Adding Multiple Interface Support in NS-2

Ramón Agüero Calvo
University of Cantabria
ramon@tlmat.unican.es

Jesús Pérez Campo
University of Cantabria
jesus@tlmat.unican.es

January, 2007

Acknowledgments

The author would like to thank all the people who have contributed to enhance this document with their comments and suggestions. Ramón Agüero would like to acknowledge the help provided by Rubén Ansótegui Boada, who compiled all the changes which were needed on the AODV code, and helped to build the corresponding section on the document.

Contents

List of Figures

Listings

Chapter 1

Introduction

There is now commonly accepted that the presence of multi-interface enabled devices is going to be very likely in the near future. The rapid growth of IEEE 802.11 technology has eased the sharp decrease of the corresponding products' prices and therefore, their presence is each day more and more common. This has gathered the interest of the *Network Simulator* (*ns*) community, since a lot of researchers are willing to extend their simulation models to incorporate multiple interfaces.

This document aims at being a guide for all researchers that want to incorporate multiple channel support to the core of the current version of the simulator *ns*, *ns-2*. In this sense, there has been quite a lot of discussion about this topic in the corresponding mailing-lists and fora. There are some other people who have already addressed the same aspect; however, our understanding is that the available information is not complete or it is very specific to certain problems, so our goal is to provide a more generic solution, allowing the user to have complete flexibility when configuring the scenario.

1.1 Related Work

As has been mentioned before, we have seen quite a lot of interest from the *ns* community in trying to accommodate multiple interfaces in the simulator model. Some of the approaches that have been used are, or have been, public available. In the remaining of this section, we will analyze three of the most relevant approaches. Although none of them completely fulfilled our requirements, they definitively provided us with interesting ideas. Below we discuss the main characteristics of each of them, enumerating also the main drawbacks they had, according to our view.

1.1.1 MITF

This is a project not longer active, which was carried out at the University of Río de Janeiro. The goal was to implement multiple interfaces, and to adapt the AODV routing protocol accordingly, and it was done using *ns-2.28*. However, since the project stopped, it was not possible to fully evaluate concrete results of this research.

Most of the modifications that were made on the simulator were within its C++ files. More specifically, for all the different modules which are part of the *MobileNode* architecture (see Chapter 1.3, e.g. LL, ARP, MAC, etc, arrays (lists of variables) with as many elements as the maximum number of channels that could be simulated were used, instead of simple variables, which is the original approach used by the simulator. In this way, it was possible to refer to the appropriate module (array model) using the correct channel as an index to locate the corresponding target within the aforementioned list. In addition, two new arrays were created in the *MobileNode* class, so as to manage the lists of nodes associated to each channel, again using the channel as the index to access these two new arrays.

On the other hand, both the *Tcl* and the implementation of the AODV routing agent were modified so that the multi-interface capability could be exploited from the routing protocol. Although the development was not completely finished, we got a number of interesting ideas, which we partially used in our own development.

1.1.2 TENS

This project [1] was done at the Indian Institute of Technology of Kanpur, India. Its main objective was to improve the *ns-2.1b9a* implementation of the IEEE 802.11 protocol on various aspects, like the MAC and physical layers' model, as well as adding multiple interface support for that specific *ns* version. The implemented multi-interface model is based on multiplexing, within the C++ implementation of the physical layer; a channel number, specified from the *Tcl* script, was used to select the appropriate channel. This multi-interface model aimed at emulating the different channels used by the DSSS version of the IEEE 802.11 standard (also accounting for the interference) and does not really reflect the requirements we originally had (see Chapter 2), since we were willing to create a number of orthogonal interfaces, mainly on *Tcl*, bringing about the possibility of implementing heterogenous interfaces.

This implementation modified different C++, as well as *Tcl*, files. One of the most important aspects was the way different interfaces were incorporated into the node from the *Tcl* code; the approach is quite similar to the one used by the Hyacinth project (see Section 1.1.3). In this case a loop was added to the `add-interface` procedure of the `ns-mobilenode.tcl` file, so as to create more than one complete physical layers, i.e. embracing MAC, LL and IFQ (see Chapter 2), per node. This method is of particular interest in our implementation, as will be discussed later.

1.1.3 Hyacinth

This is probably the closest work to ours. The corresponding project was originally carried out at the the State University of New York for *ns-2.1b9a* [2], and there is available information on how to use it over *ns-2.29* [3, 4]. Its main drawback is that it provides quite a static configuration, in which all nodes within the scenario need to incorporate up to 5 different interfaces; in addition, a static (manual configurable) routing agent was implemented to use this multi-channel capability, and according to our best knowledge, there is not available information on how to modify existing routing agents (e.g. AODV) so as to be able to use the multi-interface capability.

After defining up to 11 different channels (thus emulating the IEEE 802.11b physical layer) in the simulator script, five of them are assigned to each node, by means of the `node-config` command. Hence, the corresponding procedures were added within the `ns-lib.tcl` file. Afterwards, the `create-wireless-node` procedure, also within the the same file, calls five times the `add-interface` procedure, each of them with a different channel. To our best knowledge, these code segments are always executed, no matter whether the user was interested in having such number of interfaces within an specific node. Looking at the changes that are required at the `mac-802_11.cc` file, it seems that in order to guarantee a correct behavior, all nodes within the scenario need to have the same number of interfaces (5 in this case), and, in addition, there is a strong relationship between them, since they are always ordered according to the channel they belong to.

On the other hand, as has been already mentioned, the original work was based on a static, manual routing agent, which was configured (i.e. processes to add and delete routes) from the scenario script. Therefore, it is not straightforward extending the use of multiple interfaces to different routing protocols.

1.2 Objective of the Document

We have seen in the previous section that there is not a comprehensive, documented, way to extend *ns-2* model (at least according to our best knowledge) to add multiple interfaces on a flexible way, nor instructions on how to modify routing protocols so as to be able to use this new feature. Hence, the main goal of this technical report is to provide a extensive, though concise, set of changes that need to be performed on the simulator framework, so as to allow, first, to use a flexible number of interfaces per node (i.e. not all nodes within the same scenario need to employ the same interfaces) and, second, to modify routing protocols (existing and new ones) so as to be able to benefit from this capability.

On the other hand, the document assumes some basic knowledge about the *ns* framework. The most comprehensive information can probably be found at its manual [5], but there are several other sources available.

1.3 Structure of the Document

The report is organized as follows: next Chapter presents the revised architecture that we implement, based on the original *MobileNode*, Chapter 3 describes the changes that are required in the *Tcl* code of the simulator, while Chapter 4 discusses which changes are required in the C++ files. The latter does not include how routing agents need to be adapted so as to use multiple interfaces, since this is discussed in Chapter 5. Chapter 6 describes a potential scenario script that could be used so as to introduce the multi-interface support in the simulation, while Chapter 7 introduces some open future working items, which appear after introducing the capability to incorporate multiple interfaces on the simulator architecture.

1.4 Disclaimer

We do not guarantee the correct operation of these instructions over all *ns-2* versions, nor its straightforward usage; interested users may need to perform some “additional” changes and modifications. We will appreciate any feedback on the information provided within the text, so as to make this report as thorough as possible.

Chapter 2

Multiple Interface Model

As we have discussed in the previous chapter, one of the results of our research about the current situation on the extension of the *Network Simulator* framework to include multiple interfaces was that none of the existing solutions completely fulfilled with our requirements. In this chapter we will describe the architecture of the new *MobileNode* model that we have implemented, based on the aspects we would like to have in the simulation.

It is worth mentioning that although the simulator has two different models for mobile nodes, we will just focus on *MobileNode*, since the other one (*SRNode*) is only used by the DSR protocol. However, we understand that the information provided within this document should be enough so as to face the required changes also within the *SRNode*. Figure 2.1 shows the original architecture of the *MobileNode*, which consists, below the "Routing Agent", of a chain of modules, emulating the different protocol stack entities that any host would have in the real life: "Link Layer", "MAC Protocol", "ARP", "Interface Queue", "Network Interface", all of them connected to the same shared wireless channel. In addition, the "Propagation Model" is used so as to simulate the effect of the real wireless channels on the transmitted signal; more specifically, the propagation loss is modeled, either on a deterministic or a random way.

2.1 Requirements and Working Assumptions

In this section we present all the requirements that we would like to fulfill with our development, and we also enumerate the working assumptions that we have made. First, we opted for using different instances of the wireless channels at the *Tcl* level, rather than multiplexing them on a single object (as was done e.g. by the TENS project, see Section 1.1.2), since this is probably better aligned with the intrinsic architecture of the simulator. Using different instances of the channel at the *Tcl* level provides also a greater flexibility and eases the changes that are required within the corresponding C++ files. Another additional advantage of this approach is that in this way it is easier to change their characteristics (e.g. transmitting power or energy levels) from the scenario script. Hence, one aspect that will not be added to our implementation is the inter-channel interference.

Furthermore, and in contrast with the previous works on this issue, one of the most relevant aspects of our implementation is that it should allow the user to define a different number of interfaces per node, i.e. not all nodes need to implement the same number of interfaces. In addition, the number of channels used in a single simulation could also be parameterized and nodes should be able to randomly connect to a subset of the defined wireless channels,

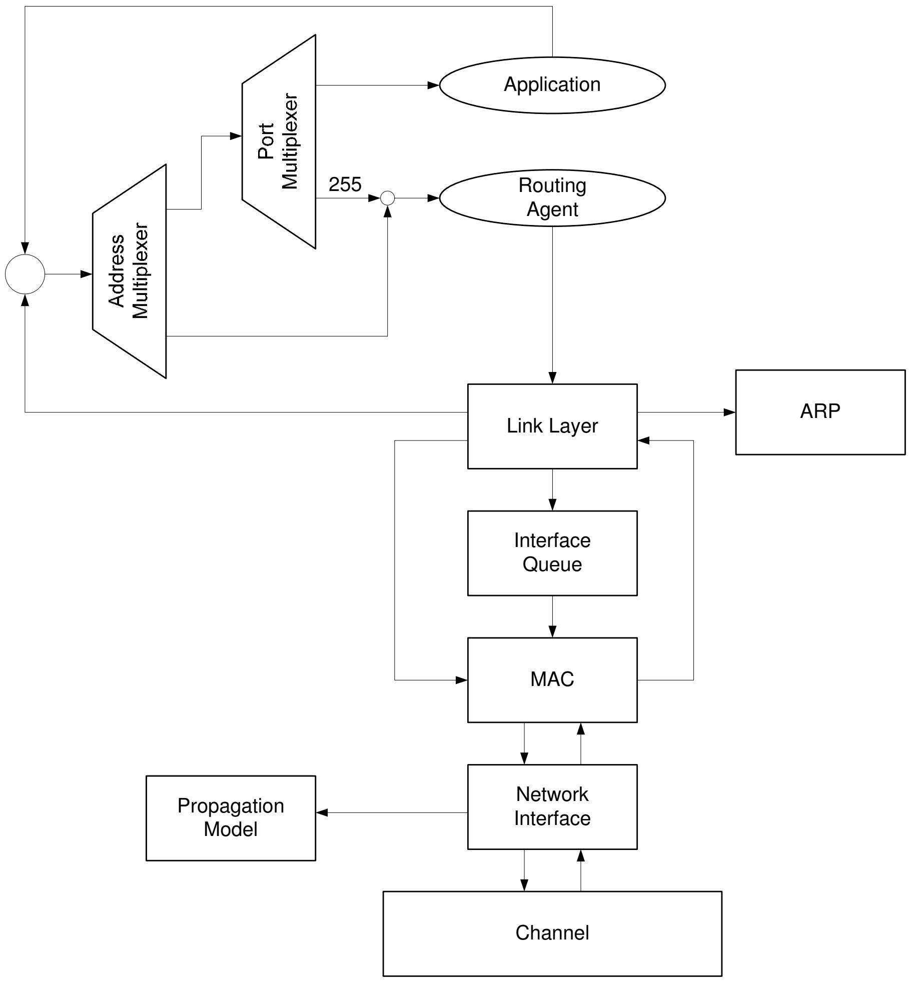

Figure 2.1: *MobileNode Architecture*

thus giving a complete flexibility to the user. We understand that this level of flexibility, that needs to be accomplished from the scenario script, would be really important so as to evaluate different types of situations.

In addition, our intention is that the modified model could be used with any of the existing (or new) routing agents (but the ones based on the *SRNode*), but it would also be nice being able to maintain the legacy behavior of the simulator, so that already existing scripts would still be valid. One of the drawbacks that we observed on the previous works on this aspect is that they usually force the simulations to use their particular characteristics or, otherwise, the simulator will not probably work properly.

Taking all the above into consideration we can summarize the requirements we would like to cope with as follows:

- **[REQ.1]** The number of channels in a particular scenario should be modifiable.

- [**REQ.2**] The number of interfaces per node is variable, and do not need to be the same for all nodes within a single scenario.
- [**REQ.3**] Each node within the same scenario could connect to a different number of channels (of the ones that had been previously defined).
- [**REQ.4**] Routing agents may take advantage of the modified model, but legacy operation of the simulator must be preserved, so as to ensure backwards compatibility.

2.2 Multiple Interface Node Model

Taking the discussion of the previous section into consideration, Figure 2.2 presents the high level architecture of the "modified" *MobileNode*. As can be seen, each node would have as many copies of the original chain of entities (the one shown before) as many interfaces it has. In addition, the single module which is not repeated is the "Propagation Model", since our initial assumption was to work exclusively with IEEE 802.11 networks, in which nodes could use more than one channel at the same time; in these circumstances, the use of a single propagation model is sensible. However, it should not be too complicated being able to extend the current model so as to be able to add flexibility also on the number and types of propagation models to be used.

For incoming traffic, there are not many differences to the original operation of the simulator. Incoming packets arrive through the corresponding channel and travel through the different entities in ascending order; since the last module of every interface, the "Link Layer" is connected to the same common point (the "Address Multiplexer", all packets are handled by the appropriate agent (either the routing protocol or the application), independently of the interface the originally arrived through.

On the other hand, for outgoing traffic, it is worth highlighting that the intelligence of selecting the appropriate interface needs to be within the routing agent; as can be seen, this is the point in which the decision needs to take place. In Chapter 5, the changes that are required in their C++ implementation to be able to select the appropriate interface are extensively discussed.

The following chapters describe the changes that are required in the simulator implementation so as to use the described model. As has been already mentioned, most of the changes are carried out within the *Tcl* code (see Chapter 3), but in addition some C++ files need to be modified (see chapter 4).

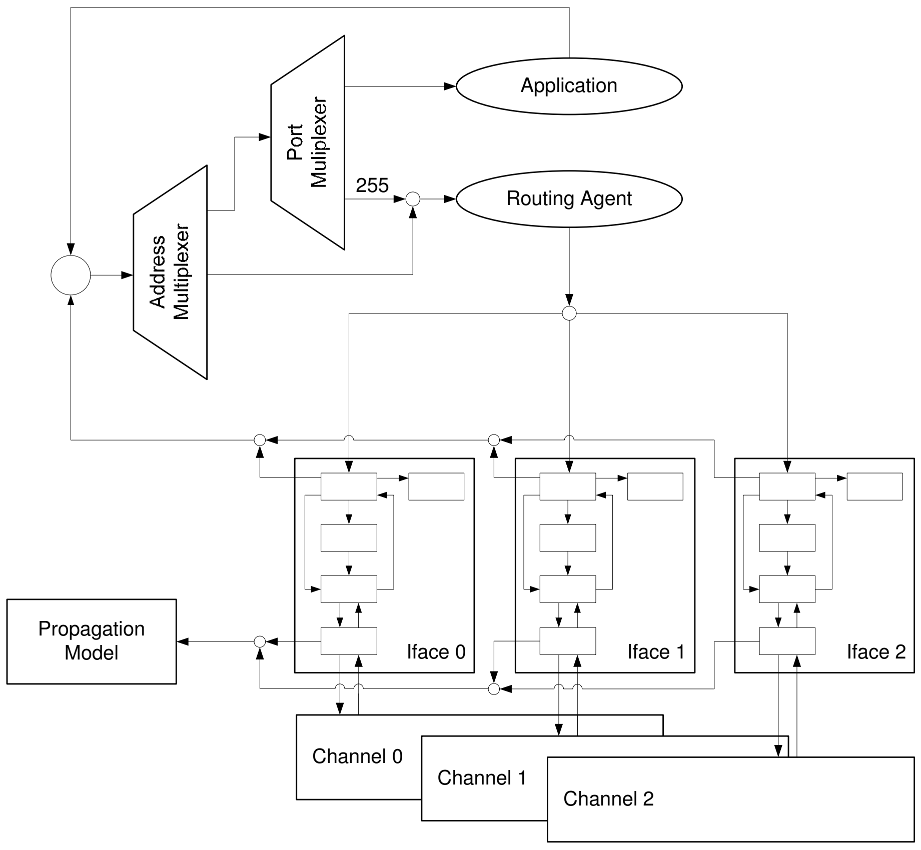

Figure 2.2: *Modified MobileNode architecture, with multiple interface support*

Chapter 3

Changes on Tcl Code

3.1 Introduction

As has been discussed within Chapter 2, we decided to base the multi-interface extension mostly on the *Tcl* implementation of the simulator, since we believed that this would probably better fit the intrinsic operation of the *ns* architecture. Most of the used *Tcl* procedures are either within the `tcl/lib/ns-lib.tcl` or `tcl/lib/ns-mobilenode.tcl` files. In this chapter we detail the changes that are required within each of them so as to extend the simulator to be able to add multiple interfaces per node.

3.2 Changes on `ns-lib.tcl`

One aspect that is worth mentioning is that the developments described herewith do require creating the channel before calling the `node-config` procedure (providing it as an argument), instead of specifying the type of channel within the `node-config` call. Chapter 6 extensively discusses the script that needs to be used so as to simulate multiple interfaces.

We need to create four new procedures. The first one `change-numifs` (see Listing 3.1) is called before creating the wireless node, and allows the user to specify a different number of interfaces per node. Obviously, if called only once, it will affect all nodes. This procedure, as will be seen later, is called from the scenario script, and needs one argument, being the number of interfaces that a particular node has.

Listing 3.1: (`ns-lib.tcl`) Procedure to change the number of interfaces

```
Simulator instproc change-numifs {newnumifs} {
    $self instvar numifs_
    set numifs_ $newnumifs
}
```

The second one allows to add an interface (channel) to a node; it must be called, also from the scenario script, before the node is created, and requires two arguments: the first one is the index of such interface within the node, while the second one is the channel itself (*Tcl* object previously created).

The third procedure we have added within the `ns-lib.tcl` does not need to be called from the scenario script, but is required so that we can gather the number of interfaces from

Listing 3.2: (`ns-lib.tcl`) Procedure to add an interface on a node

```
Simulator instproc add-channel {indexch ch} {
    $self instvar chan
    set chan($indexch) $ch
}
```

other parts of the *Tcl* architecture, as will be seen later. It is called `get-numifs` and it is shown in Listing 3.3.

Listing 3.3: (`ns-lib.tcl`) Procedure to get the number of interfaces

```
Simulator instproc get-numifs { } {
    $self instvar numifs_
    if [ info exists numifs_ ] {
        return $numifs_
    } else {
        return ""
    }
}
```

Last, we need to create another procedure, so that we can add the number of interfaces as an argument to the `node-config` command of the *Tcl* script, as will be seen in Chapter 6.

Listing 3.4: (`ns-lib.tcl`) Procedure to add multiple interfaces as an argument to node-config label

```
Simulator instproc ifNum {val} {$self set numifs_ $val}
```

In addition to the new ones shown above, two of the already existing procedures need to be modified. The first one is `node-config`, in which we have to initialize the `chan` variable, either as a single variable, if normal operation is being used, or as an array, when the multi-interface is enabled. Furthermore, we add the `numifs_` variable to the list of arguments passed to the method. The changes are shown on Listing 3.5.

Listing 3.5: (`ns-lib.tcl`) Changes on node-config procedure

```
Simulator instproc node-config args {
        # Object::init-vars{} is defined in ~tclcl/tcl-object.tcl.
        # It initializes all default variables in the following
            way:
        #  1.  Look for pairs of {-cmd val} in args
        #  2.  If "$self $cmd $val" is not valid then put it in a
            list of
        #      arguments to be returned to the caller.
        #
        # Since we do not handle undefined {-cmd val} pairs, we
            ignore
        # return value from init-vars{}.
        set args [eval $self init-vars $args]
```

```
    $self instvar addressType_  routingAgent_ propType_
        macTrace_ \
    routerTrace_ agentTrace_ movementTrace_ channelType_
        channel_ numifs_\
    chan topoInstance_ propInstance_ mobileIP_ rxPower_ \
    # change wrt Mike's code
txPower_ idlePower_ satNodeType_ eotTrace_

    if [info exists macTrace_] {
    Simulator set MacTrace_ $macTrace_
}
    if [info exists routerTrace_] {
    Simulator set RouterTrace_ $routerTrace_
}
    if [info exists agentTrace_] {
    Simulator set AgentTrace_ $agentTrace_
}
    if [info exists movementTrace_] {
    Simulator set MovementTrace_ $movementTrace_
}

# change wrt Mike's code
if [info exists eotTrace_] {
            Simulator set EotTrace_ $eotTrace_
    }

    # hacking for matching old cmu add-interface
    # not good style, for back-compability ONLY
#
# Only create 1 instance of prop
if {[info exists propInstance_]} {
    if {[info exists propType_] && [Simulator set
        propInstCreated_] == 0} {
         warn "Both propType and propInstance are set. propType
             is ignored."
    }
} else {
    if {[info exists propType_]} {
         set propInstance_ [new $propType_]
         Simulator set propInstCreated_ 1
    }
}

# Add multi-interface support:
# User can only specify either channelType_ (single_interface
    as
# before) or channel_ (multi_interface)
# If both variables are specified, error!
if {[info exists channelType_] && [info exists channel_]} {
    error "Can't specify both channel and channelType, error!"
} elseif {[info exists channelType_] && ![info exists
    satNodeType_]} {
            # Single channel, single interface
            warn "Please use -channel as shown in tcl/ex/
                wireless-mitf.tcl"
            if {![info exists chan]} {
                set chan [new $channelType_]
            }
        } elseif {[info exists channel_]} {
            # Multiple channel, multiple interfaces
            if {[info exists numifs_]} {
                set chan(0) $channel_
            } else {
                set chan $channel_
            }
        }
        if [info exists topoInstance_] {
            $propInstance_  topography $topoInstance_
        }
        # set address type, hierarchical or expanded
        if {[string compare $addressType_ ""] != 0} {
            $self set-address-format $addressType_
        }
        # set mobileIP flag
        if { [info exists mobileIP_] && $mobileIP_ == "ON"} {
            Simulator set mobile_ip_  1
        } else {
            if { [info exists mobileIP_] } {
                Simulator set mobile_ip_ 0
            }
        }
}
```

The changes are highlighted with bold font on the above listing. As can be seen, we maintain the legacy operation of the simulator, and we do only modify it provided that the multi-interface extension has been set from the scenario script.

Listing 3.6 shows the other procedure which needs to be modified, `create-wireless-node`. In this case, when the extension is being used, the `add-interface` procedure, which is defined in the `ns-mobilenode.tcl` file, has to be called as many times as the number of interfaces the node has, and a `for` loop is used for this.

Listing 3.6: (`ns-lib.tcl`) Changes on create-wireless-node procedure

```
Simulator instproc create-wireless-node args {
        $self instvar routingAgent_ wiredRouting_ propInstance_
            llType_ \
        macType_ ifqType_ ifqlen_ phyType_ chan antType_
            energyModel_ \
        initialEnergy_ txPower_ rxPower_ idlePower_ \
        topoInstance_ level1_ level2_ inerrProc_ outerrProc_
            FECProc_ numifs_

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

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

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

# Adding Interface
if {[info exist numifs_] } {
    for {set i 0} {$i < $numifs_} {incr i} {
        # Add one interface per channel
        $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_ == "OMNIMCAST" ||
    $routingAgent_ == "Directed_Diffusion" } {
    $ragent port-dmux [$node demux]
    $node instvar ll_
    $ragent add-ll $ll_(0)
}
if { $routingAgent_ == "DumbAgent" } {
    $ragent port-dmux [$node demux]
}


# Bind routing agent and mip agent if existing basestation
# address setting
    if { [info exist 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 namattach $namtracefd
        }
        if [info exists energyModel_] {
            if  [info exists level1_] {
                set l1 $level1_
            } else {
                set l1 0.5
            }
            if  [info exists level2_] {
                set l2 $level2_
            } else {
                set l2 0.2
            }
            $node addenergymodel [new $energyModel_ $node \
                           $initialEnergy_ $l1 $l2]
            }
            if [info exists txPower_] {
            $node setPt $txPower_
            }
            if [info exists rxPower_] {
            $node setPr $rxPower_
            }
            if [info exists idlePower_] {
            $node setPidle $idlePower_
            }

        $node topography $topoInstance_

        return $node
    }
```

3.3 Changes on `ns-mobilenode.tcl`

In this case no new procedures were required, but rather some modifications have to be performed to some of the already existing ones. These are explained in the remaining of this section.

The first procedure which was modified is the `add-target`, which can be seen on Listing 3.7. First of all, the `get-numifs` procedure that was discussed before is called, so that we can assess whether we are using the multi-interface extension and, if such is the case, the number of interfaces that the current node has. Later, this number is used to call the "new" `if-queue` command of the routing agent (see Chapter 5) as many times as the number of interfaces the node has.

Listing 3.7: (`ns-mobilenode.tcl`) Changes on add-target procedure

```
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

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

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

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

    #<zheng: add>
    # Special processing for ZBR
    #set zbronly [string first "ZBR" [$agent info class]]
    #if {$zbronly != -1 } {
    #   $agent if-queue [$self set ifq_(0)]   ;# ifq between LL and
        MAC
    #}
    #</zheng: add>

    if { $port == [Node set rtagent_port_] } {
        # Special processing when multiple interfaces are supported
        if {$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
    }
}
```

The second procedure that needs to be modified is the `add-target-rtagent`, which is called from the previous one, when the agent is attached to the RT_PORT port and, thus it is a routing agent. As we did before, we use the get-numifs procedure to get the number of interfaces that the node has (provided that the multi-hop extension is being used) and we later use this variable so as to link the routing agent with the corresponding link layer (ll_) entities, which were initialized before, after the subsequent calls to the add-interface procedure. The variable numIfsSimulator allows us to preserve the original behavior of the simulator, since the legacy code is still used when this variable does not have a valid value. We do this both when the tracing support is activated and when it is not.

Listing 3.8: (ns-mobilenode.tcl) Changes on add-target-rtagent procedure

```
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]

    # We see whether we have multiple interfaces 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
            }
            $sndT target [$self set ll_(0)]
        } else { ;# no IMEP
            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" } {
            [$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
                $classifier_ defaulttarget $rcvT2
            }
        } else {
            $rcvT target $agent
            $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
                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 last procedure that needs to be modified is the `add-interface`; originally we did not touch this one, since the multi-interface support was brought about by the `for` loop that was added into the `create-wireless-node` procedure. However, if no changes were made, the model that has been presented on Figure 2.2 would not have been completely accurate, since the original `add-interface` method would have just created one ARP table per node, instead of one ARP table per interface. Although it could be argued that having one ARP per node would have been closer to a realistic case, we noticed that this could lead to a wrong behavior. E.g. if a node has already used one interface to communicate with another one, it will not be possible trying to use another interface, as the request to the ARP entity would be answered with previous entry, which would not be longer valid. Hence, functionality-wise it is more appropriate having one ARP table per interface. To achieve this, as already mentioned, we needed to make some changes on the `add-interface` procedure, as shown on Listing 3.9.

Listing 3.9: (`ns-mobilenode.tcl`) Changes on add-interface procedure

```
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]

$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)

# We also create one ARP table per interface
```

```
set arptable_($t) [new ARPTable $self $mac]
set arptable $arptable_($t)

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
$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
}
#
# 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 != ""} {
    $mac down-target $fec
    $fec down-target $netif
} else {
    $mac down-target $fec
    $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 != ""} {
    $netif up-target $fec
    $fec up-target $mac
} else {
    $netif 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 eoT [cmu-trace EOT "MAC" $self]
               }
               $mac eot-target $eotT
        }



    # ==========================================================================

    $self addif $netif
}
```

This latter change affects the way the *MobileNode* is created (see Listing 3.10) and reset (see Listing 3.11).

Listing 3.10: (`ns-mobilenode.tcl`) *MobileNode* init procedure

```
Node/MobileNode instproc init args {
#   # I don't care about address classifier; it's not my business
#   # All I do is to setup port classifier so we can do broadcast,
#   # and to set up interface stuff.
#   $self attach-node $node
#   $node port-notify $self

    eval $self next $args

    $self instvar nifs_ arptable_ X_ Y_ Z_ nodetype_
    set X_ 0.0
    set Y_ 0.0
    set Z_ 0.0
#   set arptable_ ""                    ;# no ARP table yet
    set nifs_   0       ;# number of network interfaces
    # Mobile IP node processing
        $self makemip-New$nodetype_
}
```

Listing 3.11: (`ns-mobilenode.tcl`) *MobileNode* reset procedure

```
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
            }
            if { $arptable_($i) != "" } {
                $arptable_($i) reset
            }
    }
}
```

Chapter 4

Changes on C++ Code

4.1 Introduction

As we have already said, most of the changes have been done within the *Tcl* implementation of the simulator. However, some modifications need to be done, also within the C++ files, basically to be able to adapt to the new framework. Most of the changes affect how the simulator deals with the *MobileNode* class and are discussed in the remaining of this section.

4.2 Changes on `mobilenode.[cc,h]`

After creating the multiple interface structures for mobile nodes on *Tcl*, it is necessary to correctly associate them to the appropriate channel. The simulator controls the nodes which are connected to a channel by means of a list which is managed using two pointers (one to the previous and another one to the next node on the list). These pointers were originally simple variables; however, if we wish to manage several channels, it is required to create two arrays of pointers with as many elements as channels exist within the simulation scenario. In this sense, it becomes easy managing the nodes of one particular channel, i.e. referring to it using the channel number as the array index, thus being able to move to either the previous or the next element of the list. These modifications are made in the `mobilenode.h` file as shown in Listing 4.1; obviously the `MAX_CHANNELS` variable needs to be defined before.

Listing 4.1: (`mobilenode.h`) New declaration of `MobileNode` lists within `MobileNode` class

```
...
/* For list-keeper */
MobileNode* nextX_[MAX_CHANNELS];
MobileNode* prevX_[MAX_CHANNELS];
...
```

After performing all the changes needed to adapt to the new definitions of the `nextX_` and `prevX_` variables (see next Section) we detected quite a weird behavior on the simulator. The call to the original `getLoc` method, which was declared as `inline` within the `MobileNode` class did not work properly, as it always returned a zero distance, thus leading to wrong packet receptions, no matter the real distance between nodes. In order to solve this problem, we changed the method declaration, so that it was not `inline` anymore, as shown in Listing 4.2.

Listing 4.2: (`mobilenode.h`) New `getLoc` method declaration within `MobileNode` class

```
...
void start(void);
void getLoc(double *x, double *y, double *z);
inline void getVelo(double *dx, double *dy, double *dz) {
    *dx = dX_ * speed_; *dy = dY_ * speed_; *dz = 0.0;
}
...
```

In addition, we added the method definition within the `mobilenode.cc` file (see Listing 4.3).

Listing 4.3: (`mobilenode.cc`) `getLoc` method definition

```
void
MobileNode::getLoc(double *x, double *y, double *z)
{
    update_position();
    *x = X_;
    *y = Y_;
    *z = Z_;
}
```

4.3 Changes on `channel.cc`

The two arrays mentioned above are used within the `channel.cc` file so as to manage the corresponding node lists (e.g. attaching a new node to a channel, removing, updating, etc). In order to refer to the appropriate list, the index of the corresponding channel has to be used, as shown in Listing 4.4, where `this->index()` refers to the correct one. Note that this has to be changed throughout the whole `channel.cc` file.

Listing 4.4: (`channel.cc`) Accessing the appropriate MobileNode list

```
nextX_[this->index()]
prevX_[this->index()]
```

Furthermore, when a packet is sent, a previous evaluation procedure is performed so as to ensure that it is sent to the correct destination. The first step is to assess which nodes are close enough to the source and are also connected to the channel that will be used for this communication (this last condition is automatically checked because the right list of nodes for this channel had been previously selected, thanks to the management changes explained before). Then, the packet could be sent to all of the interfaces in the destination node. However, this makes little sense, since the packet should only be received from the interface connected to the appropriate channel. In this sense, we had to add a new condition so as to check which of the interfaces of the destination node is connected to the same channel that will be used to transmit the packet. The code already modified in `channel.cc` is shown in Listing 4.5.

Listing 4.5: (`channel.cc`) `affectedNodes` method from the *channel* class

```
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()){
        if(rifp->channel() == this) {
             s.schedule(rifp, newp, propdelay);
        }
    }
}
delete [] affectedNodes;
```

4.4 Changes on `mac-802_11.cc`

The last change in the C++ code is needed so as to be able to identify the interface which a message was received through. This is mandatory for the correct handling of multiple interfaces by the routing agents, as will be explained in Chapter 5. The code which is shown in Listing 4.6 needs to be added to the `mac-802_11.cc` file.

Listing 4.6: (`mac-802_11.cc`) Registering the correct MAC receiving interface within the `recv` method of the `Mac802_11` class

```
...
if(tx_active_ && hdr->error() == 0) {
    hdr->error() = 1;
}
hdr->iface() = addr();
if(rx_state_ == MAC_IDLE) {
...
```

Chapter 5

Changes on Routing Protocol Code

5.1 Introduction

It goes without saying that the final goal of implementing the multi-interface model is to make use of it. Hence, external agents from the simulator architecture must be changed so as to use this new feature. In this section we show how a routing agent needs to be adapted in order to use the multiple interface structure previously discussed, so that it is possible to assess the benefits of the modified model.

These changes have been tested on a proprietary implementation of an ad-hoc routing protocol, which follows the same approach as the original AODV implementation from the simulator does. Nonetheless, they are generic enough, so it should not be too complicated extending them to other agents.

In order to fully understand the discussions of this chapter, a good knowledge of how routing protocols are implemented within the simulator is required. The reader may refer to [6] for a good overview on this issue.

5.2 Changes on routing agent implementation

Since we want the number of interfaces per node to be flexible, and furthermore, we want to maintain the legacy behavior of the simulator, it is required that the routing agent keeps track of the number of interfaces it is managing. A new member of the routing agent class, `nIfaces`, is declared so as to keep this information. In our approach, the interfaces will be defined stepwise from the scenario script (see Chapter 6), so at the beginning its value is set to 0 (i.e. in the `constructor` of the agent).

As can be seen on Figure 2.2, it is the routing agent that needs to decide upon the outgoing interface it needs to pass the packet to. Instead of using the traditional single `ifqueue` and `target` that any routing agent may use, we declare two arrays, `targetlist` and `ifqueuelist`. The first one stores the LL modules for all the interfaces a particular node has, whilst the second one keeps their corresponding queues. Listing 5.1 shows the lines that are required to be added within the header file of the routing agent (within the class declaration). `MAX_IF` needs to be declared beforehand.

Listing 5.1: (`routingAgent.h`) New class members to manage multiple interfaces

```
int nIfaces;
```

```
NsObject *targetlist[MAX_IF];
PriQueue *ifqueuelist[MAX_IF];
```

The next step would be to modify the `command` method of the routing agent class, so as to initialize the values of the aforementioned variables from the *Tcl* script and to make use of them. Listing 5.2 shows how the aforementioned arrays, `ifqueuelist` and `targetlist` get populated, taking the corresponding values while the interfaces are being created in the nodes. At the same time we increase the value of the variable that maintains the number of interfaces used by the node, for each of the interfaces which are added.

Listing 5.2: (`routingAgent.cc`) Changes on `command` method of the routing agent class

```
else if(argc == 4) {
    if(strcmp(argv[1],"if-queue") == 0) {
        PriQueue * ifq = (PriQueue *) TclObject::lookup(argv[3]);
        int temp_ = atoi(argv[2]);
        if(temp_ == nIfaces) {
            nIfaces++;
        }
        ifqueuelist[temp_] = ifq;
        if (ifqueuelist[temp_]) {
            return TCL_OK;
        } else {
            return TCL_ERROR;
        }
    }
    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;
        }
    }
}
```

With all the previous changes it is indeed possible to create multi-interface nodes. The next thing is to add the required intelligence within the routing agent implementation, so that it can decide the interface which needs to transmit each packet. On the other hand, it is also well known that the use of broadcast transmissions is quite relevant in routing protocols for ad hoc networks (e.g. during the *Route Discovery* process); when there is more than one available interface, a broadcast packet (typically a *Route Request*) needs to be transmitted through all the interfaces a node has. Listing 5.3 shows how this can be accomplished. We use a loop to send the packet through all the interfaces, adding some random time to avoid collisions. Note that for each of the interfaces a copy of the original packet is sent, since their route through the simulator entities will be different onwards. In addition, to preserve the traditional behavior of the simulator, the new code is only executed if the multi-interface extension has been initialized from the scenario script, i.e. `nIfaces` $\neq 0$. If such is not the case, the routing agent performs the broadcast transmission as it would have done without the extension.

Listing 5.3: (`routingAgent.cc`) Sending a broadcast packet

```
if(nIfaces) {
    for(int i = 0; i < nIfaces;i++){
        Packet *p_copy = pkt->copy();
        Scheduler::instance().schedule(targetlist[i],p_copy,JITTER)
            ;
    }
    Packet::free(pkt)
else {
    Scheduler::instance().schedule(target_,pkt,JITTER);
}
```

On the other hand, for unicast transmissions, an index `Iface` (see Listing 5.4) is used so as to select the appropriate `target` (i.e. the LL entity of the interface the packet needs to be sent to). This value must be carefully selected and it needs to be kept at the routing table, together with the rest of information that needs to be therein (see Section 5.3). In this sense, the method used to create a new routing table entry needs to be updated so as to indicate the output interface that has to be used to reach the destination. It obviously belongs to the range of interfaces that are used by the particular node (`Iface` $\in [0, \texttt{nIfaces} - 1]$). Note that, as was done for the broadcast case, we apply the multi-interface extension only when the user had previously initialized such extension.

Listing 5.4: (`routingAgent.cc`) Sending a unicast packet

```
if(nIfaces) {
    Scheduler::instance().schedule(targetlist[Iface], pkt,0);
} else {
    Scheduler::instance().schedule(target_,pkt,0);
}
```

The challenge is to be able to associate the interface with the `Iface` index, e.g. when a new entry has to be introduced within the route table. In order to accomplish this, Listing 5.5 shows the code that needs to be used. `cmn->iface()` stores the address of the incoming interface, as discussed in Section 4.4; on the other hand the second term is the address of the first interface of the node. Taking advantage from the fact that the interfaces are gradually added (see Chapter 6), this simple expression allows us to easily refer to the appropriate interface. If the multi-interface extension is not being used, we assign the index a non-valid value.

Listing 5.5: (`routingAgent.cc`) Getting the interface index

```
if(nIfaces) {
    Iface = cmnh->iface()-((Mac *)ifqueuelist[0]->target())->addr()
        ;
} else {
    Iface = −1;
}
```

5.3 Changes on the Route Table

We have just seen how the routing agent is able to ascertain the interface from which any packet had been received. This information must be stored in the route table entry for the corresponding destination so that it can be used on future transmissions; that is to say, in order to route a packet to a destination it is not enough to know the next hop, but the routing agent must be aware of which output interface it needs to use so as to reach it. Hence, a new variable `interface` must be added to the route entry definition.

This variable stores the index for the corresponding interface, so as to be able to refer to the appropriate array member, as has been previously explained. Using the corresponding method the entry is created or updated, including this information for multi-interface support in the route table.

5.4 Illustrative example: AODV

One of the most used routing protocols within the *Network Simulator* framework is AODV. The main reason for this is that it is included, by default, in the different *ns-2* distributions. It is sensible, thus, to compile all the changes which are needed in such routing protocol, following the guidelines provided before in this section.

5.4.1 Changes in `aodv.h`

Following Listing 5.1, there are two changes which are needed in the AODV class declaration.

First, we need to define a new constant `MAX_IF`, as shown in Listing 5.6, since it is used afterwards (see Listing 5.7) to declare the arrays which handle the list of targets and interface queues.

Listing 5.6: (`aodv.h`) Declaring the MAX_IF constant

```
// We declare the maximum number of interfaces
#define MAX_IF                    11
```

Once we have defined the new constant we need to add the new members at the end of the AODV class declaration, as shown in Listing 5.7.

Listing 5.7: (`aodv.h`) New members of the AODV class

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

    /*
```

```
         * 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_;


        // New members required for the multi-interface extension
        int nIfaces;
        NsObject *targetlist[MAX_IF];
        PriQueue *ifqueuelist[MAX_IF];
};
```

Another change which needs to be applied in this file deals with how the routing table needs to be handled. As was explained before, each routing table entry must also incorporate an index so that the routing agent is able to identify the interface through which the packet needs to be forwarded; in this case, we need to change the way the `rt_update` is called, as shown in Listing 5.8.

Listing 5.8: (`aodv.h`) New members of the AODV class

```
    ...
        /*
         * 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, u_int8_t
                                    interface);
        void            rt_down(aodv_rt_entry *rt);
        void            local_rt_repair(aodv_rt_entry *rt, Packet *
            p);
public:
        void            rt_ll_failed(Packet *p);
        void            handle_link_failure(nsaddr_t id);
protected:
        void            rt_purge(void);

        void            enque(aodv_rt_entry *rt, Packet *p);
        Packet*         deque(aodv_rt_entry *rt);
    ...
```

5.4.2 Changes in `aodv.cc`

In this case we need to apply those changes which are depicted in Listings 5.2, 5.3, 5.4, 5.5. Some of them need to be applied in more than one AODV method, while there are other additional places in which the interface with the routing table needs to be adjusted accordingly. It is also convenient to ensure that the number of interfaces per node is initialized in the constructor, so as to maintain the legacy behavior of the AODV protocol. This can be seen in Listing 5.9.

Listing 5.9: (`aodv.cc`) Changes on the AODV 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;
}
```

Before describing the changes which are required on those methods which deal with the transmission and reception of packets, the `command` method needs to be modified, in order to adapt it to the new architecture, as it was previously discussed in Listing 5.2. As can be seen in Listing 5.10, the modifications are very much the same for the case of the AODV agent.

Listing 5.10: (`aodv.cc`) Changes on the command method

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

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

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

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

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

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

  else if(strcmp(argv[1], "log-target") == 0 || strcmp(argv[1], "
     tracetarget") == 0) {
    logtarget = (Trace*) TclObject::lookup(argv[2]);
    if(logtarget == 0)
      return TCL_ERROR;
    return TCL_OK;
  }
  else if(strcmp(argv[1], "drop-target") == 0) {
  int stat = rqueue.command(argc,argv);
    if (stat != TCL_OK) return stat;
    return Agent::command(argc, argv);
  }
  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;
  }
}
else if(argc == 4) {
  if(strcmp(argv[1],"if-queue")==0) {
      PriQueue * ifq = (PriQueue *) TclObject::lookup(argv[3]);
      int temp_ = atoi(argv[2]);
      if(temp_ == nIfaces) {
          nIfaces++;
      }
      ifqueuelist[temp_] = ifq;
      if (ifqueuelist[temp_]) {
          return TCL_OK;
      } else {
          return TCL_ERROR;
      }
  }
  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;
            }
        }
    }
    return Agent::command(argc, argv);
}
```

Listing 5.3 has to be used whenever the AODV needs to send a broadcast packet, which happens, as far as we can tell, in the following methods of the `aodv.cc` file: `sendRequest`, `sendError` and `sendHello`. The corresponding changes are highlighted in the following listings.

Listing 5.11: (`aodv.cc`) Changes on the sendRequest method

```
void
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_timeout = CURRENT_TIME + MAX_RREQ_TIMEOUT;
    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
  // Also note that we are making timeouts to be larger if we have
  // done network wide broadcast before.

  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 id 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;

if(nIfaces) {
    for(int i=0; i<nIfaces; i++) {
        Packet *p_copy = p->copy();
        Scheduler::instance().schedule(targetlist[i], p_copy, 0.0);
    }
    Packet::free(p);
 }
 else {
    Scheduler::instance().schedule(target_, p, 0.0);
 }
}
```

Listing 5.12: (`aodv.cc`) Changes on the sendError method

```
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;

  if (jitter) {
        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());
             }
           Packet::free(p);
        }
        else {
            Scheduler::instance().schedule(target_, p, 0.01*Random
               ::uniform());
        }
 }
 else {
        if(nIfaces) {
             for(int i=0; i<nIfaces; i++) {
                   Packet *p_copy = p->copy();
                   Scheduler::instance().schedule(targetlist[i],
                      p_copy, 0.0);
             }
            Packet::free(p);
        }
        else {
            Scheduler::instance().schedule(target_, p, 0.0);
        }
 }

}
```

Listing 5.13: (aodv.cc) Changes on the sendHello method

```
void
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;


 if(nIfaces) {
    for(int i=0; i<nIfaces; i++) {
        Packet *p_copy = p->copy();
        Scheduler::instance().schedule(targetlist[i], p_copy, 0.0);
    }
    Packet::free(p);
 }
```

```
  else {
     Scheduler::instance().schedule(target_, p, 0.0);
  }
}
```

Additionally we also need to take into account the changes which are required whenever a unicast transmission needs to be performed (see Listing 5.4). For the AODV case, these included the `sendReply` method and the `forward` method, which also includes some broadcast mechanisms. Note that in both cases, we change the `Iface` index with the one which is provided by the routing table entry.

Listing 5.14: (`aodv.cc`) Changes on the sendReply method

```
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;
  if(nIfaces) {
        Scheduler::instance().schedule(targetlist[rt->rt_interface],
            p, 0);
  } else {
        Scheduler::instance().schedule(target_, p, 0);
  }
}
```

Listing 5.15: (`aodv.cc`) Changes on the forward method

```
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);

  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 (ih->daddr() == (nsaddr_t) IP_BROADCAST) {
  // If it is a broadcast packet
```

```
    assert(rt == 0);
    if (ch->ptype() == PT_AODV) {
      /*
       *  Jitter the sending of AODV broadcast packets by 10ms
       */
         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());
         }
         Packet::free(p);
         } else {
                 Scheduler::instance().schedule(target_, p, 0.01 *
                    Random::uniform());
         }
     } else {
         if(nIfaces) {
                 for(int i=0; i<nIfaces; i++) {
                    Packet *p_copy = p->copy();
                    Scheduler::instance().schedule(targetlist[i],
                       p_copy, 0.0);
                 }
                 Packet::free(p);
           } else {
                 Scheduler::instance().schedule(target_, p, 0.0);
           }
     }
  } else { // Not a broadcast packet
      if(delay > 0.0) {
         if(nIfaces) {
                 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) {
                 Scheduler::instance().schedule(targetlist[rt->
                    rt_interface], p, 0);
         } else {
                 Scheduler::instance().schedule(target_, p, 0);
         }
     }
  }
}
```

The last group of changes need to be made in order to correctly manage the routing table. In particular, and as it was already described before (see Listing 5.5), whenever we add an entry to the routing table we must include the interface which corresponds to such entry. We need to make the corresponding changes in the `recvRequest` and `recvReply` methods (see Listings 5.16 and 5.17, respectively). Note that we also need to modify the handling of the routing table, since we have to include the interface as an argument to the `rt_update` method.

Listing 5.16: (`aodv.cc`) Changes on the recvRequest method

```
void
AODV::recvRequest(Packet *p) {
struct hdr_ip *ih = HDR_IP(p);
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_aodv_request *rq = HDR_AODV_REQUEST(p);
u_int8_t Iface;
aodv_rt_entry *rt;


  /*
   * Drop if:
   *      - I'm the source
   *      - I recently heard this request.
   */
  //DBG_INFO("Node %d receives request from %d",addr(), rq->rq_src)
      ;
  if(rq->rq_src == index) {
#ifdef DEBUG
    fprintf(stderr, "%s: got my own REQUEST\n", __FUNCTION__);
#endif // DEBUG
    Packet::free(p);
    return;
  }

 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;
 }

 /*
  * Cache the broadcast ID
  */
 id_insert(rq->rq_src, rq->rq_bcast_id);



 /*
  * 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_rt_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);
  }

  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.

 if(nIfaces) {
     Iface = ch->iface()-((Mac *)ifqueuelist[0]->target())->addr()
         ;
 } else {
     Iface = -1;
 }
 rt_update(rt0, rq->rq_src_seqno, rq->rq_hop_count, ih->saddr(),
              max(rt0->rt_expire, (CURRENT_TIME + REV_ROUTE_LIFE))
                  , Iface);
    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++;

    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) ) {

    //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,
              rq->rq_timestamp);
    // 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

#ifdef RREQ_GRAT_RREP
```

```
    sendReply(rq->rq_dst,
              rq->rq_hop_count,
              rq->rq_src,
              rq->rq_src_seqno,
              (u_int32_t) (rt->rt_expire - CURRENT_TIME),
              //             rt->rt_expire - CURRENT_TIME,
              rq->rq_timestamp);
#endif

// TODO: send grat RREP to dst if G flag set in RREQ using rq->
    rq_src_seqno, rq->rq_hop_counT

// DONE: Included gratuitous replies to be sent as per IETF aodv
    draft specification. As of now, G flag has not been dynamically
     used and is always set or reset in aodv-packet.h --- Anant
    Utgikar, 09/16/02.

        Packet::free(p);
 }
 /*
  * Can't reply. So forward the  Route Request
  */
 else {
   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);
 }
}
```

Listing 5.17: (`aodv.cc`) Changes on the recvReply method

```
void
AODV::recvReply(Packet *p) {
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_reply *rp = HDR_AODV_REPLY(p);
u_int8_t Iface;
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


 /*
```

```
   *  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.
   */
  // Note that rp_dst is the dest of the data packets, not the
  // the dest of the reply, which is the src of the data packets.

   //DBG_INFO("Receive reply from ih->src()");
  rt = rtable.rt_lookup(rp->rp_dst);

  /*
   *  If I don't have a rt entry to this host... adding
   */
  if(rt == 0) {
    rt = rtable.rt_add(rp->rp_dst);
  }

  /*
   * 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

   if(nIfaces) {
       Iface = ch->iface()-((Mac *)ifqueuelist[0]->target())->addr()
           ;
   } else {
       Iface = -1;
   }
   rt_update(rt, rp->rp_dst_seqno, rp->rp_hop_count,
                rp->rp_src, CURRENT_TIME + rp->rp_lifetime, Iface);

   // 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);
      // Delay them a little to help ARP. Otherwise ARP
      // may drop packets. -SRD 5/23/99
        forward(rt, buf_pkt, delay);
        delay += ARP_DELAY;
      }
    }
  }
  else {
    suppress_reply = 1;
  }

  /*
   * If reply is for me, discard it.
   */

 if(ih->daddr() == index || suppress_reply) {
    Packet::free(p);
  }
  /*
   * Otherwise, forward the Route Reply.
   */
  else {
  // Find the rt entry
 aodv_rt_entry *rt0 = rtable.rt_lookup(ih->daddr());
    // If the rt is up, forward
    if(rt0 && (rt0->rt_hops != INFINITY2)) {
         assert (rt0->rt_flags == RTF_UP);
      rp->rp_hop_count += 1;
      rp->rp_src = index;
      forward(rt0, p, NO_DELAY);
      // Insert the nexthop towards the RREQ source to
      // the precursor list of the RREQ destination
      rt->pc_insert(rt0->rt_nexthop); // nexthop to RREQ source

    }
    else {
    // I don't know how to forward .. drop the reply.
#ifdef DEBUG
      fprintf(stderr, "%s: dropping Route Reply\n", __FUNCTION__);
#endif // DEBUG
      drop(p, DROP_RTR_NO_ROUTE);
      }
    }
  }
```

Last, but not least, we must adapt the `rt_update` method, according to the new definition which was shown in Listing 5.8.

Listing 5.18: (`aodv.cc`) Changes on the rt_update method

```
void
AODV::rt_update(aodv_rt_entry *rt, u_int32_t seqnum, u_int16_t
    metric, nsaddr_t nexthop, double expire_time, u_int8_t
    interface) {

      rt->rt_seqno = seqnum;
      rt->rt_hops = metric;
      rt->rt_flags = RTF_UP;
      rt->rt_nexthop = nexthop;
      rt->rt_expire = expire_time;
      rt->rt_interface = interface;
}
```

5.4.3 Changes on the routing table implementation `aodv_rtable.[cc,h]`

As already anticipated before, the way the routing table is handled must be adapted according to the multi-interface extension. The only change which needs to be done is to uncommment the interface component of the route table entry (`aodv_rt_entry`).

Listing 5.19: (`aodv_rtable.h`) Changes on the aodv_rt_entry class definition

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

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

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

        double           rt_req_timeout;          // when I can send
            another req
        u_int8_t         rt_req_cnt;              // number of route
            requests
 protected:
        LIST_ENTRY(aodv_rt_entry) rt_link;

        nsaddr_t          rt_dst;
        u_int32_t         rt_seqno;
        u_int8_t          rt_interface;
        u_int16_t         rt_hops;                // hop count
        int               rt_last_hop_count;      // last valid hop
           count
        nsaddr_t          rt_nexthop;             // next hop IP
           address
        /* list of precursors */
        aodv_precursors rt_pclist;
        double            rt_expire;              // when entry
           expires
        u_int8_t          rt_flags;

#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.
        u_int8_t          rt_errors;       // error count
        double            rt_error_time;
#define MAX_RT_ERROR             4         // errors
#define MAX_RT_ERROR_TIME        3         // seconds
         */

#define MAX_HISTORY       3
        double            rt_disc_latency[MAX_HISTORY];
        char              hist_indx;
        int               rt_req_last_ttl;        // last ttl value
           used
        // last few route discovery latencies
        // double                 rt_length [MAX_HISTORY];
        // last few route lengths

        /*
         * a list of neighbors that are using this route.
         */
        aodv_ncache             rt_nblist;
};
```

And, in order to ensure a correct operation, we should initialize this member to 255 (non-valid value) in the corresponding constructor.

Listing 5.20: (`aodv_rtable.cc`) Changes on the aodv_rt_entry constructor

```
aodv_rt_entry::aodv_rt_entry()
{
```

```
int i;

 rt_req_timeout = 0.0;
 rt_req_cnt = 0;

 rt_dst = 0;
 rt_seqno = 0;
 rt_interface = 255;
 rt_hops = rt_last_hop_count = INFINITY2;
 rt_nexthop = 0;
 LIST_INIT(&rt_pclist);
 rt_expire = 0.0;
 rt_flags = RTF_DOWN;

 /*
 rt_errors = 0;
 rt_error_time = 0.0;
 */

 for (i=0; i < MAX_HISTORY; i++) {
   rt_disc_latency[i] = 0.0;
 }
 hist_indx = 0;
 rt_req_last_ttl = 0;

 LIST_INIT(&rt_nblist);
}
```

Chapter 6

Scenario Script

One of the cornerstones of this work has been to create a flexible multi-interface model where the number of interfaces per node, as well as the overall number of channels which is being used within the scenario can be easily configured by the user, by tweaking the *Tcl* script which establishes the simulation scenario.

At the beginning of the script we must initialize the values that will be used afterwards as arguments to the `node-config` command, as it is usually done. Obviously, one of the parameters that must be included is the channel type, which has to be set to `WirelessChannel`. Furthermore, there is a new parameter, required to set the maximum number of interfaces that the nodes within the scenario may use. In the example below, this parameter is set to 3.

Listing 6.1: (`scen-script`) Initialization of simulation variables

```
set val(chan)           Channel/WirelessChannel     ;
set val(ni)             3                           ;
set val(nn)             20                          ;
```

As has been already mentioned, the creation of several channels is done from the *tcl* scenario script. Listing 6.2 shows how channels are created (as many as the maximum number of interfaces previously established), using a `for` loop.

Listing 6.2: (`scen-script`) Creation of wireless channels

```
for {set i 0} {$i < $val(ni) } {incr i} {
    set chan_($i) [new $val(chan)]
}
```

In order to ensure that an appropriate memory management is performed, the initialization of the `god` has to include as many interfaces as there may be overall, as shown below:

Listing 6.3: (`scen-script`) Initialization of the `god`

```
create-god [expr $val(nn)*$val(ni)]
```

As we explained in Chapter 3, a new procedure, to allow the inclusion of the number of interfaces as an argument to `node-config`, was added to the `ns-lib.tcl`. Listing 6.4 shows how the number of interfaces is included as a new argument. It is also worth mentioning that we do not specify the type of channel, but rather one channel, this was done so as not to require too many changes within the corresponding *tcl* procedure and, as will be seen later, the channels are added afterwards, before actually creating the wireless node.

Listing 6.4: (`scen-script`) `node-config`

```
$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)
```

Indeed, before creating a node we need to indicate how many interfaces it has, using the new procedure `change-numifs`, as well as associating them with the corresponding channel, i.e. by means of the `add-channel` procedure. These procedures have been added into the `ns-lib.tcl` file, and have been described in Chapter 3.

Thanks to the model flexibility, we can perform quite a broad range of combinations. For instance, Listing 6.5 shows an easy way to configure all nodes so as they use the same number of interfaces, connected to all previously defined wireless channels.

Listing 6.5: (`scen-script`) Creating a number of nodes with the same number of interfaces associated to the same wireless channels

```
$ns_ change-numifs $val(ni)
for {set i 0} {$i < $val(ni) } {incr i} {
    $ns_ add-channel $i $chan_($i)
}
for {set i 0} {$i < $val(nn) } {incr i} {
    set node_($i) [$ns_ node]
    $node_($i) random-motion 0
}
```

However, we may want to have a more flexible configuration, in which some nodes have a different number of interfaces than the others, connected to different wireless channels. As an example, Listing 6.6 shows a possible configuration, in which the first node has 2 interfaces (associated to channels 0 and 2), while the second one only has one interface, associated to channel 2.

Listing 6.6: (`scen-script`) Creating two nodes with different number of interfaces

```
$ns_ change-numifs 2
$ns_ add-channel 0 $chan_(0)
```

```
$ns_ add-channel 1 $chan_(2)
set node_(0) [$ns_ node]
$node_(0) random-motion 0

$ns_ change-numifs 1
$ns_ add-channel 0 $chan_(2)
set node_(1) [$ns_ node]
$node_(1) random-motion 0
```

Chapter 7

Future Work

The work that has been accomplished is, already, quite flexible; the different interfaces can be accessed from the *Tcl* script, bringing about the possibility to modify some of their working parameters (e.g. transmission power or coverage, etc) on a rather straightforward way. One additional aspect that might be quite interesting would be the extension of the whole model so as to really include multiple technologies, and not only different interfaces belonging to the same technology, as has been our case. Another topic which may be of interest would be to address the same changes on the *SRNode* architecture, so that source routing protocols could also benefit from the new feature.

Anyhow, new ideas to improve the current model and to extend its capabilities are more than welcome.

Bibliography

[1] The Enhanced Network Simulator. `http://www.cse.iitk.ac.in/users/braman/tens`.

[2] Tzi cker Chiueh, Ashish Raniwala, Rupa Krishnan, and Kartik Gopalan. Hyacinth: An IEEE 802.11-based Multi-channel Wireless Mesh Network. `http://www.ecsl.cs.sunysb.edu/multichannel`, October 2005.

[3] Bo Wang. NS2 Notebook: Multi-channel Multi-interface Simulation in NS2 (2.29). `http://www.cse.msu.edu/~wangbo1/ns2/nshowto8.html`.

[4] Dapeng Wang. Make "hyacinth" run on Debian NS-2.29.2. `http://my.opera.com/HenryFD/blog/show.dml/202861`, March 2006.

[5] The VINT Project. *The ns Manual*, December 2000.

[6] Francisco J. Ros and Pedro M. Ruiz. Implementing a new MANET unicast routing protocol in ns2. Technical report, University of Murcia, December 2004.

Appendix A

GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

5. COMBINING DOCUMENTS

10. FUTURE REVISIONS OF THIS LICENSE