netraf

A Network Analyzer and Traffic Logger

http://netrafd.sourceforge.net/

The Project

***netraf** is a C written, POSIX threads specification compatible set of programs, allowing to monitor all network traffic routed through the machine it is installed on, helping administrators in bandwidth shaping and load profiling. Flexible and easy configuration of logging options, statistics and periodical summaries.*

***netraf** consist of **netrafg**, **netrafd** and **netrafl**.*

Table of Contents

1. Authors 7
2. Introduction 8
3. Project Goals 10
4. Theory 11
netraf Operation Diagram 11
netraf Shared Memory Model 12
netrafd Working Scheme 17
netrafl Working Scheme 19
"MYCAP" Packet Capture Library 21
netraf Configuration Files Syntax 23
5. Project ChangeLog 25
6. Program Documentations 30
7. Issues 30
netrafd Issues 31
netrafg Issues 32
netrafl Issues 33
8. Literature 34

1. Authors

Mateusz Styrczula – *dr4k@users.sourceforge.net*

- whole **netraf** idea and project management,
- **netrafg** (ncurses, pthreads GUI),
- shared memory model and it's implementation,
- code chunks,
- **netrafg** documentation and screenshots,
- **netraf** Operation Diagram
  and **netraf** Shared Memory Model articles,
- **netraf** project page,
- printed documentation preparation,

Michał Kraszewski – *regiss@users.sourceforge.net*

- **netrafd** packet capturing daemon,
- MYCAP packets capturing library,
- misc **netraf** utility-helpers (bash scripts),
- code chunks,
- **netrafd** documentation (man pages),
- **netrafd** Working Scheme
  and MYCAP Packet Capture Library articles,

Tomasz Jędrzejczak – *tomek_j@users.sourceforge.net*

- **netrafl** logging daemon,
- config-files handling library,
- **netrafl** documentation (man pages),
- autoconf scripts,
- code chunks
- **netrafl** Working Scheme
  and **netraf** Configuration Files and Filters Syntax articles,

2. Introduction

About

netraf is a programming project being in realization at Institute of Computer Science at Wrocław University (in Poland) as our License Work. Project's main target is to develop application helping in some aspects of Network Load Profiling.

Background

Wait a minute... There are many applications like that – so why write another? For a simple reason – we couldn't find one which solves following problems:

- let's say we need to know how much data (GB, MB, kB... whatever) are passing particular network interface on our machine monthly. How to do it? We can read that information exported from kernel (`/proc/net/dev`) via `ifconfig`, but linux is storing it in two integer (32 bit) variables (one for RX, one for TX), so after transmitting about 4GB of data via device, this counters will overflow – their contents are useless for us. Of course we have packet counters per device, but every packet has different size... Beside this; what happens when for some reason (i.e. no power for a while) machine needs to be restarted?
- second way is to run *IPTraf* or similar program as a daemon (or in a screen session), and then use one from the bunch of log-analisys script to gather information we need. But this solution (beside it's inelegant nature) has some disadvantages:

  - you can't run any log-analyzing script while logging application is working – log files are empty (at least those with statistics – like that from *IPTraf's* „LAN Station Monitor"), so you have to break logging, make analyze, and start logging again – very bad,
  - if you're doing something like:
    `tcpdump -i eth0 -n -vvv > ./somelog.txt &`
    yes, contents of logfile are accessible immediately, but wait a week and check that file size...,

  - As mentioned above this solution is also not immune to sudden, random machine restarts,

- The same problems we can met if we want to generate monthly (quarterly, yearly...) statistics for certain (or all) machines in a LAN (of course we're talking about gathering statistics on some router/firewall/NAT machine etc...).

Beside of everything; you can of course find some way to survey certain network parameters via log-inspecting scripts – I'm not claiming that it's impossible, but it would be extremely hard to automate.

Challenge

Assume hypothetical situation:
We are spreading internet connection to several users but have some transfer limit. We want to be fair to every user, and we want that everyone have equal chances to enjoy internet resources. But users – like users; one of them are only receiving/sending emails, using chats, reading web-pages etc..., while others are using P2P networks, listening to internet radios, downloading huge ISOs etc.
We must find method to measure every user transfer and restrict him for example only to ICMP echo request/answers in case he exceed his limit (e.g. `user_lmit = global_limit / number_of_users`). Of course we're not talking about buying an dedicated, expensive hardware solution for this.

Solution

Of course – **netraf**!! We can assign rule to each machine's MAC address which defines maximal transfer per some period and define action what to do when rule's limit is exceeded (it could be, for example script running `tc` or `iptables` with appropriate arguments).

Vision

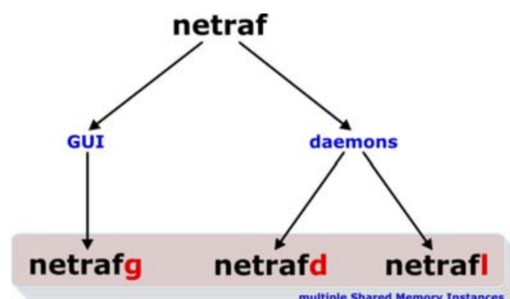
Thus, we could go further; using **netraf** an administrator can notice that while some users exceeds their transfer limits, others don't. With an eye to possibly best and efficient connection usage he can gradually increase transfer limits for first group of users and proportionally decrease for second.

3. Project Goals

- division into three parts:
    - daemon process (**netrafd**) - capturing packets and analyzing them (according to fully-configurable user-defined filters),
    - daemon process (**netrafl**) - logging statistics to files,
    - interactive (ncurses) program (**netrafg**) showing "what's happening" live, and - in some way - remote control for daemon processes described above,
- immune to sudden restarts of machine and electric power shortage,
- ability to log certain network traffic informations (depending on applied filter type):
    - gathering statistics by network interface(s):
        - amount of data (RX and TX) bytes,
        - count of transferred packets and IP packets,
        - count of transferred Broadcast, Multicast and "routed-through" packets,
    - gathering statistics by network MAC addresses:
        - amount of data (RX and TX) bytes per MAC,
        - count of transferred packets and IP packets per MAC,
        - average data rates,
    - gathering statistics by network TCP connections:
        - amount of data (DOWN and UP) bytes per one TCP connection,
        - count of transferred IP packets per connection,
        - average data rates for choosen connection,
    - gathering statistics by network IP addresses:
        - amount of data (IN and OUT) bytes per IP address,
        - count of transferred IP packets,
- modularized structure witch allow users to write their own filters and logging rules,
- user-defined filter consist of:
    - source or destination MAC address,
    - source or destination IP address,
    - source or destination TCP/UDP port,
    - network interface to listen on,
    - perl-compatible regular expression to be matched to packet data,
- user-defined logging rule consist of:
    - whether captured packets should be logged to file,
    - if above is true, what is the period the log-file should be updated,
    - directory path, where the log-file should be stored,

4. Theory

netraf Operation Diagram



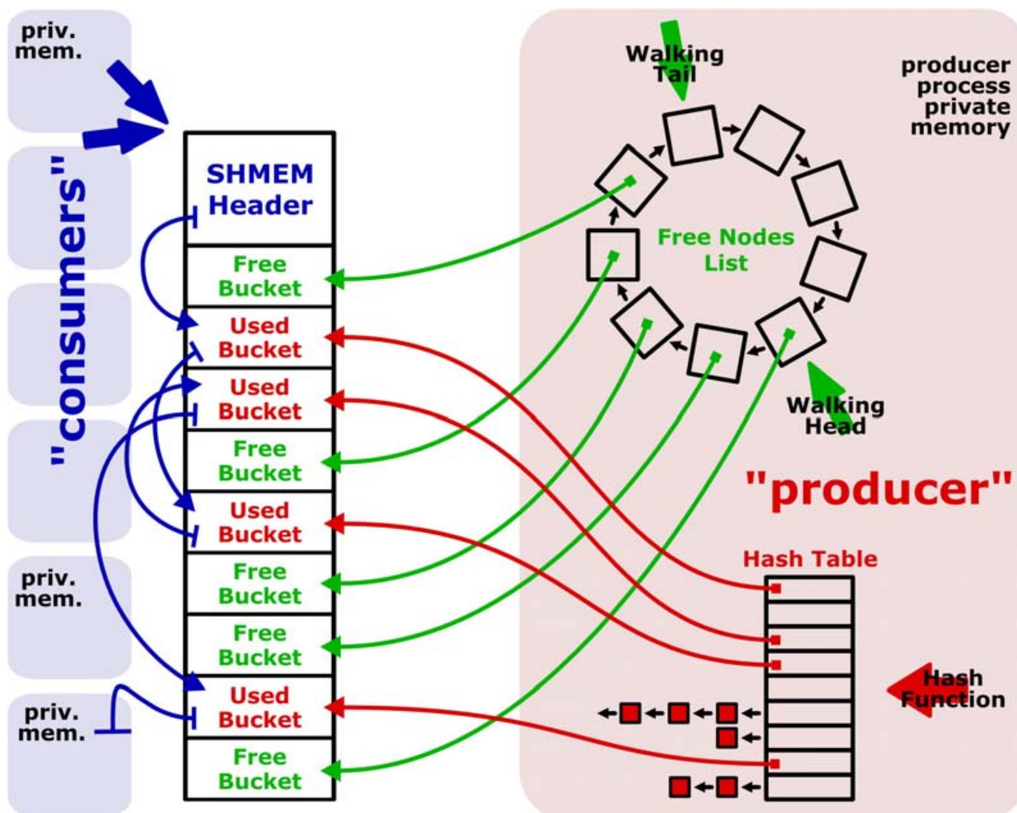
netraf – Network Analyzing Tool – consist of three independent, autonomous programs:

- **netrafg** - Graphical User Interface program. It is "remote control" allowing user to steer particular parameters of daemons described below. It is also the only "window" showing everything live.
- **netrafd** - This is in fact "work horse" of whole project. It is responsible for packet capturing, applying filters to them and generating statistics.
- **netrafl** is a logging daemon. It's only work is writing statistics gathered by **netrafd** to files.

Beside the complexity of every component in **netraf** project, major thing is communication. We have to realize, that every of the three programs are independent processes that have their own memory space allocated by system separately. Also we must take in consideration fact, that simple communication protocol (implemented for example on UNIX sockets) wouldn't be sufficient - **netrafd** is "producing" lot of data that have to be read by **netrafg** as well as by **netrafl**. Thus we need data structure with random access allowing many processes to read at one time and at least one process with a possibility to write. And that is **netraf** Shared Memory Model which meets these assumptions.

When **netrafd** is starting to gather statistics, it is creating one shared memory segment and inform readers about that memory unique identifier. Then every interested reader can "connect to" that memory and read data that are interesting to him. It is important to know, that one working instance of **netrafd** can create multiple shared memory segments - one segment per one type of logging.

netraf Shared Memory Model



Preface

IPC (InterProcess Communication) mechanisms was introduced in UNIX System V.2 by AT&T corp. It consist of:

- semaphores,
- shared memory,
- message queues,

All of above are often called "System V IPC". We're interested in shared memory implementation. For details of use I recommend N. Matthew and R. Stones book - "Beginning Linux Programming". Basically, the idea is quite simple: one program requests for memory block and then gets from operating system two things - pointer to that memory and special unique identifier. Next, every other processes that want to gain in access to shared memory must use that unique identifier to "attach memory" and gets pointer in their own address spaces. Every processes can use shared memory exactly as if it was allocated by `malloc()` function. If one process performs write to shared memory, changes are immediately visible for every other processes having access to it. If work with shared memory is done, it has to be detached by all

processes and deallocated by exactly one. Shared memory is very efficient way of passing data between many non related processes, but have some limits:

- doesn't have it's own synchronization method (thus programmers are using semaphores, or messages sending via pipes to synchronize),
- number of shared memory blocks and total blocks size is limited,

According to above limits (especially number of shared memory blocks limit) we can't treat request for new shmem block and `malloc()` function equally. That's not all - we must remember that every request for memory (it is not important which type) is very expensive. If we want built efficient, dynamic data structure, we have to take care of appropriate management of one continuous data block.

Let's characterize what we want from our data-structure:

- it has to be multi-accessible. More precise: we want to allow one process writing to it, and many (at least two) processes reading it at a same time,
- writing process must have possibly fastest random access to every part of data,
- writing process must have possibility of deleting, changing existing or adding new portions of data,
- adding, searching and deleting data must be as fast as possible,
- every "reading process" must be able to read data in some logical order: from "beginning" to "end",
- it has to be guaranteed that "path" from logical begin of memory to logical "end" will be available in every moment,
- when "reading process" finishes reading memory it have to check if amount of data it reads are equal to data stored in memory (writing process could make some changes during "reading process" work),

Multi - accessibility

Multi-accessibility to our data structure is provided by shared memory IPC mechanism. Must take into consideration, that shared memory is a special addresses range, created by IPC and appearing in processes user spaces. The point is, that physical memory pointers (pointing to the same shared memory segment) peculiarly differs in different processes. So, if we want to "recover" original producer pointer value (it is explained later, for what reason) we have to store it in shared memory. Thus we describe **SHMEM header** structure:

```
/** header of shared memory */
typedef struct shmem_header_t{

   /** based on this pointer every reader
   can calculate offset to his mapped memblock */
   void *producer_mem_point;

   /** pointer to first used bucket */
   void *head;

   /** number of buckets allocated in memory */
   size_t n_buckets;

   /** number of "used" buckets */
   size_t used_buckets;

   /** we don't know what kind of structures will be stored,
   thus we need their size */
   size_t bucket_size;

} shmem_header_t;
```

This kind of structure will be placed at "top" of every shared memory block. Because above type is known at compilation time, every "memory consumer" can cast his own shared memory pointer to this type and gain access to every information stored in header.

Random access

Fast random access can be achieved by Hash Table implementation. Instead of storing "real data", every hash-table bucket is storing pointer to appropriate place in shared memory. Such hash table is created in "producer's" private memory, thus accessible only for him.

Modifying data

When shared memory block is created (let's assume N buckets), the corresponding cyclic list (build of N elements pointing to free buckets in shared memory) is created in producer's private memory. Therefore operating on memory can be described as follows:

- adding new bucket to memory:
  1. pointer to free bucket is taken from cyclic's list "walking head",
  2. "walking head" is moved to next logical position,
  3. appropriate data is copied to shared memory,
  4. every used shared memory bucket contains pointer to next used bucket (or NULL), so appropriate pointers (newly allocated bucket, logically previous bucket and eventually "head" pointer in SHMEM header structure) is set,

5. using hash-function (we're using Phong's congruential linear hash implementation), based on key of actually adding bucket, hash-code is generated,
6. pointer taken from "free nodes list" (in p. 1.) is stored in hash_table,

- searching for bucket (testing if it exists):
    1. based on currently searched bucket key, hash-code is generated,
    2. if in hash table such hash code isn't empty, and keys (searched one and founded in hash table) are equal, bucket exists.

- deleting bucket:
    1. check if bucket exists,
    2. update "pointer to next used bucket" in logically previous bucket in shmem,
    3. add pointer to "free nodes list" to element pointed by "walking tail",
    4. move "walking tail" to next logical position,

As you can see, all above functions boil down to some operations on pointers and calculating hash-codes. In fact, only cost is in copying memory (when adding a new bucket).

Reading data

As mentioned above every used shared memory bucket has pointer to logically next bucket. Every "consumer" that connects to shared memory, can read "head" pointer (to first used bucket) and then pass whole list till NULL... But - little nuance is hiding here. When we was talking about multi-accessibility we were saying that pointer to beginning of shared memory block can have different values in individual processes.

Let's assume that we have two buckets in shared memory. One bucket have pointer called "next" and pointing it to second bucket. We have two independent processes which have pointers to first bucket. First process is "writer" and it created both buckets (especially it sets value of "next" pointer in first bucket). Second process is "reader" and it is trying to read buckets data from shared memory. If it tries to read first bucket - everything will be OK. But then, it tries to read data from

place pointed by "next" pointer of first bucket and it gets segmentation fault. This is because both processes have different address spaces assigned by system. The way getting appropriate pointers from "reader" process is count difference between "writer" and "reader" pointer to first bucket values and then add that difference to every pointer value. That's why we have "producer_mem_point" pointer stored in SHMEM header.

Error immunity

Taking into consideration, that our data structure is multi-accessible (means that many uncommon processes can operate on one portion of data at the same time) we have to prevent deadlocks, races and starvation. Normally we would use semaphores or any other synchronization techniques, but in this case it isn't necessary (or rather plain unwanted). Let's review how our shared memory is used:

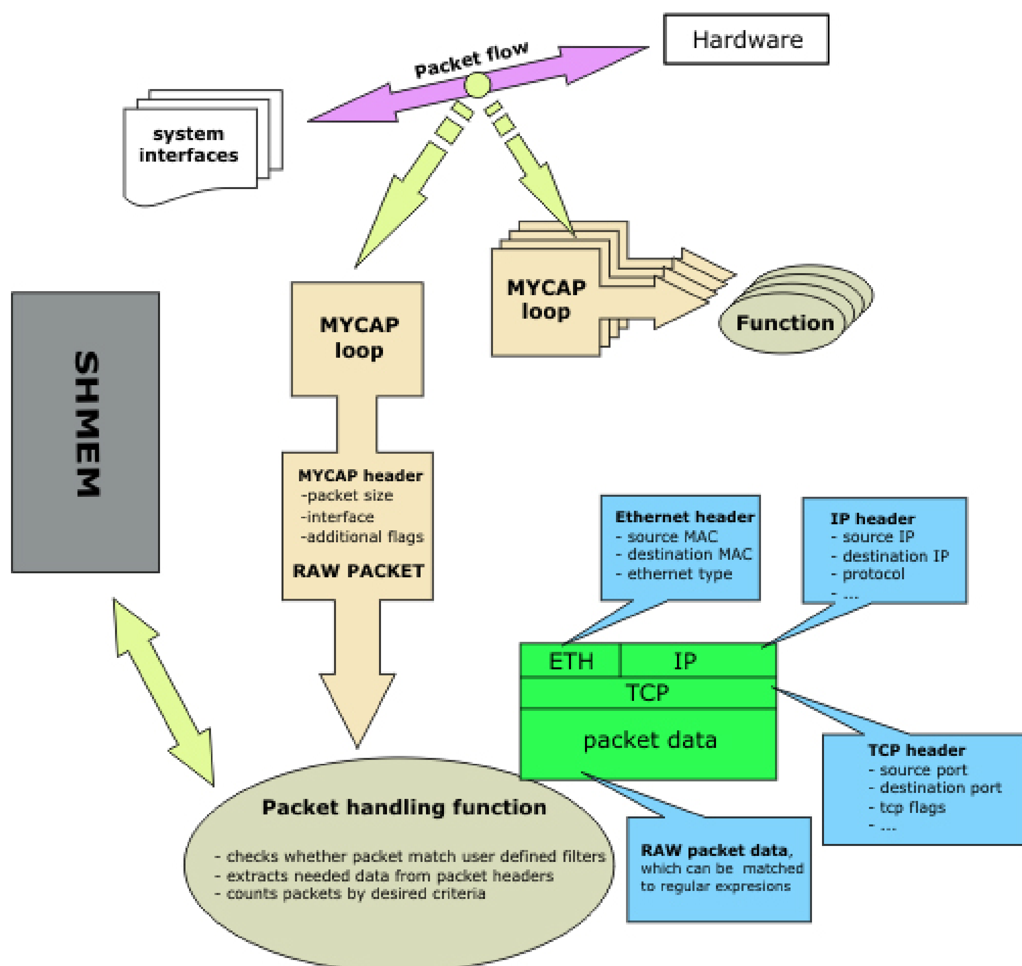
- one memory producer with permission to write (having all "helper" structures described above), doing random memory modifications,
- many (in **netraf** project only two) readers, "walking" through list,

Please notice, that efficiency of **netrafd** daemon strongly depends on shared memory structure random access speed. If we were using semaphore for synchronization purposes, it could happen that one of "reading process" closes semaphore and starts reading. During that time, system's planner can assign CPU time for "writing process". If there were some ethernet packets waiting for processing, "writing process" will try to close semaphore and will hung. What’s more - there could be more "reading processes" waiting, so during that time "writing process" loose many packets (I'm omitting starving situation - IPC semaphores implementation assures that processes waiting in queue to close semaphore will be chosen randomly). It is unacceptable. Our shared data structure must allow "writing process" doing it's job immediately in every time quantum. This entails following problems: what happens if during reading "writing process" changes some data (maybe several times)? After a moment of thinking, we can specify following situations:

- just after reading, it could be more data buckets in memory than "reading process" read,
- it could be less data buckets in memory than "reading process" read,
- number of buckets read and that stored in shared memory structure are equal, but data differs,

Solution of above problems isn't easy in general situation. But in our case it is simple; we must remember, that "reading processes" are renewing their work periodically (in particular **netrafg** renewing reading process form scratch about 10 times per second). So... we've decided that we can put "used_buckets" variable into **SHMEM header** structure and apply comparing rule to every reader: depending on application of reader (some readers can perform reading often - such as **netrafg**, some of them can read memory relatively rarely - such as **netrafd**) there will be some "epsilon" value. If difference between number of read buckets and "used_buckets" value (as for absolute value) will be greater than epsilon, all read buckets are treated as "false" and reading process is started immediately.

netrafd Working Scheme



netrafd - network traffic capturing and analyzing daemon.

The main goal of the **netrafd** is to capture packets from the interfaces installed on the machine, process them and create statistics, dependent on the user defined packet filters.

As you can see on the picture above, the whole idea of the **netrafd** is quite simple. It uses the MY[p]cap library to open many packet capturing threads associated with packet handling functions. The mycap loop waits until a packet arrives and after basic preprocessing passes it to the associated function.

Packet handling functions compare the data from packet headers with applied filter and if the packet match the desired criteria it is further processed. Function extracts needed data from the raw packet and creates or updates an appropriate structure in the shared memory.

And now, a short info about **netrafd** internals. In fact, **netrafd** is a bunch of packet logging threads managed by one 'mother' process. It is the mother process that receives and process signals, read the configuration file and create new (or stop) the logging threads. Every logging thread, during the initialization process, is given it's own shared memory block, filtering rule, and is assigned to one of the packet handling functions. There are few types of packet handling functions, that process the received packets in their own specific way.

User defined filters may consist of source and destination mac addressed, source and destination IP and ports, interface that the logging thread should listen at, and the regular expression that the data transferred by packet should be matched to.

After meting the filter criteria, the data from packet headers is extracted. Every packet is built of layers encapsulated one by another, from the top layers, which create easy to use interface for the user applications, through the network layer to the datalink layer. Every ethernet packet contains an ethernet header, almost the lowest layer protocol, in it's header are defined the hardware (MAC) addresses of the receiver and sender and a field telling what kind of next layer protocol was used. In most cases the next layer protocol will be IP protocol which among many more or less interesting flags will tell us the source and destination IP addresses, also the protocol version, 'time to live' and of course the kind of the fourth layer protocol encapsulated.
The fourth layer is the session layer, and its two main protocols are UDP (connectionless) and TCP. In their headers are defined the source and destination ports, in the TCP header there are many interesting flags used to maintain the connection, by reading those flags we can, for example figure out the current state of the connection.

And finally after the fourth layer protocol header, in most cases, till the end of the captured packet would be the data carried by the packet. Having all this data, the packet handling functions can easily group packets by machines in the network, their IP addresses or the interfaces that the packets came through. When placed on main node in the local network, **netrafd** could easily count the number of computers in the network, or manage any task that has something to do with analyzing network traffic. All it requires is a packet handling function, which will define the way packets are analyzed.

netrafl Working Scheme

netrafl - network logging daemon.

This tool is used to log data gathered by **netrafd** into files. It may be useful if we want to analyze traffic much older than from last server reboot, which allows to define traffic shaping scripts more precisely. **netrafl** working scheme is quite simple. First, it reads his configuration file, where are specified writers to be logged. Then, creates one thread for every writer. Here begins main part: logging and it lasts until interruption. Before **netrafl** closes, it cleans all backup log files and leaves one log file per writer, which actually has been logged.

Now the details. **netrafl** configuration file consist of "globals" section and sections named as writers we want to log. "globals" can have two entries. "files" indicating number of files created by one writer, and "default_period" logging time per file. This means, that writer will be logged every "default_period" / "files" seconds. We can insert entries: "files" and "period" into the writer sections. Their meaning is similar to those from "globals", but they are more important, and simply override them. Let's look at the example:

```
#globals section should always be the first one in the
file
[[globals]]
files = "4";
default_period = "360";
[[]]

[[writer_1]]
[[]]
```

```
[[writer_2]]
files = "6";
[[]]

[[writer_3]]
period = "160";
[[]]

[[writer_4]]
files = "5";
period = "50";
[[]]
```

As we can see, `writer_1` will be logged in 4 files every 360 seconds, `writer_2` in 6 every 360 seconds, `writer_3` in 4 files every 160 seconds, and `writer_4` in 5 files every 50 seconds.

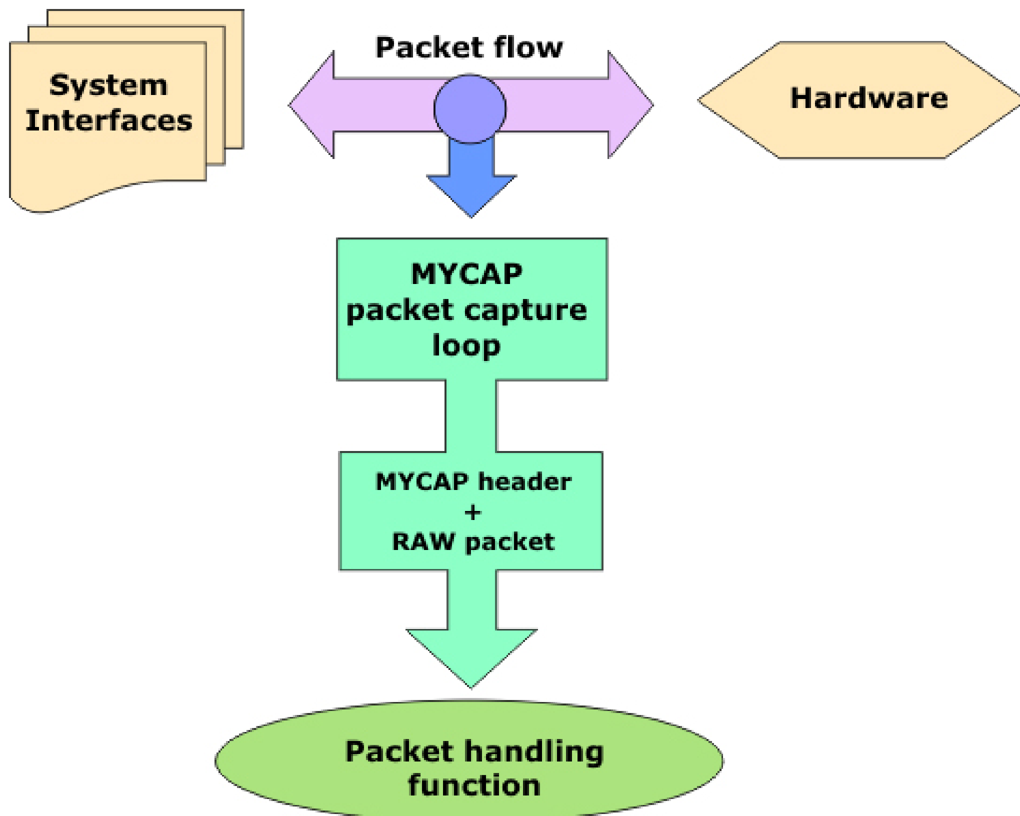
After reading configuration file, **netrafl** prepares log files and creates threads. Every thread waits until his time period elapsed and chooses the oldest backup file, then reads shared memory filled by **netrafd**, and dumps all data into that file.

When **netrafl** is interrupted, it names newest log as writer name with ".log" suffix, and the rest of files is removed. Before **netrafd** starts to capture the packets it can open any log file and fill shared memory with data form it. What happens if there was a blackout, and **netrafl** didn't manage to clean up logs? There is no problem, because **netrafd** uses special functions, which look for the newest backup.

netrafd has also ability to memorize logs of writers which are deleted by **netrafg**. Suppose user has deleted writer 'if_stat'. It will be renamed to 'if_stat_currenttimestamp_.log', instead of 'if_stat.log'. This operation prevents loss of logged data, when user defines new writer named as the old one.

Data in log files have the same syntax as configuration files, which allows to read logs using configuration files functions. There is one limitation to **netrafl**. It can log only ifstat and macstat types, because connstat and ipstat have too many data, so logging it could seriously slow down the machine.

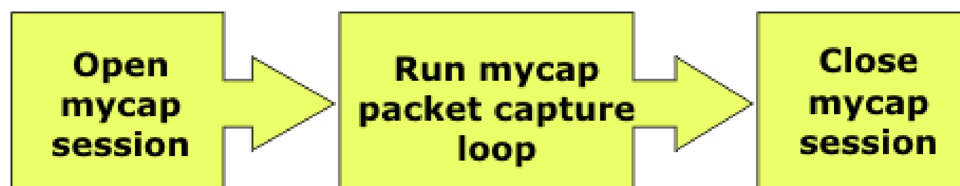
MYCAP Packet Capture Library



MY[p]**CAP** - simple packet capture library...

MYcap is a small set of functions that make packet capturing easy. It allows users to write their own packet processing functions and don't care about the methods to obtain the raw packets. All that user have to know is the way packets are built, and how to extract data from the raw packets.

There are only three steps in capturing packets with mycap:



Creating a new session with mycap, which causes a new raw packet listening socket is created in the system. Through this socket we'll be able to receive a raw binary stream from packets flow, going from and to any network interface installed on the system. Running a packet capturing loop. This is the heart of mycap, here all the packets are actually captured and passed to packet handling function. Mycap loop

is in fact a passive loop waiting for incoming packets, when called inside other program it will stop execution of code until it ends (end loop function). Starting the packet capturing loop requires a packet handling function, which will be associated with current session. Every time the packet loop captures a packet it passes it to an associated packet handling function.

The only limitation is that the packet handling function must be of specific prototype:

```
void *function(void *, mycap_header *, unsigned char *);
```

In the first argument it gets the parameters passed by user, then the header prepared by mycap loop (code below) and the pointer to the buffer containing the raw packet.

```
/* The header passed to the packet handling function,
contains additional information about the captured packet */
typedef struct mycap_header_s{
   /* size of the captured packet in bytes */
   int caplen;

   /* name of the interface the packet came through */
   char ifname[8];

   /* special field describing whether is is an incoming,
   outgoing or maybe a broadcast or mutlicast packet */
   unsigned char pkttype;

} mycap_header_t;
```

After passing the packet to the function, the loop will wait for the function to finish its work, before it will pass it another packet. It's because the time required by the function to process the packet depends on the packet filters applied to that function and the execution time of the function itself, which can vary on slower cpu types. While reading packets, loop uses the system buffer, but if the packet processing function requires more time then the rate of incoming packets, instead of running another function to process the excess of packets, the loop drops them. This way even if some packets are dropped, the loop won't cause any overload to the system. Using the end loop function on a mycap session will stop the packet reading loop and continue executing the code below the call to the mycap loop. The loop may be restarted at any later time.

To end packet capturing finally, user closes the mycap session, the packet socket is closed and memory used by the session is freed. After this point, to run packet capturing loop, user must create another mycap session.

netraf Configuration Files Syntax

Configuration files are used by all **netraf** programs (**netrafd**, **netrafg** and **netrafl**) to read or write every configurable aspects of them. Their syntax is clear and human readable, so the user can easily change settings to meet his demands. Operations on configuration files are divided on two levels:

- low level - allows simple operations like inserting text or reading single line
- high level - looks after correct syntax, proper positioning of inserting text, etc.

This division makes eventual changes much more easier to develop.

Configuration interface

It is a collection of functions allowing low-level operations on specified file. First of all it must be opened. All necessary structures will be initialized and ready to work. We do this by calling iconf_open() function. It will return pointer to iconf_t structure, and will be used in every operation until it is closed. After then, we can read the file line by line using iconf_getline function, insert any text at given position, clean area in the file appointed by start position and end position, finally override selected line. When we finished working with file, it should be closed by iconf_close() function.

Configuration file syntax

Basically, all configuration files consist of sections. Each section begins with name, which must be enclosed with a pair of '[[' and ']]' strings. Moreover every section must end with '[[]]' string. Section can have any number of entries. Entry is simply a line containing entry name, equation mark, quoted value, and semicolon as terminator. We can also insert comments into the file. It must begin with '#' character and all data after it will be ignored until end of line. Let's look at the configuration file example:

```
#here is comment
[[ section_name ]]
entry_name = "value"; #comment
another_entry_name = "value";
[[]]
```

```
[[ section_name_2 ]]
entry_name = "value";
another_entry_name = "value";
[[]]

#this section has no entries
[[ section_name_3 ]]
#entry_name = "value";
[[]]
```

Now, knowing how configuration file should look like, let's explain how to control it. At the beginning it must be opened (as we did with configuration interface) by conf_open() function. Pointer to conf_t structure will be returned. How to read it? There are two ways. First one is sequential. We call get_first_section() function to move to the first section. It'll return it's name or NULL if there is no section in the file. Then, using next_section() we get other sections, until NULL is returned. Second way is to call goto_section() function. But we must know exact name of section we want to go. There is also an option to get name of current section, by calling current_section(). Retrieving entry values from current section is similar operation. get_first_entry() and next_entry() are used to review following entries giving their names. Finally, to get entry value current_entry() function is used. There is also a shortcut. We can use get_value() with entry name as one of the arguments, to search whole section for it. We will get value, or NULL if there was no such entry.

Inserting new sections or entries into the configuration file is even more easy than reading it. We use new_section() to add new section, new_pair() to add new entry in the current section and delete_section() to delete one. An example showing the usage of all, mentioned functions is available in code chunks.

5. Project ChangeLog

24.02.2005	OK... Here we are. It's time to invent something interesting.
26.02.2005	I've register some free domain, configure primary and secondary DNS. It's time to bring some http server up.
03.03.2005	We are working on describing major Project Goals. Those are in VERY unofficial state, and probably they will change.
05.03.2005	**netraf** project "web-page creating process" has been started. From now http://netrafd.sign.a.la/ is the official web-address of project.
09.03.2005	Michał is trying to create an account and register **netraf** at SourceForge.net.
10.03.2005	First steps in network programming with pcap library in linux. It's quite easy. I'm diggin' in documentation.
	I've configured some test-environments (linux 2.4.xx, FreeBSD 5.1 with generic kernel, Cygwin under Windows®) wondering if some `ioctl()` calls will be portable *(NO :(, at least it won't be a child's play)*.
13.03.2005	**netraf** code parts (hmm... "early stages") are available.
	After looking at opensource.org licenses page we have chosen BSD License - it's short and plain.
	Short introduction (and explanation *"why?"*) to **netraf** is available.
15.03.2005	**netraf** project is registered at SourceForge.net! Thus, new addresses are available: http://www.sourceforge.net/projects/netrafd/ and http://netrafd.sourceforge.net/.
	Project page is a little modernized (added a bit of CSS and some cosmetic changes).

28.03.2005	our project page can be found via google.com since 25.03.2005! PS: Happy Easter!!
02.04.2005	I'm starting work with GUI. Learning NCURSES Library from wonderful NCURSES Programming HOWTO.
05.04.2005	Our second "first steps" in pcap (now it's Michał turn to figure out how it all works), some packets captured, now we know how this big machinery works, lots of fun with tracing packet flow through system.
07.04.2005	NCURSES library is amazing, but insufficient to build convenient User Interface. We need good thread-model; solution: POSIX-threads specification. It's well described in Mark Mitchell's book - "Advanced Linux Programming".
08.04.2005	Tons of packets captured. Work on counting packets by MAC addresses begun.
09.04.2005	Found some implementation of hash table, which would nicely fit into our project. Tweaking some features, assimilation process in progress.
10.04.2005	We've noticed the need for some kind of communication protocol (GUI and daemons have to communicate in some way).
	Tomek is learning all about autoconf. We will have "professional" configure script and Makefiles :).
	First worth mentioning success with **netrafd** part of the project: packets are flowing through machine, counters are spinning... Looks like everything will eventually work.
12.04.2005	Small trip into the world of advanced linux programming. First thoughts that tracing packets going through system might not be as easy as it looked... Back to the drawingboard, time to refresh knowledge about packet structure, size of particular fields in packet headers.
14.04.2005	Finally - we've described how **netraf** will be working. We'll invite you to reading changed project goals.
	Amazed by the fact that small and simple things

	tend to become big and complex when they come in large numbers. There are so many network protocols, from link layer to session layer, and every one of them have its own header. Great, big world of possibilities opened its gates for us, and crushed us with endless complexity of the problem. Another step in specifying the goals of the project helped to overcome the crisis.
17.04.2005	The packets from one interface are counted by MAC addresses. Don't know why, but something is going wrong with linux pseudo interface "any" (way to capture on all interfaces). Investigation in progress.
18.04.2005	Another part of code almost working, first steps to connection logging. Connections will be remembered by two pairs (IP and PORT). Reading about the three way handshake in TCP protocol, and trying to understand the process of closing connection.
19.04.2005	Encountered another problem; this time it is about the hash table implementation - it doesn't fit our needs. It implements the LRU algorithm which causes it to dynamically shift physical position of the hashes in table. So... when we have one writer and many readers, different readers would have different readings from the structure. What's more, even the same reader could read some entry twice or crash when trying to read entry that is being moved. We need a hashtable that would give some interface for readers, where data is static.
22.04.2005	Shared memory model born and is described now. Mateusz starts coding it.
	Can't focus on one thing, started work on counting packets for interface statistics.
23.04.2005	Stopped working. Thinking and reading about interface "any" in linux kernel. The problem is that when capturing packets from interface "any", the source MAC is set to something like 00:00:01:00:00:06, and I don't know what to do with it - how to recover the original source mac.
25.04.2005	First GUI screenshot is available! "Menu system" based on NCURSES menus library is very near...
26.04.2005	**netrafg** "Menu system" is passing tests (screenshot available).

02.05.2005	Working with existing **netrafd** code, rewriting and optimizing functions, commenting code, trying to figure out how to put everything together. Working on threading, and thinking how to split code into autonomous parts.
04.05.2005	Shared memory model is ready to use (available in Code Chunks).
05.05.2005	Rewrote parts of **netrafd** to use shmem.
06.05.2005	***"... I was who you are... You will be who I am..."*** *Goodbye...* *I'm sure we'll meet each other again some day.* M.S.
12.05.2005	Goodbye to pcap. Change of plans; we'll write our own, simple library to capture packets. Why? As mentioned before, Michał couldn't figure out how to get machine source hardware address while listening on interface "any".
18.04.2005	**netrafg** "Menu system" is ready to use. It is thread-safe and using NCURSES menu and panel libraries.
	The "mycap" set of functions is ready. It presents a simple interface similar to the one presented by pcap, so it shouldn't be difficult to rearrange existing code to use it instead of pcap.
19.05.2005	Finished the MAC statistics module. Interface statistics module is also capturing packets on all interfaces.
20.05.2005	Organized all **netrafd** functions into one interface, allowing simple adding and managing of modules. Using shmem made whole work much easier.
21.05.2005	"Literature" section added to project page.
22.05.2005	Interface statistics module is finished. Now it listens on all interfaces, and counts incoming and outgoing bytes/packets. Also knows how to recognize broadcast, multicast and loopback packets.
23.05.2005	We've invented Config-file module interface. Tomek is working on whole library.

	Started to write parts of filtering module, which will allow us to add user filters to packet capturing functions.
26.05.2005	We have moved our project page to sourceforge.net since "a.la" domain no longer exists.
27.05.2005	We began working on "man" pages to **netraf**.
28.05.2005	"Theory" section added. You're invited to look at "**netraf** Operation Diagram".
30.05.2005	Tomek introduced first working version of Config-file module.
	netraf Shared Memory Model, **netrafd** working scheme and "MYCAP" packet capture library articles are finally done.
03.06.2005	There are more and more dialog boxes in **netrafg** (`form.h` library usage).
06.06.2005	First working version of **netrafl** is ready for tests.
08.06.2005	**netrafg** Interface Statistics, MAC Statistics, IP Statistics and TCP Connection Statistics windows are done. Every window is owned by different thread, so every change in individual **netrafd** structure are visible immediately.
09.06.2005	A major bug in shmem implementation found and fixed (Code Chunks updated). It was mistake in deleting element from hash table.
10.06.2005	Change of technology in configuration files, because usage of `mmap` wasn't stable enough
16.06.2005	**netrafg** has two "screen-savers" (available in code chunks separately).
17.06.2005	Configuration Files library with example is available in Code Chunks.
18.06.2005	New articles in "Theory" section: **netrafl** Working Scheme and **netraf** Configuration Files and Filters Syntax
19.06.2005	"Documentation" section appeared.

20.06.2005	New screenshots of **netrafg** added.
21.06.2005	**netraf** finally has professional "configure" and "make" scripts!
22.06.2005	We invite you to download first official beta of **netraf** project. (Everything is here).

6. Program Documentations

All program documentations are available on included CD and on **netraf** project page.

7. Project Issues

The first release is finally published.

Many hours of work have finally paid off, everything seems to work great. We have tried to make a program that would be in some way useful for linux administrators, helping them to monitor their network traffic for longer periods, so they could shape their bandwitch more accurately. Does it really work? For now **netraf** allows to monitor network traffic and analyze it in few basic aspects, meeting our primary goals, and is a good platform for future development.

During our work, our goals didn't go that far away from our primary specification, the main idea has been kept intact, we only changed technologies used to achieve it. Bugs? None we could find but there are some possible fixes or improvements though.

One thing we didn't achieve is portability - we simply run out of time to test it and check dependencies on systems other than Linux. The main reason is we didn't use pcap, that would ensure us the portability of **netrafd**. The whole software is made in a way that it can be easily rewritten to operate on other UNIX-like systems.

Second thing that has to wait till next release are actions assigned to writers, that would be triggered if some user defined criteria are met. This actions would help to automate reactions to some statistics states, like amount of bytes transfered by given user. The reason is same as above - time, there is never too much of it.

netrafd Issues

Things that could be done.

As I mentioned earlier, linux offers so many possibilities that even after another few months of work there would still be many things that could be added to **netrafd**. For now **netrafd** (on most writer types) recognizes only ethernet packets, and only IP version 4 protocol.
I think, that 0.1beta status is very adequate, and clearly shows the relation between things already implemented and things I would like to implement.
In most cases I completed the primary goals that were set at the beginning. From things that could be added to this version, and because of lack of time will appear in next release are:

- *simple connection tracking* - by now, **netrafd** uses very simple method to recognize connections, based on two pairs of IP and port, and packet flags, to guess the connection state. This method merely allows us to tell whether the connection is established or has it already ended, but we can't associate any additional info about that connection.

- *advanced filters* - by now, filters allow to filter by specific values from packet headers, like particular ip, mac address or interface and I think it would be useful if one could define a value range (like IP range from 172.12.0.0 to 172.12.100.1, or port range) or define a wild card (like IP addresses 192.168.0.*)

- *more writer types* - maybe later we could add more writer types, for more specific uses, like ones that allow to write data stream from packets to file (combined with connection tracking this could open many interesting possibilities).

Things that have changed in specification during implementation.

The main difference is the pcap library, we intended to use it to capture the packets for further processing, but I've encountered problems with using it in multithreaded application, and later on with interface 'any' - I couldn't find a way to retrieve the source MAC address from packets that came from that "virtual" interface, I didn't even try to figure out how to check from which physical interface they came.

That was the main reason I wrote my own simple packet capturing library, that perfectly fits my needs, it may be lacking many of the pcap functionality, but can be easily extended.

Things I intend to fix.

For now, while closing, **netrafd** occasionally hangs for about 4 seconds before exiting, it is because **netrafd** has to stop all writers, and if any conn_stat type writer is running with periodical statistics cleanup enabled, **netrafd** has to wait for that "garbage collector" to wake up and terminate, before he can shut down the main process.

netrafg Issues

Things that will be done in near future. They aren't implemented now because lack of time. Options and features mentioned below are grown up during **netrafg** creation process. Every major assumptions that we've made at the begin of our work are implemented now. So, here follows details to do:

- search facilities in every writer window,
- sort facilities in every writer window (by IP-addr, MAC, interface etc...),
- resizing writer windows,
- miscellaneous key-bindings (eg. `[shift]`+`[tab]`, correct `[esc]` key operation, `[alt]`+letter menu calls...),
- more configuration dialogs (screen refreshes, saving actual desktop state...),
- increase security (it is sufficient to have only one daemon that requires root privileges - **netrafd**. **netrafg**, and **netrafl** can

obtain needed permissions - via some simple autenthication method - for operations that really need them and just after such operation get rid of them),
- more user-conveniences (e.g. when creating new writer it will be nice to have "default" name given to it),
- add "filtering" to GUI "view" options (we're talking only about filtering at GUI level, not about "filters" mechanism implemented in **netrafd**),

netrafl Issues

There are still many things that can be done to improve **netrafl** and configuration files. It has some minor bugs, or features that due to lack of time couldn't be done yet.

- change config files technology to mmap again - First of all, mmap technology had to be changed because of instability of config files implementation. Still I don't know what caused it, I think that mmaping is really interesting subject and I'm sure it will be brought back to project. Although new implementation is more stable, operating on mapped file is much more faster, and more elegant way of file access. This could cause logging more data in shorter time, and less CPU usage.

- logging ipstat and connstat into database (PostgreSQL) - Ipstat and connstat can't be logged as ifstat or macstat. There is one reason. Simply, in short period of time, there will be too many data to log. So the machine, on which **netrafl** runs could slow down seriously. The idea to solve this problem is professional database, for example PostgreSQL. It is designed to store large amount of data, and has very optimal implementation of inserting, deleting and searching operations. At first, logging supposed to be done in the way as ifstat and macstat is done. But problems with configuration files (change the technology to slower one, to be precise) forced me to forbid logging writers, which generate too many data.

- some minor bugs related with user privileges - There is a bug occurring sometimes, when user is trying to insert or delete data from configuration file without root privileges.

8. Literature

- **Programming:**
  - ***"The C Programming Language"***
    Brian W. Kernighan, Dennis M. Ritche
    ISBN: 83-204-2719-3
  - ***"Beginning Linux Programming"***
    Neil Matthew, Richard Stones
    ISBN: 83-7243-020-9
  - ***"Advanced Linux Programming"***
    Mark Mitchell, Jeffrey Oldham, Alex Samuel
    ISBN: 83-7243-217-1
  - ***"Professional Linux Programming"***
    Neil Matthew, Richard Stones
    ISBN: 83-7197-495-7
  - ***"Linux Kernel Development"***
    Robert Love
    ISBN: 83-7361-439-7

- **Programming techniques and theory:**
  - ***"Introduction to Algorithms"***
    Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
    ISBN: 83-204-2879-3
  - ***"Data Structures and Algorithms"***
    Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman
    ISBN: 83-7361-177-0
  - ***"The Design and Analysis of Computer Algorithms"***
    Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman
    ISBN: 83-7197-770-0
  - ***"Programming Pearls"***
    John Bentley
    ISBN: 83-204-2672-8
  - ***"Języki Programowania"***
    ToMasz Wierzbicki
    Symbol Uniwersalnej Klasyfikacji Dziesiętnej: 004.43
    2000 Mathematics Subject Classification: 68N15
  - ***"Operating System Concepts"***
    Abraham Silberschatz, Peter Baer Galvin, Greg Gagne
    ISBN: 83-204-2961-7

- ***"Principles of Concurrent and Distributed Programming"***
  M. Ben-Ari
  ISBN: 83-204-1996-4
- ***"Hacker's Delight"***
  Henry S. Warren, Jr.
  ISBN: 83-7361-220-3

- **Software engineering:**
  - ***"The Art of UNIX Programming"***
    Eric S. Raymond
    ISBN: 83-7361-419-2
  - ***"The Pragmatic Programmer"***
    Andrew Hunt, David Thomas
    ISBN: 83-204-2672-3
  - ***"Writing solid code"***
    Steve Maguire
    ISBN: 83-7197-429-9
  - ***"The Mythical Man-Month"***
    Frederick P. Brooks, Jr.
    ISBN: 83-204-2467-4
  - ***"The Practice of Programming"***
    Brian W. Kernighan, Rob Pike
    ISBN: 83-204-2732-0
  - ***"Extreme Programming. Pocket Guide"***
    Ward Cunningham
    ISBN: 83-7361-343-9
  - ***"Inżynieria Oprogramowania"***
    Andrzej Jaszkiewicz
    ISBN: 83-7197-007-2

- **Networking:**
  - ***"TCP/IP Network Administration"***
    Craig Hunt
    ISBN: 83-7243-305-4
  - ***"TCP/IP Bible"***
    Rob Scrimger, Paul LaSalle, Mridula Parihar, Meeta Gupta, Clay Leitzke
    ISBN: 83-7197-668-2
  - ***"Teach yourself TCP/IP in 14 days"***
    Timothy Parker
    ISBN: 83-86718-55-2

- *"Introduction to the Internet Protocols"*
  Charles Hedrick
- *"Connected: An Internet Encyclopedia"*
  Brent Baccala
- *"Internet Protocols. Description and Packet Format"*
  alex@netfor2.com
- *"Internet Tutorials"*
  InetDaemon Enterprises
- *"Programming with pcap"*
  Tim Carstens

- **Miscellaneous packages and libraries documentation:**

  - *"GNU Autoconf manual"*
    Free Software Foundation, Inc.
  - *"NCURSES Programming HOWTO"*
    Pradeep Padala
  - *"Writing Programs with NCURSES"*
    Eric S. Raymond, Zeyd M. Ben-Halim, Thomas Dickey