10. Future revisions of this license

1 Introduction

During the last year, we have witnessed a lot of people asking for the same question is the reverse networking lab. How do I develop any route protocol for NS2? By reading this document we hope to help those researchers who need to add a new route protocol for NS2.27. We from our document in recent protocols. The document is aimed to those people who are somewhat familiar with performing simulations in ns2, 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 moderately understood "Max-Gain Tutorial" [5]. It would be very useful if you also take a look at "The ns Manual" [5], specially chapters 5.1, 10.12, 10.14, 13.23, and 28. 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 double while reading this document.

Before talking about programming. You need knowledge about C++ and a little bit Tcl programming. If you aren’t enough experienced with these programming languages, you should firstly read any of the excellent resources about them which are freely available on the Internet.

2 Getting Started

We are going to implement a new route protocol called protox by step. This protocol does nothing really, but it is enough enough to have several common points with other coding protocols. As you likely know (if you wrote code before), what we will do next is to implement a new protocol using C++ and then to test its simulations. In this section with Tcl scripts.

To allocate our code we will firstly create a new directory called protox under your NS2 home directory. We will create files like these:

protox.mhs. This is the header file where will be defined all necessary type (if any) and routing agent which perform protocol’s functionality.

protox.cc. This file is actually implemented all times, routing agent and Tcl hooks.

protox.proto.h. Here are declared all packets protox protocol needs to exchange among nodes in the network.

protox.proto.cc. Routing table implementation.
There is one task left: to bind our packet header to Tcl interface. We will do so in protonameprotoamemprotonamelcc with the...e of the routing agent. We will treat routing table as a new class, protonamestable.

Also we 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 protonameprotoamemprotonamelA. When the protocol needs to send packets periodically or after some time from the occurrence of an event, it is very world to event 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 time capable of doing such job. A timer should also be able to specify validity 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 have before going into details. The Time class is the base for writing log files with information about what happened during the simulation.

And the last but not for now: when you want to print a debugging message in your code, it is helpful to use the debugger function as it is suggested in chapter 23 [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 we already know the basics, let’s create a new file and call it protonamepkt.h. Here we are going to put all data structures, constants and functions related to our new packet type. The most usual example:

```
struct hdr_protoname_pkt {
  ....
};
```

4 The Routing Agent

Now we start programming the agent itself. Inside protonameproto 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:

```
struct protoname {
  ....
};
```

7

---

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

Let’s get back to the code. We have mentioned the "logical" one class. To implement a routing protocol in NSE you must create an object by inheriting from Agent class. At the very beginning of chapter 18 [1] we can read that objects representing network layer packets are unordered or accessed, 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 options 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, protonamestable.

Also we 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 protonameprotoprotokyleA. When the protocol needs to send packets periodically or after some time from the occurrence of an event, it is very world to event 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 time capable of doing such job. A timer should also be able to specify validity 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 have before going into details. The Time class is the base for writing log files with information about what happened during the simulation.

And the last but not for now: when you want to print a debugging message in your code, it is helpful to use the debugger function as it is suggested in chapter 23 [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 we already know the basics, let’s create a new file and call it protonameprotopkt.h. Here we are going to put all data structures, constants and functions related to our new packet type. The most usual example:

```
struct hdr_protoname_pkt {
  ....
};
```

4 The Routing Agent

Now we start programming the agent itself. Inside protonameproto 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:

```
struct protoname {
  ....
};
```

---
4.2 Timers

As we have learned in previous chapters, one of the critical aspects of a network simulation is the ability to schedule events and control the flow of the simulation. This is achieved through the use of timers. The ProsaSim framework provides a mechanism for defining these timers and controlling their execution.

To define a timer, we use the `protoname_vktzimer__expireJkventL` function, which is called from the `protoname_vktzimer__commandJint` method. This method is called whenever a specified command is executed.

```c
void protoname_vktzimer__commandJprotoname_vktzimer__commandJint
  (protoname_vktzimer__commandJint_t arg)
{
  // Code to handle timer command
}
```

4.3 Agent

The `Agent` class encapsulates the behavior of an agent in the simulation. It is responsible for receiving commands from the simulation environment and executing actions based on those commands.

```c
class Agent
{
  // Constructor and methods
}
```

4.4 Tcl hooks

Tcl hooks are scripts that are executed at specific points in the simulation. They can be used to control the simulation flow, monitor states, or perform other tasks.

```c
proc create_simulation {
  // Code to create the simulation
}
```

5. Conclusion

In this chapter, we have explored the ProsaSim framework for network simulation. We have learned about timers, the `Agent` class, and Tcl hooks. These components are essential for creating dynamic and interactive simulations. With a deeper understanding of these concepts, we are now ready to dive into more advanced topics and develop more complex simulation scenarios.
argv[n] contains the name of the method (always "cmd") see chapter [Z] being invoked. argv[o] is the requested... type and we use it at line [V] wine "does the same but in order to get t– header... it is defined in iplh V Y [125x218]is implemented the same... as defined in iplh V Y [125x248]is simply that the... function that gives us the routing table at a certain time from a simulation script. It assumes that we "simulate" an instance of Simulator and made it a Node created by... are passing 257 as argument because this is the number of the port where a routing agent is attached to. simulation.tcl

1: def at 15-0 "[Node, agent 255] print_table"
Another mandatory command to implement is port-dump. No implementation is provided in line 25-27. An explanation in chapter 3 of [2]. NI stores a reference to every mapped object (C=object) in a hash table to provide a fast access to each of them in your game. We make use of that facility in line 28 to obtain a Forwarder object given its name.

Specifically, there is another mandatory operation called truss-test that we add to this code because we very often use this program to test the scheme. If we don’t know how to proceed the requested command, we delegate this responsibility to base class, as we do in line 65.

4.3.3 recv()
Next function is recv() and as we know it is invoked whenever the routing agent receives a packet. Every Packet has a common header that is defined in common/packet. To access this header, there are many ways as we can use the two classes Forwarder and BasePacketType to do this, all are equivalent. Line 1 does the same but in order to get IP-header, we mapped, described in get.h.

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

Lines 3-4 get IP header and a protoname packet as packet. After that we make sure that destination parts are PROTONAME or PROTOATIVE at line 6-9. This constant is defined in common/packet and it equals 257. This is to prevent the routing agent from dropping the packet.

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: { Protocolname: recv_protons.type(Packet) pl {}
3:   struct hdr_ip ik = * HDR_IP();
4:   struct hdr_protons.type ik = HDR_PROTONAME(Ptcpn);
5: }
6: // All routing messages are sent from and to port RT_PRT,
7: // so we check it
8: if (hdr sidl() == RF_RPT) { // RFC 1122
9:   struct hdr_sid lk = * HDR_SID();
10: }
11: // */ processing of protoname packet */
12: // Release resources
13: // Packet: (free); 15: }

4.3.5 send_protons.type.plt()
In our code, there is a common routine called the function send_protons.type(), whenever it expires. We show a simple implementation of this function below. Normally you can call that protocol expires receiving something different and this is just an example.

1: void 2: { Protocolname: send_protons.type(Packet) pl {}
3:   struct hdr_ip ik = * h2r(HDR());
4:   struct hdr_protons.type ik = HDR_PROTONAME(Ptcpn); 5:   // IPv4 protocols planners
6:   if (hdr_sid() == RF_SND) { 7:     if (there exists a hop, start drop the packet 8:       if (ch_cap(formula) != 1) 9:         drop(), DROP_XFR_BROK_LOOP;
10:     return;
11:   } else if this is a packet; we are originating, add IP header
12:     else if (ch_cap(formula) != 0) 13:       return(); 14:     else;
15:   }
16: }
17: // If it is in a protoname, make protoname
18: if (ch_cap(formula) == RT_PROTONAME)
19:   recv_protons.type(pl);
20: // Otherwise, make Forwarder the packet (unless TTL has reached zero)
21:   if (1) { 22:     struct.getHeader();
23:     return;
24:   }
25: // */ format_detail();
26: // 27:
28: // */
29: // */

First thing that we do is to check we are not receiving a packet we may ignore. If that is the case we should drop the packet and return, as we do in line 9-11. 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 was spent to generate it (in bytes). We assume protoname packets over IP, as it is shown in line 15-16.

When the protoname packet is type RT_PROTONAME, we process it (line 19). If it is a data packet then we process it as we would any other packet, that is, if it was a headend packet or was destined to a site, reduce TTL and added. Line 21-26 is what we have just described making use of the format_detail() function.

We should also realize that the drop() function is used to dropping packets. Its argument is a pointer to the packet itself and a constant giving the reason for discarding it. There exist several of these constants. You can take a look at them in the file trrpc/sid.c.

// Note: In case of the packet exceeding IP header

13: char_direction() k = h2r (src); 14: char_next() = * IP_HDR(src) + hp Next();
15: char_next() = * IP_HDR(dst) + hp Next(); 16: char_next() = * IP_HDR(dst); 17: char_next() = * IP_HDR(dst); 18: char_next() = * IP_HDR(dst);
19: char_next() = * IP_HDR(dst); 21: char_next() = * IP_HDR(dst); 22: char_next() = * IP_HDR(dst);
23: char_next() = * IP_HDR(dst); 25: char_next() = * PL (protocol number);

26: // Schedule: instance) schedule([target p., CONTEXT];
27: // */

28: // To send a packet we need to allocate a new. We use the alloc() function for that. This function is defined for all Agents. Then we get common, and protoname packet header as usual (line 34). Our aim is to fill all three headers with values we want to:

29: // Protoname packet header is filled in line 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 packet specification.

30: // The common headers in NS are defined in NS_data_fields. We are mainly concerned with how we are interested (line 2-7). We need to set the packet type to a protoname packet (line 12). We also assign the packet direction in line 13. If we are sending a packet, it is going down... 33: // This is a very important field, and it’s protocol number as IP_PROTOCOL_IP for IP packets.

34: // The best field we fill is the address field. It can be NS/IP/VX/DNXNS... 36: // We choose NS_IP, IP because we are implementing an Internet protocol.

37: // Now we proceed with the configuration of the IP header. It is very simple... 39: // So there is a new constant called IP_PEER, which is defines it to correspond to the default TTL value for IP packets.

40: // The IP header has other fields used for IPV6 simulations, but we don’t need... 42: // For our example, we do (see chapter 12 of [2]) as they need to be scheduled. In fact, sending a packet is equivalent to schedule it at a certain time. Line 21 shows how to send a packet introducing some jitters. The SendPacket class inherits from the Connector class, which has a reference to a TCPsocket called tcp SOCK. This is the handler which will process the event, and is passed as an argument to the schedule() function.
4.3.6 reset_protosname_ab_timer()

Our packet sending timer performs another callback (section 4.4) to recompute
field. It's done in the function reset_protosname_ab_timer(). We show that in
next example, where protosname__reset_protoname_pkt_timer() is recompiled to expire five seconds later.

```
protosname__reset_protoname_pkt_timer()
    void
    void protosname__reset_protoname_pkt_timer()
        {  
            get_time__reversed(chrono[0, 0]);
        }
```

4.3.7 forward_data()

So far we have been mainly focused on protosname packets, but it's time to
dual with data packets. The forwarding function decides whether a packet
has to be delivered to the upper-layer protocol or to be forwarded to other nodes.
We check for the first case in line 20. When it's an incoming packet and
destination address is in the node itself or broadcast, then we the node's down,
if we consider it is a PortClassifier object to accept the incoming packet.

Otherwise, we must forward the packet. This is accomplished by properly
calling the common header with as we do in lines 12-20. 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 if 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 23.

```
protosname__forward(data)
    {  
        struct hdr_cmn
        {  
            types hdr_type;
            types hdr_len;
        } hdr;
        ts_u_intUT_t size;
        {  
            if (nsaddr_t* addr = &upשיתוף)
                return a
            else
                return a
            if (next_hop) IP_BROADCAST
                return a
            else
                if (next_hop) IP_BROADCAST
                    return a
            return;
        }
    }
```

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 address, next hop addresses, distance or cost
connected to the routes, expiration times, distances, and so on. Of course
we may want to store some simple routing table and a method to get
the only information we will store in each entry is destination and next hop
address. We use a hash table (soon) as the storage structure. This case is too
difficult to implement a new class, but we will do it in another example. The next
piece of code corresponds to protosname__proto_rtable.h.

```
protosname__proto_rtable.h
    {  
        typedef enum
        {  
            case
            {  
                void
                case
            }
        getClass
        {  
            typedef enum
            {  
                case
            }
        getClass
```

6 Needed Changes

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

6.1 Packet type declaration

If you remember we had to use a constant to indicate our new packet type,
IP_PKT_BROADCAST. We will define it inside the source file. Let's find packet.h,
where all packet types are listed. We will add IP_PKT_BROADCAST to this list as
we show in the next piece of code (line 4).

```
packet.h
    {  
        case packet_t
            {  
                void
                case
```

19
6.2 Tracing support

As we have demonstrated, one is to get a trace file describing what happened during execution. To do so, familiar with trace files need chapter 21 [3]. A Trace object is used to write wanted information of a packet every time it is received, sent or dropped. To log information regarding our packet type, we implement the (Packet protocol) function inside the CMUTrace class. Trace support for window simulation is provided by CMUTrace objects and it is described in chapter 21 [3].

Let’s edit trace/trace.c file. We must add our new function as in the line number 6 of the next example.

```
struct packet_trace; // This MUST be the last one

void vrotoname(
    struct packet_trace *pt,
    const char *msg)
{
    switch (msg) {
        case PACKET_RECV:            
            break;
        case PACKET_SENT:            
            break;
        case PACKET_BAD:             
            break;
        default:                     
            break;
    }
}
```

The next piece of code (extracted from trace/trace.c) shows different types of traces.

```
struct packet_trace;

#include <protoname/prototrace_pkt.h>

// ... definitions ...

// priv ...

void format_node(Packet vp, int offset);

void format_prototrace(Packet vp, int offset);

}
```

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 limited attributes and provide the needed information to create window nodes running our protocol.

In c/cm/tcl/tcl/tcltrace all files that you must locate the next code and add prototype to the list (as we do in line 2).

```
tcl/lib/tcl-packet.tcl

1: # format print {
2:   Prntname
3:   ADD
4:   ANP
5:   # ...
6:   HV
7: )
8: # add-packet-header $pt
9: )

# Default values for limited attributes have to be given inside $id/$default.tcl.
# We must go to the end of the file and write something like the next code:

tcl/lib/$default.tcl

1: #
2: # Default defined for Prototrace
3: Agent/Prototrace set accessible/true
```

Finally we have to modify c/cm/tcl/lib/tcl. We need to add procedures for creating a node. Our interest will be centered around creating a window node with protocol as routing protocols.

The procedure code is to create-window node procedure. This last one, among other tools, is intended to set the routing agent for a node. We must add this procedure to create an instance of our protocol.

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

1: # Simulator commands: create-wireless-nodeargs 
2: # ...
3: switch --exec RoutingAgent
4: Prototrace 
5: set agent [lindex [lsort $nodes] 0]
6: )
7: # ...
8: # ...
9: # ...
10: }
```

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

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

1: # Simulator commands: create-prototrace-agent ( node ) { 
2: # Create Prototrace routing agent 
3: # Routing Agent (new agent)
4: D (node)
5: Set at 0.0 "RoutedEventArgs" [Dnode node-wide] 
6: Bind at 0.0 "RoutedEventArgs" [Dnode node-wide]
7: Bind at 0.0 "RoutedEventArgs" [Dnode node-wide]
8: return $agent;
9: # Line 3 is a new protocol with $node's address. This agent is scheduled to start at the beginning of the simulation (line 6), and is assigned as the node's routing agent in line 5.
```

6.4 Priority queues

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 beginning of the queue. We need to tell the PriQueue class that protocol packets are routing packets and therefore treated as high priority.

We need 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.

```
priqueue.cc

1: void
2: PriQueue::recv(Packet vp, Handler *h) { 
3: }
4: struct lock_cm
5: if (PriQueue_Routing_Protocol) {
6:  switch(lock(tuple)) { 
7: case PriQueue
8: case PriQueue
9: case PriQueue
10: case PriQueue
11: case PriQueue
12: case PriQueue
13: case PriQueue
14: case PriQueue
15: case PriQueue
16: default;
17: case PriQueue
18: case PriQueue
19: case PriQueue
20: case PriQueue
21: case PriQueue
22: case PriQueue
23: case PriQueue
24: case PriQueue
25: case PriQueue
26: case PriQueue
27: case PriQueue
28: case PriQueue
29: case PriQueue
30: case PriQueue
31: }
```


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 [40] terminology). Wired-cum-wireless scripts need to use hierarchical addressing, as you used (read chapter 13 and 20) to get necessary knowledge in this type of addressing.

With minimal changes we could use our protocol in wired-cum-wireless simulations. In these cases there are fixed nodes, wireless nodes and base stations.

MobileNode class owns two functions we are interested in. First of all is base station which returns identifier of the base station which is reachable to such a mobile node. We will call this function once function we fixed() before we only have to modify line 9 of "proventame.c"

    1. static int  base_station (Node *node)
    2. {
    3.     int base_id = 0;
    4.     for (int i = 0; i < node->n_base_stations; i++)
    5.         if (node->base_stations[i] == node)
    6.             base_id = i;
    7.     return base_id;
    8. }
    9. #endif

    10. #endif

    11. #endif

MobileNode class owns two functions we are interested in. First of all is base station which returns identifier of the base station which is reachable to such a mobile node. We will call this function once function we fixed() before we only have to modify line 9 of "proventame.c"

    1. static int  base_station (Node *node)
    2. {
    3.     int base_id = 0;
    4.     for (int i = 0; i < node->n_base_stations; i++)
    5.         if (node->base_stations[i] == node)
    6.             base_id = i;
    7.     return base_id;
    8. }
    9. #endif
    10. #endif
    11. #endif

We have added a reference to MobileNode object (defined in core/sock/sock.h), which represents node to which the routing agent is attached to. To get this reference we need to add line 4 inside "proventame.c" constructor.

    1. #include "mobilenode.h"
    2. #endif
    3. #endif
    4. static int  base_station (Node *node)
    5. {
    6.     int base_id = 0;
    7.     for (int i = 0; i < node->n_base_stations; i++)
    8.         if (node->base_stations[i] == node)
    9.             base_id = i;
    10.     return base_id;
    11. #endif
    12. 
    13. }

References

A GNU Free Documentation License
Version 1.2, November 2002
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
Everyone is permitted to copy and distribute verbatim copies of this license,
but changing it is not allowed.

Preamble

The purpose of this License is to permit the free distribution of original
works, modified versions of such original works, and compilations of
these works. It is intended as a form of cope, generally available as
development sets, allowing users of the License to copy and distribute
the same works freely provided that they also provide the terms
of this License to others.

1. APPLICABILITY AND DEFINITIONS

A "Licensee" is anyone who is permitted to copy, distribute and/or
modify the Document under these terms. The "Document" means any
work covered by this License, unless specially marked as "Restricted".

A "Work" means a copyrightable work, such as a book, essay,
lecture,lawsuit, movie, sound recording, computer program, or other work
whether or not it is copyrightable.

A "Copy" means a human-readable

You may copy and distribute the Document in any medium, either
commercially or noncommercially, provided that this permission notice
is included on all copies of the Document (and might also be included
on some or all other works that make use of it).

1. COPYING IN QUANTITY

You may copy and distribute the Document in any medium, either
commercially or noncommercially, provided that this permission notice
is included on all copies of the Document (and might also be included
on some or all other works that make use of it).

1. COPYING IN QUANTITY

You may copy and distribute the Document in any medium, either
commercially or noncommercially, provided that this permission notice
is included on all copies of the Document (and might also be included
on some or all other works that make use of it).
N. Do not notify 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 not use your option to designate these as Invariant Sections.

To do this, add the titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must not 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 25 words as a Best-Case Test, and a passage of up to 25 words as a Worst-Case Test, to the end of the list of Cover Texts in the Modified Version. Only one passage of Best-Case Test and one passage of Worst-Case Test may be added by all contributors to a particular entity. If the Document already includes a Cover Test for the same cover, previously added by you or by an arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, or replace a promotion with the present promotion that added the old one.

The author(s) and publisher(s) of the Document do not give permission to use that name for publicity except to assert the implied 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 the combination of all 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 not contain any portion of the Document released under this License that appears in the combination by permission from the copyright holder unless you have received additional permission to include that portion or portion(s) from the copyright holder. If the Document is a collection, you may include an individual version of this License, or a calculation of royalty fees based on furthermore, in any other way included in or derived from the Document as a whole, provided that you indicate the basis of the royalty fees charged, and that you include an adequate notice in that notice of the most recent term of this License to be applied to any given work, in whichever language and format may be chosen for legal notices normally used by the copyright holder.

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 modifying a work specified in this paragraph, provided that you do not modify this License itself.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, as specified in section 4, may always be included in compilations otherwise complying with the rules of this License.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of the section 4, replacing the original License with the translated version, or the original License with the translation. Otherwise, translation is not considered an option to modify the Document, and it is therefore considered a modification, and any distribution of another version of it must be subject to the terms of section 4.

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/.