Implementing a New Manet Unicast Routing Protocol in NS2

Francisco J. Ros
Pedro M. Ruiz

{fjrm, pedrom}@dif.um.es

Dept. of Information and Communications Engineering
University of Murcia

December, 2004

Contents

1 Introduction

During the last year, we have witnessed a lot of people asking for the same question in the ns-users mailing list. How do I develop my own protocol for NS2? By writing this document we hope to help those researchers who need to add a manet routing protocol for NS-2.27. We focus our document in unicast protocols. The document is aimed to those people who are somehow familiar with performing simulations in ns-2, and they want to go one step forward to implement their own protocols. Everything we describe in this text is related to version 2.27 of NS2, but it might be useful for other versions as well.

We assume that the reader is familiar with the NS2 basics. That means having read and at least mostly understood “Marc Greis' Tutorial” [1]. It would be very useful if you also take a look at “The ns Manual” [2], specially chapters 3-5, 10-12, 15-16, 23, 25 and 29. We will refer to them several times throughout this text and encourage you to read them. Before coding your own routing protocol, you should know how to perform simulations with other implemented protocols and you are expected to feel familiar and comfortable with simulator. This will avoid lots of misunderstandings and doubts while reading this document.

Besides this tutorial is about programming. You need knowledge about C++ and (a little) Tcl programming. If you aren't enough experienced with these programming lenguages, you should firstly read any of the excellent resources about them which are freely available in the Internet.

2 Getting Started

We are going to implement a new manet routing protocol called *protoname* step by step. This protocol does nothing useful, but it is general enough to have several common points with other routing protocols. As you likely know (if not, please read firstly what we told you!) we are going to implement routing protocol using C++ and then we will do simulations describing scenarios with Tcl scripts.

To allocate our code we will firstly create a new directory called *protoname* inside your NS2 base directory. We will create five files there:

protoname.h This is the header file where will be defined all necessary timers (if any) and routing agent which performs protocol's functionality.

protoname.cc In this file are actually implemented all timers, routing agent and Tcl hooks.

protoname_pkt.h Here are declared all packets *protoname* protocol needs to exchange among nodes in the manet.

protoname_rtable.h Header file where our own routing table is declared.

protoname_rtable.cc Routing table implementation.

You can organize your code as you want to. That is, you can create more or less files, with those names or with others; this is only a hint. Our advice is to use at least those files and to create more as they are needed.

Now we have our "physical" structure (files), let's continue with the "logical" one (classes). To implement a routing protocol in NS2 you must create an agent by inheriting from **Agent** class. At the very beginning of chapter 10 [2] we can read "*Agents represent endpoints where network-layer packets are constructed or consumed, and are used in the implementation of protocols at various layers*". As you can figure out, this is the main class we will have to code in order to implement our routing protocol. In addition, this class offers a linkage with Tcl interface, so we will be able to control our routing protocol through simulation scripts written in Tcl.

Our routing agent will maintain an internal state and a routing table (which is not always needed). Internal state can be represented as a new class or as a collection of attributes inside the routing agent. We will treat routing table as a new class, **protoname_rtable**.

Also our new protocol must define at least one new packet type which will represent the format of its control packets. As we said these packet types are defined in *protoname/protoname_pkt.h*. When the protocol needs to send packets periodically or after some time from the occurrence of an event, it is very useful to count on a **Timer** class. We show an example in which we code our own timers for sending these packets at regular intervals. Timers are also useful in lots of other cases. Imagine *protoname* needs to store some sort of internal information which must be erased at a certain time. The best solution is to create a custom timer capable of doing such job. A timer should also be used to specify timelife of an entry in the routing table. In general, we will use a timer whenever we have to schedule a task at a given time.

There is another important class we must know before going into details. The **Trace** class is the base for writing log files with information about what happened during the simulation.

And the last hint for now: when you want to print a debug message in your code, it is helpful to use the *debug()* function as it is suggested in chapter 25 [2]. This allows you to turn debugging on or off from your simulation scripts and is easier to read for other programmers.

3 Packet Types

Now that you already know the basics, let's create a new file and call it *protoname/protoname_pkt.h*. Here we are going to put all data structures, constants and macros related to our new packet(s) type. See next example.

`protoname/protoname_pkt.h`

```
#ifndef __protoname_pkt_h__
#define __protoname_pkt_h__

#include <packet.h>

#define HDR_PROTONAME_PKT(p) hdr_protoname_pkt::access(p)
```

```
7:
8:  struct hdr_protoname_pkt {
9:
10:     nsaddr_t   pkt_src_;     // Node which originated this packet
11:     u_int16_t  pkt_len_;     // Packet length (in bytes)
12:     u_int8_t   pkt_seq_num_; // Packet sequence number
13:
14:     inline     nsaddr_t&   pkt_src()     { return pkt_src_; }
15:     inline     u_int16_t&  pkt_len()     { return pkt_len_; }
16:     inline     u_int8_t&   pkt_seq_num() { return pkt_seq_num_; }
17:
18:     static int offset_;
19:     inline static int& offset() { return offset_; }
20:     inline static hdr_protoname_pkt* access(const Packet* p) {
21:         return (hdr_protoname_pkt*)p->access(offset_);
22:     }
23:
24: };
25:
26: #endif
```

Lines 8-24 declare **hdr_protoname_pkt** struct which represents the new packet type we are defining. In lines 10-12 we can see three raw attributes our packet has. They are of following types:

nsaddr_t Every time you want to declare a network address in NS2 you must use this type.

u_int16_t 16 bits unsigned integer.

u_int8_t 8 bits unsigned integer.

All these types and more are defined in the header file *config.h*. We encourage the reader to have a look at this file and to use types defined there. It's also worth mentioning raw attributes' names are expected to finish with an underscore to distinguish them from other variables (see chapter 25 [2]).

Lines 14-16 are member functions for defined attributes. This is not mandatory but a "good practice" suggested in chapter 12 [2] (and we actually support it!).

Line 4 includes file *common/packet.h* which defines **Packet** class (see chapter 12 [2]). Packets are used to exchange information between objects in the simulation, and our aim is to add our new struct **hdr_protoname_pkt** to them. Doing so our control packets will be able to be sent and received by nodes in the simulation. How can we do that? To answer this question we must know how all packet headers are stored by **Packet** class, and the answer is by using an array of unsigned characters (bag of bits) where packets' fields are saved. To access a concrete packet header is necessary to provide the offset where it is located. And that's exactly what we do through lines 18-22. We define a static (common to all **hdr_protoname_pkt** structs) offset, a member function to access it and a function which returns a **hdr_protoname_pkt** given a **Packet**. Moreover, in line 6 we create a macro to use this last function.

There is one task left: to bind our packet header to Tcl interface. We will do so in *protoname/protoname.cc* with the following code. As we can see we are doing accesible through Tcl the offset of our packet header.

```
protoname/protoname.cc

int protoname_pkt::offset_;
static class ProtonameHeaderClass : public PacketHeaderClass {
public:
    ProtonameHeaderClass() : PacketHeaderClass("PacketHeader/Protoname",
                                              sizeof(hdr_protoname_pkt)) {
        bind_offset(&hdr_protoname_pkt::offset_);
    }
} class_rtProtoProtoname_hdr;
```

4 The Routing Agent

Now we start programming the agent itself. Inside *protoname/protoname.h* we define a new class called **Protoname** containing the attributes and functions needed to assist the protocol in doing its job. To illustrate the use of timers we assume that *protoname* is a proactive routing protocol that requires sending out some control packets periodically. The next code shows such an example.

```
protoname/protoname.h

#ifndef __protoname_h__
#define __protoname_h__

#include "protoname_pkt.h"
#include <agent.h>
#include <packet.h>
#include <trace.h>
#include <timer-handler.h>
#include <random.h>
#include <classifier-port.h>

#define CURRENT_TIME         Scheduler::instance().clock()
#define JITTER               (Random::uniform()*0.5)

class Protoname;             // forward declaration

/* Timers */

class Protoname_PktTimer : public TimerHandler {
public:
    Protoname_PktTimer(Protoname* agent) : TimerHandler() {
        agent_ = agent;
    }
protected:
    Protoname*   agent_;
```

```
26:     virtual void expire(Event* e);
27: };
28:
29: /* Agent */
30:
31: class Protoname : public Agent {
32:
33:     /* Friends */
34:     friend class Protoname_PktTimer;
35:
36:     /* Private members */
37:     nsaddr_t           ra_addr_;
38:     protoname_state    state_;
39:     protoname_rtable   rtable_;
40:     int                accesible_var_;
41:     u_int8_t           seq_num_;
42:
43: protected:
44:
45:     PortClassifier*    dmux_;      // For passing packets up to agents.
46:     Trace*             logtarget_; // For logging.
47:     Protoname_PktTimer pkt_timer_; // Timer for sending packets.
48:
49:     inline nsaddr_t&        ra_addr()        { return ra_addr_; }
50:     inline protoname_state& state()          { return state_; }
51:     inline int&             accessible_var() { return accessible_var_; }
52:
53:     void  forward_data(Packet*);
54:     void  recv_protoname_pkt(Packet*);
55:     void  send_protoname_pkt();
56:
57:     void  reset_protoname_pkt_timer();
58:
59: public:
60:
61:     Protoname(nsaddr_t);
62:     int   command(int, const char*const*);
63:     void  recv(Packet*, Handler*);
64:
65: };
66:
67: #endif
```

Lines 4-10 are used to include the header files required by our agent. Below we explain what are they useful for.

protoname/protoname_pkt.h Defines our packet header.

common/agent.h Defines **Agent** base class.

common/packet.h Defines **Packet** class.

common/timer-handler.h Defines **TimerHandler** base class. We will use it to create our custom timers.

trace/trace.h Defines **Trace** class, used for writing simulation results out to a trace file.

tools/random.h Defines **Random** class, useful for generating pseudo-random numbers. We will use it soon.

classifier/classifier-port.h Defines **PortClassifier** class, used for passing packets up to upper layers.

Line 12 defines a useful macro for getting current time in the simulator clock. That's done by accesing the single instance of **Scheduler** class. This object manages all events produced during simulation and simulator's internal clock (see chapter 4 [2]).

Another macro is in line 13. It is just an easy way to obtain a random number inside [0–0.5] interval. This is commonly used to randomize the sending of control packets to avoid synchronization of a node with its neighbors, that would eventually produce collisions and therefore delays at time of sending these packets[1].

Lines 19-27 declare our custom timer for sending periodical control packets. Our **Protoname_PktTimer** class inherits from **TimerHandler** and has a reference to the routing agent which creates it. This is used as a callback for telling routing agent to send a new control packet and to schedule the next one. We shall see more on this later on, when we describe how to overload the *expire()* method. To do these callbacks routing agent needs to treat **Protoname_PktTimer** as a friend class (line 34).

The **Protoname** class is defined within lines 31-65. It encapsulates its own address, internal state, routing table, an accessible variable from Tcl and a counter for assigning sequence numbers to ouput packets (lines 37-41). *protoname_state_* could be a class itself or a set of attributes needed by the **Protoname** class for doing its job. *accessible_var_* is thought to be read and written from Tcl scripts or shell commands. This is useful in many situations because it allows users to change simulation behaviour through their scripts without re-compiling the simulator.

A **PortClassifier** object is declared in line 45. You should read chapter 5 [2] in order to understand node's structure. There you will see how a node consists of an address classifier and a port classifier. The first is used to guide incoming packets to a suitable link or to pass them to the port classifier, which will carry them to appropiate upper layer agent. That's why the routing agent needs a port classifier. When it receives data packets destined to itself it will use *dmux_* in order to give them to corresponding agent[2]. The detailed architecture of a mobile node is explained in chapter 16 of [2].

Another important attribute is the **Trace** object (see line 46). It is used to produce logs to be store in the trace file. In our example we use it to write the

[1]This is a typical feature in manet routing protocols, but the real aim is to provide an example of getting random numbers.

[2]Really this is not true. In fact data packets are directly delivered to their corresponding agent, so the port classifier isn't necessary for our routing agent. However we maintain this explanation because NS-2.27 requires routing agent to accept *port-dmux* operation (see section 4.3.2) as part of its API

contents of the routing table whenever the user requests it from the Tcl interface. This is not necessary if you are only interested in writing tracing information regarding packets. In that case, those logging functions are implemented in other location (as we shall see in section 6).

Line 47 declares our custom timer. And lines 49-51 are member functions to access some internal attributes.

Function at line 53 will be used to forward data packets to their correct destination. The one at line 54 will be called whenever a control packet is received, and that at line 55 is invoked for sending a control packet. Line 57 declares a function used in order to schedule our custom timer expiration.

Lines 61-63 contain public functions of class **Protoname**. Constructor receives as an argument an identifier used as the routing agent's address. **Protoname** inherits from **Agent** base class two main functions which need to be implemented: *recv()* and *command()*. *recv()* is called whenever the agent receives a packet. This may occur when the node itself (actually an upper layer agent such as UDP or TCP) is generating a packet or when it is receiving one from another node. The *command()* function is invoked from Tcl as is described in chapter 3 [2]. It's a way to ask the C++ object to do some task from our Tcl code. You will understand this better once we'll go through section 4.3.

Now that you know how **Protoname**'s interface is, it's time to go on with its implementation. The next subsections are related to the *protoname/protoname.cc* file.

4.1 Tcl hooks

We saw in section 3 how to bind our own packet to Tcl. Now we will do the same for our agent class. The aim is to let **Protoname** to be instantiated from Tcl. To do so we must inherit from the class **TclClass** as depicted in the next code.

```
protoname/protoname.cc

1: static class ProtonameClass : public TclClass {
2: public:
3:     ProtonameClass() : TclClass("Agent/Protoname") {}
4:     TclObject* create(int argc, const char*const* argv) {
5:         assert(argc == 5);
6:         return (new Protoname((nsaddr_t)Address::instance().str2addr(argv[4])));
7:     }
8: } class_rtProtoProtoname;
```

The class constructor is in line 3 and it merely calls the base class with the string "*Agent/Protoname*" as an argument. This represents class hierarchy for this agent in a textual manner.

In ines 4-7 we implement a function called *create()* which returns a new **Protoname** instance as a **TclObject**. *argv* is of the form "*<object's name> <$self> <$class> <$proc> <user argument>*" (see chapter 3 of [2] for more information). In this particular case it is "*<object's name> <$self> Agent/Protoname create-shadow <id>*". Because of this, at line 6 we return a new **Protoname** object with the identifier stated in *argv[4]*. We use the **Address** class to get a *nsaddr_t* type from a string.

4.2 Timers

All we have to code in *protoname/protoname.cc* about timers is the *expire()* method. Timers are detailed in chapter 11 [2]. Implementing this is pretty easy because we only want to send a new control packet and to reschedule the timer itself. According to our design decisions these two tasks must be executed by the routing agent, so we invoke these callbacks as in the next example.

`protoname/protoname.cc`

```
void
Protoname_PktTimer::expire(Event* e) {
     agent_->send_protoname_pkt();
     agent_->reset_protoname_pkt_timer();
}
```

4.3 Agent

4.3.1 Constructor

Let's begin with constructor implementation. As we can see in line 1 below, we start by calling the constructor for the base class passing PT_PROTONAME as an argument. This constant will be defined later (see section 6) and it is used to identify control packets sent and received by this routing agent. In the same line we create our **Protoname_PktTimer** object.

Just after that we bind *accessible_var_* as a boolean attribute which now may be read and written from Tcl. To bind this variable as an integer, we must use the *bind()* function instead of *bind_bool()*.

Line 3 saves the given identifier as the routing agent's address.

`protoname/protoname.cc`

```
1: Protoname::Protoname(nsaddr_t id) : Agent(PT_PROTONAME), pkt_timer_(this) {
2:    bind_bool("accessible_var_", &accessible_var_);
3:    ra_addr_ = id;
4: }
```

Accessing from Tcl scripts is fairly simple. The next example sets the value of *accesible_var_* to *true*.

`simulation.tcl`

```
Agent/Protoname set accesible_var_ true
```

4.3.2 command()

The next piece of code is a little bit more complicated. It consists of the implementation of the *command()* method that our agent inherites from the **Agent** class.

`protoname/protoname.cc`

```
int
Protoname::command(int argc, const char*const* argv) {
    if (argc == 2) {
        if (strcasecmp(argv[1], "start") == 0) {
            pkt_timer_.resched(0.0);
            return TCL_OK;
        }
        else if (strcasecmp(argv[1], "print_rtable") == 0) {
            if (logtarget_ != 0) {
                sprintf(logtarget_->pt_->buffer(), "P %f _%d_ Routing Table",
                    CURRENT_TIME,
                    ra_addr());
                logtarget_->pt_->dump();
                rtable_.print(logtarget_);
            }
            else {
                fprintf(stdout, "%f _%d_ If you want to print this routing table "
                    "you must create a trace file in your tcl script",
                    CURRENT_TIME,
                    ra_addr());
            }
            return TCL_OK;
        }
    }
    else if (argc == 3) {
        // Obtains corresponding dmux to carry packets to upper layers
        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;
        }
        // Obtains corresponding tracer
        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;
        }
    }
    // Pass the command to the base class
    return Agent::command(argc, argv);
}
```

argv[0] contains the name of the method (always "cmd", see chapter 3 [2]) being invoked, *argv[1]* is the requested operation, and *argv[2..argc-1]* are the rest of the arguments which were passed. Within this function we must code some mandatory operations as well as any other operation that we want to make accesible from Tcl. As an example we will code an operation called *print_rtable* which dumps the contents of the routing table's to the trace file.

We focus our code only in cases where we have two or three arguments, so that you can see how to process them. Each case must finish its execution returning either *TCL_OK* (if everything was fine) or *TCL_ERROR* (if any error happened).

Lines 4-7 describe a mandatory command that we always have to implement:*start*. The expected behaviour of this command is to configure the agent to begin its execution. In our case it starts its packet sending timer. We should implement here all the required actions that the routing agent must perform in order to begin its operation.

Lines 8-23 implement our *print_rtable* command. We firstly check if *logtarget_* is initialized (line 9). Then we dump the table into the trace file as is showed in lines 10-13. To understand this piece of code it would be useful that you take a look into the *trace/trace.h* header file. There is where the **Trace** class is defined. It has a reference to *pt_* of the **BaseTrace** class. This last class implements *buffer()* and *dump()* functions which are used to get the variable where output is buffered and to flush that buffer to the output file respectively. Finally, line 14 calls the *print()* function of our routing table for writing into trace file its own content. The TCL code below shows how to execute the *print_rtable* operation at a certain time from a simulation script. It assumes that *ns_* contains an instance of **Simulator** and *node_* is a **Node** created by *ns_*. We are passing 255 as argument because this is the number of the port where a routing agent is attached to.

```
simulation.tcl

1: $ns_ at 15.0 "[$node_ agent 255] print_rtable"
```

Another mandatory command to implement is *port-dmux*. Its implementation is provided in lines 27-37. As explained in chapter 3 of [2], NS stores a reference to every compiled object (C++ object) in a hash table to provide a fast access to each of them given its name. We make use of that facility in line 28 to obtain a *PortClassifier* object given its name.

Similarly, there is another mandatory operation called *tracetarget* (note that we a allow it to be called *log-target* as well) which simply obtains a **Trace** object given its name.

If we don't know how to process the requested command, we delegate this responsability to base class, as we do in line 48.

4.3.3 recv()

Next function is *recv()* and as we know is invoked whenever the routing agent receives a packet. Every **Packet** has a common header called *hdr_cmn* defined in *common/packet.h*. To access this header there is a macro like the one we defined before for our own packet type, and we use it at line 3. Line 4 does the same but in order to get IP header, *hdr_ip*, described in *ip.h*.

```
1:  void
2:  Protoname::recv(Packet* p, Handler* h) {
3:      struct hdr_cmn* ch  = HDR_CMN(p);
4:      struct hdr_ip* ih   = HDR_IP(p);
5:
6:      if (ih->saddr() == ra_addr()) {
7:          // If there exists a loop, must drop the packet
8:          if (ch->num_forwards() > 0) {
9:              drop(p, DROP_RTR_ROUTE_LOOP);
10:             return;
11:         }
12:         // else if this is a packet I am originating, must add IP header
13:         else if (ch->num_forwards() == 0)
14:             ch->size() += IP_HDR_LEN;
15:     }
16:
17:     // If it is a protoname packet, must process it
18:     if (ch->ptype() == PT_PROTONAME)
19:         recv_protoname_pkt(p);
20:     // Otherwise, must forward the packet (unless TTL has reached zero)
21:     else {
22:         ih->ttl_--;
23:         if (ih->ttl_ == 0) {
24:             drop(p, DROP_RTR_TTL);
25:             return;
26:         }
27:         forward_data(p);
28:     }
29: }
```

First thing we should do is to check we are not receiving a packet we sent ourselves. If that is the case we should drop the packet and return, as we do in lines 8-11. In addition, if the packet has been generated within the node (by upper layers of the node) we should add to packet's length the overhead that the routing protocol is adding (in bytes). We assume *protoname* works over IP, as it is shown in lines 13-14.

When the received packet is of type PT_PROTONAME then we will call *recv_protoname_pkt()* to process it (lines 18-19). If it is a data packet then we should forward it (if it is destined to other node) or to deliver it to upper layers (if it was a broadcast packet or was destined to ourself), unless TTL[3] reached zero. Lines 21-28 do what we have just described making use of the *forward_data()* function.

You would have realized that the *drop()* function is used for dropping packets. Its aarguments are a pointer to the packet itself and a constant giving the reason for discarding it. There exist several of these constants. Uou can take a look at them in the file *trace/cmu-trace.h*.

[3] *Time To Live* of the packet according to IP header.

4.3.4 recv_protoname_pkt()

Let's assume that the routing agent has received a *protoname* packet, making the *recv_protoname_pkt()* to be invoked. The implementation of this function will vary a lot depending on the concrete protocol, but we can see a general scheme in the next example.

Lines 3-4 get IP header and *protoname* packet header as usual. After that we make sure source and destination ports are RT_PORT at lines 8-9. This constant is defined in *common/packet.h* and it equals 255. This port is reserved to attach the routing agent.

After that, the *protoname* packet must be processed according to our routing protocol's specification.

Finally we must release resources as we do in line 14.

```
1: void
2: Protoname::recv_protoname_pkt(Packet* p) {
3:     struct hdr_ip* ih           = HDR_IP(p);
4:     struct hdr_protoname_pkt* ph = HDR_PROTONAME_PKT(p);
5:
6:     // All routing messages are sent from and to port RT_PORT,
7:     // so we check it.
8:     assert(ih->sport() == RT_PORT);
9:     assert(ih->dport() == RT_PORT);
10:
11:     /* ... processing of protoname packet ... */
12:
13:     // Release resources
14:     Packet::free(p);
15: }
```

4.3.5 send_protoname_pkt()

We saw in section 4.2 how our custom timer calls the function *send_protoname_pkt()* whenever it expires. We show a sample implementation of this function below. Needless to say that each protocol requires something different and this is just an example.

```
protoname/protoname.cc

void
Protoname::send_protoname_pkt() {
    Packet* p                     = allocpkt();
    struct hdr_cmn* ch            = HDR_CMN(p);
    struct hdr_ip* ih             = HDR_IP(p);
    struct hdr_protoname_pkt* ph  = HDR_PROTONAME_PKT(p);

    ph->pkt_src()          = ra_addr();
    ph->pkt_len()          = 7;
    ph->pkt_seq_num()      = seq_num_++;

    ch->ptype()            = PT_PROTONAME;
```

```
13:     ch->direction()         = hdr_cmn::DOWN;
14:     ch->size()              = IP_HDR_LEN + ph->pkt_len();
15:     ch->error()             = 0;
16:     ch->next_hop()          = IP_BROADCAST;
17:     ch->addr_type()         = NS_AF_INET;
18:
19:     ih->saddr()             = ra_addr();
20:     ih->daddr()             = IP_BROADCAST;
21:     ih->sport()             = RT_PORT;
22:     ih->dport()             = RT_PORT;
23:     ih->ttl()               = IP_DEF_TTL;
24:
25:     Scheduler::instance().schedule(target_, p, JITTER);
26: }
```

To send a packet we need fist to allocate it. We use the *allocpkt()* function for that. This function is defined for all **Agent**s. Then we get common, IP and *protoname* packet headers as usual (lines 3-6). Our aim is to fill all these headers with values we want to.

Protoname packet header is filled in lines 8-10. In our simple example we only need source address of the agent, length (in bytes) of the message and a sequence number. These fields are completely dependent on *protoname*'s packet specification.

The common header in NS has several fields. We focus only on those in which we are interested (lines 12-17). We need to set the packet type to a *protoname* packet (line 12). We also assign the packet direction in line 13. As we are sending a packet, it is going down, what is represented by *hdr_cmn::DOWN* constant. The size of the packet is given in line 14. It is in bytes and this is the value used for NS2 computations. What we mean is that it doesn't matter real size of your **hdr_protoname_pkt** struct. To calculate things such as propagation delay NS2 will use the value you put in here. Continuing with common header, in line 15 we decide not to have any error in transmission. Line 16 assigns the next hop to which the packet must be sent to. This is a very important field, and in our protocol it is established as *IP_BROADCAST* because we want all of the neighboring nodes to receive this control packet. That constant is defined in *common/ip.h* and you can check there for other macros. The last field we fill is the address type. It can be NS_AF_NONE, NS_AF_ILINK or NS_AF_INET (see *common/packet.h*). We choose NS_AF_INET because we are implementing an Internet protocol.

Now we proceed with the configuration of the IP header. It is very simple as we can see in lines 19-23. There is a new constant called *IP_DEF_TTL* which is defined in *common/ip.h* and represents the default TTL value for IP packets. The IP header has other fields used for IPv6 simulations, but we don't need them for our example.

Now we can just proceed sending the packet. Packets are events (see chapter 12 of [2]) so they need to be scheduled. In fact, sending a packet is equivalent to schedule it at a certain time. Line 25 shows how to send a packet introducing some jitter. The **Packet** class inherits from the **Connector** class, which has a reference to a **TclObject** called *target_*. This is the handler which will process the event, and is passed as an argument to the *schedule()* function.

4.3.6 reset_protoname_pkt_timer()

Our packet sending timer performs another callback (section 4.2) to reschedule itself. It's done in the function *reset_protoname_pkt_timer()*. We show that in next example, where *pkt_timer_* is rescheduled to expire five seconds later.

protoname/protoname.cc

```
void
Protoname::reset_protoname_pkt_timer() {
    pkt_timer_.resched((double)5.0);
}
```

4.3.7 forward_data()

So far we have been mainly focused on *protoname* packets, but it's time to deal with data packets. The *forward_data()* function decides whether a packet has to be delivered to the upper-layer agents or to be forwarded to other node. We check for the first case in lines 6-10. When it is an incoming packet and destination address is the node itself or broadcast, then we use the node's *dmux_* (if we remember it is a **PortClassifier** object) to accept the incoming packet.

Otherwise, we must forward the packet. This is accomplished by properly setting the common header with as we do in lines 12-28. If the packet is a broadcast one, then next hop will be filled accordingly. If not, we make use of our routing table to find out the next hop (line 17). Our implementation returns *IP_BROADCAST* when there is no route to destination address. In such a case we print a debug message (lines 19-22) and drop the packet (line 23). If everything goes fine then we will send the packet as we do in line 29.

protoname/protoname.cc

```
1:  void
2:  Protoname::forward_data(Packet* p) {
3:      struct hdr_cmn* ch = HDR_CMN(p);
4:      struct hdr_ip* ih = HDR_IP(p);
5:
6:      if (ch->direction() == hdr_cmn::UP &&
7:          ((u_int32_t)ih->daddr() == IP_BROADCAST || ih->daddr() == ra_addr())) {
8:          dmux_->recv(p, 0.0);
9:          return;
10:     }
11:     else {
12:         ch->direction() = hdr_cmn::DOWN;
13:         ch->addr_type() = NS_AF_INET;
14:         if ((u_int32_t)ih->daddr() == IP_BROADCAST)
15:             ch->next_hop() = IP_BROADCAST;
16:         else {
17:             nsaddr_t next_hop = rtable_.lookup(ih->daddr());
18:             if (next_hop == IP_BROADCAST) {
19:                 debug("%f: Agent %d can not forward a packet destined to %d\n",
20:                     CURRENT_TIME,
```

```
                    ra_addr(),
                    ih->daddr());
                drop(p, DROP_RTR_NO_ROUTE);
                return;
            }
            else
                ch->next_hop() = next_hop;
        }
        Scheduler::instance().schedule(target_, p, 0.0);
    }
}
```

5 The Routing Table

You might not need a routing table, but if your protocol uses it then read this section. We can implement the routing table as a different class or as any other data structure (e.g. a hash table). We are going to show a class encapsulating the functionality that a routing table is supposed to have. Internal information may vary a lot from protocol to protocol. For each entry in routing table one might want to store destination addresses, next hop addresses, distances or cost associated to the routes, sequence numbers, lifetimes, and so on. Of course our example illustrates a very simple routing table and a method to print it. The only information we will store in each entry is destination and next hop addresses. We use a hash table (map) as the storage structure. This case is too simple to implement a new class, but we will do it as an example. The next piece of code corresponds to *protoname/protoname_rtable.h*.

`protoname/protoname_rtable.h`

```
#ifndef __protoname_rtable_h__
#define __protoname_rtable_h__

#include <trace.h>
#include <map>

typedef std::map<nsaddr_t, nsaddr_t> rtable_t;

class protoname_rtable {

    rtable_t  rt_;

public:

    protoname_rtable();
    void       print(Trace*);
    void       clear();
    void       rm_entry(nsaddr_t);
    void       add_entry(nsaddr_t, nsaddr_t);
    nsaddr_t   lookup(nsaddr_t);
```

```
    u_int32_t  size();
};

#endif
```

The implementation of these functions is quite easy. In fact the constructor is so simple that there is nothing to do inside it.

protoname/protoname_rtable.cc

```
protoname_rtable::protoname_rtable() { }
```

The *print()* function will dump the contents of the node's routing table to the trace file. To do that we use the **Trace** class which we mentioned in section 4.3.

protoname/protoname_rtable.cc

```
void
protoname_rtable::print(Trace* out) {
    sprintf(out->pt_->buffer(), "P\tdest\tnext");
    out->pt_->dump();
    for (rtable_t::iterator it = rt_.begin(); it != rt_.end(); it++) {
        sprintf(out->pt_->buffer(), "P\t%d\t%d",
            (*it).first,
            (*it).second);
        out->pt_->dump();
    }
}
```

The following function removes all entries in routing table.

protoname/protoname_rtable.cc

```
void
protoname_rtable::clear() {
    rt_.clear();
}
```

To remove an entry given its destination address we implement the *rm_entry()* function.

protoname/protoname_rtable.cc

```
void
protoname_rtable::rm_entry(nsaddr_t dest) {
    rt_.erase(dest);
}
```

The code below is used to add a new entry in the routing table given its destination and next hop addresses.

protoname/protoname_rtable.cc

```
void
protoname_rtable::add_entry(nsaddr_t dest, nsaddr_t next) {
    rt_[dest] = next;
}
```

Lookup() returns the next hop address of an entry given its destination address. If such an entry doesn't exist, (that is, there is no route for that destination) the function returns *IP_BROADCAST*. Of course we include *common/ip.h* in order to use this constant.

protoname/protoname_rtable.cc

```
nsaddr_t
protoname_rtable::lookup(nsaddr_t dest) {
    rtable_t::iterator it = rt_.find(dest);
    if (it == rt_.end())
        return IP_BROADCAST;
    else
        return (*it).second;
}
```

Finally, *size()* returns the number of entries in the routing table.

protoname/protoname_rtable.cc

```
u_int32_t
protoname_rtable::size() {
    return rt_.size();
}
```

6 Needed Changes

We have almost finished. We have implemented a routing agent for protocol *protoname* inside NS2. But there are some changes we need to do in order to integrate our code inside simulator.

6.1 Packet type declaration

If we remember we had to use a constant to indicate our new packet type, *PT_PROTONAME*. We will define it inside file *common/packet.h*.

Let's find *packet_t* enumeration, where all packet types are listed. We will add *PT_PROTONAME* to this list as we show in the next piece of code (line 6).

common/packet.h

```
1: enum packet_t {
2:     PT_TCP,
```

```
    PT_UDP,
    PT_CBR,
    /* ... much more packet types ... */
    PT_PROTONAME,
    PT_NTYPE // This MUST be the LAST one
};
```

Just below in same file there is definition of **p_info** class. Inside constructor we will provide a textual name for our packet type (line 6).

common/packet.h

```
1: p_info() {
2:     name_[PT_TCP]= "tcp";
3:     name_[PT_UDP]= "udp";
4:     name_[PT_CBR]= "cbr";
5:     /* ... much more names ... */
6:     name_[PT_PROTONAME]= "protoname";
7: }
```

6.2 Tracing support

As we know simulation's aim is to get a trace file describing what happended during execution. To feel familiar with traces please read chapter 23 [2]. A **Trace** object is used to write wanted information of a packet everytime it is received, sent or dropped. To log information regarding our packet type we implement the *format_protoname()* function inside the **CMUTrace** class. Trace support for wireless simulations is provided by **CMUTrace** objects and it is described in chapter 16 [2].

Let's edit *trace/cmu-trace.h* file. We must add our new function as in the line number 6 of the next example.

trace/cmu-trace.h

```
1: class CMUTrace : public Trace {
2:     /* ... definitions ... */
3: private:
4:     /* ... */
5:     void    format_aodv(Packet *p, int offset);
6:     void    format_protoname(Packet *p, int offset);
7: };
```

The next piece of code (extracted from *trace/cmu-trace.cc*) shows different types of traces.

trace/cmu-trace.cc

```
#include <protoname/protoname_pkt.h>

/* ... */

```

```
void
CMUTrace::format_protoname(Packet *p, int offset)
{
    struct hdr_protoname_pkt* ph = HDR_PROTONAME_PKT(p);

    if (pt_->tagged()) {
        sprintf(pt_->buffer() + offset,
            "-protoname:o %d -protoname:s %d -protoname:l %d ",
            ph->pkt_src(),
            ph->pkt_seq_num(),
            ph->pkt_len());
    }
    else if (newtrace_) {
        sprintf(pt_->buffer() + offset,
            "-P protoname -Po %d -Ps %d -Pl %d ",
            ph->pkt_src(),
            ph->pkt_seq_num(),
            ph->pkt_len());
    }
    else {
        sprintf(pt_->buffer() + offset,
            "[protoname %d %d %d] ",
            ph->pkt_src(),
            ph->pkt_seq_num(),
            ph->pkt_len());
    }
}
```

We can deduce from above code that there are three different trace formats: tagged traces, new format traces and classical traces. The syntaxis followed by each, although different, is very easy and intuitive as you can tell. Both in tagged and new trace formats there exists tags used to identify each field of information being printed. We have decided to use “o” as source address (origin), “s” as sequence number and “l” as length of corresponding packet.

In order to call this recently created function we must change the *format()* in *trace/cmu-trace.cc*.

`trace/cmu-trace.cc`

```
void
CMUTrace::format(Packet* p, const char *why)
{
    /* ... */
    case PT_PING:
        break;

    case PT_PROTONAME:
        format_protoname(p, offset);
        break;
```

```
    default:
        /* ... */
}
```

6.3 Tcl library

Now we need to do some changes in Tcl files. Actually we are going to add our packet type, give default values for binded attributes and provide the needed infraestructure to create wireless nodes running our *protoname* routing protocol.

In *tcl/lib/ns-packet.tcl* you must locate the next code and add *protoname* to the list (as we do in line 2).

`tcl/lib/ns-packet.tcl`

```
1: foreach prot {
2:     Protoname
3:     AODV
4:     ARP
5:     # ...
6:     NV
7: } {
8:     add-packet-header $prot
9: }
```

Default values for binded attributes have to be given inside *tcl/lib/ns-default.tcl*. We must go to the end of the file and write something like the next code:

`tcl/lib/ns-default.tcl`

```
# ...
# Defaults defined for Protoname
Agent/Protoname set accessible_var_ true
```

Finally we have to modify *tcl/lib/ns-lib.tcl*. We need to add procedures for creating a node. Our interest will be centered around creating a wireless node with *protoname* as routing protocol.

The procedure *node* calls to the *create-wireless-node* procedure. This last one, among other tasks, is intended to set the routing agent for a node. We need to hack this procedure to create an instance of our *protoname* protocol.

`tcl/lib/ns-lib.tcl`

```
Simulator instproc create-wireless-node args {
    # ...
    switch -exact $routingAgent_ {
        Protoname {
            set ragent [$self create-protoname-agent $node]
        }
        # ...
    }
    # ...
}
```

Then *create-protoname-agent* will be coded below as shown in the next example.

```
tcl/lib/ns-lib.tcl

1: Simulator instproc create-protoname-agent { node } {
2:         # Create Protoname routing agent
3:         set ragent [new Agent/Protoname [$node node-addr]]
4:         $self at 0.0 "$ragent start"
5:         $node set ragent_ $ragent
6:         return $ragent
7: }
```

Line 3 creates a new *protoname* agent with the node's address. This agent is scheduled to start at the begining of the simulation (line 4), and is assigned as the node's routing agent in line 5.

6.4 Priority queue

It's very likely you will use priority queues in your simulations. This queue type treats routing packets as high priority packets, inserting them at the begining of the queue. But we need to tell the **PriQueue** class that *protoname* packets are routing packets and therefore treated as high priority.

We must modify the *recv()* function in *queue/priqueue.cc* file. Line 13 in the next piece of code is the only modification we need to do.

```
queue/priqueue.cc

1:  void
2:  PriQueue::recv(Packet *p, Handler *h)
3:  {
4:      struct hdr_cmn *ch = HDR_CMN(p);
5:
6:      if (Prefer_Routing_Protocols) {
7:
8:          switch(ch->ptype()) {
9:              case PT_DSR:
10:             case PT_MESSAGE:
11:             case PT_TORA:
12:             case PT_AODV:
13:             case PT_PROTONAME:
14:                 recvHighPriority(p, h);
15:                 break;
16:
17:             default:
18:                 Queue::recv(p, h);
19:         }
20:     }
21:     else {
22:         Queue::recv(p, h);
23:     }
```

```
}
```

6.5 Makefile

Now everyting is implemented and we only need to compile it! To do so we will edit *Makefile* file by adding our object files inside *OBJ_CC* variable as in following code (line 4).

Makefile

```
1: OBJ_CC = \
2:     tools/random.o tools/rng.o tools/ranvar.o common/misc.o common/timer-handler.o \
3:     # ...
4:     protoname/protoname.o protoname/protoname_rtable.o \
5:     # ...
6:     $(OBJ_STL)
```

As we modified *common/packet.h* but not *common/packet.cc* we should "touch" this last file for being recompiled. After that we can execute *make* and enjoy our own routing protocol (or perhaps solve all compilation problems!).

```
[ns-2.27]$ touch common/packet.cc
[ns-2.27]$ make
```

7 Receiving Information from Layer-2 Protocols

Some routing protocols might be interested in reacting when a packet can't be sent from layer-2. This can be easily accomplished by our routing agent, as we explain below.

How does it work? The common header of a packet has a field where you can specify a function that will be called if the packet can't be sent by the layer-2 agent. Let's call that function *protoname_mac_failed_callback()*. We will use this function to call another one within the routing agent being in charge of reacting to such a layer-2 failure. We will call this second function *mac_failed()*. So we only have to modify line 9 of *protoname/protoname.h*.

protoname/protoname.h

```
1:  class Protoname : public Agent {
2:      /* ... */
3:
4:  public:
5:
6:      Protoname(nsaddr_t);
7:      int    command(int, const char*const*);
8:      void   recv(Packet*, Handler*);
9:      void   mac_failed(Packet*);
10: };
11: #endif
```

The *protoname/protoname.cc* file requires more changes. First of all we must implement the function which is registered inside the common header. That function will simply call to the *mac_failed()* function of the **Protoname** class. You can see the implementation below.

`protoname/protoname.cc`

```
static void
protoname_mac_failed_callback(Packet *p, void *arg) {
    ((Protoname*)arg)->mac_failed(p);
}
```

The functionality implemented by *mac_failed()* depends very much on *protoname* specification. As an example, the next piece of code prints a debug message (lines 6-9) and drops the packet (line 11).

`protoname/protoname.cc`

```
1:  void
2:  Protoname::mac_failed(Packet* p) {
3:      struct hdr_ip* ih   = HDR_IP(p);
4:      struct hdr_cmn* ch  = HDR_CMN(p);
5:
6:      debug("%f: Node %d MAC layer cannot send a packet to node %d\n",
7:          CURRENT_TIME,
8:          ra_addr(),
9:          ch->next_hop());
10:
11:     drop(p, DROP_RTR_MAC_CALLBACK);
12:
13:     /* ... do something ... */
14: }
```

If we want to know when a routing packet isn't sent by layer-2 protocols we need to modify *send_protoname_pkt()*. Similarly if we want to pay this attention to data packets *forward_data()* must be lightly modified as well. In both cases we only must add next two lines when updating common header of the packet.

`protoname/protoname.cc`

```
ch->xmit_failure_      = protoname_mac_failed_callback;
ch->xmit_failure_data_ = (void*)this;
```

What cases will *protoname_mac_failed_callback()* be called in? In NS-2.27 we can found two different situations:

mac/arp.cc When a node wants to resolve a destination address (via ARP) but maximum number of retries is exceeded.

mac/mac-802_11.cc There are two possibilities. First one occurs when a RTS is sent but no corresponding CTS is received and maximum number of retries is exceeded. Second one happens when a data packet was transmitted but never acknowledged (no ACK received) and maximum number of retries is exceeded.

8 Support for Wired-Cum-Wireless Simulations

Until now we have been only concerned about flat manets, that is, wireless-only scenarios. In this section we will introduce basic concepts to deal with hybrid manets (wired-cum-wireless scenarios, following NS2 terminology). Wired-cum-wireless scripts need to use hierarchical addressing, so you must read chapter 15 and 29 [2] to get necessary knowledge in this type of addressing.

With minimal changes we could use our protocol in wired-cum-wireless simulations. In these ones there are fixed nodes, wireless nodes and base stations. A base station is a gateway between wired and wireless domains, and every wireless node needs to know which base station it is associated to. All we need to do in order to provide wired-cum-wireless support is to set the corresponding base station for each node.

Simulation scripts describing wired-cum-wireless scenarios perform former operation on each mobile node, that is, every mobile node is attached to a base station (*base-station* function of **Node** API). But imagine we are interested in scenarios where several base stations are used, and we also want mobile nodes to dynamically change their associated base stations. This is useful if we want to code a routing protocol supporting hybrid ad hoc networks where multiple base stations are allowed. If this is your case, continue reading the section.

Let's edit *protoname/protoname.h* again as is shown in following code. Lines 1 and 11 are added, while the rest remains unchanged.

```
protoname/protoname.h

1:  #include <mobilenode.h>
2:
3:  /* ... */
4:
5:  class Protoname : public Agent {
6:
7:      /* ... */
8:
9:  protected:
10:
11:     MobileNode* node_;
12:
13:     /* ... */
14: };
```

We have added a reference to a **MobileNode** object (defined in *common/mobilenode.h*), which represents node at which the routing agent is attached to. To get this reference we need to add next line 4 inside **Protoname** constructor.

```
protoname/protoname.cc

1: Protoname::Protoname(nsaddr_t id) : Agent(PT_PROTONAME), pkt_timer_(this) {
2:    bind_bool("accessible_var_", &accessible_var_);
3:    ra_addr_ = id;
4:    node_    = (MobileNode*)Node::get_node_by_address(id);
5: }
```

MobileNode class owns two functions we are interested in. First of all is *base_stn()*, which returns identifier of the base station the mobile node is attached to. Second is *set_base_stn()* which is be able to establish suitable base station for that mobile node. So we can deal with wired-cum-wireless simulations by using these two functions. As an example, next code checks if the mobile node itself is a base station; and if not then it is assigned one.

`protoname/protoname.cc`

```
if (node_->base_stn() == ra_addr()) {
    // I'm a base station
    /* ... */
}
else {
    // I'm not a base station
    node_->set_base_stn(base_stn_addr);
}
```

Former example shows how to change associated base station dynamically. What approaches are used to perform these switches depend on the protocol itself.

References

[1] Marc Greis. *Tutorial for the Network Simulator "ns"*. http://www.isi.edu/nsnam/ns/tutorial/index.html.

[2] The VINT Project. The ns Manual, December 2003. http://www.isi.edu/nsnam/ns/ns-documentation.html.

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.

A **"Modified Version"** of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A **"Secondary Section"** is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The **"Invariant Sections"** are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The **"Cover Texts"** are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A **"Transparent"** copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called **"Opaque"**.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The **"Title Page"** means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section **"Entitled XYZ"** means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as **"Acknowledgements"**, **"Dedications"**, **"Endorsements"**, or **"History"**.) To **"Preserve the Title"** of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.