IP Noise Generator

Final Report

Submitted by:
Shlomi Fish
Roy Glasberg

Supervised by:
Lavy Libman

Content

1 General Information

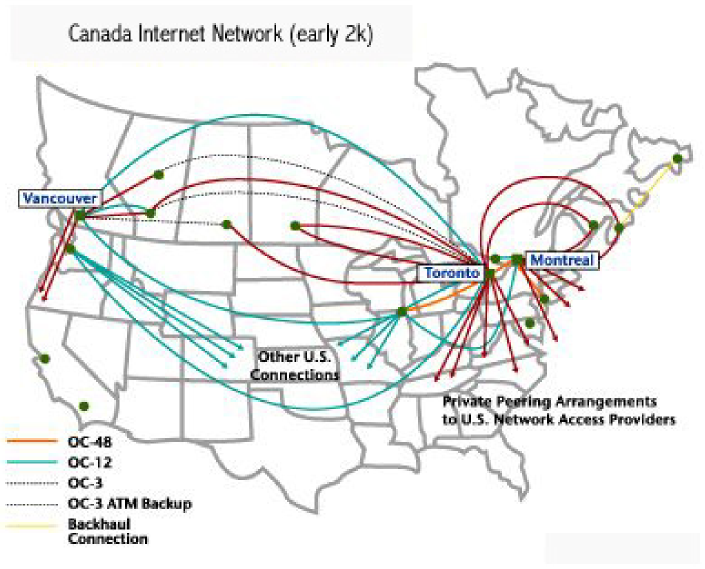
The Internet Protocol (IP) is used as the infrastructure for many computer communication protocols. Since the Internet is far from being noise free, protocols that are built above it should be relatively immune to such noise.

One way to test the merit of these new protocols is to try them out on the Internet or an intranet environment with a “known” noise behavior. However, such an environment is hard to set up, and so this method leaves a lot to be desired.

A better development process of these protocols would require a controlled and monitored environment that will simulate the different conditions one may encounter when using the IP, on demand.

The following phenomena may be encountered when routing packets on the Net:

1. Packets with the same source and destination may travel along different routes.
2. Information packets may be lost.
3. Information packets may be delayed.
4. Packets may arrive out of order or in order.
5. The behaviour of these phenomena may vary over time.

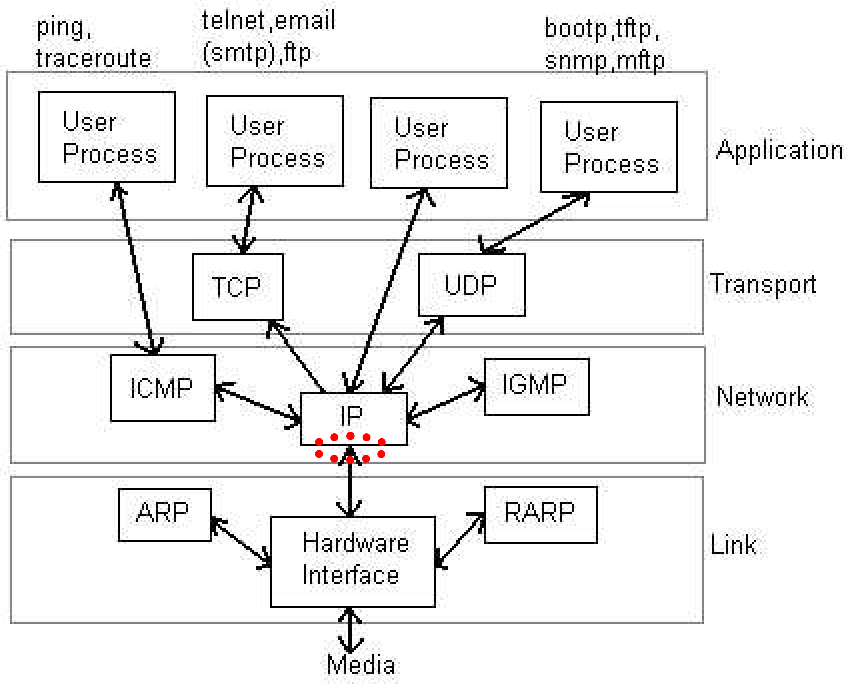

Canada Internet Network (early 2k)
Vancouver
Toronto
Montreal
Other U.S. Connections
Private Peering Arrangements to U.S. Network Access Providers
OC-48
OC-12
OC-3
OC-3 ATM Backup
Backhaul Connection

A representation of a small part of the Internet network.

2 Purpose

We created a simulation tool that aids the process of evaluating protocols above the IP protocol. It simulates such an environment (with all the mentioned behaviours of noises).

Using this tool one can handle packets differently according to their varying parameters: source, destination, length, ToS[1], protocol; as well as the source and destination port if relevant. It simulates environments in which the tested protocol traffic is insignificant and therefore does not affect the noise behaviour.



Network Model and Point of Packet Interception.

[1] ToS stands for "Time of Service" which is a 6-bit field in the IP header that specifies a Quality-of-Service like value

3 User's Guide

3.1 Introduction

3.1.1 What is the IP-Noise Simulator

The IP-Noise Simulator is a simulator for TCP/IP networks noise that can run on top of a Linux 2.4.x system. It is available in two versions: one as a stand-alone user-level program that can run from the command line. The second one is a kernel module, which should be loaded with `insmod`.

TCP/IP noise generation involves the deliberate dropping or delaying of packets sent over the TCP/IP network. The simulator can distinguish between the various protocols above the IP level and arbitrate them differently accordingly. However, the contents of the packets cannot affect its decision in any way.

Note that it is assumed that the network congestion is independent of what occurs in the applications that are tested with the noise. Thus, the load of the network does not affect the simulator's behaviour.

3.1.2 When can it prove useful?

We intend the simulator to be used to test the robustness of protocols, and see how immune they are to network noise. The simulator is designed to be flexible and simulate various network noise conditions, in a manner that is accurate enough for most needs.

3.1.3 Credits

The project was initiated by Lavy Libman, and was written by Shlomi Fish and Roy Glasberg under his supervision. It was conducted with the help and resources of the Computer Networks Lab of The Technion.

We would like to thank the various people in the Israeli Group of Linux Users who provided important help and input during the various stages of the project: Omer Mussaev, Guy Keren, Mulix and others. We would also like to thank the developers of the Netfilter firewalling stack of the Linux Kernel 2.4, which gave us most of the necessary kernel functionality, and made our work much easier.

We would also like to thank the FOKUS group for creating and maintaining the BerliOS service, whose on-line resources we were able to use.

3.1.4 Supported Platforms

The simulator requires a GNU/Linux system running the Linux Kernel version 2.4.x. IP-Tables has to be available as a kernel module, as is the IP-Queue module.

3.1.5 License

The translator which is written in Perl is distributed under the MIT X11 license. The compiler is distributed under a dual GPL and MIT X11 license except for the following modules:

- `redblack.c` - derived work which is distributed under the LGPL.

(part of the libredblack distribution)

- `ip_queue.c` - derived work which is distributed under the GPL. (was derived from the module of the same name of the Linux 2.4.x kernel.)

The Perl arbitrator is no longer maintained and is only useful as a reference implementation. We cannot guarantee that it does not contain bugs. It is distributed under the MIT X11 license.

3.2 Installation Instructions

3.2.1 Global Instructions

3.2.1.1 Prerequisites

In order to use the simulator, the IP-Tables kernel module must be compiled, and available for loading. Specifically, the ip_queue module should be available. perl v. 5.6.1 or later must be available on the system in order to configure the simulator.

libipq which is part of the iptables userland distribution should be installed and available on the system, as the userland simulator makes use of it.

3.2.1.2 Compiling the Simulator

Download and unpack the latest distribution. `cd` to the directory `C/arbitrator`. In order to compile the userland simulator type "`make`".

In order to compile the kernel module-based simulator type "`make -f Makefile kernel`". In case you don't have an i386-compatible machine, you should edit the file `Makefile.kernel` and change the line "`I386 = 1`" to "`I386 = 0`".

3.2.2 Userland Arbitrator Specific Instructions

Designate a directory in which two named pipes will be put. These pipes will be used for communication between the arbitrator and the configuration utility. The directory should be accessible both by root and by the user, with which you intend to run the configurator (which we recommend should not be root).

In this directory type the following set of commands as root:

```
# pwd
/ip-noise
#mkfifo to_arb
#mkfifo from_arb
#chown ipnoise.ipnoise to_arb from_arb
#chmod 600 to_arb from_arb
#chmod 700 .
#chown ipnoise.ipnoise .
```

`ipnoise.ipnoise` is the user-name and group of the user that will be used to invoke the configurator. Now set the environment variable `IP_NOISE_UM_ARB_CONN_PATH` to the full path of this directory in the initial shell scripts of both root and the `ipnoise` user.

3.2.3 Kernel-module Arbitrator Specific Instructions

Under the home directory of the user which will run the module, type the following commands as root:

```
# mknod iface_dev c 254 0
# chown ipnoise.ipnoise iface_dev
# chmod 600 iface_dev
```

`ipnoise.ipnoise` is the user-name and group of the user which will run the configuration utility.

3.3 Running the Arbitrator

3.3.1 Setting up IP-Tables Rules to Transfer Packets for Noisifying

The simulator intercepts packets from the kernel by means of the IP-Tables framework. It receives those packets (and only those packets) that were designated with the `QUEUE` rule. Normally, you would like to configure the IP-Tables rules, so that important communication will not be interfered with. Only the ports, protocols and IPs of interest need to be transferred to the arbitrator in order to decide their verdict.

Describing how to set up IP-Tables rules is out of the scope of this article. We refer you to the Internet for further information. However, we will present a script that we use to set up the IP-Tables rules on our Mandrake Linux 8.1 system:

```
#!/bin/sh

UDP_PORTS="67 111 135 137 138 161 513 520 525 631 2571"
TCP_PORTS="6346 6347 5901"
IFACES="eth0 lo"

for iface in $IFACES ; do
    for port in $UDP_PORTS ; do
        /sbin/iptables -A INPUT -i $iface -j ACCEPT -p udp --
destination-port $port
    done
    for port in $TCP_PORTS ; do
        /sbin/iptables -A INPUT -i $iface -j ACCEPT -p tcp --
destination-port $port
        /sbin/iptables -A INPUT -i $iface -j ACCEPT -p tcp --source-
port $port
    done
    /sbin/iptables -A INPUT -i $iface -j QUEUE -p tcp
    /sbin/iptables -A INPUT -i $iface -j QUEUE -p udp
    /sbin/iptables -A INPUT -i $iface -j QUEUE -p icmp
done
```

Note that the IP-Noise kernel module replaces the IP-Queue mechanism of the Linux kernel, and both cannot be used simultaneously.

3.3.2 Loading and Unloading the arbitrator

3.3.2.1 Userland Arbitrator

After the IP-Table rules has been set up follow the following steps:

1. `modprobe ip_queue`

2. As root, cd to the `C/arbitrator` directory of the distribution. There type `./arb` to invoke the arbitrator.

To unload it, simply type `Ctrl+C` at the terminal in which it was invoked.

3.3.2.2 Kernel-Module Arbitrator

After the IP-Tables rules have been set up, cd to the `C/arbitrator` directory of the distribution as root. There type `insmod ./ip-noise-arb.o`.

To unload it type `rmmod ip-noise-arb` as root.

3.3.3 Configuring an Arbitrator

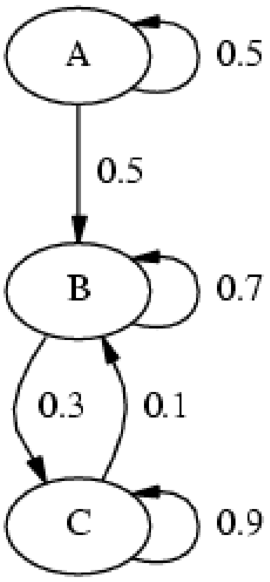
To configure an arbitrator one has to prepare a configuration file that contains the description of the noise behaviour, in the syntax that was defined for this purpose (which will be described below). After the file has been prepared, cd to the `perl/compiler` directory of the distribution. Then type `perl tests/translator.pl` or `perl tests/ker_translator.pl` (for the userland and kernel-level arbitrators respectively) followed by the path of the configuration file.

3.4 Configuration of the Arbitrator - Basic Concepts

3.4.1 Markov Chains

Markov chains are similar in concept to state machines, except that the next state is determined according to a probability factor rather than the input to the state machine. In each iteration, a probability value between 0 and 1 is determined randomly (and uniformly) and based on it, the next state is deduced.

The following figure, displays a sample Markov chain:



As can be seen, after an iteration step, "A" switches to itself with a probability of 0.5, and to "B" with a probability of 0.5. "B" and "C" switch to themselves and to one another with a various probabilities. One can notice, that once the chain leaves "A", it cannot return to it again.

The sum of the probabilities of the links that emerge out of a certain state must be equal to 1, so the chain will always know to which state to go to next. In our implementation of Markov chains, we assumed that in case, the sum of emerging probabilities is less than 1, than the remainder instructs the chain to remain in the current state. If the sum of probabilities is greater than 1, a compilation-time error will be reported.

One can model memory-less noise using a Markov chain, in which each state dictates a different statistical behaviour of the noise.

3.4.2 Basic Concepts

3.4.2.1 Delay

A packet can be delayed by a certain amount of milliseconds. This delay can be determined randomly, by specifying a certain type of statistical distribution or another.

3.4.2.2 Drop

One can specify to drop the packet altogether representing a loss of a packet along its route.

3.4.2.3 Accept

One can specify to simply "accept" the packet and send it on its way. For all practical purposes, an accept is treated as a delay of 0.

3.4.2.4 Exponential Delay

An exponential delay is a distribution of a delay according to the formula $F(t)=1-e^{-\lambda t}$. The lambda factor can be determined by the user.

3.4.2.5 Generic Delay (or Split-Linear Delay)

This is a generic delay type that can be used to model an arbitrarily complex delay function. In it one specifies points along the randomosity factor from 0 to 1. For each such point, one specifies the delay at that point. Between two adjacent points, one interpolates their end-values linearly.

3.4.2.6 Uniform Delay

In this delay type, the delay is chosen uniformly between a minimum and a maximum specified by the user.

3.4.2.7 Stable Delay

A delay can be either stable or non-stable. In a stable delay, the packets are sent on their way at the same order in which they arrived. In a non-stable delay, their order may be mixed. One can decide whether to issue a stable or non-stable delay by specifying a probability factor.

3.4.3 How the Arbitration is Done

3.4.3.1 Several Markov Chains

The arbitrator contains several Markov chains which are run and processed in parallel. Each chain has a chain filter that specifies which packets it wishes to process. When a packet arrives at the arbitrator, it is passed to all the chains, except for a special chain which is designated as the default chain. Each chain determines on its own what to do with the packet, without consulting the other chains.

A packet that is not processed by any chain is passed to the default chain, whose filter (if any) is ignored. If more than one chain processed the packet, then the following rules apply:

1. If the packet was dropped by any chain, then it will be dropped. (i.e: each chain can veto the verdict to a drop)

If none of the chains dropped it, then the delay the packet would experience would be the sum of the delays of the chains. (and as previously mentioned, an accept verdict is considered as a delay of length 0)

3.4.3.2 State Switching

In each chain, the states are switched at exponential times from the time of the last switch. Each state determines the probabilities for a packet being dropped, delayed or accepted and the distribution of the delay. It also determines the probability for a stable delay.

3.4.3.3 Chain Filters

A chain filter determines which packets are processed by a chain according to their TCP/IP information. The motivation for a chain filter is that different parts of the network being modeled may experience different conditions. Therefore, packets need to be treated differently based on their destination, source and other TCP/IP properties.

In the IP-Noise Simulator, packets can be filtered according to the following factors:

Source IPs and port combinations.

Destination IPs and port combinations.

Packet Length

Value of the Time of Service field

The Protocol type above the IP layer (e.g: TCP, UDP, ICMP, OSPF and IGMP).

3.5 The Configuration File Syntax

3.5.1 General Conventions

Empty lines are ignored. Comments start with the sharp-sign (`#`) and extend until the end of the line.

Identifiers contain letters, underscores and digits, where the first character is not a digit. Identifier are not case-sensitive.

Block constructs are declared at one line, in which a left curly bracket (`{`), must appear, and after it, nothing else. They span several lines, until they are terminated with a right curly bracket (`}`) on its own line.

Blocks contain field names followed by an equal sign (`=`) followed by a value. All of which must appear on the same line.

3.5.2 Syntax of a State

A state starts with the `State` keyword, followed by the state name followed by a left curly bracket (`{`). The state name is an identifier.

The `State` keyword, the name and the `{` must all appear on the same line. A state ends with a right curly bracket (`}`) which appears on a line of its own.

3.5.2.1 drop, and delay probabilities

If a `drop` or `delay` field/value pair appears it specifies the probability in which a packet will be dropped or delayed when the chain is at this state. The values of `drop` and `delay` will be assumed to be 0 if not specified, and the rest of the probability will be assumed to send the packet as is.

Example:

```
State MyState {
    drop = 0.2
    delay = 0.3
}
```

This state drops packets with a probability of 0.2 and delays them with a probability of 0.3

3.5.2.2 time_factor

The `time_factor` field specifies the time factor (in milliseconds) of the state. This will be supplied as input to an exponential delay calculation that will determine the time the state is switched.

3.5.2.3 move_to's

The `move_to` field specifies the probabilities the state will switch to other states. After a state is switched, the next state is determined according to a random value.

The syntax is `move_to = {` at the first line followed by a list of `[Probability] = [State]` lines followed by a line containing a right curly bracket (`}`). If the probabilities sum below 1, the remainder will be assumed to remain in the current state.

Here's an example:

```
State MyState {
    drop = 0.2
    delay = 0.1
    time_factor = 500
    move_to = {
        0.2 = AnotherState
        0.1 = StateWithoutAName
    }
}
```

This state switched to `AnotherState` with a probability of 0.2, and to `StateWithoutAName` with a probability of 0.1, and remains at itself with a probability of 0.7.

3.5.2.4 delay_type

The `delay_type` field specifies the type of the delay. Its value can be any of the following:

Exponential: With a given factor in milliseconds. Marking: `E([factor])`. For example: `E(500)`, or `E(10)`.

Uniform: Marking: `U([start],[end])`. For example: `U(100,500)`.

Generic: This defines a function from 0 to 1 using intermediate points. The value field should start with `Generic {` and end with a line containing a right curly bracket (`}`). Between them, there are `[prob] = [delay]` lines. Note that 0 to 1 will be chosen within the probability range allocated for delaying a packet.

For example:

```
delay_type = Generic {
    0 = 500
    0.1 = 500
    0.2 = 300
    0.5 = 0
    0.8 = 0
    0.8001 = 1000
    1      = 2000
}
```

3.5.2.5 stable_delay

The `stable_delay` field specifies the stable delay probability of the state.

3.5.2.6 Complete State Example:

```
State A {
   drop = 0.2
   delay = 0.1
   delay_type = E(200)
   time_factor = 500 # This is the time factor to move to the other
state
   stable_delay = 0.2  # This is the probability for a stable delay.
   move_to = {
        0.2 = B
        0.3 = C
        0.5 = E
   }
}
```

3.5.3 Chain Syntax

A chain starts with the keyword `Chain` followed by the chain's identifier followed by a left curly bracket (`{`) - all on the same line. It is terminated with a right curly bracket on a line of its own (`}`).

3.5.3.1 States

A chain can contain an arbitrary number of states, that are specified using the state syntax. There is one special state which ought to be named `Start`, which is the initial state of the chain.

3.5.3.2 The Chain Filter

3.5.3.2.1 length

The length field specifies the length of the IP packets that is accepted by the chain. Its syntax is `length = l < [max_len]` , or `length = l > [min_len]` or `length = [min_len] < l < [max_len]`.

3.5.3.2.2 tos

The `tos` field specifies a valid range for the Time-of-service specifier. The syntax is similar to the `length` field except that the variable `tos` is used instead of `l`. For example: `tos = tos > 30` or `tos = 2 < tos < 5`.

3.5.3.2.3 dest and source

`source` specifies a list of IP addresses and their corresponding ports for the source of packets to be processed by the chain. `dest` is the same only for the destination of the packets.

The value of these fields is a semicolon delimited list of IP specifications. An IP specification contains an IP address (4 decimal digits separated by a dot (`.`)); an optional netmask width preceded by a slash (`/`). The last part is an optional port specifier that begins with a colon (`:`) and contains a comma-delimited list of port ranges or individual port numbers. A port range is of the form `[start_port]-[end_port]`.

Note: if the protocol above the IP does not use ports, then the port specification will be ignored.

Examples:

```
source = 127.0.0.1:80                   # Only HTTP from the localhost
source = 10.0.0.0/24:80,8080            # Only HTTP and 8080 from
10.x.x.x
dest = 132.69.253.254 ; 132.68.7.4      # Either of the two IP
addresses.
```

3.5.3.2.4 protocol

The protocol specifies the protocols above IP that will be accepted by the chain. Each can be a numeric value or it can be a name of the protocol (according to the file /etc/protocols). The individuals protocols are separated by a comma.

Example:

```
protocol = tcp, udp
```

3.5.3.3 Complete Arbitrator Example

```
# This is the chain which handles packets which were not accepted
# in any of the other chains.
Chain Default {
    State A {
        time_factor = 100000
    }
}

Chain vipe_len_lt_100 {
    State Start {
        time_factor = 5000
        drop = 0
        move_to = {
            0.5 = Noisy
        }
    }
    State Noisy {
        time_factor = 5000
        drop = 1
        move_to = {
            0.7 = Start
        }
    }
    length = l < 100
    dest = 127.0.0.1 ; 132.68.52.118 ; 132.69.253.254
    source = 127.0.0.1 ; 132.68.52.118 ; 132.69.253.254
}

Chain vipe_len_gt_100 {
    State Start {
        time_factor = 5000
        drop = 0
        move_to = {
            0.5 = Noisy
        }
    }
    State Noisy {
        time_factor = 5000
        drop = 0
        delay = 1
        delay_type = U(3000,3000)
        move_to = {
            0.7 = Start
        }
    }
    length = 1 > 100
    dest = 127.0.0.1 ; 132.68.52.118 ; 132.69.253.254
    source = 127.0.0.1 ; 132.68.52.118 ; 132.69.253.254
}


Chain t2 {
    State Start {
        time_factor = 5000
        drop = 0
        move_to = {
            0.5 = Noisy
        }
    }
    State Noisy {
        time_factor = 5000
        drop = 1
        move_to = {
            0.7 = Start
        }
    }
    dest = 127.0.0.1 ; 132.68.52.118 ; 132.68.7.4
    source = 127.0.0.1 ; 132.68.52.118 ; 132.68.7.4
}
```

3.6 Final Notes

3.6.1 Known Issues

1. The kernel module arbitrator does not work with the new virtual memory manager by Andrea Archanageli. We are inspecting what can be done to change the situation.

2. It is possible for a malicious compiler that communicates with the kernel module to cause the kernel to crash by having it allocate a lot of memory.

3.6.2 Disclaimer

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE

3.6.3 Links and References

The IP-Noise Simulator Project's Homepage

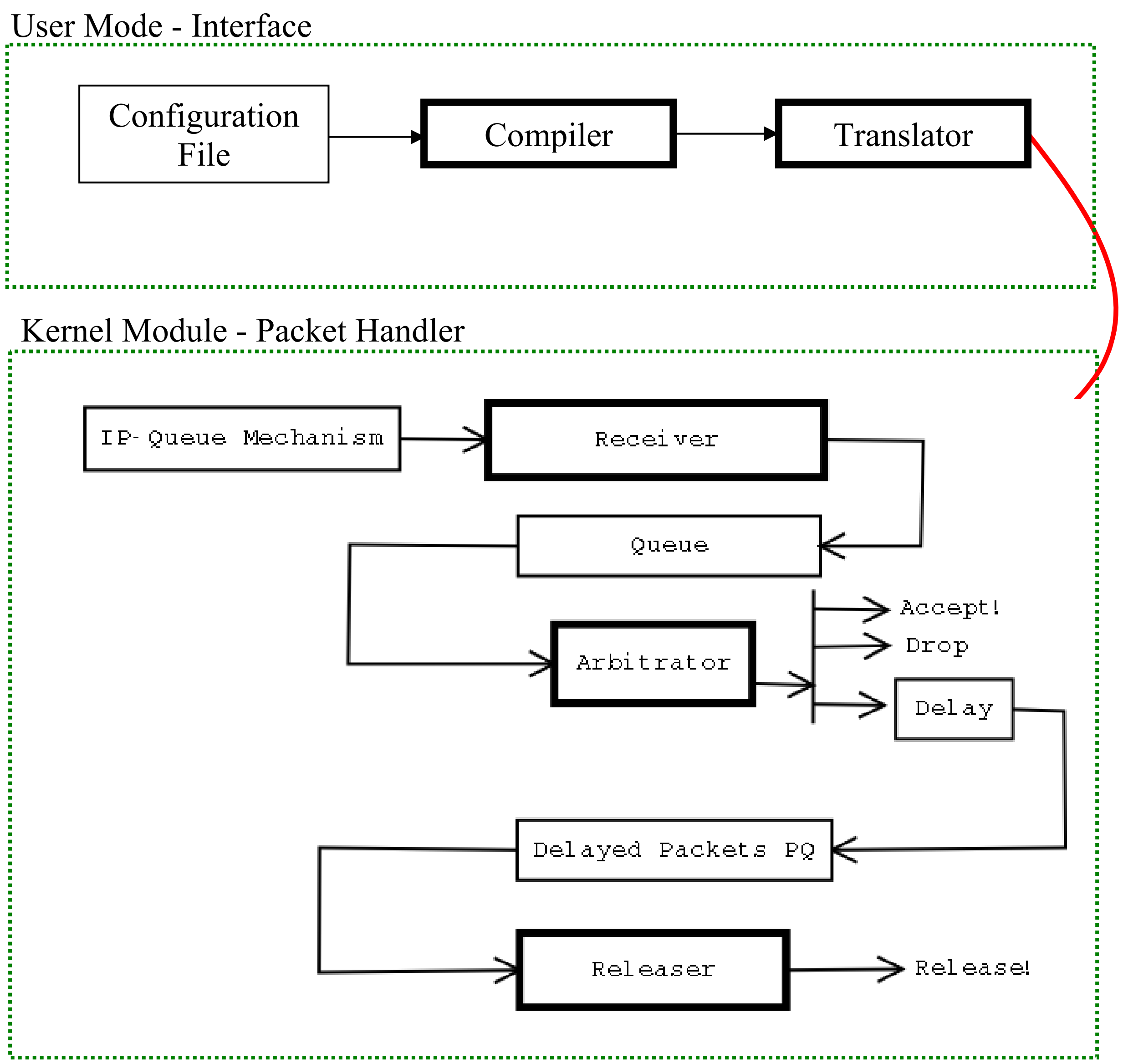
Netfilter - the IP Firewalling stack of the Linux kernel version 2.4.x.

User Mode Linux - Run the Linux as a process. Easy debugging with GDB, run a system as a non-privileged user, and other cool features.

BerliOS - a software hub, which proved very helpful to us.

The Linux Documentation Project

4 Architecture Description[2]



A representation of the software structure.

🟥 - See Supplemental, Communication link protocol.

4.1 General Structure

4.1.1 User Interface

The user interface works in user mode and handles as many of the tasks that do not demand "real time" processing as possible. The purpose of the user-interface is to compile the syntax description and pass it to the packet handler.

[2] We used a perl coded construct to evaluate our method of implementation

The compiler is split into two passes: the parser and the translator. The parser's objective is to parse the syntax file which is edited by the user and to form a hierarchal perl data structure that describes it 1-by-1.

The translator's purpose is to feed this data structure into the arbitrator by serializing it.

4.1.1.1 The Parser

The Parser uses a class named IP::Noise::Text::Stream::In to parse the files line by line and to strip comments. It is not a very sophisticated class, but it was adequate for this purpose.

The parser itself is written as one monolithic class (IP::Noise::C::Parser) where every data type has its own parsing function. In this stage, we have allowed ourselves to throw exceptions freely. Refer to Section 5.2.6.

4.1.1.2 The Translator

The translator uses the class IP::Noise::Conn or IP::Noise::Conn::Ker (pardon the pun) to communicate with the arbitrator. The choice of the class is determined by whether he is communicating with the kernel or with the user-land arbitrator. In any case, the translator's constructor is given the class reference as a parameter, so everything works transparently.

The translator contains a map called %transactions, which maps the transactions' names to their opcodes, parameters and output parameters. The function transact is used to perform a transaction with the specified arguments and to return the return value as well as the input codes. For more information consult Section 6.1 – the protocol description.

The function load_arbitrator() is a monolithic function, which is used to load the arbitrator. It performs various operations, one by one, while sometimes relying on the return results of the previous transactions.

In this code we have also used exception throwing to abruptly end a connection.

4.1.2 Packet Handler

The packet handler has three parts: a receiver, an arbitrator and a releaser.

The Receiver: its object is to receive new packets from the IP-Queue mechanism (or its equivalent in kernel mode), stamp them with the time in which they were arrived and pass them to the arbitrator.

The Arbitrator: receives packets from the receiver, decides whether to drop, release or delay them. If they are dropped or released, it does so immediately. Else, it puts the packets in a priority queue for the release to release.

The Releaser: Polls the priority queue in order to release the delayed packets on time.

In the user-mode application, each one of them is implemented in its own thread. However, in the kernel, everything is implemented using call-backs and timers. (see section 5.2.3 - “Why we did not use kernel threads?”)

4.1.2.1 The Receiver

The receiver plugs into the IP-Queue mechanism. It receives packets from it, stamps them with the time of their arrival and places them in a queue. The queue is a thread safe queue that was implemented in the module queue.c.

The receiver thread can be found at the end of the main() function in the main.c module (for the user-level arbitrator) or in the function ipq_enqueue in the ip_queue.c module (for the kernel-mode arbitrator). Note that, in kernel mode the receiver does not place the packets in a queue, but rather calls the packet logic verdict determination function directly to find its verdict.

4.1.2.2 The Releaser

In user-mode, the releaser is modeled around an infinite loop, in which a thread waits for the time of the release of the element with the minimal time to arrive. If an element with a lower time has arrived, the thread is woken up prematurely so it can re-schedule its next awakening.

In kernel-mode, the releaser has a timer function, which is manipulated in much the same manner using the Linux kernel timer functions.

One can find the releaser's code in the module delayer.c. The priority queue that is used by it can be found in pqueue.c and is based on the code by Justin H. Jones.

4.1.2.3 The Arbitrator:

The arbitrator is split into three logical parts:

1. The User Interface Back-end: this interacts with the user-level compiler and is used to modify the behavior of the arbitrator.

2. Switcher: the purpose of this part is to switch the chains to their new states.

3. Packet Logic: this part does the actual decision of what should be done with a packet.

This distinction is also kept in the kernel module version of the packet handler.

Note that the data structure used by the arbitrator for holding the information of the chains and their states needs to be synchronized to be kept from multiple simultaneous accesses. Not only that, but the interface back-end actually duplicates the data, and works on a copy, in order not to block the other tasks.

4.1.2.3.1 The User-interface Back-end

Since data written to the back-end is not necessarily received at once in the case of a kernel module, we have performed the communication with the compiler using transactions. The data that arrives into the device or named pipe is cached. The back-end attempts to read data from this cache. If it cannot read enough data to complete the transaction, it rollbacks the transaction. Else, it commits it, which causes it to be flushed.

The steps in evaluating a transaction are:

1. Receiving the opcode and according to it retrieving the record that corresponds to it.

2. From that record, the parameter types of the opcode's parameters are deduced and each parameter is received from the line in its turn.

3. The record's handler is called with all the parameters. It performs the actual operation on the data's copy.

4. One record handler reads other data from the line: set_move_probs().We could not think of a more elegant way to resolve its needs.

After the connection had been closed, the interface replaces the data with its data copy and waits for another connection to be opened.

The back-end can be found in the modules iface.c and iface_handlers.c of the source code.

4.1.2.3.2 The Switcher

The switcher is implemented using a priority queue in user-mode, and using one timer for each chain in kernel mode. Testing the priority queue every 50 milliseconds does polling on the priority queue in user-land.

The switcher can be found in the module switcher.c

4.1.2.3.3 The Packet Logic

The packet logic refers to the data in order to deduce what is the verdict of the packet. The function get_packet_info() retrieves packet information from the packet's headers. Based on it, is_in_chain_filter() decides whether the packet enters the chain or not.

The function chain_decide() decides what is the verdict for each chain, while the function decide accumulates their decisions.

The function that serves as the interface for the rest of the code is ip_noise_arbitrator_packet_logic_decide_what_to_do_with_packet().

4.2 Implementation Details

4.2.1 Coding Conventions

4.2.1.1 Identifiers

We used lowercase names for all of our identifiers. The words or components are separated by underscores.

All exportable symbols start with "ip_noise" in order to avoid namespace collition. Following the ip_noise, comes the class name. E.g: "rand", "arbitrator_iface", "delayer" etc. The last component is the name of the method.

Most classes have an alloc method that allocates a new object, and a destroy method that destroys it.

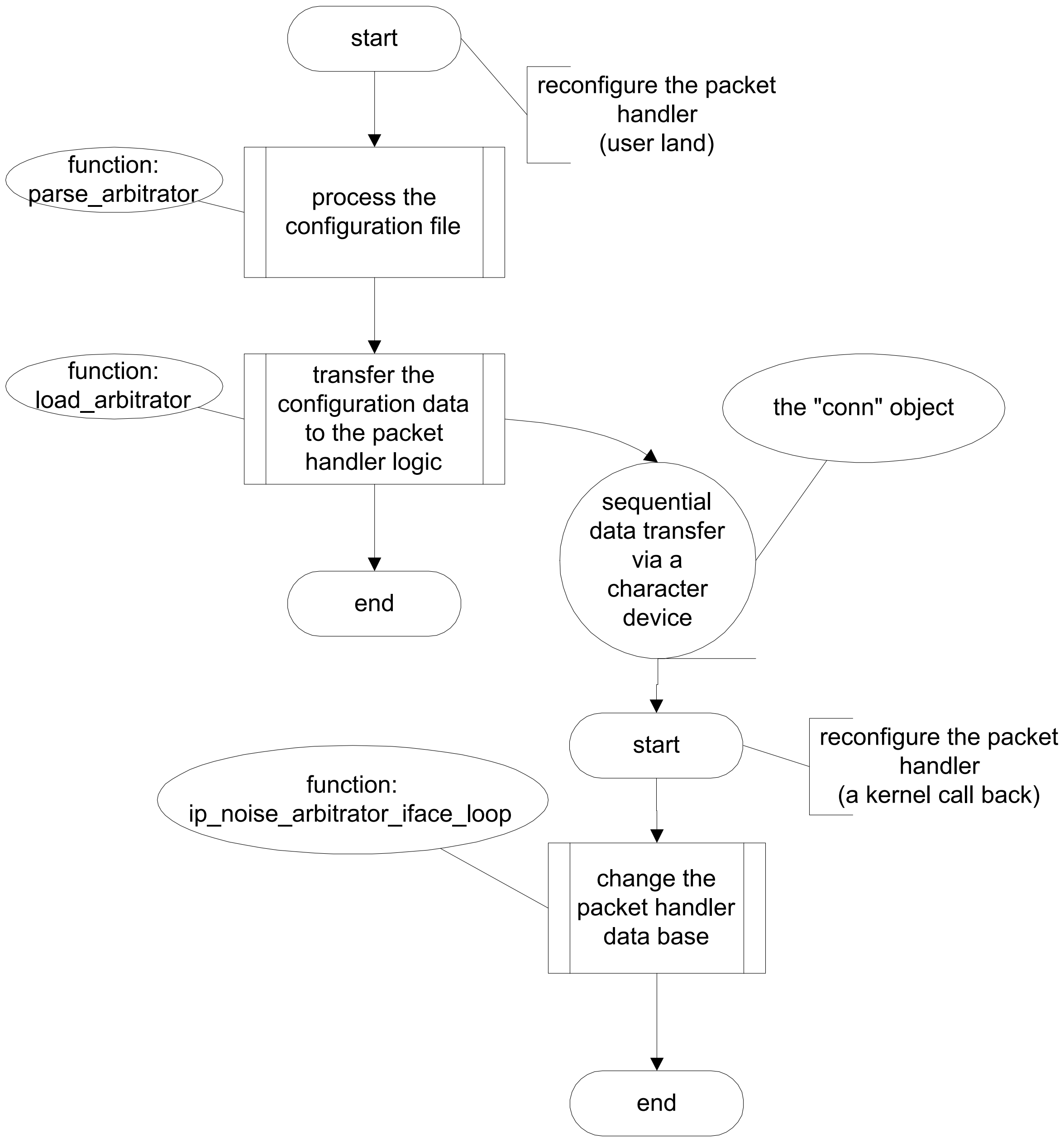
4.2.1.2 Typedefs

Typedefs also start with ip_noise, but end with a "_t" suffix. They are usually declared as "struct ip_noise_foo_struct" where foo is the data type being declared.

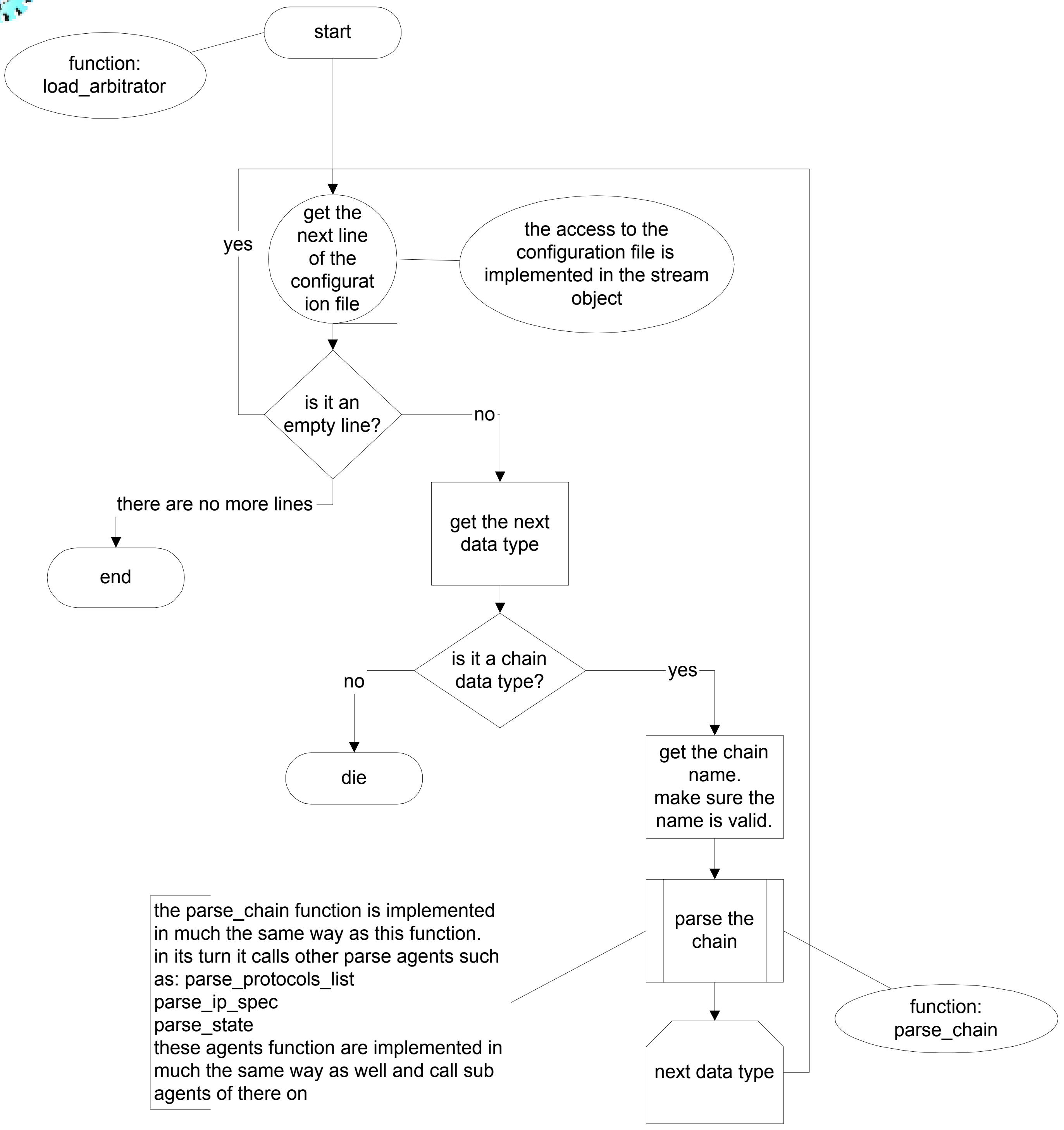
4.2.1.3 Common Abbreviations

- alloc – allocate.
- bsearch - binary search.
- cmp – comparison.
- com – comulative.
- ip - IP Address.
- len – length.
- pq - priority queue.
- prob – probability.
- ptr – pointer.
- rand - random, randomizer etc.
- ret - return value.
- tos - type of service.
- tv - timeval (an internal C struct that specifies the time in seconds and microseconds).
- w – with.

4.2.2 Arbitrator Re-configuration

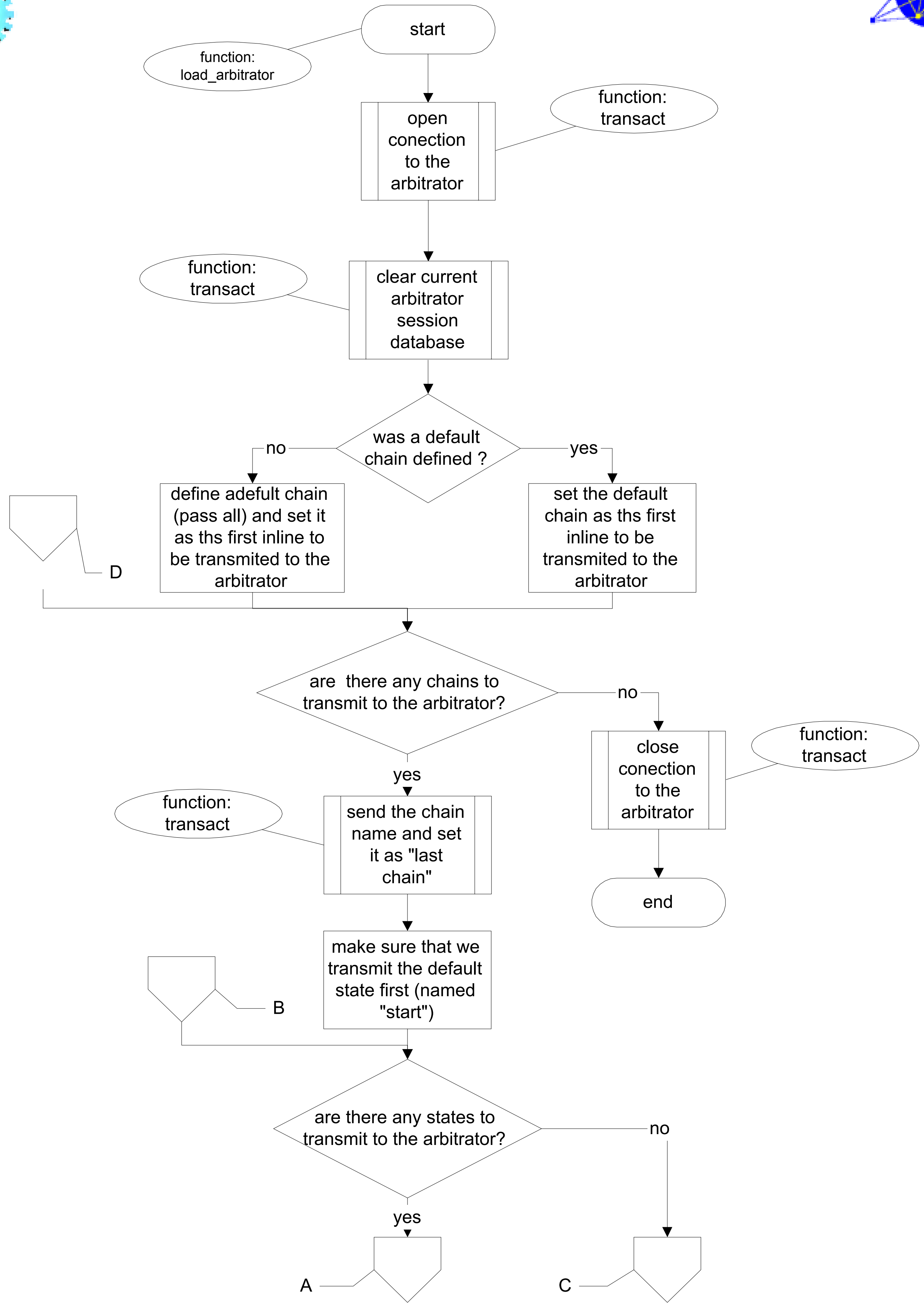

start
reconfigure the packet handler (user land)
function: parse_arbitrator
process the configuration file
function: load_arbitrator
transfer the configuration data to the packet handler logic
the "conn" object
sequential data transfer via a character device
end
start
reconfigure the packet handler (a kernel call back)
function: ip_noise_arbitrator_iface_loop
change the packet handler data base
end

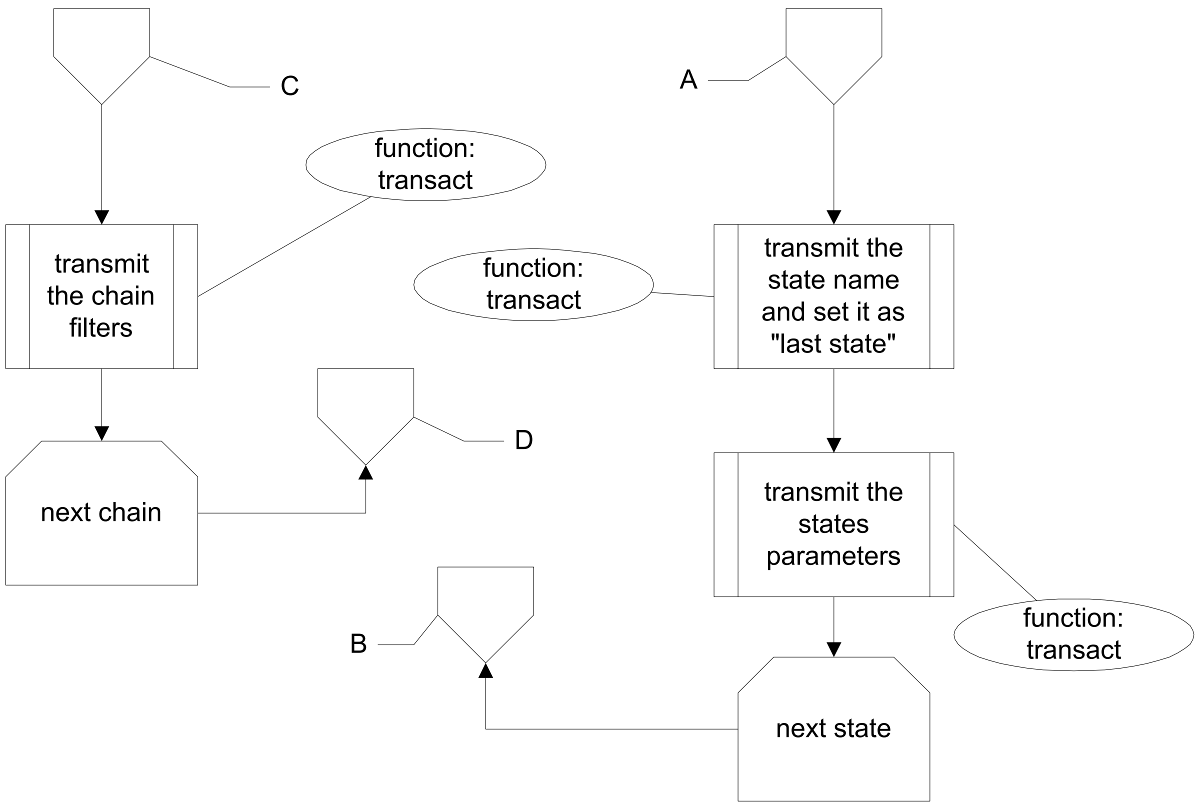
This flowchart describe the configuration process of the arbitrator


start
function: load_arbitrator
yes
get the next line of the configuration file
the access to the configuration file is implemented in the stream object
is it an empty line?
no
there are no more lines
end
get the next data type
is it a chain data type?
no
yes
die
get the chain name. make sure the name is valid.
parse the chain
function: parse_chain
next data type

the parse_chain function is implemented in much the same way as this function.
in its turn it calls other parse agents such as: parse_protocols_list
parse_ip_spec
parse_state
these agents function are implemented in much the same way as well and call sub agents of there on

This flowchart describes how we process the configuration file

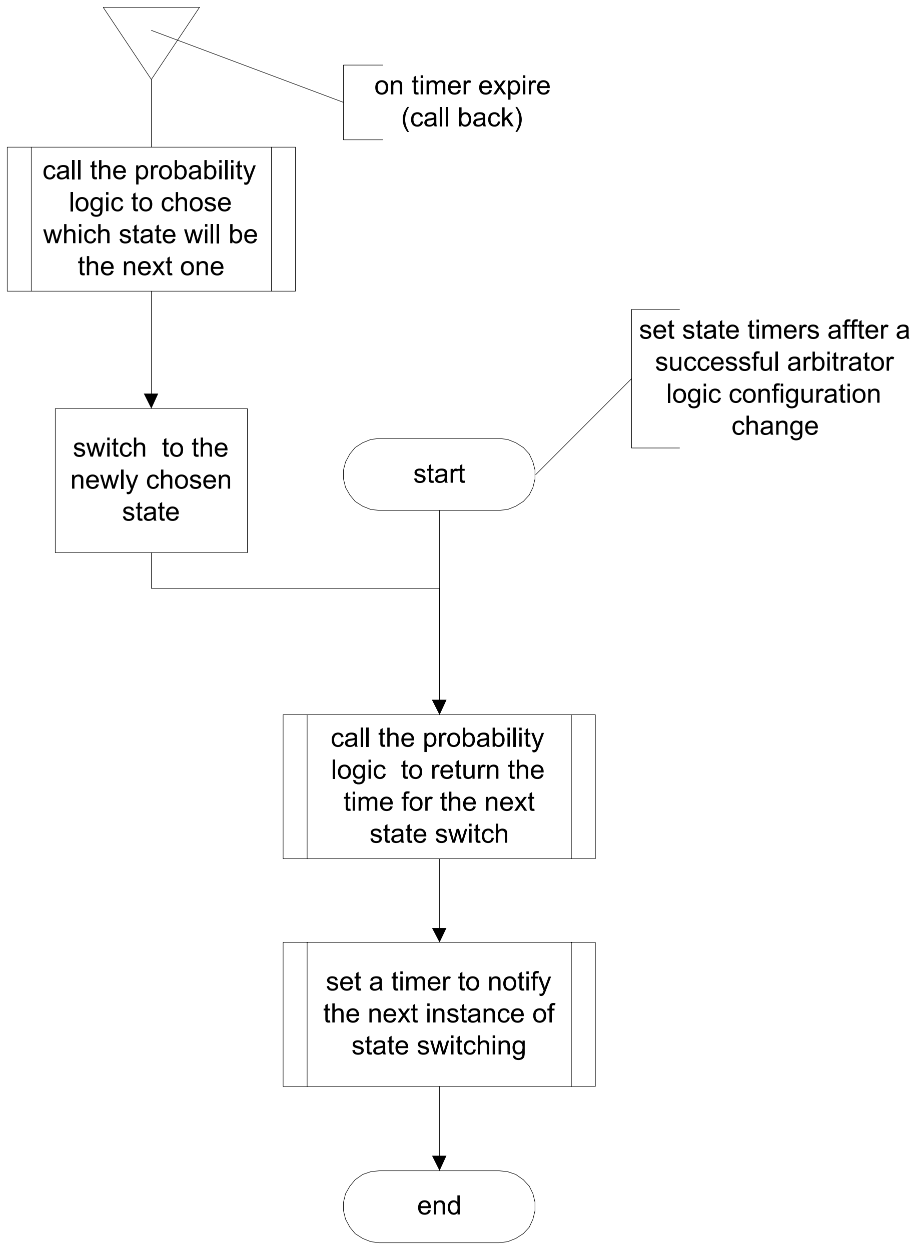

start
function: load_arbitrator
open conection to the arbitrator
function: transact
clear current arbitrator session database
function: transact
was a default chain defined ?
no
yes
define adefult chain (pass all) and set it as ths first inline to be transmited to the arbitrator
set the default chain as ths first inline to be transmited to the arbitrator
D
are there any chains to transmit to the arbitrator?
no
close conection to the arbitrator
function: transact
end
yes
send the chain name and set it as "last chain"
function: transact
make sure that we transmit the default state first (named "start")
B
are there any states to transmit to the arbitrator?
no
yes
A
C


C
A
function: transact
transmit the chain filters
function: transact
transmit the state name and set it as "last state"
D
next chain
transmit the states parameters
function: transact
B
next state

Transmission of the Configuration Database to the Arbitrator

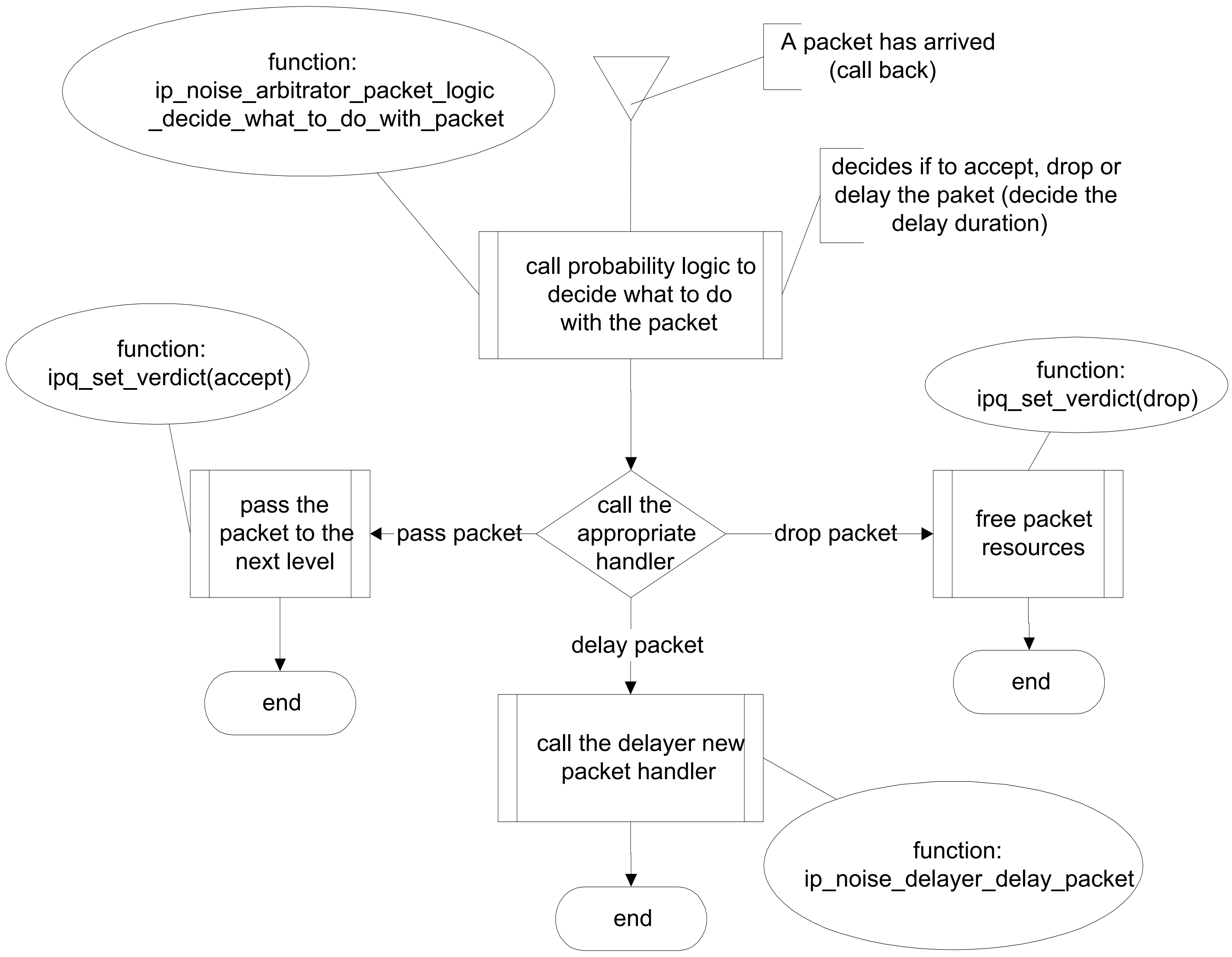
Receiving the information by the arbitrator is achieved by the use of similar functions that receive data from the connection. Instead of transmitting them, they store the data in the computer memory.

4.2.3 Changing a chain's state


on timer expire
(call back)
call the probability logic to chose which state will be the next one
switch to the newly chosen state
set state timers affter a successful arbitrator logic configuration change
start
call the probability logic to return the time for the next state switch
set a timer to notify the next instance of state switching
end

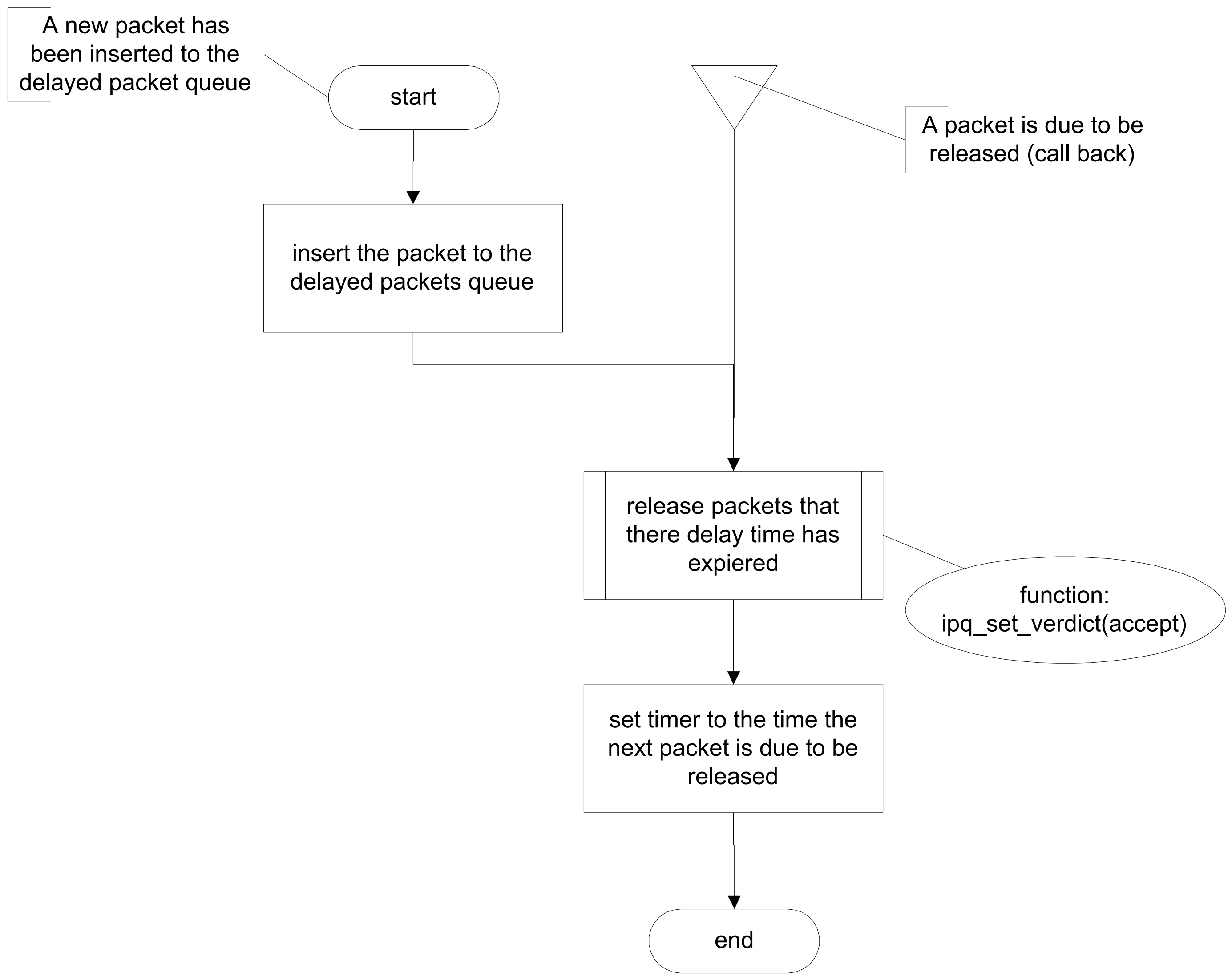
Changing a Chain's State

4.2.4 Deciding what to do with a packet



Deciding what to do with a Packet

4.2.5 Delaying the arrival of a packet


A new packet has been inserted to the delayed packet queue
start
insert the packet to the delayed packets queue
A packet is due to be released (call back)
release packets that there delay time has expiered
function: ipq_set_verdict(accept)
set timer to the time the next packet is due to be released
end

Delaying the Arrival of a Packet

5 Conclusions

5.1 Tools that were used

5.1.1 VNC

VNC, which stands for "Virtual Networking Computer" is similar to remote X. However, it has two very important advantages:

1. It retains the same session between two consecutive logins.
2. Its Win32 viewer is only one file, and does not require administrator permissions to run.

During our first days, when we were given a very low quality screen, we set up a VNC server on our Linux workstation, a viewer on a Windows 2000 machine, and used the Linux workstation over the network.

VNC is not without its flaws, however. For instance, using a tiled image wall-paper resulted in a very slow rendering of the screen.

The VNC home page is:
http://www.uk.research.att.com/vnc/

5.1.2 Perl

Perl is a programming language commonly used for writing server-side Web applications and for many other system administration tasks. We found it useful to program both the compiler and the initial version of the arbitrator in Perl, because a Perl code takes a fraction of time to write, in comparison to the equivalent C code.

Furthermore, Perl is similar enough to C (if you don't abuse its more powerful features) that we were able to benefit from having a working Perl code to base upon.

We would recommend anyone, who can afford to, to write the initial code base in Perl (or a similar language such as Python or Ruby), before converting it to C.

We plan to keep the compiler in Perl, because Perl is much better suited for tasks like that, because it is available on most UNIX systems, and because the compiler is not a real-time application.

The Perl home page is:

http://www.perl.com/

5.1.3 User-Mode Linux

User-Mode Linux is a version of the Linux kernel that runs as a process on top of Linux. It gives one a full-fledged (albeit slow) Linux system that can be run as a user, even without any SUID games.

Kernel modules that the user writes can be loaded and debugged using gdb, and if there is a bug inside one of them, it only crashes the process, while the system remains intact.

The User-Mode Linux home page is:
http://user-mode-linux.sourceforge.net/

5.1.4 CVS

CVS stands for Concurrent Versioning System. It is a tool to maintain several revisions of a source code directory tree. By using BerliOS' remote CVS server (see below), we worked against CVS and found it extremely helpful.

One of the advantages of using CVS is that we were able to grab the most up-to-date version of the code from the Net, with a few command-line commands. Not only that, but we were able to grab or view older versions, and compare between them.

There is a software project under works called Subversion that will eventually offer an even better alternative to CVS. Both CVS and Subversion are open source software.

The CVS home page is:
http://www.cvshome.org/

5.1.5 BerliOS

BerliOS is a software hub that is physically located in Germany. We did not use all of the services that BerliOS offered us, but we made extensive used of the fact that it gave us a CVS repository.

At the time of this writing there are two similar services to BerliOS on the Web: SourceForge and GNU Savannah. All of them offer a good service for hosting open-source software.

The BerliOS home page is:
http://developer.berlios.de/

5.2 C and Kernel Programming Conclusions

5.2.1 jiffies instead of gettimeofday Function

When programming in kernel mode it would make a vast speed improvement to use jiffies instead of gettimeofday(). jiffies is an integer that represents the internal time since the machine's startup. It is measured in units of 1/HZ seconds, where HZ is an architecture-specific constant.

gettimeofday() requires some extra calculations, but retrieving the current jiffies does not even requires a function call. Furthermore, timers are set according by jiffies-time, so it saves the overhead of converting from jiffies to struct timeval and vice versa.

5.2.2 Don't have too many Kernel Timers

During the initial version of the kernel module we allocated one timer for each packet that was to be delayed. That turned out to be a wonderful way to crash the kernel.

We strongly recommend to avoid this practice, and to try to allocate one timer at a time, so that each timer handler will re-allocate the timer.

(Note that we knew we would eventually have to do it the other way, but one should know in advance that the kernel cannot handle too many active timers)

5.2.3 Why we did not use Kernel Threads

We did not use kernel threads for several reasons, despite the fact that kernel 2.4 supports them well:

1. Most people try to avoid using them inside their kernel modules.

2. The Netfilter architecture parallelizes handling the incoming packets for us, assuming it is an SMP machine.

3. It would be faster and have fewer overheads to write the code asynchronously rather than having several threads running inside the kernel.

5.2.4 Excessive kmalloc is your Enemy

If possible use vectors of structs instead of vectors to pointers to structs. I.e.:

```
struct mystruct myarray[100];
struct mystruct * myarray;
```

That is, assuming mystruct is relatively small. Otherwise, it would probably be a better idea to use kmalloc() or your own data allocation system with fixed records.

5.2.5 The Internet is your Friend

We found a lot of useful information about programming in kernel-mode, either by consulting people of various Linux-related mailing lists or by searching the web using Google and other resources.

We encourage everyone who reads this document to make liberal use of the Internet to get help with his project.

5.2.6 Exceptions should be Used with Care

Exceptions are Evil (with a capital "E") in C++, much less in C. We decided not to use them due to the fact that C does not have garbage collection and because it may mean our code will be killed due to an exception that was not caught.

In Perl, however, which supports exceptions and has garbage collection, we used exceptions extensively. We did not use them in the arbitrator and the translator, but it made our life when writing the parser easier, because we did not have to pass an error code.

We strongly recommend C++ programmers not to use exceptions in their code. But programmers of Perl and similar languages may use them with care.

6 Supplemental

6.1 Protocol Description

The communication protocol will use two one-way channels.

State - If 0 is transmitted - <int32> - index
If 1 is transmitted - <string> - its name
<string> is string_len+1 bytes that terminate with a NULL character.
string_len should be 127 bytes or so.

Chain - If 0 is transmitted - <int32> - index
If 1 - <string> - its name

Return Values - 0 - OK
1. Wrong Input

Packet Range - start=<int16> end=<int16>
If start > end then it's a terminator.

Packet Ranges Specifier - <Packet Range><Packet Range> .... <Terminator>

IP Component - ip=<int32>
netmask_width=<int32>
inverse=<bool>
<Packet Ranges Specifier>

IP Component Terminator - ip=255.255.255.255 but there is still a netmask and a packet range specifier for parity.

IP Filter - <IP Component> <IP Component> <IP Component> <Terminator>

Split Linear Specifier Token:
prob=<Probability Type>
delay=<Delay Type>

terminator (last element) is when prob=1.

Split Linear Specifier:

<Split Linear Token> <Split Linear Token> ... <Terminator>

Delay Type - any of:

1. Exponential   0x0
2. Split Linear 0x1

Protocol Operations:

*Define a new chain.

```
Input: <0x0000> chain_name=<string>
Output : <RetValue> chain_number=<int32>
```

Note: even if RetValue indicates failure we still broadcast the chain number, which should be ignored by the front-end.

*Define a new state in a chain.

```
Input: <0x0001> chain=<Chain> state_name=<State>
Output : <RetValue> state_number=<int32>
```

*Define the move probabilities between states @source to states @dest.
(by using a matrix of inputs) inside a chain.

```
Input: <0x0002>
            chain=<Chain>
            scalar(@source)=<int32>
            scalar(@dest)=<int32>
$             source[0]=<State>
$             source[1]=<State>
.
.
$             dest[0]=<State>
$             dest[1]=<State>
$             dest[2]=<State>
            probs($s -> $d)=<Probability Type>
(scalar(@source)*scalar(@dest) of  them)

Return: <RetValue>
```

*Chain Filters:

- Set source to something.
  Input: <0x0003>
  chain=<Chain>
  filter=<IP Packet Filter>
  Output:
  <RetValue>

- Set dest to something
  Input: <0x0004>
  chain=<Chain>
  filter=<IP Packet Filter>
  Output:
  <RetValue>

- Enable/Disable Protocol X.
  Input: <0x0005>
  chain=<Chain>
  protocol_id=<int32>
- If = 256 then all protocols should be enabled or disabled enable_or_disable=<bool>
  Output:
  <RetValue>

- Set TOS Precedence Bits
  Input: <0x0006>
  chain=<Chain>
  Precedence = <int16>
  Output:
  <RetValue>

- Set TOS bits mask
  Input: <0x0007>
  chain=<Chain>
  tos_bits_mask=<int16>
  Output:
  <RetValue>

- Set Min. packet length
  Input: <0x0008>
  chain=<Chain>
  Min_Packet_Length=<int16>

Output:
    <RetValue>

– Set Max. packet length
    Input: <0x0009>
        chain=<Chain>
        Max_Packet_Length=<int16>
    Output:
        <RetValue>

– Set Which packet length
    Input: <0x000a>
        chain=<Chain>
        Which_Packet-Length=<int16>
    Output:
        <RetValue>

Can be - 0 - don't care.

1. Greater than min
2. Lower than max
3. Between min and max
4. Not between min and max.

*Retrieve the index of the chain (by name).
    Input: <0x000b>
        chain=<Chain>

    Output:
        <RetValue>
        chain#=<int32>

*Retrieve the index of a state within a chain (by name).
    Input: <0x000c>
        chain=<Chain>
    Output:
        <RetValue>
        state_index=<int32>

*Retrieve the move probabilities between @source and @dest.
    Input: <0x000d>
        chain=<Chain>
        scalar(@source)=<int32>
        scalar(@dest)=<int32>
$ source[0]=<State>

```
$                          source[1]=<State>
.
.
$                          dest[0]=<State>
$                          dest[1]=<State>
$                          dest[2]=<State>
              Output:
                        <RetValue>
                        prob[$s][$d]=<Probability Type> for each source
and dest.
```

*Set the drop/delay prob of a state.

```
Input: <0x000e>
          chain=<Chain>
          state=<State>
          drop_prob=<Probability Type>
          delay_prob=<Probability Type>
Output:
      <RetValue>
```

*Set the delay function type of a state.

```
Input: <0x000f>
          chain=<Chain>
          state=<State>
          delay_type=<Delay Type>
Output:
      <RetValue>
```

*Set the split-linear delay function of a state.

```
Input: <0x0010>
          chain=<Chain>
          state=<State>
          split_linear=<Split Linear Spec>
Output:
      <RetValue>
```

*Set the delay function's lambda. (in case it is Exponential).

```
Input: <0x0011>
          chain=<Chain>
          state=<State>
          lambda=<lambda_type>
Output:
      <RetValue>
```

*Set the time factor of a state.

    Input: <0x00013>
        chain=<Chain>
        state=<State>
        time_factor=<Delay Type>
    Output:
        <RetValue>

*Set the stable delay probability of a state. (we mean that packets are sent in the order in which they were received)

    Input: <0x0014>
        chain=<Chain>
        state=<State>
        prob=<Probability Type>
    Output:
        <RetValue>

*Delete a state from a chain.

    Input: <0x0015>
        chain=<Chain>
        state=<State>
    Output:
        <RetValue>

*Delete an entire chain.

    Input: <0x0016>
        chain=<Chain>
    Output:
        <RetValue>

*Set a chain's current state.

    Input: <0x0017>
        chain=<Chain>
        state=<State>
    Output:
        <RetValue>

*Dump all the information of all the chains.

    Input: <0x0018>
    Output :

```
                    <RetValue>
                    num_chains=<int32>
                    chains=@<Output_Chain> where
                        Output_Chain=
                            name=<string>
                            current_state=<int32>
                            time_of_last_packet=<Time Type> where
                                Time Type=
                                    sec=<int32>
                                    usec=<int32>
                            filter_protocols=<Bitmask of 32 bytes>
                            filter_tos_precedence=<int16>
                            filter_tos_bits=<int16>
                            filter_min_packet_len=<int16>
                            filter_max_packet_len=<int16>
                            filter_which_packet_len=<int16>
                            filter_source_ip_spec=<IP Specification>
                            filter_dest_ip_spec=<IP Specification>
                            states_num=<int32>
                            states=@<Output_State> where

                                Output_State=
                                    drop_prob=<Probability Type>
                                    delay_prob=<Probability
Type>
                                    delay_type=<Delay Function
Type>
                                    delay_type_param= Depends
on delay_type:
                                        If Exponential=

 lambda=<lambda_type>
                                        If Split-linear=

 split_linear_spec=<Split Linear Spec>
                                    time_factor=<Delay Type>

 stable_delay_prob=<Probability Type>
```

6.2 A sample configuration file

```
# By default - pass the packet.
Chain Default {
   State A {
      move_to = {
         1 = A
      }
      drop = 0
      delay = 0
      time_factor = 100000
   }
}

# The purpose of this chain is to simulate a situation where the host
# "t2" is normally fine, but there are sometimes periods when it becomes
# noisy and delays packets.
Chain T2 {
   State Normal {
      time_factor = 5000
      move_to = {
         0.8 = Normal
         0.2 = Noisy
      }
   }
   State Noisy {
      move_to = {
         0.8 = Noisy
         0.2 = Normal
      }
      delay = 0.3
      drop = 0.3

      delay_type = U(100,3000)

      time_factor = 20000

      stable_delay = 0
   }

   # These are the IPs of t2.technion.ac.il and the local host
   source = 132.68.7.4 ; 132.68.52.118
   dest = 132.68.7.4 ; 132.68.52.118
}
```

```
Chain TX {
   State Normal {
      time_factor = 5000
      move_to = {
         0.8 = Normal
         0.2 = Noisy
      }
   }
   State Noisy {
      move_to = {
         0.8 = Noisy
         0.2 = Normal
      }
      delay = 0.3
      drop = 0.3

      delay_type = U(100,3000)

      time_factor = 20000

      stable_delay = 0
   }

   # These are the IPs of t2.technion.ac.il and the local host
   source = 132.68.1.28 ; 132.68.52.118
   dest = 132.68.1.28 ; 132.68.52.118
}

Chain ElecEng {
   State Normal {
      time_factor = 5000
      move_to = {
         0.9 = Normal
         0.1 = SlightLoad
      }
      drop = 0.1
   }
   State SlightLoad {
      time_factor = 5000
      move_to = {
         0.7 = SlightLoad
         0.2 = HighLoad
         0.1 = Normal
      }
```

```
        drop = 0.2
        delay = 0.2

        delay_type = E(500)

        stable_delay = 1
    }
    State HighLoad {
        time_factor = 2000
        move_to = {
            0.8 = HighLoad
            0.2 = Normal
        }
        drop = 0.5
        delay = 0.3

        delay_type = U(2000,4000)

        stable_delay = 1
    }

    source = 132.68.52.0/8
}

Chain Technion {
    State Normal {
        time_factor = 5000
        move_to = {
            0.9 = Normal
            0.1 = Noisy
        }
        delay = 0.1

        delay_type = U(0,500)

    }

    State Noisy {
        time_factor = 10000
        move_to = {
            0.7 = Noisy
            0.3 = Normal
        }
        delay = 0.3
        drop = 0.2
```

```
            delay_type = Generic {
                0 = 0
                0.5 = 1000
                1 = 5000
            }

            stable_delay = 0.5
        }

        source = 132.68.0.0/16
        dest = 132.68.0.0/16
    }
```