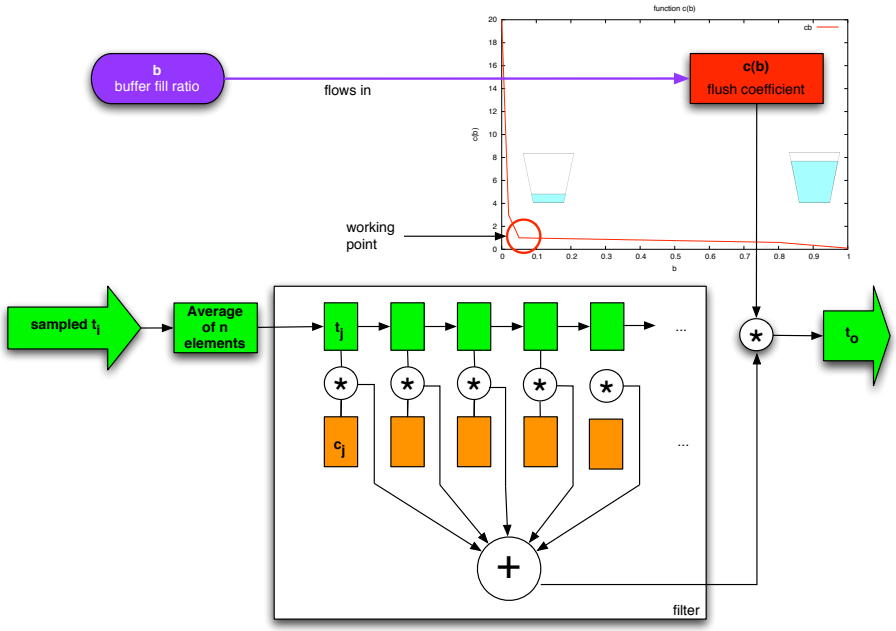
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Caspar Schlegel

Realtime UDP NetFlow Processing Framework



Diploma Thesis DA-2003.26
May 2003 to September 2003

Supervisors: Thomas Dübendorfer, Arno Wagner
Professor: Bernhard Plattner

Abstract

For the detection of an ongoing infection or attack phase of a distributed denial of service attack it is important to be able to characterize the current Internet traffic. Therefore network packets containing NetFlow data periodically generated by Cisco routers have to be captured and buffered. Near realtime plugins can process the data. For optional resending of the received data on a network, bursts in the input traffic have to be filtered out. From different possible designs of such a receiving framework a variant has been chosen and implemented where shared memory buffers circulate between the data capturing entity and the framework clients. The implemented framework offers a function library to easily implement client plugins. This library also contains a traffic shaping unit which filters out the data bursts and generates a smooth traffic characteristics. The framework has been tested with simulation data and data from an Internet provider backbone router.

Um eine laufende Infektions- oder Angriffsphase einer distributed denial of service Attacke zu erkennen ist es wichtig, den aktuellen Internet-Verkehr charakterisieren zu können. Hierzu müssen Netzwerk-Pakete, welche NetFlow-Daten enthalten die periodisch von Cisco-Routern generiert werden, aufgefangen und gebuffert werden. Near-Realtime-Plugins bearbeiten diese Daten. Falls die Daten über ein Netzwerk weitergesendet werden, müssen Lastspitzen in der Datenrate geglättet werden. Unter verschiedenen möglichen Designs eines solchen UDP-Empfangs-Frameworks wurde eine Variante gewählt und implementiert, welche mit Shared-Memory-Buffern arbeitet. Diese Buffer zirkulieren zwischen dem Daten-Empfänger und den Framework-Clients. Das implementierte Framework bietet eine Funktionsbibliothek an, mit der einfach Client-Plugins erstellt werden können. Diese Funktionsbibliothek enthält des weiteren eine Traffic-Shaping-Einheit, welche Datenhäufungen vermeidet und eine glatte Verkehrscharakteristik produziert. Das Framework wurde mit simulierten Daten und auch mit Daten eines Internet Provider Backbone Routers getestet.

Contents

Chapter 1

Motivation

The subject of the DDoSVax research project is the following (taken from the project home page [2]):

"Distributed Denial of Service Attacks are a growing threat to the whole Internet. The DDoS-Vax Project researches possible detection and analysis algorithms, general attack mitigation strategies and possible (semi-)automatic countermeasures.

The DDoSVax Project's objectives are

- Detection of infection phases while infection takes place
- Detection and analysis of massive DDoS attacks when they start in near real-time.
- Provision of methods and tools that support countermeasures during both phases.

To achieve these objects data of the current and past Internet traffic has to be gathered. Having the possibility to process the Border-Gateway-Router generated NetFlow-Data[1] of a Backbone Provider (in our case of SWITCH), one can study all flows which cross the boundary of the Switch-Backbone."

The *Realtime UDP NetFlow Processing Framework* provides a research workbench to experiment with different algorithms working on NetFlow.

This framework should provide the following key functionality to it's client plugins:

- Reception of the NetFlow data from several routers
- Provision of a buffering mechanism to cope with bursts of data
- Protect the client plugin from the bulky traffic characteristics generated by the routers
- Provide watchdog functionality for the clients

With this framework, it is save to experiment with different types of detection algorithms because the client plugins are protected from each other, the crash of one will not lead to disruption of the framework functionality for the other clients.

[1] NetFlow is Cisco's data format for collecting IP data flow statistics entering router or switch interfaces, uses UDP packets and is described in [3]

Chapter 2

Problem

2.1 Today's situation

There exist tools (Fromplicator and Samplicator, [1]) from SWITCH which collect the produced NetFlow data from the routers, do a protocol conversion from NetFlow version 7 to NetFlow version 5 while at the same time filling in fields not provided by the router and send the UDP packets to other hosts which process the data for example for accounting and billing reasons. This tool chain runs on x86 based machines with Linux as operating system.

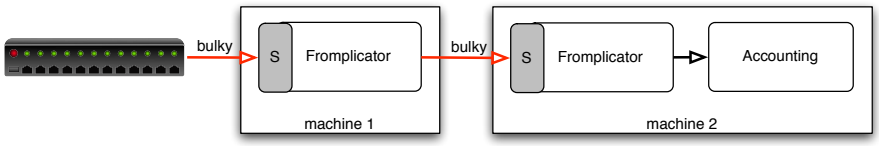


Figure 2.1: Status Quo

This solution has some drawbacks, especially

- There is no plugin support. The only possibility to add or remove subscribers of the data is to reconfigure the tool chain (see below).
- It has a static configuration. For reconfiguration the tools have to be restarted, which implies data loss. For example it is not possible to add new receiving hosts at runtime, but as a research workbench, it is required that it is be possible to add and remove targets without interrupting the data capturing since during development plugins have to be tested with real world data without interrupting productive plugins
- The UDP reception functionality has to be implemented for every member of the tool chain. As the routers cache the NetFlow Data and flush every few seconds, the traffic is very bursty. This burstyness is preserved during the forwarding in the tool chain. This leads to the need of very large socket buffers for each process in the tool chain[1].
- This bursty traffic is forwarded on the Switch networking backbone to the central processing facility. This leads to congestion in the backbone and again to data loss.

[1] Socket buffers under Linux are per default rather small. Because of that and the fact that Linux does not have realtime scheduling, bursty traffic leads to data loss.

2.2 Requirements for the plugin framework

It's the goal of this Diploma Thesis to design a Realtime UDP NetFlow Processing Framework as a workbench for the further analysis of the NetFlow Data Stream. One can define the following requirements for the framework:

- The data is as complete as possible.
- The framework is runtime configurable.
- All data that is resent from the framework is debursted. There is an optional rate limit imposed on the data stream.
- The framework provides a data buffering mechanism
- There is a plugin mechanism, plugins can join and leave the framework at any time.
- No plugin can ever hinder other plugins.
- Optionally, plugins are on surveillance by a watchdog process
- Statistics over several aspects of the framework are generated

2.3 Possible solution: in Hardware

2.3.1 Custom Flow Replicator Hardware

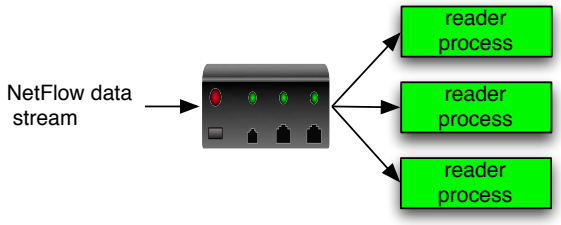


Figure 2.2: In Hardware

The replication of the data in multiple output streams can be done very efficiently in hardware. Plugins could be implemented by designing extension cards using co-processors or by using reconfigurable hardware (for example with FPGA's).

This solution is expensive as it is not widely available like PC hardware.

It is not suitable as experimental research platform because either new extension cards have to be constructed and built or algorithms have to be implemented in hardware description languages like VHDL. There is no easy and fast way to test a prototype of some algorithm.

2.4 Possible solution: in Software

Using a PC as basis for the Network Receiving Framework has the advantage that the hardware is widely available and cheap compared to custom built hardware. Algorithms can be implemented using well known programming languages, ideas can be tested using scripting languages. The development cycles are fast.

2.4.1 Extending Fromplicator/Samplicator

One possibility for the implementation would be the extension of the existing Fromplicator/Samplicator tools like described in chapter 2.1 with the missing functionality. As practically all required functionality is missing this corresponds to a new implementation of such a system.

2.4.2 Extending another NetFlow Receiving Framework

There exists the traffic flow analysis tool cflowd from CAIDA [4]. It implements the receiving of NetFlow data, the processing of the data and the storage of both raw NetFlow Data and of derived statistics. The buffering of the data is done using the “Buffer Switching” technique which has some major drawbacks like limited buffer space or high latency. See chapter 2.5 for a discussion of different buffering techniques and their advantages and disadvantages.

It would have been possible to extend a third party framework with the missing functionality like extended buffering, plugin functionality and a traffic shaping facility. This would have meant a major rework of all parts of cflowd as for example the processing of the data is done monolithically in the cflowd process. One would have had to extend the buffering facility, to implement the traffic shaping facility etc.

The only part that would stay untouched would be the receiving part but it would have required a considerable amount of time to understand the existing data structures and to find locations where the desired functionality could have been added.

2.4.3 Result of the evaluation: Implementation of a new Framework

Because of the reasons stated above neither extending Fromplicator/Samplicator nor extending another framework like CAIDA would taken considerably less time than implementing a new approach of a receiving framework. Another advantage is that a custom built framework can be tailored exactly to it’s employment’s needs, therefore all requirements stated above can be met.

The variant finally chosen is to build a new framework while recycling some code from Samplicator in order not to have to implement basic code like reception of an UDP packet etc.

2.5 Data Buffering

The buffer component is one of the crucial components of the system. During the implementation two different approaches have been tested:

2.5.1 Buffer Switching

This is the first mechanism considered for a preliminary version of the framework and is also implemented in the CAIDA Traffic Flow Analysis Tool cflowd[4]. Therefore it is explained with cflowd as example:

Two shared memory segments[2] are used. The received data is entered in one segment by cflowdmux while cflowd reads the data from the other segment. As soon as cflowdmux has finished filling the first segment, these two segments are interchanged, the segment formerly read by cflowd is now written on by cflowdmux and vice versa.

Despite of it’s simplicity this approach has some major drawbacks which makes it unusable for our needs and therefore was replaced with another approach, “Circulating Memory Segments”:

Looking at the requirements defined above, One can find the following drawbacks:

[2] In this thesis the term “shared memory” is always used as a synonym to “System V Shared Memory”, a Unix kernel facility where multiple processes can share memory. Each memory segment is identified by a shared memory key (analogical to a file name) and a shared memory identifier (analogical to a file descriptor)

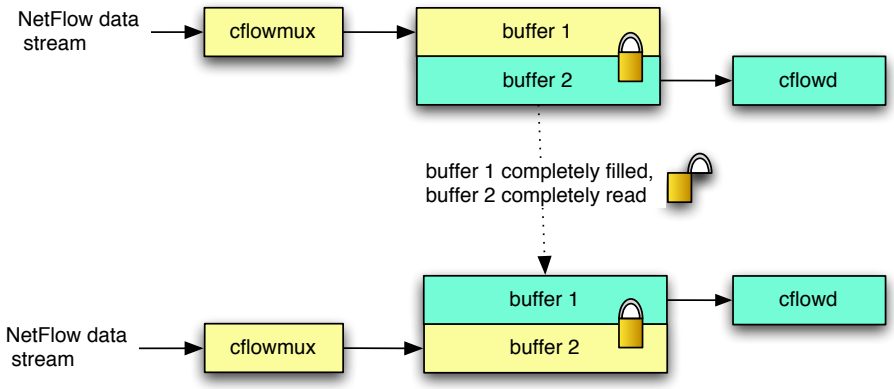


Figure 2.3: Switching Buffers

The data should be as complete as possible. Due to the limited socket buffer of the Linux kernel to prevent data loss the receiving component has to gather the data from the socket buffer as fast as possible. During the switching of the two shared memory segments the reading and writing process have to be synchronized. In the example of cflowd, this is achieved by semaphores. It is possible that the receiving component looses data during waiting for this semaphore.

The framework should provide a data buffering mechanism. As this solution only uses two memory segments to provide a significant amount of buffer space (approx. 100M) these segments have to be very large, which leads to a huge amount of latency, these memory segments would be allocated statically and could not grow or shrink which causes a big memory footprint. As this buffering mechanism is one of the desired key features of the framework, this solution is not applicable.

2.5.2 Circulating Memory Segments

The “Circulating Memory Segments” solution is the replacement solution for the “Buffer Switching” approach considered first for the framework. It fixes the shortcomings stated above and also implements the plugin functionality.

There are not only two shared memory segments available but a variable number of shared memory segments. These segments circulate between the writing component and the reading components.

As seen in figure 2.4, a shared memory segment is taken by a Memory Management Process from a free memory segment repository in step 1. It’s then passed in step 2 to the Writer Process which subsequently fills this shared memory segment with received UDP payloads and passes it (step 3) back to the Management Process. As we’ll see in chapter 3 there’s no locking required in moving segments from or to the Writer Process.

In step 4, the now filled segment is enqueued in the Read List. The segment is then read by Reader Processes (Client Plugins). Each Client Plugin can access this list on a different position and individually advances to younger segments (step 5). There is a client library which provides a *iterator-like* functionality to the Reader Processes: A Reader Plugin is locked on a certain shared memory segment, therefore all operations working on a shared memory segment work on this particular segment. The plugin can *get* all UDP-Payloads stored in that shared memory segment and then switch to the next shared memory segment via the *advance* primitive. The framework guarantees that locked segments are never being overwritten or deleted without noticing the client plugin (later more on this).

As soon as a segment is read by all Client Plugins, it is dequeued from the Read List (6) by the Management Process and put back into the Free List (step 7) where it moves to the head

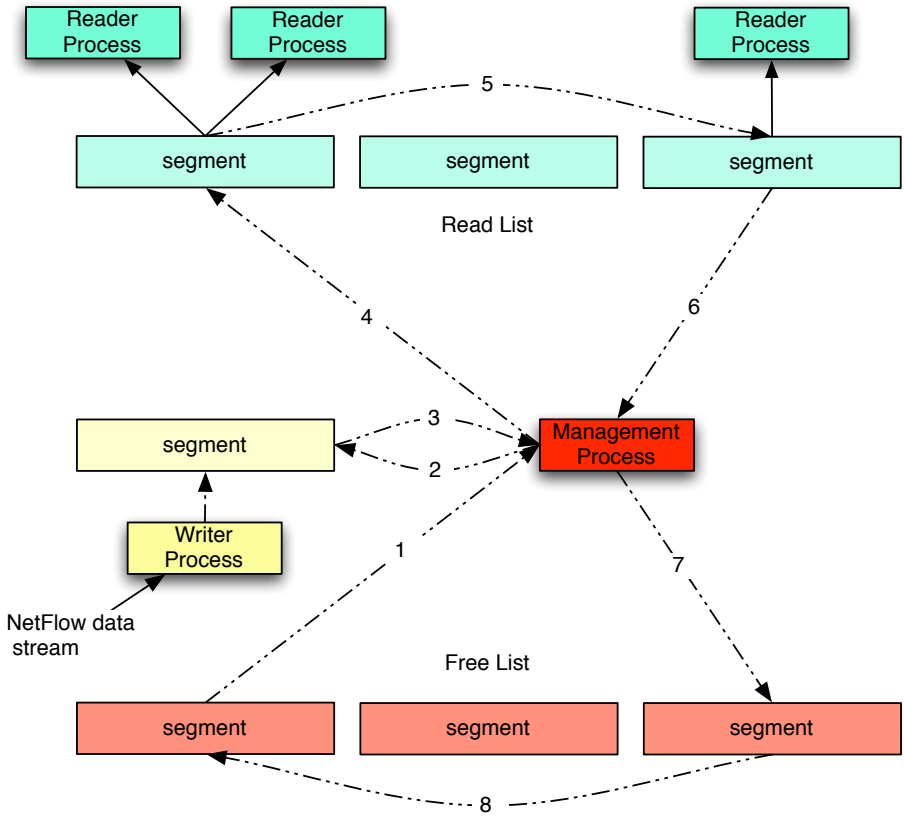


Figure 2.4: Circulating Memory Segments

(8) and is later again consumed by the Manager (step 1), which closes the Life Circle of a shared memory segment. It has to be noted that while circulating the content of the shared memory segment is never copied, it is merely an identifier the operating system assigns to a shared memory segment (shared memory Id, ShmemId) that is enqueued in the different lists and copied around. (As a matter of fact, shared memory segments are not attached to the address space of any process most of the time...)

2.6 Chosen Design and the Design Requirements

The chosen design uses the “Circulating Segments” buffer. In addition to the reader and writer used in the Buffer Switching approach, this it uses a central arbiter: the Management Process. This process has several duties:

- Maintaining the Life Cycle of a shared memory segment as described above.
- Handling of plugin attaches and detaches of the Writer Process and the Client Plugins
- Handling of internal sequence numbers to ensure data consistency

With the help of some additional processes this design approach fulfills the stated requirements: (the mentioned mechanisms are explained in detail in the next chapter)

The data is as complete as possible. Using lock- and wait free data structures as decoupling device between the Management Process and the Writer Process the latter can concentrate on receiving UDP packets and writing them into the buffer. In contrary to the “Buffer Switching” approach from above the Writer Process never has to wait for a Semaphore or something similar.

The data consistency is maintained with sequence numbers: in case the framework looses a memory segment, the clients are notified.

The framework is runtime configurable. The configuration resides in a central shared memory segment. Every time a component needs a configurable value, it is directly taken from this configuration repository. The configuration can be altered with a configuration tool at runtime.

All data that is resent from the framework is debursted. There is an optional rate limit imposed on the data stream. The framework provides the clients an optional traffic shaper mechanism which flattens and limits the data speed. Clients resending the data over the network will use this mechanism while clients working locally with the data work at full speed and minimal latency.

The framework provides a data buffering mechanism As there are more than two segments circulating in the framework, there can be a variable number of segments waiting to be read. So, small bursts of data can be flattened out. Because there is not only one memory segment Reader Plugins can work at their own speed. It is possible that a Reader Plugin stops reading data for a moment, for example during garbage collection, disk access or while resending data via TCP, and can catch up later.

There is a plugin mechanism, plugins can join and leave the framework at any time. The Management Process administrates the attached plugins and allows a plugin to join or leave the framework at any time.

No plugin can ever hinder other plugins The plugins individually process the data segments provided by the Management Process. If a plugin is too slow or crashes during the processing of a memory segment, the manager steals this segment from the plugin when this memory segment has to be reused for fresh data.

Optionally plugins are on surveillance by a Watchdog Process. There is a watchdog client library the plugins can use for the communication with a Watchdog Process. With the help of this library, the plugin sends heartbeat messages to the Watchdog Process. If a process stops sending these heartbeat messages, it is restarted after a configurable amount of time.

Statistics over several aspects of the framework is generated. As all data segments are circulated through the framework by the Management Process, this process can gather statistics over several parameters of the framework.

Chapter 3

Design

The taken design follows the mentioned “Circulating Buffers” approach. Figure 3.1 shows a more exact picture of a shared memory segment life cycle as it is implemented in the framework.

The different functional units of the framework, like Management Entity, Writer and Reader Plugins can be implemented as threads of one process or each as one process. For the desiderata of the framework the thread approach has several shortcomings:

- If one of the threads catches a operating system exception (like a segmentation fault etc.) the whole process and therefore the whole system crashes.
- Every thread potentially can damage data structures of the other threads, therefore a badly programmed Reader plugin can crash the whole system
- Using System V Shared Memory has the welcome side effect that it is not bound to a lifetime of a process. In contrary to the memory shared between threads, System V Shared Memory survives the crash of the whole framework and the data later can be recovered.

Therefore the functional units are chosen to be processes.

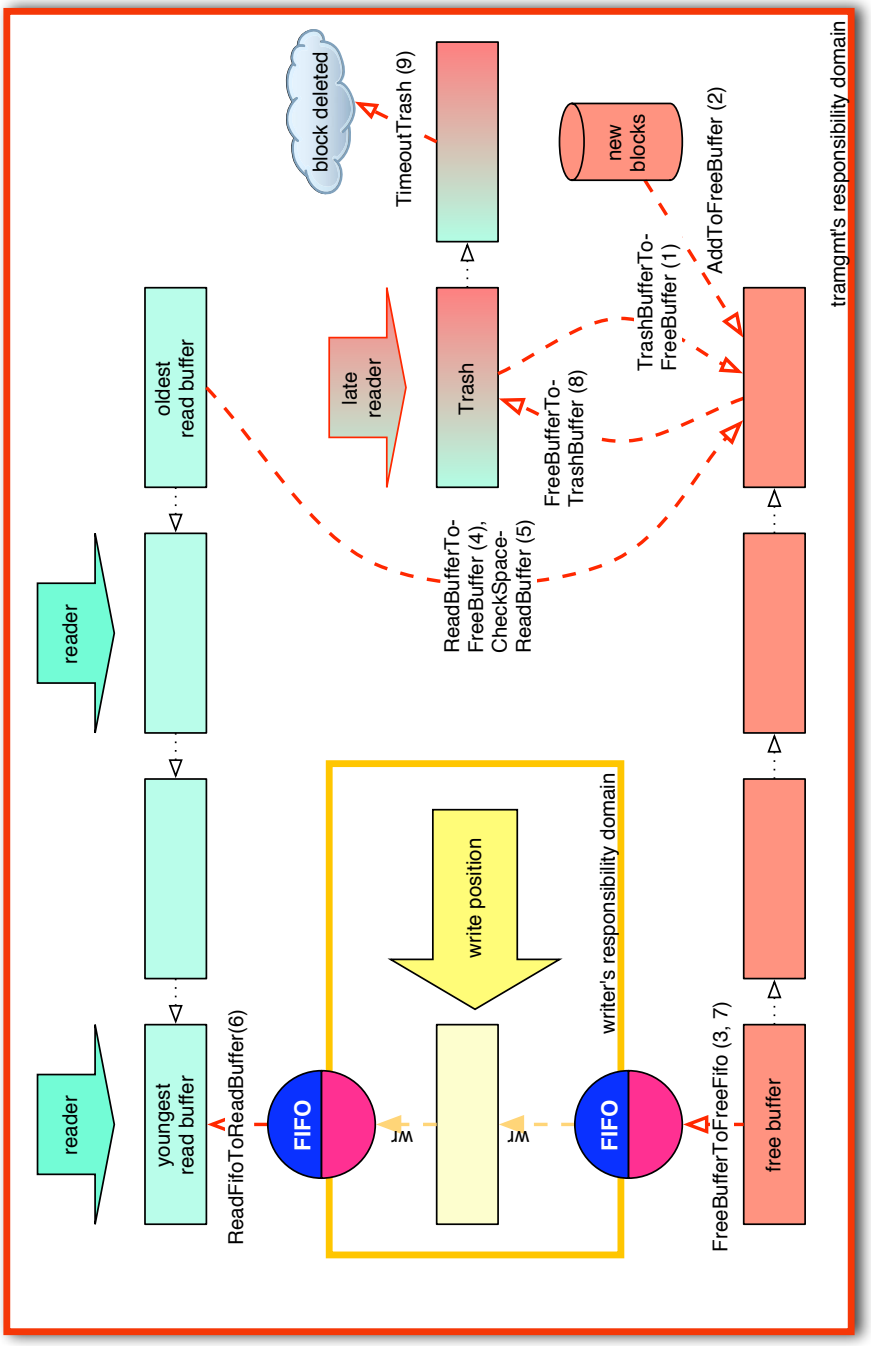


Figure 3.1: Life cycle of Shared Memory Segments

The basic life cycle is exactly as explained in figure 2.4. To lessen the used memory of the framework during quiet periods of network load, the number of shared memory segments circulating in the framework is not fixed but can grow and shrink dynamically, the memory footprint is bounded though. This leads to certain design requirements:

- To guarantee that the Writer Process has always free segments, the Management Process has to ensure that the list of free segments has always a certain length. If this length falls below a certain limit, new segments are requested from the operating system and enqueued. The reason why this is not done by the Writer Process itself is that each operating system call is a point where a Linux kernel can preempt a process and set another process in the state “running”. By loosing processor time, the probability of loosing data because of a socket buffer overrun increases.
- To lessen the memory requirement after an incoming burst of data, shared memory segments have to be freed if the length of the free segment list exceeds a certain limit.
- A to be freed segment is entered into the Trash List where it is held for a certain grace period and subsequently deleted. This Trash List helps Reader Plugins which still read a to be freed memory segment to be able to finish reading the segment. If a new burst of incoming data should lead to a need of empty memory segments, these are taken from the Trash List, as this is cheaper than requesting new segments from the operating system.
- Only if the Trash List is empty, the Management Process will request new shared memory segments from the operating system.

The Framework is split into simultaneously running processes as shown in figure 3.2. Therefore it is important to address the problems of concurrent accesses from more than one process to some data structure.

To address concurrency problems, I established the notion of a *Responsibility Domain (RD)*. A RD designates domains where one process has the guarantee that no other process will ever interfere with the data therein. One can identify two RD’s in the framework:

The Management Process is responsible to feed empty segments to the Writer Process, to collect them and to provide access to the Client Plugins.

The Writer copies the UDP payload into the provided segments.

The Reader Plugins never work with shared memory segments. All accesses are delegated to the Management Process and the Reader Process never has direct access to the Read List, so the Read List completely lies in the Management Process’ RD.

The two identified RD’s can be seen in figure 3.1.

There are two possible conflicts at the border between the identified RD’s: These arise when

- The Management Process passes segments from the Free List to the Writer and
- The Management Process gathers a filled segment from the Writer and passes it to the Read List.

A FirstInFirstOut(FIFO)-buffer is used as a decoupling element between two RD’s. The FIFO is split in two parts, each part lies in one of the two RD’s. The FIFO now allows a one way flow of data between the two RD’s.

The implemented FIFO therefore defines one side as Producer and one as Consumer. At the moment it suffices to understand that the FIFO implemented allows concurrent access of both the Producer and the Consumer at the same time without any required locking. For more details of the implementation of this FIFO see chapter 4.2.

As seen in figure 3.2 in addition to the Management Process, the Writer Process and the Reader Processes there exists a Watchdog Process, a Monitor Facility and a Configuration Facility. The function of all mentioned processes will now be explained in turn.

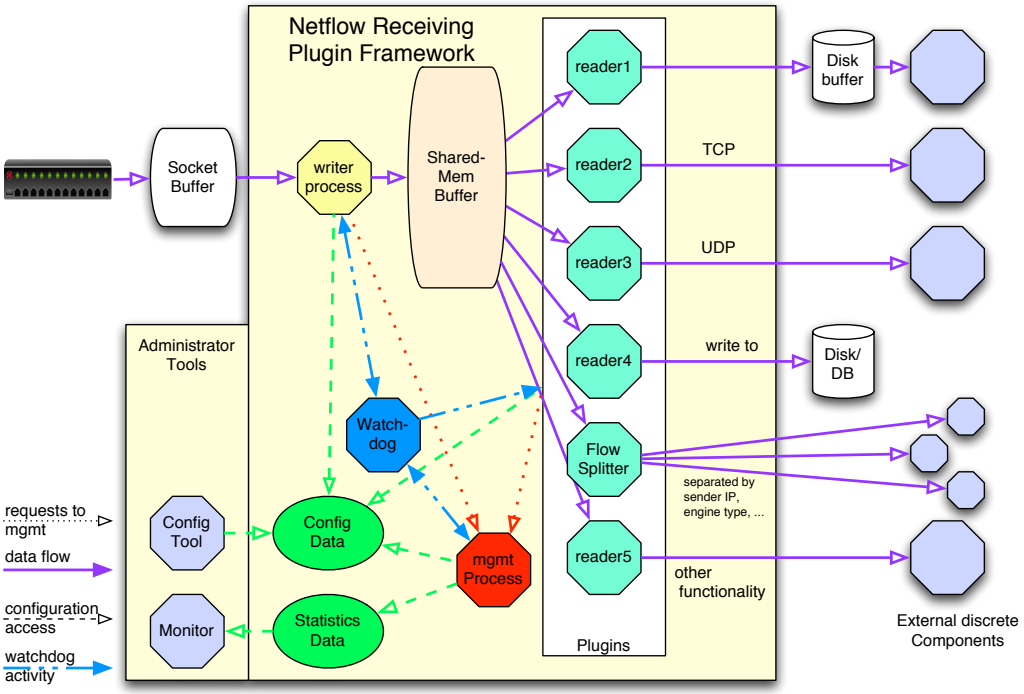


Figure 3.2: Architecture

3.1 Writer Process

The Framework is designed in such a way that the Writer Process is as simple as possible. Library routines hide the communication between the Writer Process and the Manager as well as the access to the FIFO. For am explanation of the usage of libwrite, please see chapter A.2, for information about the communication between processes of the framework chapter 4.1.

```
int main(int argc, char **argv) {

    /* writer specific initialization like opening UDP socket
     *  etc.
     */

    initialize();

    writerAnnounce();
    while(1) {
        data = getUdpPacket();
        statistics();
        writerPut(data, sizeof(data));
    }
}
```

Listing 3.1: Writer Skeleton (simplified)

Before using the Framework, the Writer Process has to *announce* itself at the Management Process. Libwrite sends an announce packet to the named pipe of the Manager (see chapter 4.1.1). The Management Process replies by sending the answer to the named pipe of this Writer. It includes the ShmemId of the central administration memory space, called adminpage. The contents of this structure is explained in chapter 4.1.2. The library function attaches the adminpage and has now access to configuration, common data structures and the ShmemId's of the two FIFO's. With this information the library can attach to the FIFO. Specific information

about the use and implementation of the FIFO can be found at chapter 4.2. One has to note that all these complicated actions are hidden from a programmer and happen in the background steered by the function library. The programmer of a Writer process can use these function calls.

The Writer now can start receive the (bursty) UDP traffic. The Library consumes a shared memory segment from the Free FIFO. The network packets buffered in the socket buffer of the operating system are now read and copied into the shared memory segment. There are logical *receivers* which receive the data from exactly one set of routers. Statistics is gathered about these receivers, for example how many packets are received on this receiver or how fast.

At some time the memory segment is filled with UDP payloads. The framework then detaches the segment and produces (copies) the ShmemId into the Read FIFO. Thereafter it consumes a new shared memory segment from the Free FIFO and continues copying.

3.2 Reader Plugins

The Reader Plugins are the Clients of the Framework. They will contain various functionality like statistical algorithms which check the data stream, forwarders which send the received data to External Discrete Components or store procedures which save the data to disk. How one writes a Reader Plugin using the TrafficShaper or by using the underlying library calls is described in chapter 6.

```
int send(bufferPacketp_t data, int result) {
    /* perform plugin functionality */
}

int main(int argc, char **argv) {

    /* Reader Plugin specific initialization like opening files */
    initialize();
    readerAnnounce();
    readerTrafficShaper(maxBandwidth, send);
}
```

Listing 3.2: Reader Skeleton using TrafficShaper (simplified)

The first step is again the *announce* at the Management Process. In “readerAnnounce” again a announce packet is sent to the named pipe of the Management Process which subsequently answers this packet on this Plugin’s named pipe which is created from the function library. This packet either tells the library that no more plugins are allowed or the shared memory id of the adminpage is returned. The function library will in this case attach the adminpage.

The last step is calling readerTrafficShaper with the allowed bandwidth the TrafficShaper should generate and some callback (send in the example) which will be called by TrafficShaper for every UDP packet contained in a shared memory segment. The traffic shaper automatically reads the UDP payloads out of the shared memory segment and switches to the next shared memory segment if needed

The timing of the callback calls is calculated so that the maximum bandwidth will never be exceeded. The produced traffic pattern is low pass filtered to eliminate spikes.

3.3 Management Process

The Management Process is responsible for various housekeeping tasks in the framework:

- Process incoming requests from Readers and Writers
- Memory Management

- Waiting List Processing
- Gather Statistics

The main loop of the Management Process looks like listing 3.3.

```
 while (1) {

    commandDispatcher();
    memoryCycle();
    processWaitingList();
    sleep();
}
```

Listing 3.3: Management Main Loop (simplified)

3.3.1 Request Processing: commandDispatcher()

As first step in the loop, incoming requests on the named pipe are processed. Commands to the Management Process include “announceReader”, “announceWriter” to announce a Plugin and as counterpart “shutdownReader” and “shutdownWriter” to sign off from the Management Process. With these commands the Plugin mechanism is implemented. There are additional commands for the Reader Plugins to navigate in the Read List: “advance” advances to the next younger segment, “toHead” allows to jump to the youngest segment of the Read List. With the command “yield”, a read lock is given back so a segment can be trashed (see chapter 3.3.2 for more exact information). It’s also possible to get the information which byte sequence number the youngest segment has (with “getByteNo”) without advancing to this segment. This functionality is useful for sampling reasons.

For some commands a response is sent. More about this type of inter process communication between the Writer Process, the Client Plugins and the Management Process is explained in chapter 4.1.1.

A special command: Advance

The most important functionality of the Management Process is the *advance* command. As mentioned above, every Client Plugin has access to exactly one shared memory segment. the command “advance” switches this visible data segment to the next younger data segment. If there is no younger segment the request cannot be answered immediately but the Reader Plugin has to wait for an incoming new segment from the Writer Process. For this case in the central administration structure (adminpage) the current segment number is marked in the waiting list.

In the main loop it is checked regularly by processWaitingList() if a new segment is ready to be read. As soon as this is the case, a reply message is sent to the Reader Plugin (which blocked on the reception of this reply packet).

There is one important exception to the fact that advance always jumps to the next younger block:

SoftLimit Functionality: If the newly visible data segment would be more than SoftLimit (a configurable limit) segments away from the youngest data segment in the Read List, the advance call skips some segments and jumps to the segment at position SoftLimit (see figure 3.3). This skipping of segments enlarges the possibility that a very slow Reader Plugin can successfully read the whole visible data segment.

As seen below the Management Process has to ensure that there’s always space to insert one data segment in the Read List. If this condition has to be enforced, one segment has to be removed even if a (late) Reader Plugin is still reading this segment. It can be that this deleted

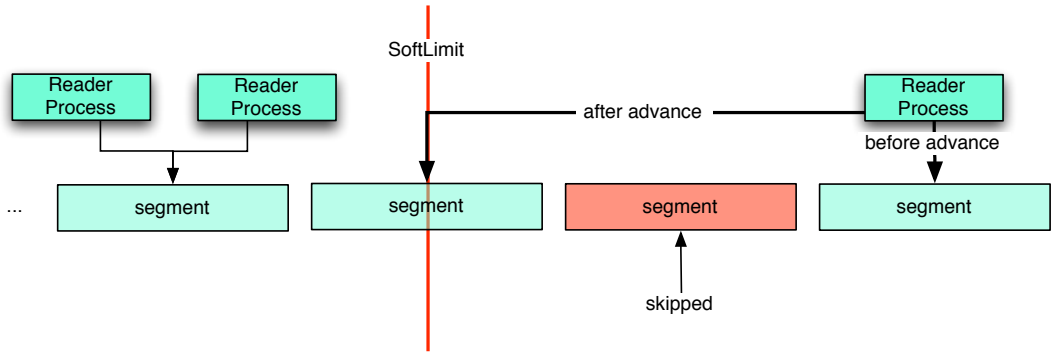


Figure 3.3: Skipping blocks with the SoftLimit functionality

segment is immediately re-used and it's content overwritten. The libread library checks for these conditions and aborts reading from this segment which leads to incomplete read segments.

Therefore it makes sense to give this Reader Plugin some head start so this condition does not (repeatedly) happen.

3.3.2 Memory Management and Statistics: memoryCycle()

The core functionality of the Memory Management Process is to circulate the shared memory segments between the Writer and Reader Plugins. This is done in the function memoryCycle(). In figure 3.1 the edges are marked with the corresponding function call from listing 3.4. The numbers in parenthesis designate the number of the step in the algorithm below.

```
saveStats();

trashBufferToFreeBuffer();
saveStats();

addToFreeBuffer();
saveStats();

freeBufferToFreeFifo();
saveStats();

readBufferToFreeBuffer();
saveStats();

checkSpaceReadBuffer(1);
saveStats();

readFifoToReadBuffer();
saveStats();

freeBufferToFreeFifo();
saveStats();

freeBufferToTrashBuffer();
saveStats();

timeoutTrash();
saveStats();
```

Listing 3.4: memoryCycle (simplified)

I split up the memory cycle in small functions which get inlined at compile time so different building blocks are more clear visible.

saveStats()

saveStats() is called at the beginning, the end and between every memory management function. It takes a sample of various statistical values like free and read Buffer length, segments received and sent etc and it calculates the filtered input stream in behalf of the TrafficShapers of the Reader Plugins (see 5. These samples are saved in the adminpage. The statistic viewer “stats” retrieves this data periodically and shows it. To see effects happening between the single steps of the memory management function, samples are taken after every step.

trashBufferToFreeBuffer()

At the beginning of memoryCycle the management Process tries to generate as much free segments as possible. Because it is cheaper to recycle segments from the Trash Buffer, they are dequeued from the Trash List and enqueued into the Free Buffer.

addToFreeBuffer()

If after filling the Free Buffer with segments from the Trash Buffer the configured minimal value is not reached, new shared memory segments are created and enqueued in the Free Buffer.

freeBuffertoFreeFifo()

Now the Free FIFO which provides the Writer Process lock- and wait free access to shared memory segments is filled with segments from the Free Buffer.

readBufferToFreeBuffer()

It is important that all shared memory segments which are read by all Reader Plugins are removed from the Read Buffer to make space for new segments.

checkSpaceReadBuffer(1)

To enforce at least one free space in the Read Buffer, if there’s no space, one shared memory segment (the oldest) is unconditionally removed and enqueued in the Free List. It is more important that newly arrived segments can be enqueued in the Read List and are read by many Plugins as that also the pathologically behaving Plugins can finish read their packets. The Reader Plugin whose segment has been stolen is notified from libread. This situation should rarely happen because of the SoftLimit functionality. This limit should be configured to be early enough that every Plugin can finish reading a segment before the segment is reused.

readFifoToReadBuffer()

Now there is space in the Read Buffer for at least one shared memory segment. The segment is dequeued from the FIFO and enqueued into the Read Buffer.

again freeBuffertoFreeFifo()

In the meantime, the Writer Process could have emptied the Free FIFO, so supply the Free FIFO with fresh shared memory segments.

freeBufferToTrashBuffer()

If there are more segments in the Free Buffer than the configured maximum, segments are moved to the Trash Buffer until this maximum is reached.

timeoutTrash()

Segments in the Trash List are traversed and segments older than a configurable timeout are deleted and the memory given back to the system.

3.3.3 Sleep

In order not to burn CPU cycles, the Main Loop is sleeping here.

3.4 Watchdog Process

The Watchdog Process ensures that the system stays operational. Every process that would like to be monitored by the Watchdog Process announces this by using the respective watchdogInitHeartBeat() function of the watchdog client library. This function takes a restart command as argument. This command (most likely a shell script) will be run as soon as the Watchdog Process recognizes that the respective process is hanging or crashed by not sending heartbeats anymore. For a Reader Plugin the watchdogInitHeartBeat() call happens automatically if the needed information is passed to announceReader().

It is very important that the monitored process regularly calls sendHeartBeat(). This will write the current time into a timestamp field of the adminpage corresponding to that process so the Watchdog recognizes that this process is still living.

The Watchdog Process regularly checks that all timestamps are up-to-date. If one timestamp is too old, it's corresponding process is either hanging or crashed. Watchdog tries to kill that process and restarts it using the supplied start command.

3.5 Statistics Watcher

The Statistics Watcher prints out the information found in the statistics section of the adminpage.

3.6 Configuration

The online configuration tool allows to change the configuration at runtime by feeding new configuration data into the adminpage. Some entries are not allowed to change at runtime: for example if the shared segment size would be changed there would be a transition phase with two sizes of segments. This is error prone, would complicate the whole framework and is therefore not allowed.

Chapter 4

Implementation Details

I'm describing here technical details of the more interesting parts of the framework.

4.1 Communication between Processes

As seen in the previous chapter, multiple processes work together to build the framework. It is vital that these processes have a means of communication. I'm using two mechanisms for inter process communication (ipc): for "rare" events like announce or shutdown, the "named pipe" facility of the operating system is used. For "not so rare" events, especially for statistics and the communication with the Watchdog process, accesses to a shared memory segment are used.

4.1.1 Via Named Pipes

Named pipes are a means for non related processes[1] to pass messages. A Named Pipe behaves like an ordinary file and also resides in a directory. In the contrary of a plain file, data written to a named pipe is not written to disk but are held in memory.

Message Format

```
typedef struct message_s {
  int magic;
  int command;
  int sender;
  int memid;
  unsigned long long int byteseq;
} message_t;
```

Listing 4.1: Format used on Named Pipes

All messages sent on the Named Pipe channel have the format specified in listing 4.1. The value `magic` prevents data accidentally written to the Named Pipe from being handled as a command. It always contains the same value defined in message.h: `PACKETMAGIC`. The type of command is specified in the field `command`, the field `sender` specifies the PID (process id) of the sender.

There exist these defined commands:

[1] Processes that have a parent – child relationship created by the parent forking and calling exec are called related. They have additional ways of communication.

Command	Sent by	Answered with	Usage
ANNOUNCEREADER	reader	ANNOUNCEREADER SHUTDOWNREADER	announce a reader
ANNOUNCEWRITER	writer	ANNOUNCEWRITER	announce a writer
SHUTDOWNREADER	reader	n/a	sign off a reader
SHUTDOWNWRITER	writer	n/a	sign off a writer
ANNOUNCEREADER	mgmt	n/a	send adminpage shared memory Id in memid
SHUTDOWNREADER	mgmt	n/a	no more readers allowed
ANNOUNCEWRITER	mgmt	n/a	send adminpage shared memory Id in memid
ADVANCE	reader	ADVANCELATE ADVANCEOK	request the next younger segment
TOHEAD	reader	ADVANCELATE ADVANCEOK	request jump to youngest segment
YIELD	reader	n/a	yield interest in current segment
ADVANCEOK	mgmt	n/a	send new ShmemId in memid
ADVANCELATE	mgmt	n/a	like ADVANCEOK but it took the reader too long to read the last segment, some segments were skipped

4.1.2 Via Shared Memory

The central administration shared memory page is called adminpage[2].

```
typedef struct adminpage_s {
  int tag; // for retrieving after crash, contains adminpage-tag.
  int id;  // id to separate different frameworks. is initialized to
           // the adminpage shmemkey

  // reader section
  reader_t readers[MGMT_MAX_CLIENTS];
  int waiting[MGMT_MAX_CLIENTS];

  // writer section
  writer_t writer;

  // mgmt section
  mgmt_t mgmt;

  // pid of watchdog
  int wdpid;

  // shared data structures & mem keys

  int freeFifoMemKey;
  int readFifoMemKey;

  // configuration space
  config_t config;

  // statistics
  stat_t stat;
```

[2] It used to be 4K large (one memory page) but has grown since, therefore the name

```
} adminpage_t;
```

Listing 4.2: central administration location

As seen in listing 4.2 it is divided into different sections. After the tag which contains a unique value for adminpage shared memory segments for crash retrieval (see chapter 4.4 for more information) and the id, which designates to which framework this adminpage belongs (more than one can run concurrently and have separate adminpages) there is a section for the Reader Processes, for the Writer Process, the Management Process, configuration of the Management Process and room to store statistical information. In here are also stored information like the pid of the watchdog and the shared memory Id's of the two wait free FIFO's.

The configuration tool works by overwriting some of the values in the config structure, the statistics tool by reading the values of the statistics section.

```
typedef struct reader_s {
  unsigned int fd;           // fd of named pipe to this reader
  char fifopath[AP_STRLEN];  // and it's path
  int pid;
  int restart;               // does this reader like to be restarted from ↩
      the watchdog?
  char command[AP_STRLEN];   // if yes, use this command...
  unsigned int currentBlock; // current segment, for advance
  time_t lastBeat;           // time of last heartbeat for watchdog
} reader_t;
```

Listing 4.3: reader part of the central administration location

The watchdog library sets the lastBeat value to the current time. The Watchdog Process checks these times.

4.2 FIFO Algorithm

For a lock-less memory cycle, some decoupling device is needed to transfer data over the border of the two responsibility domains. This mechanism has to provide write access to one and read access to the other process. The opened path between the RD's is one way. The accesses can happen simultaneously, no locking is allowed in the algorithm.

The FIFO acts as this interface between the RD of the Writer and Management Process.

To grant both processes access to the data structures of the FIFO, not only the data segments it administrates but also the data structures have to reside in shared memory.

This leads to problems:

- In SYSV shared memory, every process attaches the memory at a different position in the address space. It's hard to work with linked lists, as pointers have to be relative to the start of the address segment. To follow a pointer one has to add the relative pointer to the segment start and access the memory there. This is not an atomic operation anymore and would require locking to guarantee consistency.
- Shared memory segments can't grow
- There is no memory management (like malloc and free) on the shared memory Segments

Therefore an algorithm using an array has to be employed, which limits the maximum fill level of the FIFO.

4.2.1 Ring Buffer

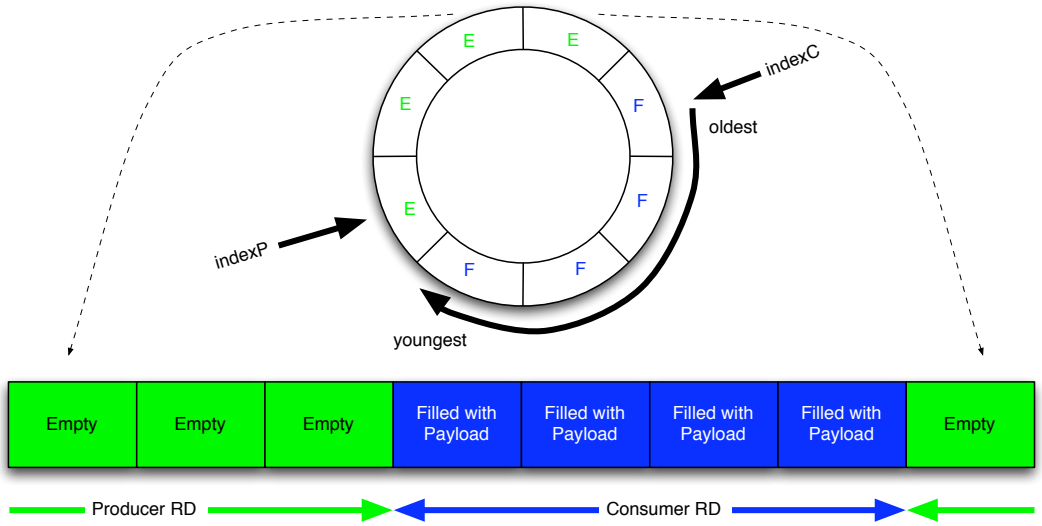


Figure 4.1: A Ring Buffer in an Array

The used algorithm is a straightforward Ring Buffer implementation using an array. The pointers indexP and indexC designate the next production or consumption element in the ring. When moving these indices over the edge of the array a wrap around has to be performed. The value level helps to check for buffer overrun (level >= size) and buffer underrun (level < 0).

```
typedef struct {
    int tag;       // contains FIFOTAG to recognize this
                   // shmem buffer as fifo buffer
    int id;        // contains adminpage memkey to
                   // distinguish different frameworks

    int dirtyC;    // if 1: crashed during consume
    int dirtyP;    // if 1: crashed during produce
    int locked;    // during shutdown fifo is locked
    int memkey;    // memory key of this shared memory segment
    int indexP;    // next production index;
    int indexC;    // next consumption index;
    int size;      // size of the fifo
    int level;     // fill level

    // for the statistics:
    int hiwater;   // highest fill
    int lowater;   // lowest fill
    int threshold; // when to issue a warning
    int type;      // source or sink, for warning
    int warncount; // how many warnings where issued?
    int warnstate; // in warn state right now?

    memblk ary[];  // contains fifo
} fifo_t;

// this pointer is passed to all fifo operations
typedef fifo_t *fifop_t;
```

Listing 4.4: Layout of the FIFO Shared Memory Segment

The layout of the shared memory segment of the FIFO is described in listing 4.4.

4.2.2 Responsibility Domains

Revisiting the Responsibility Domains (RD) of chapter 3, this FIFO represents the interface between two RD's. Therefore also in the array are elements belonging to the producer's RD and elements of the consumer's RD.

How are these RD's separated?

All elements containing data ready to be consumed clearly are in the consumer's RD, all elements which are empty (all elements between indexP and indexC, the element indexP inclusive, element indexC exclusive) are in the producer's RD.

4.3 List processing

4.3.1 Requirements

Lists are used as Free List, Read List and Trash List. These lists have special needs the List Subsystem has to accomplish:

For all lists:

- Initialization
- FIFO functionality with
  - Produce
  - Unconditional Consume

For the Read List:

- Conditional Consume: consume if read
- Buffer Increment: get next newer segment, it won't be consumed
- Return: return read lock on segment
- GetHead: get youngest segment, it won't be consumed

For the Trash List:

- bufferClean: run callback function on all entries older than a timeout

For Crash Recovery:

- Insert in Sequence: searches the right position to insert an element

4.3.2 Storage of the Lists

Since these lists are only accessed from the Management Process, no locking is required. The Reader Plugins can never directly access these lists since they do not reside in shared memory but in the Management Process' private memory.

The elements of these lists store the shared memory identifier of a memory segment. To accomplish the timeout facility, the time has to be recorded when an element is produced, to have access to the sequence number of a shared memory segment this is also stored to allow access to the sequence number without attaching it first.

```
struct listblk_s {
    struct listblk_s *next;          // next segment in list
    unsigned int memid;              // memid of this segment
    time_t time;                     // time of insertion (secs)
    unsigned long long int byteseq; // sequence number of first
                                     // byte in segment
};
```

Listing 4.5: List Segment Entry

4.3.3 Distances

The distance of a Reader Plugin, meaning the number of shared memory segments from the segment the Reader Plugin is currently reading to the youngest segment in the List is used in two places in the framework:

- It flows into the timing calculation of the TrafficShaper
- It allows the Conditional Consume command mentioned above to recognize segments read by all Reader Plugins. The distance of every Plugin is known (it is calculated by the Management Process for every Plugin). The largest of these distances sets a border which designates which blocks are still in use and which are save to be removed.

4.4 Crash Recovery

A crash of one or more processes of the system is always possible, either if there are programming errors (likely) or because of faulty hardware (less likely). It is therefore important to have some means to recover from a crash. The Watchdog Process (see chapter 3.4) tries to restart crashed processes.

Since shared memory segments not attached to any running process are preserved and can be attached from new processes, it is possible to recover from a crash. Every shared memory segment is tagged, is marked with an identifier of it's origin. The most important segment to retrieve is the adminpage. The adminpage contains the shared memory id of the Read- and Free FIFO, which allows to retrieve these and the segments contained in the FIFOs.

Only the memory segments belonging to the Read List have to be retrieved now. The entries of the Free- and the Trash List contain no data and are deleted. Since the administrative data structure of the list is in private memory of the Management Process it is lost in case of a crash of this Process. However it can rebuilt out of the information stored in the shared memory data segments itself:

The field “loc” designates the data structure this segment belonged to, for example to the Read List. In this case, the sequence number can be used to re-insert the segments into the Read List in the correct order.

At the end of the day the framework is back running without having lost data of the last incarnation of the framework. If the socket buffer of the Writer Process is large enough to hold the incoming data during the down time of the framework, no data is lost at all!

```
typedef struct {
  int tag;                        /* marks a buffer shmem segment */
  int id;                         /* id of framework this segment
                                  /* belongs to */
  int loc;                        /* current location eg. FreeFIFO etc. */
  int size;                       /* total size of mem segment */
  int fill;                       /* how much data in segment */
  int memid;                      /* shmem id */
  unsigned long long int seq;     /* sequence number counted in Segments */
  unsigned long long int bytes;   /* sequence number counted in Bytes */
  long long int recTime;          /* how long it took to receive
                                  /* the data (usec) */
  char data[];                    /* payload */
} shmemblk_t;
```

Listing 4.6: Shared Memory Data Segment

Chapter 5

Traffic Shaping

One of the requirements to the Buffering Framework stated that all traffic leaving the framework on a network connection has to be debursted. This is performed by the (optional) Traffic Shaping facility[1].

5.1 Input Traffic Characteristics

The incoming data is already pre-smoothened by the socket buffer. Very short spikes of data are absorbed, the traffic pattern is more even but the input characteristic is still irregular. The shaper-generated data pattern should be as regular as possible.

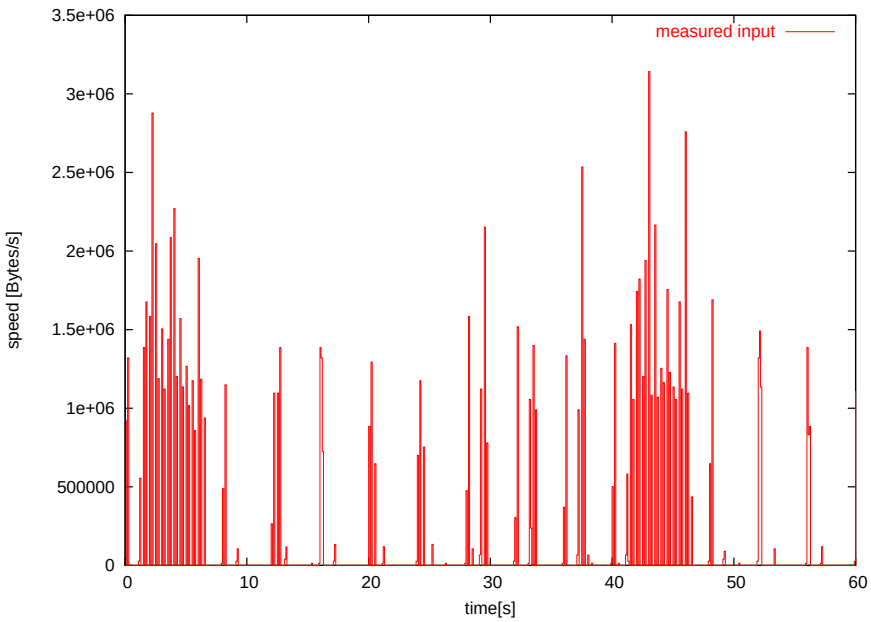


Figure 5.1: Input Traffic

[1] Optional because for locally performed tasks, eg. storing to disk no shaping should be made

5.2 Output Traffic Speed Calculation

Following must be true so the buffer is in equilibrium:

$$t_o = t_i$$

t_o is the time used to write one data segment, t_i time to read one data segment.

It is not necessary that this condition holds over short time though. Therefore the framework uses a low pass filtered version of the sampled data as smoothed version.

The samples of t_i are done every second. The sample is taken from a average over the last four elements written into the buffer. The input traffic is very spiky. If one samples without averager, the risk is high that only spikes are sampled and the average is way too high.

The filtered version t_f is calculated as following:

$$t_f = \sum_{j=1}^{n} t_j * c_j$$

holds.

with t_i as the historical input rate i seconds ago and c_i the i-th filter coefficient. n is the degree of the filter.

In the plots presented here these filter coefficients are set to:

$$c_j = \begin{cases} 20 & 1 \leq j \leq 10 \\ 5 & 11 \leq j \leq 70 \\ 1 & 71 \leq j \leq 100 \end{cases}$$

I'm using a traffic characteristics like in figure 5.3 to test the filter response. The result can be seen in figure 5.4. The characteristics is reproduced by a NetFlow replay tool. It sends the captured data and waits the time between two captured packet arrivals. This pretty well simulates the given input. As a challenge for the framework between 100 sec and 200 sec times between two arriving packets are divided by ten, therefore the arrival rate is ten times faster.

Only using the low pass filtered version with an input like 5.3 reveals two problems:

- The filter is too inertial, the maximum input speed is not reached until 200 sec (I'm using a 100 elements filter, sampling rate for the filter is one element per second).
- After processing a burst of data, $t_o = t_i$ holds. This means that the buffer fill ratio b of the Read buffer stays high, the buffer won't be emptied.

The buffer fill ratio b is defined as

$$b = b_f / b_s$$

with b_f as number of filled segments of the Read List unread by this Reader Plugin and b_s the size of the Read List. b_f is calculated as the distance of the current memory segment to the youngest memory segment.

To allow the TrafficShaper to filter out bursts without loosing packets because of buffer overflow the buffer fill ratio flows into the speed calculation as a feedback factor:

$$t_o = t_f c(b)$$

where $c(b)$ is defined as

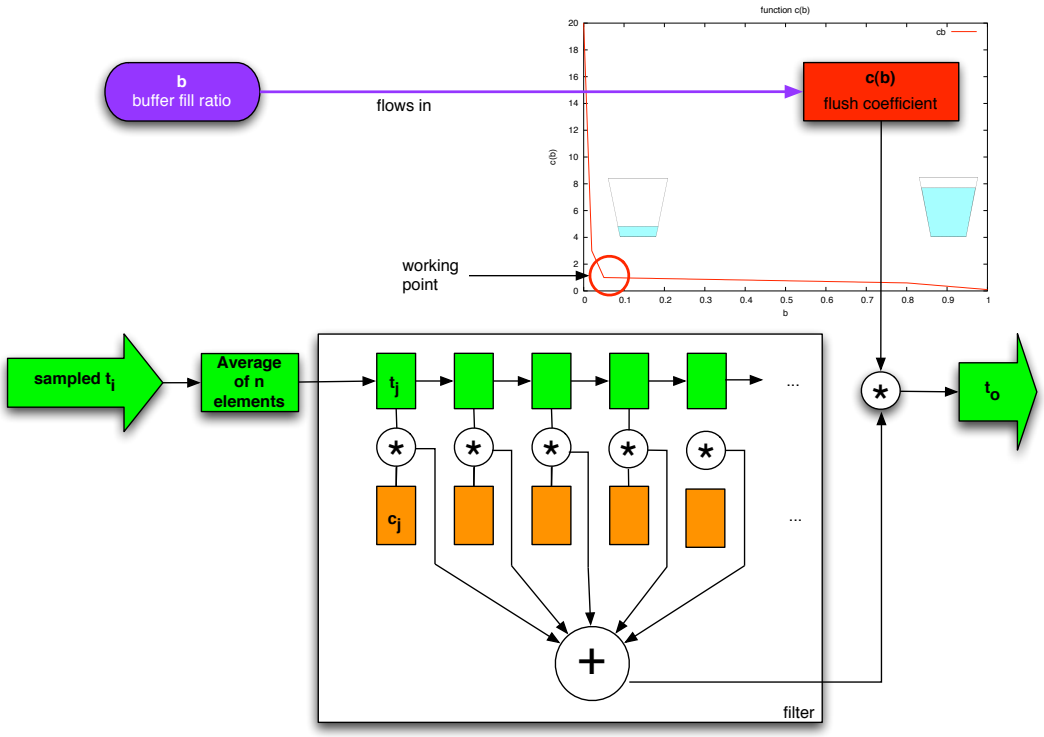


Figure 5.2: Block Diagram of the Output Time Calculation

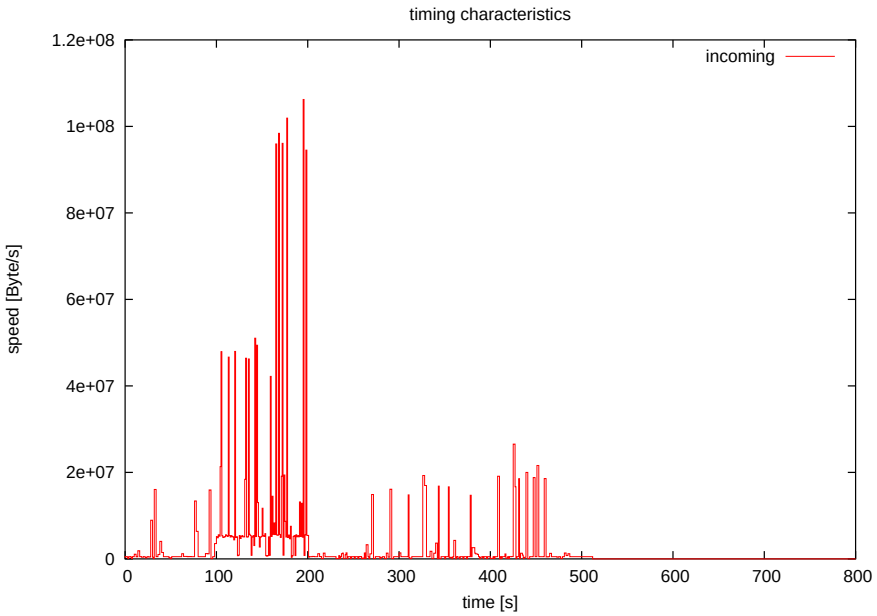


Figure 5.3: Test Input Traffic, 10 times more traffic between 100 sec and 200 sec

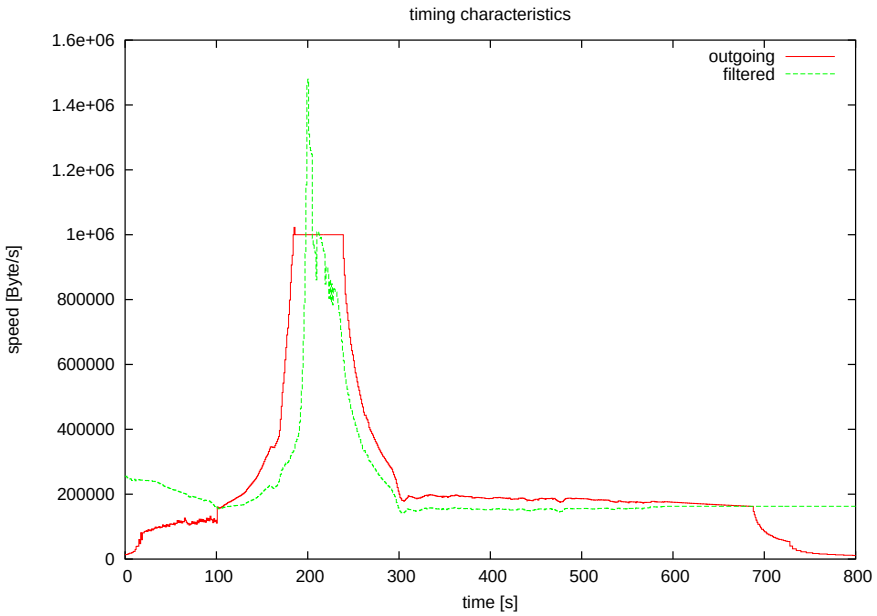


Figure 5.4: Test run with above input showing the filtered input speed and the calculated output speed

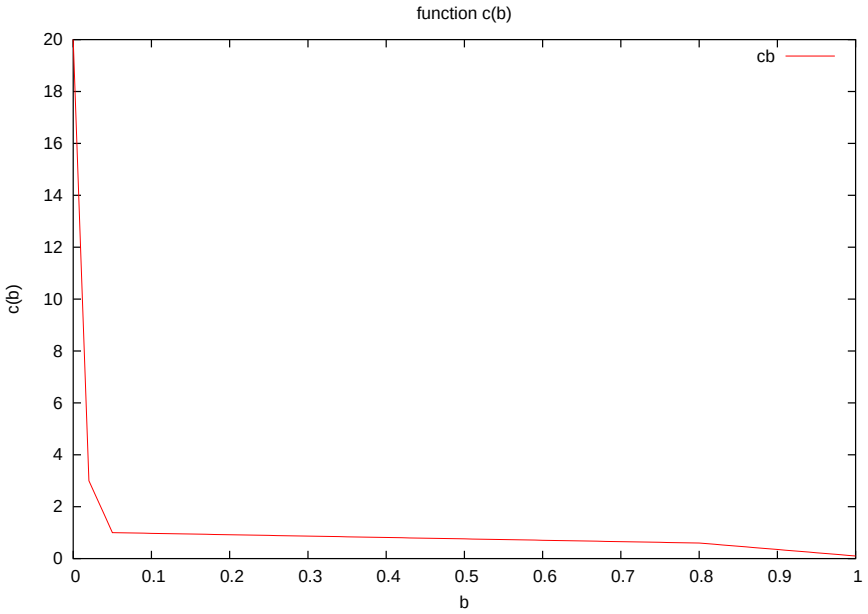


Figure 5.5: c(b)

$$c(b) = \begin{cases} 20 - 850 * b & b \leq 0.02 \\ 3 - 66(b - 0.02) & 0.02 < b \leq 0.05 \\ 1 - 0.53 * (b - 0.05) & 0.05 < b \leq 0.8 \\ 0.6 - 2.5 * (b - 0.8) & \text{otherwise} \end{cases}$$

As stated above the different t values are not measured in Bytes/s but in μsec/segmentsize. It's measured how long it takes to read one data segment. Therefore the parameter values are inverse to what one assumes thinking with speed values. Smaller values in this notation always mean faster sending.

The function for $c(b)$ tries to hold b between 2% and 5%. Smaller b values are punished with a $c(b)$ much larger than 1. The TrafficShaper sends out less packets and b will grow. With a b bigger than 5%, the outflow is boosted and b will go back to 5%. A b bigger than 80% is treated as an emergency. The buffer will amplify the outflow much stronger until at 100% $c(b)$ is 0.1, therefore the outflow is ten times bigger than the filtered input. One could omit the function segment between 0.2% and 0.5%. This would work but results in a jaggy output traffic because the high slope amplifies the inevitable changes in b very much.

The reason why the ideal position is between 2% and 5% is that the estimated output speed could be a little bit too fast. If the buffer would be holding only one element and the data would be sent faster than new data came in, holes would appear in the output traffic.

Dependent on the number of concurrent processes in the system the framework is running on the sleep times cannot be guaranteed. This leads to the situation that the equilibrium is not reached at 5% as planned but much later. Here further research is needed (see chapter 10).

The effects of the parameter can be seen very well in the above plot 5.4. As visible in 5.7 between 0 and 100 seconds b is bigger than 1. Therefore in 5.6 b is slowly growing to 5%. At 100 sec b starts to rise very fast therefore $c(b)$ drops below 1 and enforces the outflow of the data. After the burst is finished after 200 sec b recovers slowly. The buffer tries to get ridden of the surplus data of the burst. After 500 sec the input traffic stops and the buffer fill level b falls rapidly until at 700 sec the buffer is empty again. At 700 sec the buffer fill level falls below 5% and a little bit after below 2%. This can be seen clearly in figure 5.6.

These plots have been made a filter with 100 elements, the first 10 elements were set to 20, the next 60 to 5 and the rest to 1. This configuration is not ideal because the surplus data of the burst is only flushed out slowly. One has to experiment with the filter coefficients to find the best configuration for a certain traffic characteristic.

Also the parameters of the Traffic Shaper (like the function $c(b)$) will have to be tweaked for the final installation of the framework to accommodate with differences of network load, CPU load etc.

5.3 Rate Limiting and Output Traffic Generation

The Reader Plugin programmer can use the TrafficShaper by providing a callback which performs the needed Plugin action.

This callback is called for every packet in regular intervals t_w. This interval is dependent from t_o, the time available to write the whole segment:

$$t_w = \frac{t_o * s_p}{s_b}$$

with s_p the size of a data packet and s_b the size of a whole data segment.

The TrafficShaper measures the time it takes to get the data, to calculate the shaper parameters and to call the callback. It then waits the rest of the calculated time. Since Linux has a minimum granularity for waiting of about 10 ms, shorter wait times will be round up to 10 ms. Therefore

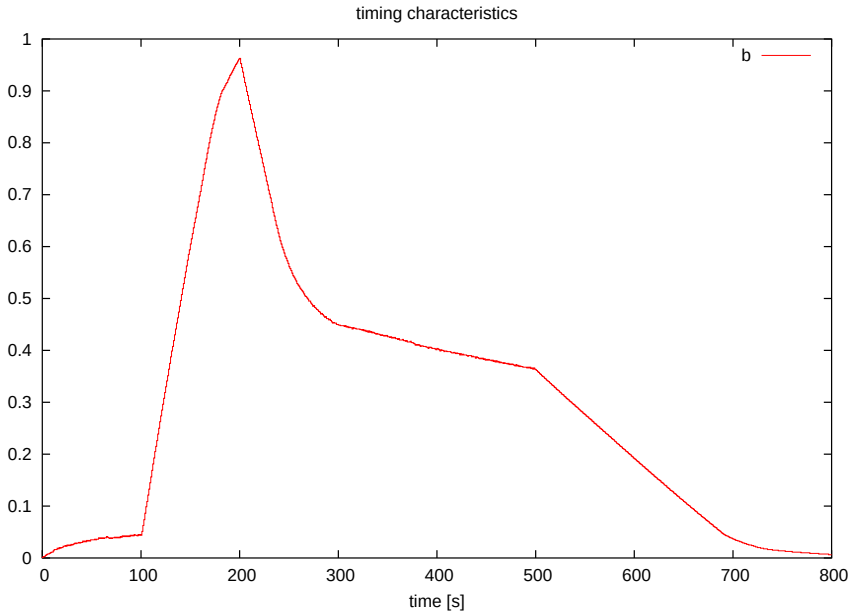


Figure 5.6: Buffer Fill Level

wait times shorter than 10 ms are accumulated until 10 ms are reached. Since the slept time is not exactly the time passed to the system call as parameter, the slept time is also measured and flows into the calculation of the next cycle.

The Rate Limiter sets a lower bound to the waiting time. This prevents the TrafficShaper to call the callback function too often and so limits the traffic speed. Picture 5.8 shows the output generated by the TrafficShaper with a rate limit set to 1'000'000 Bytes/s.

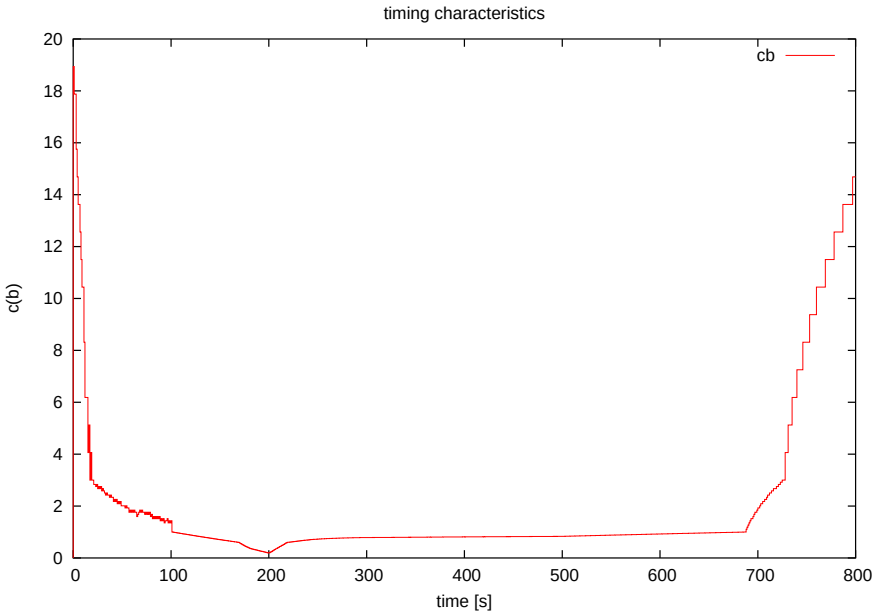


Figure 5.7: Buffer Coefficient

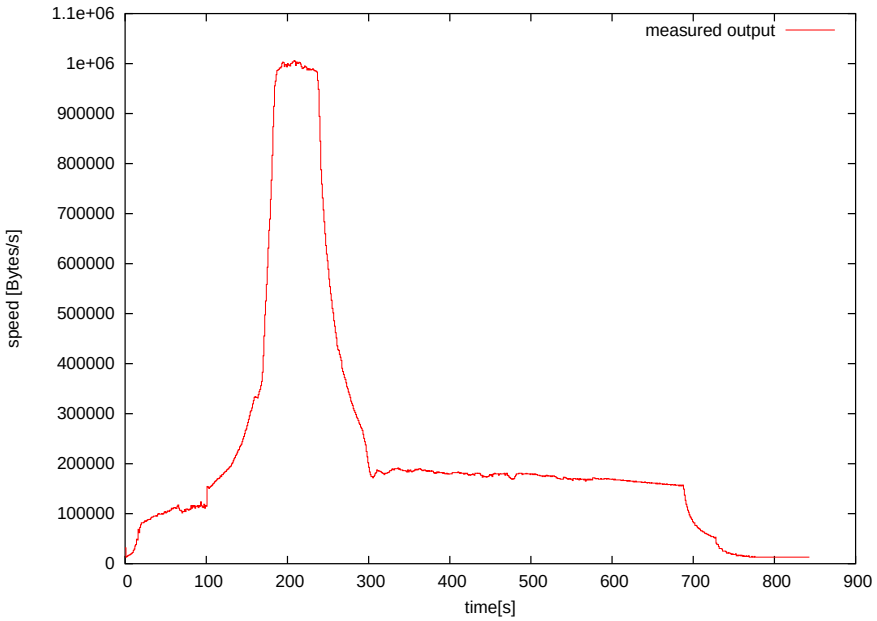


Figure 5.8: Generated Output with Rate Limiting set to 1’000’000 Bytes/s

Chapter 6

How to write a Plugin

There are two possibilities to write a Plugin dependent on the application the Plugin should fulfill:

- A “forwarding” Plugin which does resend the data over the network can use the Traffic-Shaper. This code will do all duties for the Plugin, especially call advance() when needed and serve the Watchdog Process.
- A “local” Plugin which does not resend the data over the network won’t use the Traffic-Shaper. The above mentioned duties have to be done by the Plugin itself, therefore the code is more complicated.

There’ll be a skeleton of a Plugin for both variants in the next sections.

Common to both approaches is that they consist of calls to functions provided by libread.

The data structure used to store a packet in the segment is described in bufferpacket.h.

```
typedef struct {
    unsigned int size; // size of payload
    char data[];       // payload
} bufferPacket_t;

typedef bufferPacket_t *bufferPacketp_t;
```

Listing 6.1: bufferpacket.h: buffer PDU

The Writer includes the received UDP packets as payload using the data structure defined in udppacket.h.

```
typedef struct {
    unsigned long long int seq; // sequence number: bytes since start
    struct sockaddr_in source;  // receiver of packet
    struct sockaddr_in target;  // sender of packet
    struct timeval captureTime; // time of capture
    unsigned int size;          // size of data
    char data[];                // payload
} UDPPacket_t;

typedef UDPPacket_t *UDPPacketp_t;
```

Listing 6.2: udppacket.h: UDP packet PDU

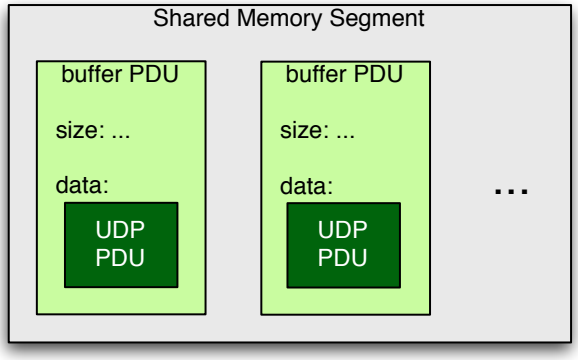


Figure 6.1: Encapsulation of PDU's in the Shared Memory Segment

6.1 Forwarding Plugin using the TrafficShaper

```
1  void samplisend(bufferPacketp_t packet, int result) {
2      /** this is the customizable part,
3          below is an example customization:
4
5      static int i = 0;
6
7      UDPPacketp_t nfPacket = (UDPPacketp_t) packet->data;
8
9      resendUDP(nfPacket, recipient);
10
11     if ((i++ % 100) == 0) {
12         printf("%d\n", i);
13     }
14
15     end of customizable part */
16     return 0;
17 }
18
19 int main(int argc, char **argv) {
20
21     int ret;
22     whandler_t whandler;
23     char *fifoPath = "/tmp/nerf";
24     char *restartCommand = "ts_skel";
25
26     while(1) {
27         while ((ret = readerAnnounce(fifoPath, restartCommand,
28             &whandler)) == -2) {
29             sleep(1);
30         };
31
32         if (ret == -1) {
33             exit(1);
34         }
35
36         if (readerTrafficShaper(1000000, samplisend) < 0) {
37             exit(1);
38         }
39     }
40 }
```

Listing 6.3: Skeleton of a Reader Plugin using the TrafficShaper (simplified: look at samplireader.c in the software distribution for a running example)

Lines 1 to 17 define the callback that will be called by the TrafficShaper regularly. Line 22 defines the watchdog handler variable. In this variable watchdog related information is stored. Line 26 starts a endless loop, though it is not required that every Plugin lives endlessly.

Lines 27 to 30 try to announce this Plugin at the Management Process. The argument “fifoPath” designates the directory where the named pipes reside. The argument “restartCommand” tells the Watchdog Process how to restart this Plugin. The watchdog client library stores this information in the watchdog handler “whandler”. The announce function saves a copy of this handler in behalf of the TrafficShaper so it has not to be passed to the TrafficShaper call in line 36.

It is possible that the connection to the Management Process can’t be established, for example because the Management Process is not running at the moment because of a recent crash and is restarted by the Watchdog Process in any moment. The function readerAnnounce will return ADVANCENOMGMT in this case and after one second, another try is started.

Lines 32 to 34 exit the program in case of another error during announce (ANNOUNCEERROR).

Lines 36 finally starts the TrafficShaper passing the bandwidth limitation value of 1’000’000 Bytes/s and the callback function “callback”.

The TrafficShaper now starts to regularly call the function samplisend. The first argument of the callback samplisend is the received UDP packet. The second argument tells if a data loss has been detected between the last and the current invocation: Either its READOK, READSTALE (data segment already reused or no more memory available) or READSEQ (sequence check failed in Management Process).

Normally the TrafficShaper is in an endless loop and the function does not return but there are cases that the function returns: if the Watchdog Process restarts the Management Process the Plugins must re-announce to make themselves known to the new Management Process. The Watchdog Process sends a SIGHUP signal to the Plugins. The TrafficShaper finishes reading the current segment and then returns with a positive return value. This is the reason for the outer endless loop starting at line 26. If the return value is negative, something non-fixable happened in the TrafficShaper and the program should abort.

6.2 Local Plugin

```
1  // signal handler
2  static void sighupHandler(int signal) {
3      myreconnect = 1;
4  }
5
6  // process data
7  inline int process(bufferPacketp_t packet, int result) {
8
9      /** this is the customizable part,
10         below is an example customization:
11
12     static int i = 0;
13
14     UDPPacketp_t nfPacket = (UDPPacketp_t) packet->data;
15
16     saveToDisk(nfPacket, fd);
17     saveStatsToDisk(nfPacket, fp, &nr, &start);
18
19     if ((i++ % 100) == 0) {
20         printf("%d\n", i);
21     }
22
23     end of customizable part */
24
25     return 0;
26 }
```

```
27
28 int main(int argc, char **argv) {
29     int ret;
30     struct sigaction sa;
31     whandler_t whandler;
32     bufferPacketp_t rec=NULL;
33     struct timeval wait;
34     int len;
35     int loss;
36
37     fd = saveToDiskOpen(dumpfile, 0);
38     fp = saveStatsToDiskOpen(statfile, 0, &start);
39
40     while(1) {
41
42         while ((ret = readerAnnounce(path, restartCommand, &whandler)) ↩
               == -2) {
43             sleep(1);
44         }
45
46         if (ret == -1) {
47             exit(-1);
48         }
49
50         sa.sa_handler = sighupHandler;
51         sigemptyset(&sa.sa_mask);
52         sa.sa_flags = 0;
53         sigaction(SIGHUP, &sa, NULL);
54
55         while ( (ret = readerAdvanceToHead()) == READANODATA) {
56             wait.tv_sec = 1;
57             wait.tv_usec = 0;
58             select(0, NULL, NULL, NULL, &wait);
59         }
60
61         while(1) {
62             ret = readerGet(&rec, &len);
63
64             if (restartCommand != NULL) {
65                 watchdogSendHeartBeat(&whandler);
66             }
67
68             if (ret <= 0) { // error occured during readerGet
69
70                 if (ret == -2) {
71                     exit(1);
72                 }
73
74                 if (myreconnect) {
75                     // finished reading segment now go back for reannounce
76                     break;
77                 }
78
79                 if (ret == -1) {
80                     loss = READSTALE; // data lost: segment no more there
81                 }
82
83                 if ( (ret = readerAdvance()) == READGSEQ) {
84                 // segment loss detected in sequence number
85                     loss = READGSEQ;
86                 } else if (ret == READAERR) { // error in communication
87                     wait.tv_sec = 1;
88                     wait.tv_usec = 0;
```

```
89                      select(0, NULL, NULL, NULL, &wait);
90
91                  } else if (ret == READANODATA) { // no data
92                      wait.tv_sec = 0;
93                      wait.tv_usec = 100;
94                      select(0, NULL, NULL, NULL, &wait);
95                  }
96              } else { // data is ok.
97                  process(rec, loss);
98                  loss = READOK;
99              }
100         }
101     }
102 }
```

Listing 6.4: Skeleton of a Reader Plugin not using the Traffic Shaper

By not using the functionality of the TrafficShaper all work has to be done in the Reader Plugin itself. This has the advantage that one is in full control of the timing and of the advance-calls. For example one could implement a sampling routine by only reading the first UDP PDU of every segment

This example code does also use a callback so it's similar to the example of using TrafficShaper. In principle, one can write a new local Plugin by changing the samplireader function.

This code is (except for the missing traffic shaping mechanism) similar to what the TrafficShaper is doing internally.

This Plugin lacks some minor details (command line argument interpretation, log file handling) and can be found complete in the software distribution. The skeleton implements a disk logging functionality. The payload of the UDP packets is written to disk together with a statistics file.

In the next few lines the structure of this skeleton is explained. I'm starting at the main function:

Lines 30 and 31 open some files for the functionality implemented in process(). In line 33, the endless loop starts that guarantees that this Plugin reannounces at the Management Process when a SIGHUP is received in case the Management Process is killed by the Watchdog Process and the Plugins have to reannounce.

Lines 35 and 36 try to announce this Plugin at the Management Process. Again the parameter "path" describes the path to the directory where all named pipes reside. This time, the parameter "whandler" is of importance, as this Plugin has to serve the Watchdog itself. Therefore the information in "whandler" will be used later.

Like in the example above, if the communication to the Management Process cannot be established, this Plugin keeps trying. If any other error happens, the Plugin quits.

As mentioned the watchdog sends a SIGHUP to all Reader Plugins when it kills the Management Process to let the Plugins reannounce itself at the new Management Process. The function readerAnnounce sets a function handler for SIGHUP, but this Plugin cannot use this handler as it is closely related to the TrafficShaper which is not used in this example. Therefore the existing SIGHUP-handler is overwritten in lines 43 to 46.

Lines 48 to 52 call the function readerAdvanceToHead to get an initial data segment to read from. If there is no data to read, the Plugin waits one second and retries.

At line 54 the main processing loop starts, the Plugin will spend most time in here.

Now in line 55 readerGet() tries to get one data packet. Parameter rec is a pointer to a buffer-Packet structure as defined in line 25, len describes it's size and is 0 at start. The readerGet function dynamically increases this rec buffer size if a packet is too large and accordingly adjusts len. The variable ret saves the result of this get operation.

Lines 57 to 59 serve the watchdog. The condition $restartCommand \neq NULL$ shows if the watchdog is switched on. The call to watchdogSendHeartBeat() will perform the needed actions. The information it needs to do so is fetched from the watchdog handler whandler which is returned on announce.

Line 61 checks if any error happened during readerGet(). Possible conditions are READOK for “no error”, READNODATA for “no more data available”, READLATE for “segment invalid, call readerAdvance() for valid data” and READERR for “fatal error”.

Line 63 to 65 terminate the program on fatal error, therefore for all other conditions the Plugin has to call readerAdvance to get to a new data segment.

As mentioned, the Plugin is sent a signal SIGHUP if the watchdog restarts the Management Process. The installed signal handler will set the global variable myreconnect. Line 67 checks this variable. Delaying the check from line 55 to here enables the Plugin to finish reading the data segment it is locked on and afterwards going into “announce” mode when having finished reading this segment.

If readerGet returned READLATE, the attached shared memory segment does not hold the data segment this Plugin advanced to. This Plugin had very long to process this data segment. The Management Process reused this shared memory segment for a new data segment. Therefore the unread part of this segment is not read by this Plugin, therefore a data loss is detected.

If readerAdvance returned READLATE, the data segment this Plugin was locked was not in memory anymore. This means that there is no direct successor to jump to. Using the SoftLimit mechanism described in chapter 3.3 a segment in a save distance to the end of the buffer is chosen and locked to. The data segments in between are not read by this Plugin, a data loss is signaled in line 73.

In line 76 the readerAdvance call is issued. If readerAdvance returned -1, a sequence number mismatch was detected by libread. Some segments in-between are lost. Therefore a data loss is signaled in line 78.

In line 79 is checked if there was a communication error. If this is the case the Plugin waits for 1 sec and retries.

Finally in line 84 is detected that there’s no data segment to advance to. After sleeping 100 μsec the Plugin retries.

In line 61 was checked if there was a problem getting data. The lines 62 to 88 handled the various reasons. On the other side if there was no problem, in line 90 the worker function is called and the loss signaling variable is set back. Now another inner loop cycle starts.

Chapter 7

Testing and Validation

7.1 First Setup: Simulation of NetFlow Data

7.1.1 Setup

To test the framework during the implementation phase a previously recorded trace of NetFlow data has been replayed with the help of an extended version of netflow_replay, originally written by Arno Wagner.

The flow capture tool of the DDoSVax Team not only saves the raw UDP payloads of the NetFlow packets but also saves time stamps for each packet arrival. Using these time stamps it is possible to mimic the traffic characteristics produced by the routers.

The replayed NetFlow packets is sent over the loop back interface to the framework. Attached to the framework is a plugin which is using the TrafficShaper interface to achieve a smooth version of the traffic. The plugin again saves the UDP payload to disk together with a timestamp.

The test for completeness of the data can be done by comparing the input- and the output file, the generated traffic pattern can be extracted from the time stamps.

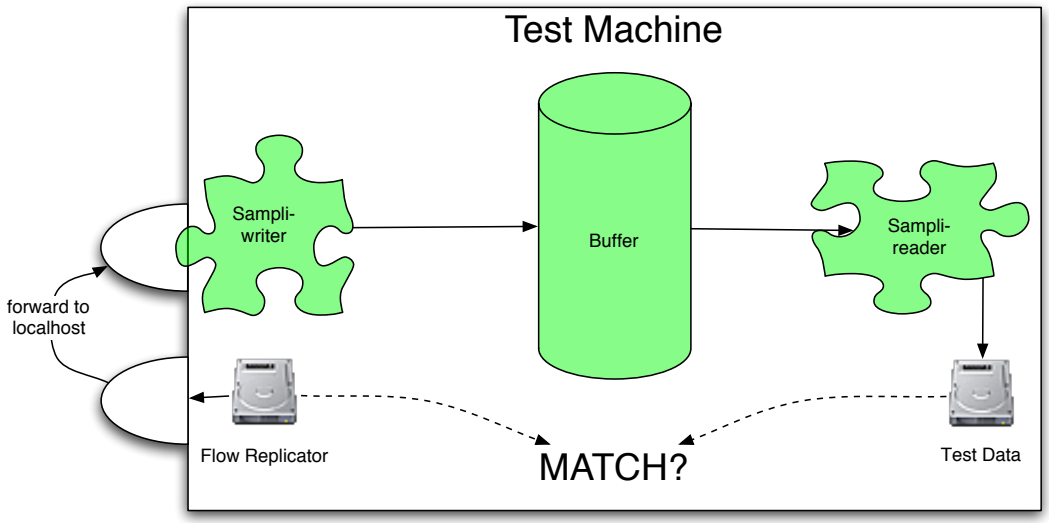


Figure 7.1: Test Setup

7.1.2 Results

The results are already presented in chapter 5 as this test was used as a test case in the development of the TrafficShaper. During the tests no data was lost.

7.2 Second Setup: Real Data at Switch

7.2.1 Setup

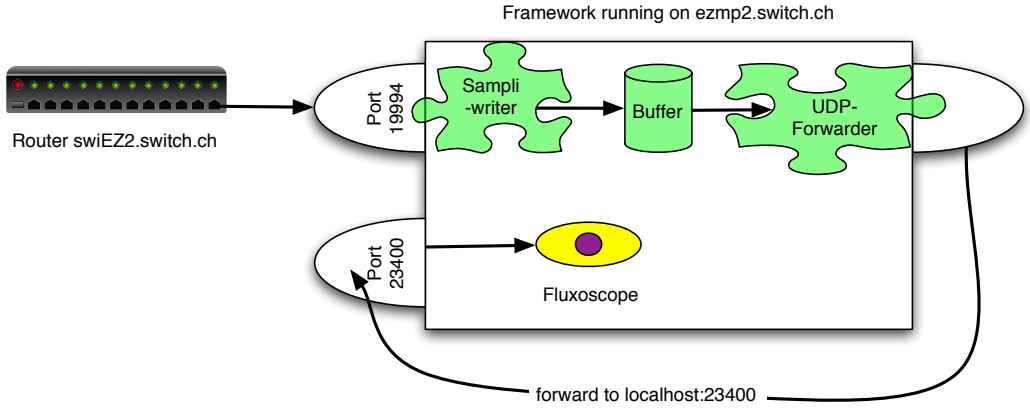


Figure 7.2: Test Setup

The setup of this test was the following:

The router swiEZ2 (which connects the ETH to the Switch Backbone) exports NetFlow Version 7 packets to port 19994 of the machine ezmp2. A Sampliwriter instance reads the data and puts it into the Framework Buffer. The data is then read by the Client Plugin UDPForwarder. This plugin converts the V7 data to V5 data (using existing conversion code from Switch) and resends it to the port 23400 of the same machine using the Traffic Shaper. Here a external discrete component, the Fluxoscope reads the data.

This test is to show that the framework is able to receive real NetFlow data. Until now only captured NetFlow data was used. The machine ezmp2 usually has high load. There are many memory-intensive processes running so from the point of view of resource allocation this is a worst-case scenario for the TrafficShaper (which normally needs a machine with no or only small load so the TrafficShaper is exact and lots of memory).

7.2.2 Results

Figures 7.3 to 7.6 show the captured traces of the experiment run at 7.9.2003 at 17:00. Since this is a Sunday less traffic as usual is expected.

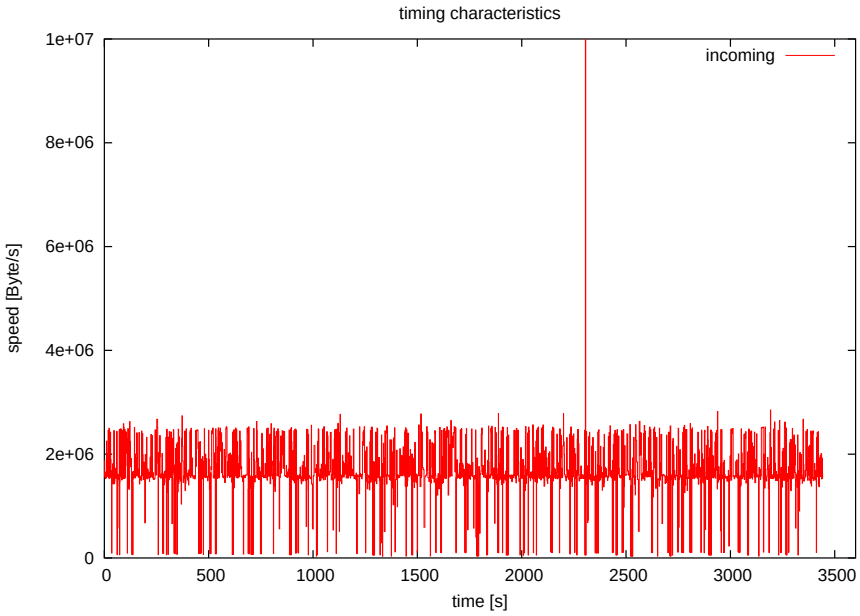


Figure 7.3: Captured data at Switch Backbone

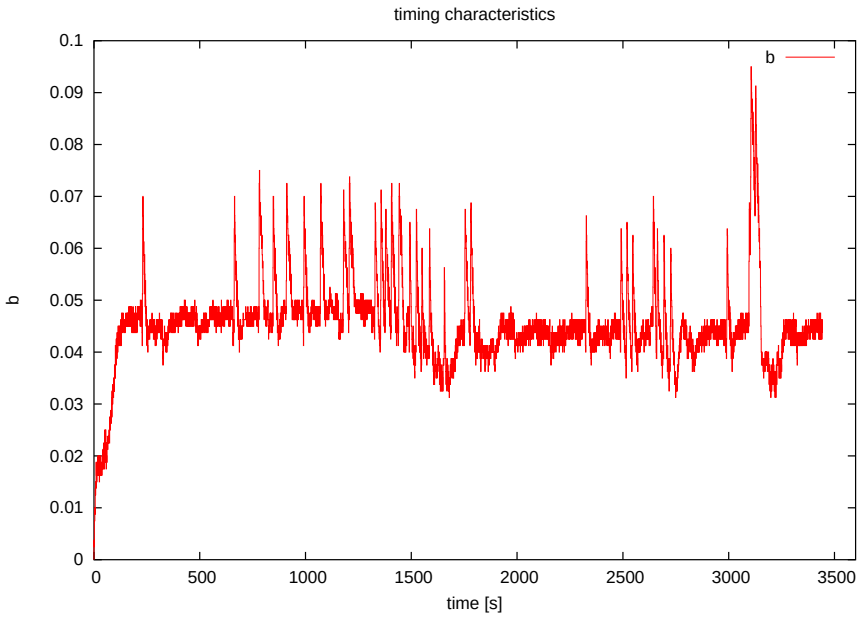


Figure 7.4: Buffer Fill Level b

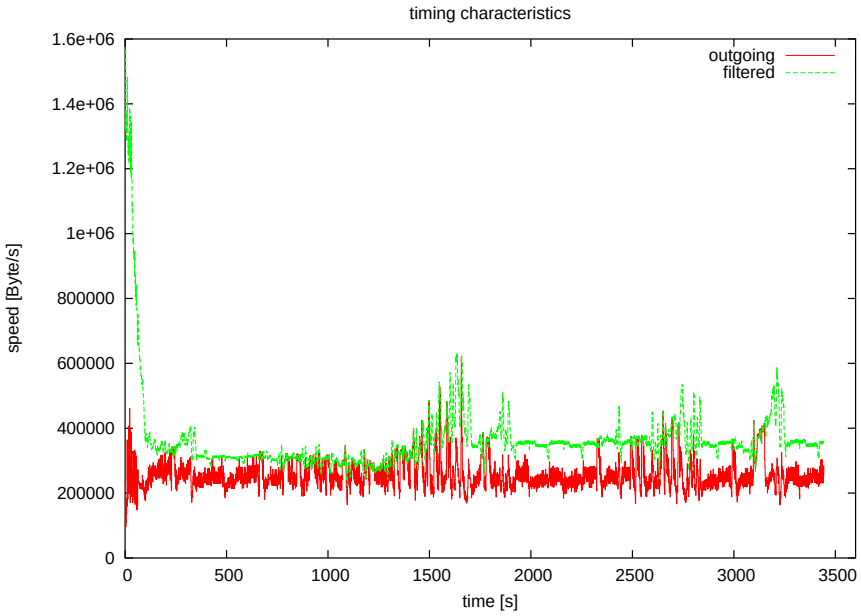


Figure 7.5: Estimation and Output Speed

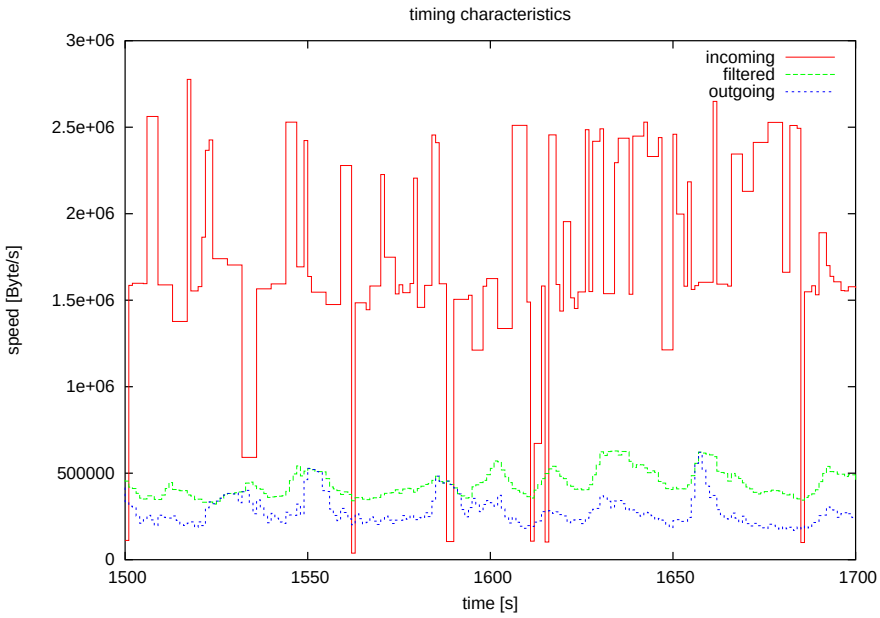


Figure 7.6: Estimation and Output Speed, magnified between 1500 sec and 1700 sec

7.2.3 Discussion

Input

Figure 7.3 shows the highly irregular traffic pattern of the incoming traffic. Since the router the captured data came from is not a border router but connects the ETH Zurich more traffic is generated as expected from a border router. At approx. 2400 sec there is a large spike. The plot is chopped at 10^7 Bytes/s but the spike went up to $9 * 10^7$ Bytes/sec. We will see later how the framework coped with this spike.

Output: Fluctuations

The output plot 7.5 and the b in7.4 shows large fluctuations between 1000 and 1700 sec and at 2500 and 3000 sec. There seem to be fluctuations in the input the overview plot 7.3 cannot explain but a closer look to plot 7.6 which magnifies the region between 1000 sec and 1700 sec and relates the input to the estimate and the output rate shows a correlation between the input data, the estimation and the output data.

The large peak at 1500 sec in the input is followed after some seconds by a peak in the estimation and the output, the dent at 1600 sec is followed by a decrease of the output speed. The difference between the peak and the dent is about 1'000'000 Bytes/s.

The estimation is largely affected by these big fluctuations in the input rate, additionally the buffer fill level rises at these large peaks and also leads to an accentuated outflow.

These two effects cumulated lead to the fluctuations seen in plot 7.5.

As a remedy one could experiment with different filter coefficients in order to flatten the estimation curve or functions for c(b) so that changes in b do not lead to fast changes in the outflow rate.

On the other hand the input has peaks up to $1.7 * 10^6$ Bytes/s where the output only has peaks up to 600'000 Bytes/s. The very high peak at 2400 sec is completely flattened out.

Output: Artifacts in Filter

An artifact of the filter mechanism can be seen in plot 7.5: The filtered value at first is very high and only after 100 sec is near the average inflow. The framework was freshly started for the capture. The first value recorded for the estimate was a spike. With this value the filter was initialized by setting all array elements to that value and it took 100 sec to flush out this spike information.

The filtered values are higher than the output values. This is still an impurity in the sampling for the estimated data and is compensated by the feedback function c(b). If the output is too high the equilibrium is not reached at 5% but earlier. In the course of these tests the averager (figure 5.2) was implemented, without it the estimation was way too high. One can experiment with a larger number of samples to improve the situation.

7.3 Third Setup: UDP Forwarding Setup

This test examines the performance of the TrafficShaper under heavy network load.

7.3.1 Setup

Four machines are acting as routers each sending a recorded trace of the Sobig F attack recorded at the 19. 8. 2003, 14:20 (a section of the data is shown in plot 7.8). The timing and the size of the packets is correct but the payload of the UDP packets is faked (the machines

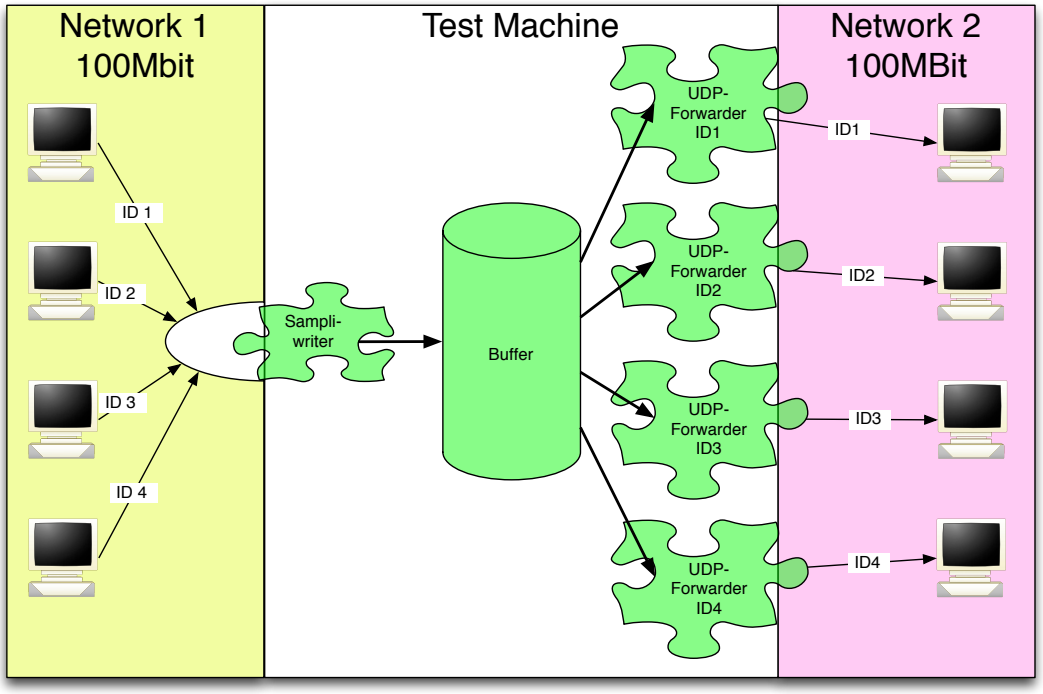


Figure 7.7: Setup of UDP Forwarding Test

used as sender and receiver did not have enough disk space to hold the data files) and replaced by an Id and a sequence number.

The UDP Forwarding Plugins checks the id field of the packet against some command line switch and only forwards matching packets to the configured external discrete component.

On for machines a external discrete component is running that receives the UDP packets and checks for the sequence number. Like this, lost packets are detected immediately. These components also write timestamps into a file for each received packet. This enables the comparison of the input data and the captured data at the receivers.

To minimize packet collisions on the network the server has two network interface cards. The senders are on one net, the receivers on another. I tested both with 100Mbit Hubs and Switches but there was no influence detectable.

7.3.2 Results

One Sender

The first plot, 7.9 shows a configuration where only one sender sends data therefore only the corresponding receiver receives data. The received data looks much more smooth. Input peaks go up to 1'000'000 Bytes/s where the reception of the shaped data only shows peaks to 450'000 Bytes/s. Again fluctuations are visible, therefore the filter parameters definitively have to be tweaked for production use of the framework.

No data was lost during the experiment.

Four Senders with Selective Routing

In another configuration four sender sent data each using an individual id, each plugin filtered for one id and forwarded it to one receiver (selective routing). The total amount of data flowing in

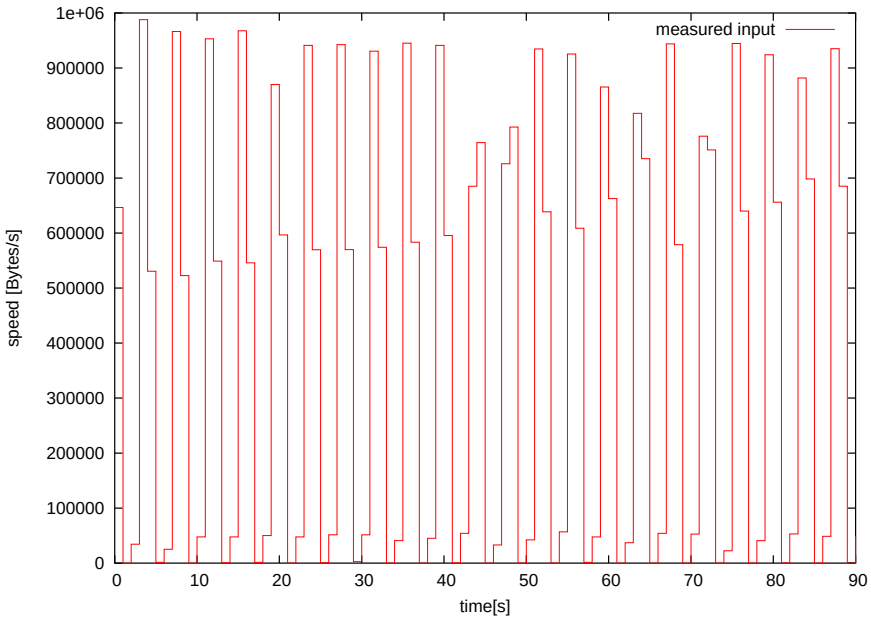


Figure 7.8: Capture of Sobig F attack

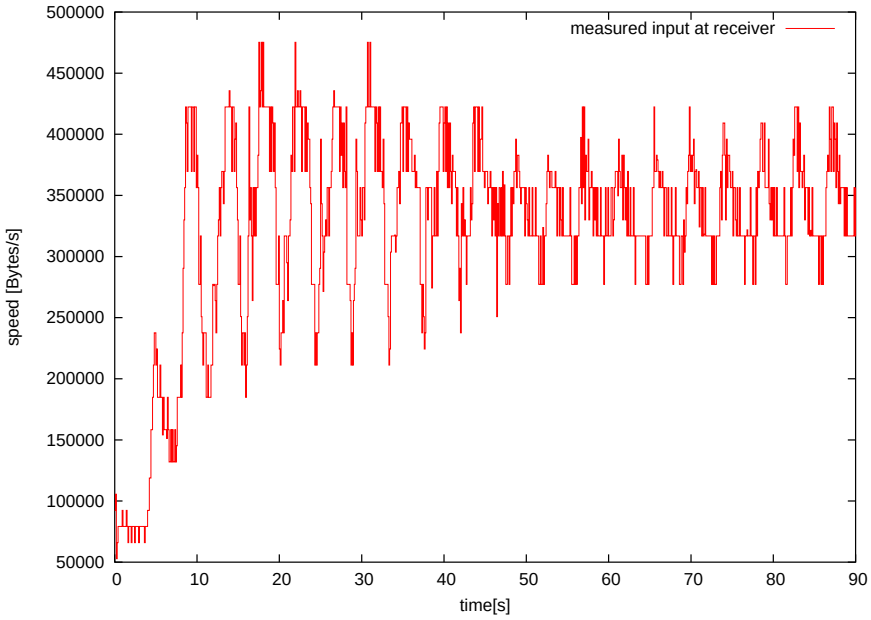


Figure 7.9: Received data, only one id is used

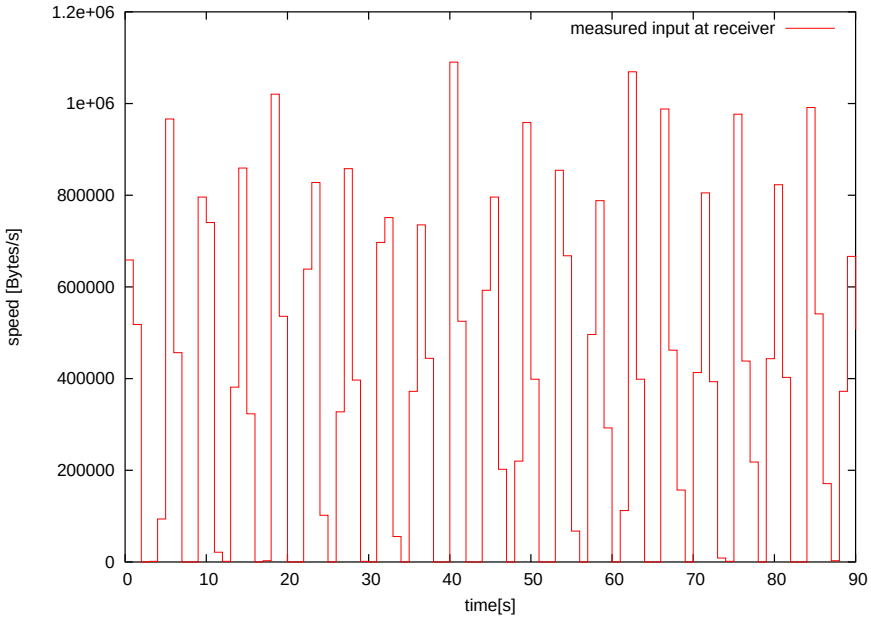


Figure 7.10: Received data, Selective Routing

the framework is now four times larger, therefore about 1'200'000 Bytes/s. The result is shown in plot 7.10.

It is interesting that there are gaps in this plot where no data is arriving at all. The peaks exactly go up to the calculated 1'200'000 Bytes/s. Where are these gaps coming from?

Looking at plot 7.11 shows that the reception of the data in the framework does not happen in round robin fashion but the framework receives multiple packets from the same sender in a row. The packets are entered into the framework in that order.

The Plugins read the packets in that order. The TrafficShaper assumes that all packets for which a callback is called are being sent out. The selective routing mechanism now fools the TrafficShaper because some packets (those with wrong id) are not sent out by a certain plugin. What can be observed in plot 7.10 is traffic shaped for all packets but only a fourth of the packets is really being sent out, the other packets build the gaps between the peaks.

This test resulted in the realization that plugins which perform a selective retransmittion of the data and therefore do not send out all packets need another type of TrafficShaper than the one presented here. The assumption $t_o = t_i$ does not hold anymore.

Only a very small percentage of the packets where lost during these tests.

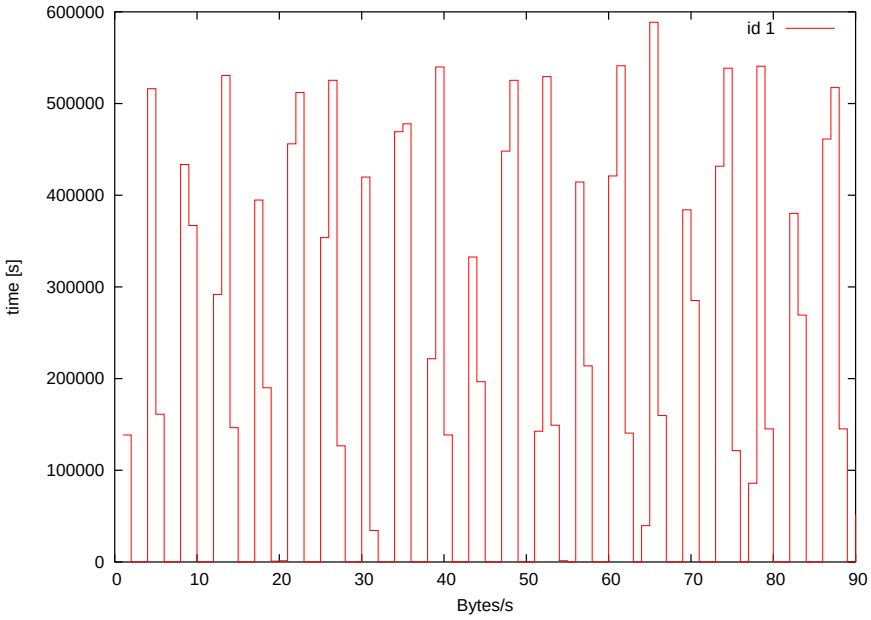


Figure 7.11: Received data on Framework for ID 1

7.4 Fourth Setup: High Load on Gigabit Ethernet

7.4.1 Setup

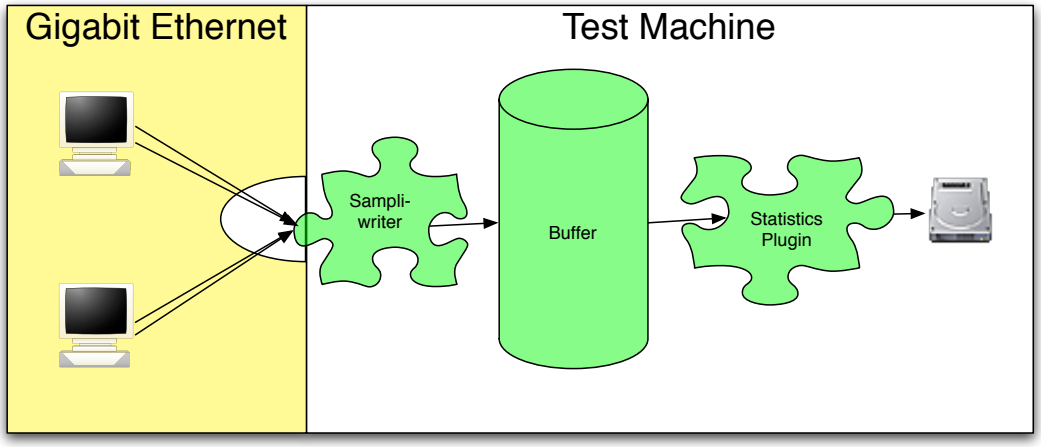


Figure 7.12: Setup of Load Test

Goal of this test is to examine the reaction of the framework under heavy load, especially the fill level of the buffer and the test for possible data segment loss. The test setup was like shown in figure 7.12: Two machines were sending two copies of the Sobig F trace at the same time. This again resulted in average in a load of 1'200'000 Bytes/s. The usage of Gigabit Ethernet was chosen because the routers normally export their data on Gigabit Ethernet, in addition the danger of collisions before the arrival of the data at the framework is minimized and all data has to be stored by the framework.

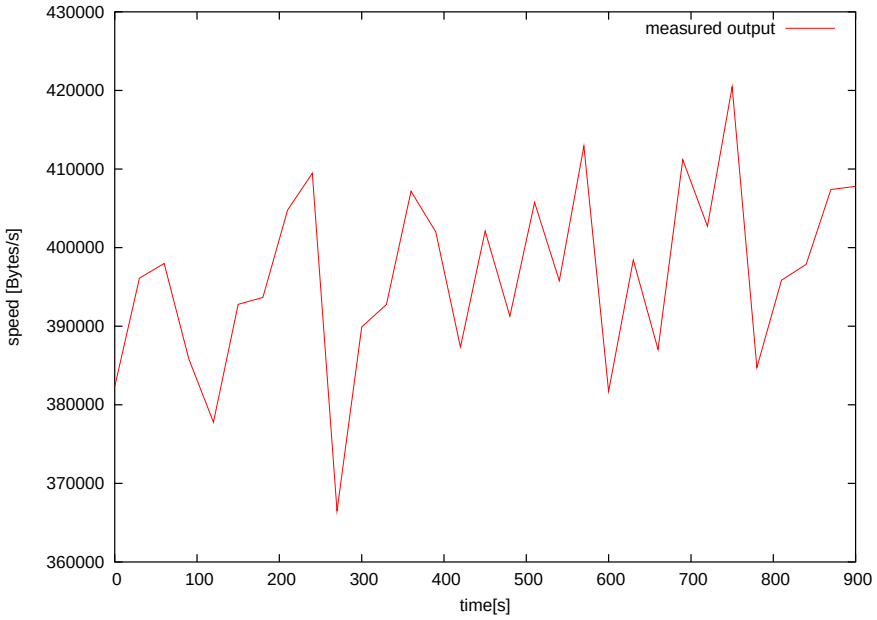


Figure 7.13: Data rate of the used Sobig F plot, averaged over 15 seconds. This data was sent to the framework four times simultaneously, so the actual data rate arrived at the framework is four times the data rate in the plot. The data shows a rising tendency.

7.4.2 Results

It can be seen that there's a relation between the input speed and the buffer fill level. Although there is a rising tendency of the inflow but the filter should adapt and the buffer fill level should stay low. This relationship is not intended and should be subject of further research.

It seems that the point of equilibrium where the inflow is equal to the outflow is not reached at 5% as intended but dependent of the inflow between 50% and 60%. This indicates that the time slept in the TrafficShaper is too long and the additional speed up of c(f) is needed to reach the equilibrium. One can see that the TrafficShaper can empty the buffer during the more quiet period after 700 sec.

Although the framework does not loose any data (as long the buffer is large enough to hold the data), the latency of the buffer rises dramatically.

The produced output 7.15 looks rather smooth and shows the same increase in network output as the original input 7.13. The produced output is four times higher than the input shown in plot 7.13 because the two machines each sent two such streams.

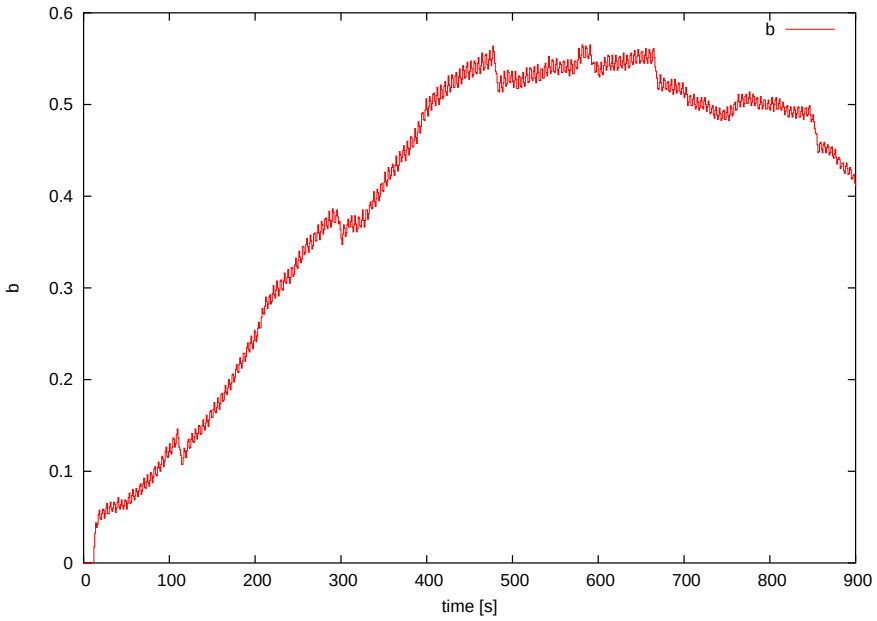


Figure 7.14: Buffer Fill Level

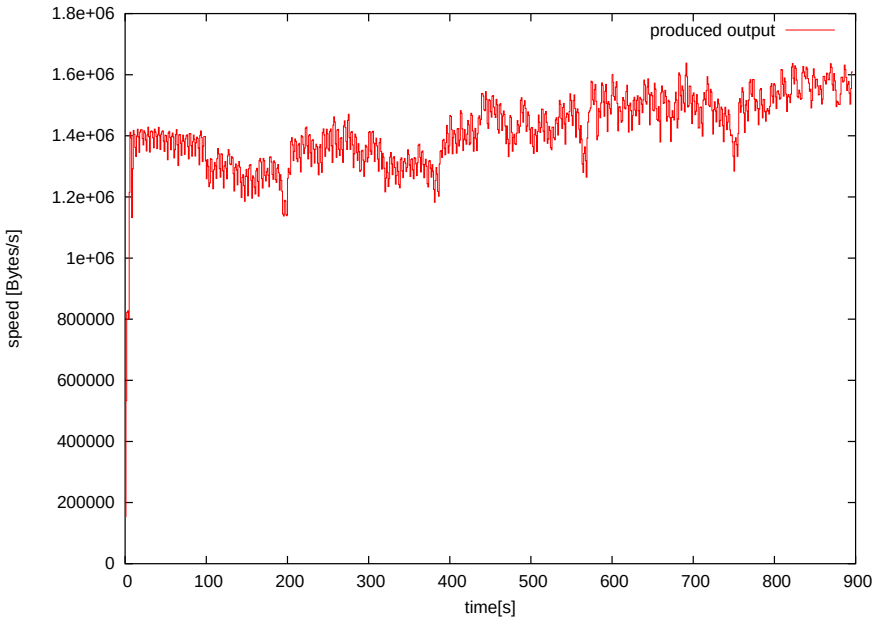


Figure 7.15: Produced Output

Chapter 8

Other Use of the Framework

The framework is not limited to NetFlow data, As a matter of fact every packet oriented payload can be buffered and read by multiple Reader Plugins. Here is a small example:

8.1 Message Broadcast System for Wireless Networks

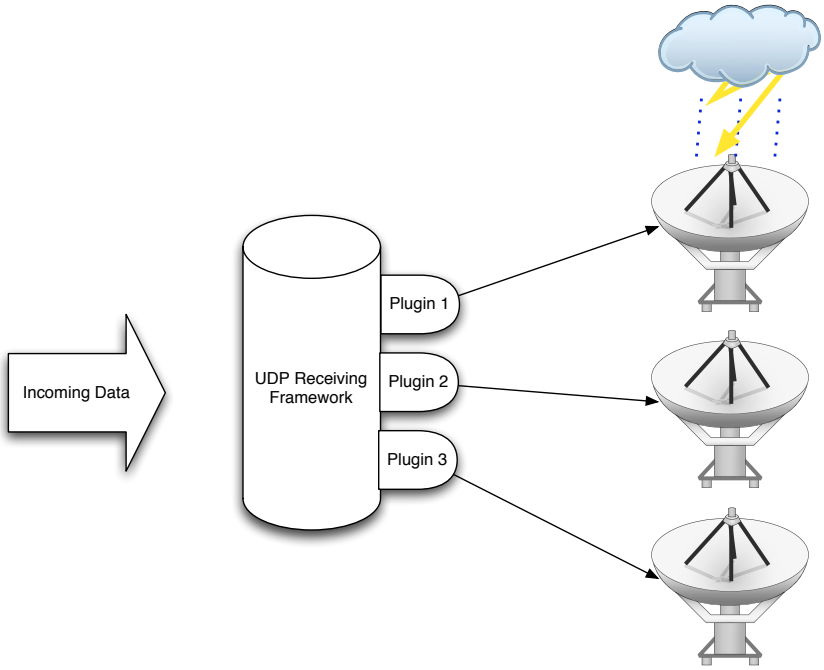


Figure 8.1: Message Broadcast System: Service interruption at the first sender

One can think of a broadcast system for wireless networks: Messages to be sent over four different satellite antennas (each located in a different region) are received at the Writer Process over the Internet. The Writer Process enters the data into the framework, the Reader Plugins read the data and can decide if their antenna have to route that particular packet. Imaging that the Wireless network works with acknowledge packets, it can take a considerable amount of time until this acknowledge arrives (for example if there is a thunder storm in the area of one antenna. The other antennas are not disturbed by the stall as they continue reading the messages.

If the thunder storm doesn't take very long, the data for that antenna is still in the buffer and can be sent out. The reader for that antenna can now catch up.

Chapter 9

Related Work

9.1 CAIDA cflowd

The cflowd suite from CAIDA [4] is taken as example in chapter 2.5.1. The reason why it is not suitable is also mentioned there.

9.2 Cisco Network Data Analyzer

There exists the (non free) Network Data Analyzer suite from Cisco [6]. It includes the NFCollector which is the working horse of the system, collects the data and writes them to disk. The data is then again read and analyzed.

It would be possible to have a plugin framework that works on these files but a plugin could not start analyzing the data before a data file is completely written to disk which would lead to high latency.

9.3 Netfltools

The Netfltools by Andrey Slepuhin [7] also implement a NetFlow receiving server. There is the possibility to process the gathered data including summarizing and logging. The system is rather monolithic, there exists a simple export tool, but no pronounced plugin functionality.

9.4 Crannog Software Netflow Monitor

Netflow Monitor by Crannog Software [8] is a Microsoft Windows based capture and analysis tool. It is currently in use at ETH Zurich but showed a very high memory demand in high traffic situations (like the mentioned Sobig F attack in August 2003).

Chapter 10

Further Work

10.1 Plugins

There now exists a framework for plugins, but the plugins still have to be written. One can think about plugins that:

- monitor the NetFlow data in (near-)realtime and perform several statistics and/or evaluations
- generate plots of the current network traffic

10.2 Surveillance via RRDTool

The possibilities of surveillance of the framework is minimal at the moment. There is only the statistics tool gathering the various values and sending mail in case of a problem. A graphical representation could be created using RRDTool from Tobi Oetiker [5]. This would help the administrator of the system to recognize and trends of buffer fill level etc.

10.3 TrafficShaper

The TrafficShaper algorithm is far from perfect. Due to the fact that Linux is not a realtime operating system the correct timing of the callback calls is nearly impossible. With many running processes the probability of sleeping too long is growing. Despite of corrective measures the state of equilibrium is not always reached at the intended point but at a much higher buffer fill level. This leads to a constant memory wastage and to high latencies.

One can think of two remedies:

10.3.1 Additional corrective measures

There could be some additional factors speeding up the outflow depending on the number of concurrent processes, the load of the CPU or depending on the equilibrium reached.

10.3.2 Port to a realtime operating system

Fiddling with additional speed up factors only hide the problem. Using a realtime operating system the wait times can be followed much more exact. It would be worthwhile to test the framework on a realtime system.

10.4 NetFlow V9

The existing tool chain at SWITCH is based on NetFlow version 5. The current version developed by Cisco is version 9. To unify all protocol versions in the framework data could be converted to NetFlow V9 so all plugins could base on NetFlow V9 independent of the version the routers deliver.

Chapter 11

Summary

11.1 Design and Implementation

The task of this Diploma Thesis was the design and implementation of a UDP framework as a research platform for the DDoSVax Project. The framework will receive UDP packets containing Cisco NetFlow data as payload.

The incoming data is buffered for two reasons: the routers exporting the NetFlow packets do this on an interval of two seconds. Therefore the generated traffic is irregular. On the other hand the stored data is read by Reader Plugins. These processes may pause reading data on several occasions, for example garbage collection, writing a file to disk or by resending the data with TCP. Therefore the framework provides a buffer of substantial size, able to hold several seconds to minutes of incoming data.

The focus during the design of the framework laid on robustness and safety, therefore the framework consists of several simple processes. The algorithms used are wherever possible straightforward. Where this wasn't possible, especially in the fifo component, the code is well commented and the details of the algorithm is explained in this thesis.

It's very easy to to adapt the sample plugins to a certain task. All framework-dependent tasks are hidden by a function library which allows access to the framework through a few simple function calls.

The plugins optionally can use a TrafficShaper which filters the bursts inherent to the incoming traffic and limits the outgoing throughput. This allows to resend the captured data over the network to do further perhaps CPU intensive processing at another machine.

Should a plugin process or even a process that belongs to the framework (like the Management Process or the Writer Process) crash, this is noticed by a Watchdog which regularly checks for hanging or crashed processes.

11.2 Major Challenges

A big challenge was achieving precise sleeps under Linux for the Traffic Shaper. Since Linux is not a realtime operating system no guarantee is given how long a sleep call will really sleep. The granularity of the sleep system call is coarse. Therefore tricks had to be used to circumvent these limitations but the TrafficShaper does not behave perfectly under high load.

11.3 Conclusion

I hope the designed and implemented framework proves usable for further research projects of the DDoS Vax Project.

I would like to thank Simon Leinen and Peter Haag of SWITCH for their help, Prof. Bernhard Plattner for his assistance and especially my supervisors Thomas Dübendorfer and Arno Wagner for their support and feedback during this diploma thesis.

The “Topsy Template” from Lukas Ruf [16] and Tobi Oetiker’s “The not so Short Introduction to LaTeX” [17] proved invaluable for writing this documentation.

Appendix A

Interfaces

For an up-to-date version of the framework interface one should consult the include files of the source distribution. The include files are the authoritative source of information.

A.1 Read Plugin Library

int readerAnnounce()		
Parameter	char *commandpath char *command	path to directory where named pipes reside command to be executed to restart Plugin
Returns	ANNOUNCEERROR ANNOUNCENOMGMT ANNOUNCEOK	fatal error unable to create connection to Management Process announcement successful
The function readerAnnounce announces a Reader Plugin at the Management Process. If this plugin should be monitored by the Watchdog Process, the parameter command holds the command which will be executed when the crash of this Plugin is detected. The parameter command can be NULL. In this case the Plugin is not announced to the Watchdog process. The first command after announce must be readerAdvanceToHead, if the TrafficShaper is not used.		
see also readerShutdown()		

void readerShutdown()		
Parameter		
Returns		
This function signs off from the Management Process		
see also readerAnnounce()		

int readerAdvance()		
Parameter		
Returns	READERR READLATE READSEQ READOK	Communication Problem with Management Process Data block lost Sequence number error, framework lost block Everything is ok
This function locks to the next younger data block than the one currently locked to. readerGet then reads data from currently locked to data block. If the current data block is already re-used by the framework or if the current data block is older than a certain threshold (configuration option “SoftLimit”) the locked-on block is the one at the threshold and READLATE is returned. In this case the locked-on data block is valid.		
see also readerGet(), readerAdvanceToHead()		

int readerAdvanceToHead()		
Parameter		
Returns	READERR READNODATA READOK	Communication Problem with Management Process No data available Everything is ok
This function locks to the youngest data block. readerGet then reads data from this youngest data block.		
see also readerGet(), readerAdvance()		

int readerYield()		
Parameter		
Returns	READERR READOK	Communication Problem with Management Process Everything is ok
This function yields the current lock. Before any call of readerGet, readerAdvanceToHead has to be called.		
see also readerGet(), readerAdvance()		

int readerGet()		
Parameter	bufferPacketp_t *result int *size	Pointer to the fetched packet, see note! Size of the result memory chunk
Returns	READNODATA READLATE READERR READOK	No more data in data buffer Data buffer invalid, call advance Fatal error Everything ok
This function reads data of the locked on data buffer. The first call of this function **must** be made with *result=NULL and *size=0. The function will allocate memory and expand it if needed. If the data block locked on is re-used before the Plugin finished reading the data -1 is returned. The Plugin then has to call readerAdvance().		
see also readerGet(), readerAdvance()		

unsigned long long readerGetByteSequence()		
Parameter		
Returns	byte sequence	
Returns the Byte Sequence Number of the youngest block.		
see also readerAdvanceToHead()		

unsigned long long readerGetHeadSequence()		
Parameter		
Returns	Sequence number of youngest block	
Returns the Sequence Number of the youngest block.		
see also readerAdvanceToHead()		

unsigned long long readerGetSequence()		
Parameter		
Returns	Sequence number of current block	
Returns the Sequence Number of the current block.		
see also readerAdvanceToHead()		

unsigned long long readerGetDistance()		
Parameter		
Returns	distance of current block to head	
Returns the distance of the current block to the youngest block. The distance is 1 if current block is youngest block.		
see also readerAdvanceToHead()		

int readerTrafficShaper()		
Parameter	int maxBandwidth int (*callback) (bufferPacketp_t, int)	Maximum bandwidth in Bytes/s worker function, see below
Returns	READERR READOK	Fatal error Reannounce
This function automatically advances and gets data. It shapes the outgoing traffic so that no spikes or bursts of high load arise. For every packet in a data buffer the callback function is called.		
see also readerGet(), readerAdvance()		

int callback(bufferPacketp_t packet, int loss)		
Parameter	bufferPacketp_t packet int loss READOK READSTALE READSEQ	data to be processed READOK READSTALE READSEQ READERR Everything OK Data loss: too slow Data loss: framework error
Returns	currently ignored	
Prototype of the callback function. The function has to be implemented by the **plugin programmer**. With the parameter loss it can check for data inconsistencies.		
see also readerTrafficShaper()		

A.2 Write Library

int writerAnnounce()		
Parameter	char *commandpath queuep_t *handle int reconnect char *restartCommand	path to named pipe of mgmt will contain the buffer handle after successful announce 1 if handle contains valid handle restart command
Returns	-1 on error, 0 else	
Announce the existence of a writer process to the mgmt process. If the Management Process crashes the handle is valid. Therefore no new handle is needed on reconnect.		
see also writerShutdown()		

void writerShutdown()		
Parameter	queuep_t *handle	buffer handle
Returns		
Shutdown writer		
see also writerAnnounce()		

int writerPut()		
Parameter	queuep_t *handle void *data int size	buffer handle data to be inserted size of data
Returns	-1 on error, 0 else	
Put data into buffer		
see also writerFlush()		

int writerFlush()		
Parameter	queuep_t *handle	buffer handle
Returns	-1 on error, 0 else	
Flush buffered data, get new memory segment		
see also writerPut()		

A.3 Watchdog Library

int watchdogInitHeartBeatMgmt()		
Parameter	whandlerp_t whandler adminpagep_t adminpage char *restartCommand int timeout	handler variable adminpage restart command watchdog timeout
Returns	always 1	
The watchdog handler whandle is initialized so that subsequent calls of watchdogSendHeartBeat have their environment. The environment is configured for Management Process supervision.		
see also watchdogSendHeartBeat()		

int watchdogInitHeartBeatWriter()		
Parameter	whandlerp_t whandler adminpagep_t adminpage char *restartCommand int timeout	handler variable adminpage restart command watchdog timeout
Returns	always 1	
The watchdog handler whandle is initialized so that subsequent calls of watchdogSendHeartBeat have their environment. The environment is configured for Writer Process supervision.		
see also watchdogSendHeartBeat(), writerAnnounce()		

int watchdogInitHeartBeatReader()		
Parameter	whandlerp_t whandler adminpagep_t adminpage char *restartCommand int timeout	handler variable adminpage restart command watchdog timeout
Returns	always 1	
The watchdog handler whandle is initialized so that subsequent calls of watchdogSendHeartBeat have their environment. The environment is configured for Reader Plugin supervision. This is done by the readerAnnounce call.		
see also watchdogSendHeartBeat(), readerAnnounce()		

void watchdogSendHeartBeat()		
Parameter	whandlerp_t whandler	handler variable
Returns		
Send a heartbeat to the watchdog.		
see also watchdogInitHeartBeatMgmt, watchdogInitHeartBeatWriter, watchdogInitHeartBeatReader		

A.4 Logging

Logwrite is a syslog wrapper that can also write similar formatted messages to stdout or into a separate file. If the logger is used by including “log.h”, also <syslog.h> has to be included!

logwriteSetMedium()		
Parameter	int medium LOG_TO_STDOUT LOG_TO_FILE LOG_TO_SYSLOG char *filename char *ident	set logging medium log to stdout log to file log to syslog if logging to file: filename name of program
Returns	0 on success, -1 else	
Prepare logging medium.		
see also logwriteCloseMedium		

int logwriteClose()		
Parameter		
Returns	0 on success	
Close log medium, optional.		
see also logwriteSetMedium()		

void logwrite()		
Parameter	int priority const char *format ...	see syslog.h like printf varargs like printf
Returns	0 on success	
Write a message to the log. logwrite follows closely the syslog() function call. %m is not supported		
see also syslog()		

Appendix B

Deployment and Usage of the System

B.1 Building the System from Source Code

B.1.1 Source Distribution

The top level directory of the source distribution contains the following directories:

- *common* contains various include files and the source code for helper libraries used by the whole system.
- *fifo* contains the include file and the source code of the locking- and wait free FIFO and is used by mgmt and writer.
- *mgmt* contains the source code of the Management Process including the management of the different lists in *buffer.c* and *buffer.h*.
- The code in *netflow* is written by Arno Wagner and contains code for manipulating NetFlow V5 packets. It is used by libnetflowutil.
- *rawsend* is written by Simon Leinen and contains code to send UDP packets with faked source addresses.
- *reader* contains the plugin library *libread*, the utility library *libnetflowutil* and some sample- and test plugins.
- *scripts* contains perl programs used to generate test data and to plot traffic characteristics.
- *stat* contains the source code of the statistics viewer and the statistics watcher.
- *v7v5* contains code from Simon Leinen to convert NetFlow V7 to NetFlow V5 while adding fields which the router did not fill in.
- *watchdog* contains the watchdog code and the watchdog client library *libwatchdog*.
- *writer* contains the sample writer *sampliwriter* and the *libwrite* library.

It also contains some sample system start scripts and a Makefile.

B.1.2 Installation

There exist compile time options. These options are version specific and are documented in the README file enclosed with the software distribution. At the moment the maximum number of Client Plugins, the number of filter coefficients and the maximum number of Receivers (see chapter 3.1) are configurable at compile time in adminpage.h.

The Makefile is used to compile the system by executing `make`. `make install` installs the system. The install location is encoded in the beginning of the Makefile, per default the binaries are installed in the `nerfinst` subdirectory of the current user.

A directory structure like in figure B.1 is created.

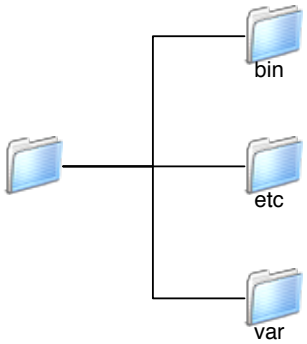


Figure B.1: Directory Structure of Software Installation

Located in `bin` are the binaries of the framework, in `etc` some sample config files. After the installation `var` is empty but it is recommended to configure the framework via the `mgmt.cfg` configuration file and the configuration switch `-p` so the named pipes are located in there. It is also recommended to keep the log files in here.

For each component should exist a start script in the etc directory that can be called from the watchdog and contains all needed command line options. These start scripts can be called from a system start script which initially starts the system.

Start Sequence

It is important that the `mgmt` process is always started first, also when using the watchdog. The rest of the programs can start in any order.

Watchdog

The usage of the watchdog functionality is easy: as soon as the watchdog process is started, all Client Plugins plus the Writer and the Management Process are under surveillance. By killing the Watchdog Process this surveillance is stopped.

B.1.3 More than one Framework

It is possible that more than one framework run parallel on one machine. One has to keep in mind that

- The adminpage shared memory id, and
- The directory for the named pipes

must be separated. One can set the adminpage shared memory id with the command line switch `-a`, this is only needed for `mgmt`, `watchdog`, `wdctl` and `stat`, the rest of the programs get the shared memory id while announcing. Therefore they need access to the named pipe directory with the parameter `-p`. This concerns `sampliwriter` and all reader plugins.

B.2 Command Line Options

Management Process: mgmt	
-h	Usage
-w *script*	script which restarts the Management Process with all required options
-c *config file* file	config file (see below)
-a *id*	Adminpage id, unique per running framework
-r	memory retrieval mode, collect shared memory segments belonging to an earlier incarnation of the framework.
-d	log to stdout
-f *filename*	log to file
-s	log to syslog

Writer Process: sampliwriter	
-h	Usage
-w *script*	script which restarts the Writer Process with all required options
-t *timeout*	watchdog timeout
-c *config file*	config file (see below)
-p	path to FIFO's
-b	socket buffer size
-d	log to stdout
-f *file*	log to file
-s	log to syslog

Client Plugin Process: udpforwarder	
-h	Usage
-w *script*	script which restarts the Writer Process with all required options
-t *timeout*	watchdog timeout
-d	log to stdout
-f *file*	log to file
-s	log to syslog
-p	path to FIFO's
–targetaddr addr	target addr
–targetport port	target port
–buffersize size:	socket buffer size

Client Plugin Process: trafficlogger	
-h	Usage
-w *script*	script which restarts the Writer Process with all required options
-t *timeout*	watchdog timeout
-d	log to stdout
-f *file*	log to file
-s	log to syslog
-p	path to FIFO's
–dumpfile *file*	file to save NetFlow dump to
–statfile *file*	file to save NetFlow stats to

Statistics Viewer: stat	
-h	Usage
-f *sampling frequency*	how often a value is sampled
-i *shmem id*	which framework is observed
-s	Summary Mode
-d	Data flow View
-m	Memory View
-r	Receiver View
-1	One-shot Mode, show only one line
-t	traffic shape debug Mode

Process Control Tool: wdctl	
-h	Usage
-a adminpageid	select framework
-d	log to debug
-f filename	log to file
-s	log to stdout
-l	list processes of framework
-r pid	remove bad behaving reader. It'll be removed from the watchdog control and killed with SIGKILL
-k	shutdown framework
-kk	kill framework and all processes

Watchdog Process: watchdog	
-h	Usage
-a *id*	adminpage id
-d	log to stdout
-f *file*	log to file
-s	log to syslog

Statistics Viewer: stat	
-h	Usage
-d	log to stdout
-f *file*	log to file
-s	log to syslog
-f *frequency*:	sample frequency
-a *shmem id*:	id of adminpage
-s:	Summary View
-d:	Dataflow View
-m:	Memory View
-r:	Receiver View
-t:	TrafficShaper View
-1:	One shot Mode: only show one line
-A *command*:	Alert mode

B.3 Configuration

The configuration file layout is always *"key value"*. The writer configuration file is context sensitive: once a receiver is defined with the keyword *"receiver"* all subsequent configuration keys belong to this receiver. The keywords are now explained in turn.

B.3.1 Manager Configuration File Format

mgmt.cfg		
keyword	default	explanation
blocksize	1048576	size of a memory segment
fifoPath	/tmp/nerf/	directory where the named pipe will be created
freeFifoSize	20	Size of Free FIFO
freeFifoWarn	15	A warning will be issued if the level is **below** the warn level.
readFifoSize	100	Size of Read FIFO
readFifoWarn	20	A warning will be issued if the level is **above** the warn level.
freeBufferMin	20	Minimal fill level of Free Buffer
freeBufferMax	50	Maximal fill level of Free Buffer
freeBufferWarn	10	A warning will be issued if the level is **below** the warn level.
readBufferSize	100	Maximal size of Read Buffer
readBufferWarn	50	A warning will be issued if the level is **above** the warn level.
softLimit	95	see chapter 3.3
trashTimeout	10	Segments in Trash List are deleted after 10 seconds
trashBufferMax	20	Maximum number of segments in trash Buffer. If trash is full, blocks are deleted unconditionally
Filter Configuration, filter has 100 elements. The filter coefficients are normalized after loading.		
filtercoeff1	20	Filter coefficients for elements below lowerlimit.
lowerlimit	10	
filtercoeff2	5	Filter coefficients for elements between lowerlimit and upperlimit.
upperlimit	70	
filtercoeff3	1	Filter coefficients for elements above upperlimit.

B.3.2 Writer Configuration File Format

sampliwriter.cfg		
keyword	default	explanation
port	19991	receive port of sampliwriter
There are 20 receivers to be configured. The framework accepts matching packets and generates statistics over the receivers.		
receiver	n/a	number of the receiver
sender	0.0.0.0	Address where the packets are coming from. This can be the address of a network. This keyword can exist multiple times in the config file, each must belong to one receiver.
mask	255.255.255.255	The netmask of the sender. The default is for a single host, if *sender* is a network then this is it's netmask.

B.3.3 Runtime Configuration

The runtime configuration is performed by using the programs mgmtcfg and sampliwritercfg. They have the same usage:

Online Configuration	
-a	adminpage id, identifies framework to configure
-g file	get configuration from memory and save in file, “-” is stdout
-p file	put configuration from file to memory.
-d	log to stdout
-f file	log to file
-s	log to syslog

B.4 Statistics and Alerting

There exists the program stat which provides statistics and alerting functionality. There are two modes:

- Statistics Viewer Mode: prints the chosen page of statistics.
- Logging and Alerting Mode: monitors critical levels of the system and informs an administrator in case of a failure or critical situation. Furthermore it logs the important statistics to stdout, to a file or to syslog.

The two modes will now be explained in turn:

B.4.1 Statistics Viewer Mode

There are several sets of statistics. The file is formatted so it can easily be used as input to gnuplot[15]. Every twenty lines a header is printed describing the data columns.

The first column, called smp for sample, is the number of the current line.

Summary View

The summary view provides fast access to the most important information. These are:

- **FreeBF:** free buffer fill level in segments
- **FreeBW:** number of free buffer warnings
- **ReadBF:** read buffer fill level in segments
- **ReadBW:** number of read buffer warnings
- **FreeFLW:** lowest reached value
- **FreeFHW:** highest reached value
- **FreeFW:** number of free FIFO warnings
- **ReadFLW:** lowest reached value
- **ReadFHW:** highest reached value
- **ReadFW:** number of read FIFO warnings
- **Rcv:** total number of received segments
- **Snd:** total number of sent segments
- **Disc:** total number of segments discarded by the framework

Dataflow View

- **Rcv:** total number of received segments
- **Snd:** total number of sent segments
- **Disc:** total number of segments discarded by the framework
- **Trashed:** number of segments put into the Trash List
- **Readers:** number of currently plugged in Reader Plugins
- **Writer:** 1 if a Writer Process is present
- **Mgmt:** 1 if the Management Process is present

Memory View

- **FreeBF:** free buffer fill level in segments
- **FreeBW:** number of free buffer warnings
- **FreeBWS:** 1 if a warning is pending
- **ReadBF:** read buffer fill level in segments
- **ReadBW:** number of read buffer warnings
- **ReadBWS:** 1 if a warning is pending
- **FreeFLW:** lowest reached value
- **FreeFHW:** highest reached value
- **FreeFW:** number of free FIFO warnings
- **FreeFWS:** 1 if a warning is pending
- **ReadFLW:** lowest reached value
- **ReadFHW:** highest reached value
- **ReadFW:** number of read FIFO warnings
- **ReadFWS:** 1 if a warning is pending

Receiver View

The Receiver View shows the following parameters per receiver:

- **RecNo:** number of receiver the values stand for
- **Packets:** number of UDP Payloads received
- **Bytes:** number of Bytes received
- **B/s:** Bytes/s averaged over the whole lifetime

TrafficShaper View

The values in the TrafficShaper View are meant for the tweaking of the TrafficShaper parameters, see chapter 5. The times are given in μseconds/segment.

- **Speed:** Calculated outgoing speed
- **rcoeff:** 1 if filter is not yet initialized, real values (wtime) is taken instead of estimated (est)
- **ecoeff:** 1 if filter is valid.
- **fcoeff:** c(b), the fill coefficient
- **fillr:** b, the buffer fill level
- **wtime:** current incoming speed
- **wtimea:** averaged incoming speed (over four values)
- **est:** filtered incoming speed

B.4.2 Alerting Mode

The alerting mode is switched on with the command line switch `-A`. In this mode `stat` checks regularly (with frequency f) for warning conditions and executes a command in case of a warning. The command is given the name of a temporary file as argument. This file contains information about the warning. The warning level (see chapter B.4.1) can be configured in the Management Process configuration file.

B.5 System and Maximum Memory Requirements

The framework was implemented and tested on the following x86 PC's, all running Linux:

- AMD Athlon 1900+, 1600 MHz with 1GB Ram
- Intel Pentium 4, 2000 MHz with 512MB Ram
- Dual Intel Pentium 3, 1400 MHz with 1GB Ram

The maximum memory requirement can be calculated from the configuration file as following:

$$mem_{max} = blocksize*(freeFifoSize+readFifoSize+freeBufferMax+readBufferSize+trashBufferMax+1)$$

The additional block stems from the Writer Process who is currently writing to a block not in any data structure of the Management Process.

Appendix C

Original Task

Realtime UDP Netflow Data Processing Framework for

Caspar Schlegel <schlegel@use.ch>

Subject

Distributed Denial of Service (DDoS) are a threat to Internet services ever since the widely published attacks on e-bay.com and amazon.com in 2000. ETH itself was the target of such an attack 6 months before these commercial sites where hit. ETH suffered repeated complete loss of Internet connectivity ranging from minutes to hours in duration. Massively distributed DDoS attacks have the potential to cause major disruption of Internet functionality up to and including severely decreasing backbone availability.

Attack Model

Most DDoS attacks share a common pattern: An *infection phase* where the initiator acquires the attack resources by compromising a large number of weakly protected hosts, ideally causing little or no visible change in host behavior, in order to make the compromise hard to notice. An infection phase can last from less then 10 minutes to several months. Attacks within the order of 100.000 and more compromised hosts have already been observed in practice (Code Red, Sapphire).

In a second phase, the *attack phase*, the attacker uses the compromised hosts to initiate actual attacks on a target computer or network. These attacks can be done autonomously or under direct or indirect control of the attacker. Although attack control increases the risk of identification for an attacker, there are possibilities to keep this risk small.

Diploma Thesis Task

A generic realtime UDP Netflow data processing framework will be developed and extensively tested to assure proper operation under high loads and bursty Netflow data traffic as observed in the Switch backbone network infrastructure.

This task is split into the following subtasks:

Understand network traffic raw data

As raw data of the network traffic, we use NetFlow[1] traffic logs that are produced by border gateway routers of the SWITCH[2] academic network. In a first step, the format of this data and its exact meaning must be understood.

Analyse the Samplicator program

Currently, SWITCH uses a program called “Samplicator” to queue the UDP Netflow data, process it and then duplicate it to two other machines. The Netflow traffic stream is currently very bursty in its nature, which is even aggravated by duplicating the data streams. In the processing step, the “Samplicator” converts the Netflow format and overwrites some data fields in the Netflow records. The source code of this program is available.

Design and Implementation of a realtime UDP Netflow data processing framework

The functionality of the “Samplicator” should be greatly improved. The result will be a realtime UDP Netflow data processing framework, which consists of modular components that very likely will be run on distributed machines.

In this thesis at least the following components of the framework will be developed:

- The *queueing component* de-bursts the Netflow data by using an efficient leaky-bucket algorithm for emptying the queue.
- The *statistics component* calculates the most essential traffic statistics.
- The *duplicating component* duplicates the Netflow data stream and sends it to 1-N hosts. The number and properties (IP address, port etc.) of the destination hosts can be configured without stopping data acquisition.
- The *logging component* writes the raw Netflow data into files.
- The *monitoring and alerting component* reads the Netflow data and shows a selection of important statistics and activates alerts upon detection of critical traffic patterns (based on given traffic signatures).

Further optional components are:

- Graphical plots of aggregated statistics over time extracted from Netflow data
- etc.

Testing

Test scenarios with artificial bursts and high loads are to be conceived and executed in order to assure the realtime capabilities of this framework.

Documentation and Presentation

A documentation that states the steps conducted, lessons learnt, major results and an outlook on future work and unsolved problems has to be written. The code should be documented well enough such that it can be extended by another programmer within reasonable time. At the end of the semester, a presentation will have to be given at TIK that states the core tasks and results of this semester thesis. If important new research results are found, a paper might be written as an extract of the thesis and submitted to a computer network and security conference.

[1] A proprietary data standard by CISCO
[2] www.switch.ch

Dates

The diploma thesis will start on May 12th, 2003 and will be finished by September 11th, 2003.

Supervisors

Arno Wagner, wagner@tik.ee.ethz.ch, +1 632 7004, ETZ G64.1
Thomas Dübendorfer, duebendorfer@tik.ee.ethz.ch, +1 632 7196, ETZ G64.1

Appendix D

Timetable

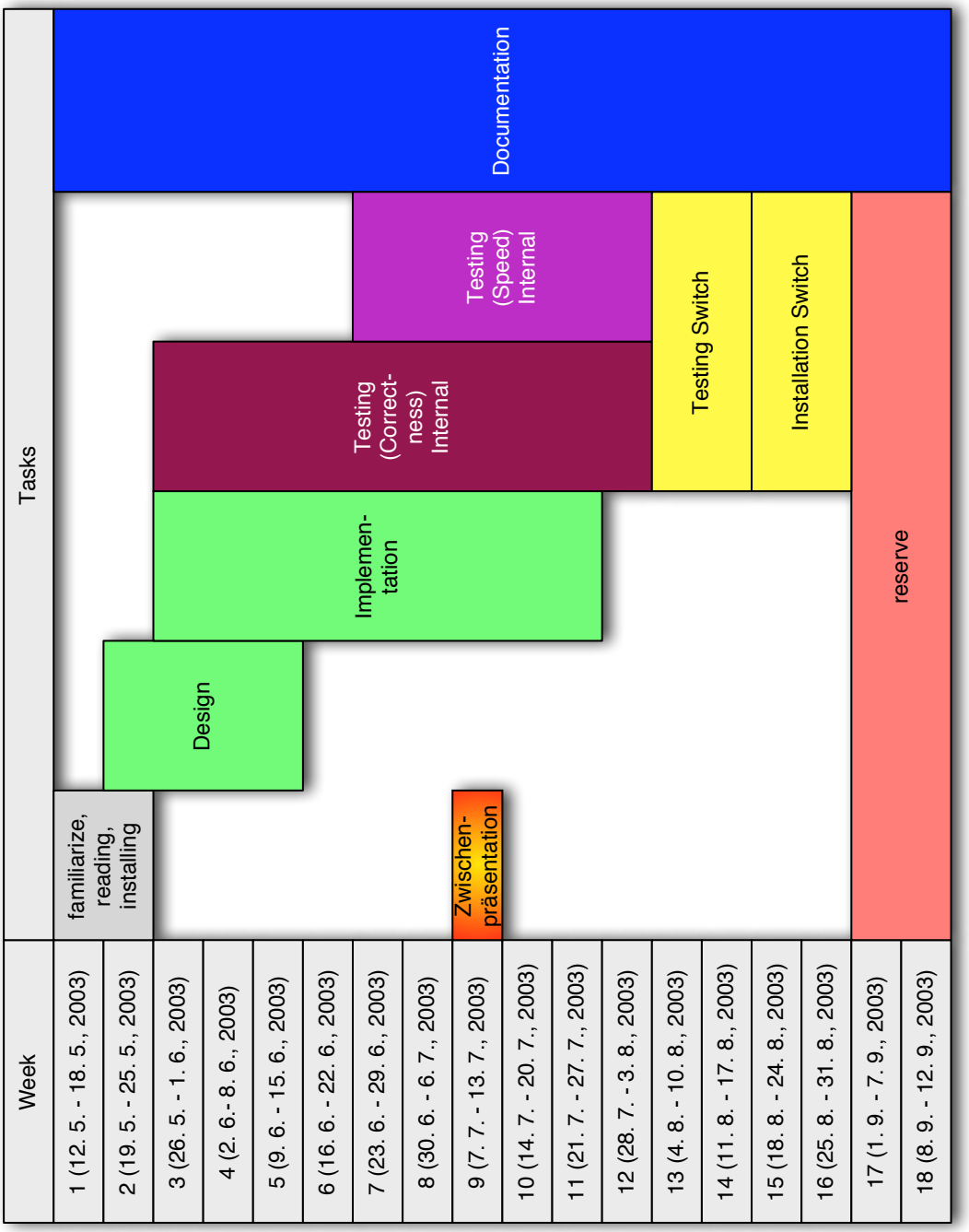


Figure D.1: Timetable

Bibliography

[1] SWITCH The Swiss Educational & Research Network, *Network Monitoring and Analysis*; http://www.switch.ch/tf-tant/floma/sw/samplicator/.

[2] Arno Wagner, Thomas Dübendorfer, *DDOSVAX Homepage*; http://www.tik.ee.ethz.ch/ ddosvax/.

[3] Cisco Systems, *NetFlow Export Datagram Format*; http://www.cisco.com/en/US/products/sw/netmgtsw/ps1964/ products_installation_and_configuration_guide_chapter09186a0080080e30.html#wp1820.

[4] Cooperative Association for Internet Data Analysis, *cflowd: Traffic Flow Analysis Tool*; http://www.caida.org/tools/measurement/cflowd/design/design.html.

[5] Tobias Oetiker, *RRDTool: Round Robin Database*; http://people.ee.ethz.ch/ oetiker/webtools/rrdtool/.

[6] Cisco Systems, *Network Data Analyzer*; http://www.cisco.com/univercd/cc/td/doc/product/rtrmgmt/nda/.

[7] Andrey Slepuhin, *Netflow processing tools*; http://sourceforge.net/projects/netfltools/.

[8] Crannog Software, *Netflow Monitor*; http://www.crannog-software.com/netflow.html

[9] Helmut Herold, *UNIX und seine Werkzeuge: UNIX-Systemprogrammierung*; Addison Wesley, 1996.

[10] W. Richard Stevens, *Networking APIs: Sockets and XTI*; Prentice Hall, 1998.

[11] Daniel P. Bovet, Marco Cesati, *Understanding the Linux Kernel*; O'Reilly, 2001.

[12] Andrew S. Tanenbaum, *Computer Networks, 3rd ed.*; Prentice Hall, 1996.

[13] Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language, 2nd ed.*; Prentice Hall, 1988.

[14] *Linux Man-Pages*;

[15] The Gnuplot Team, *Gnuplot*; http://www.gnuplot.info/.

[16] Lukas Ruf, *Topsy Template*; http://www.topsy.net/TeX/.

[17] Tobias Oetiker, Hubert Partl, Irene Hyna and Elisabeth Schlegl, *The not so Short Introduction to LaTeX*; http://people.ee.ethz.ch/ oetiker/lshort/.