

# PSC TCP Kernel Monitor

Jeffrey Semke  
Pittsburgh Supercomputing Center  
Carnegie Mellon University

May 16, 2000

PSC Technical Report # CMU-PSC-TR-2000-0001

- [Introduction](#)
- [Architecture](#)
- [Application tools](#)
  - [Pmgraph](#)
  - [Pscmonitor](#)
- [Data Structures / Kernel Monitor Internals](#)
  - [Kernel Features Flags \(psc\\_mods\)](#)
  - [Table of Snapshots \(struct tpm\\_entry\)](#)
  - [Snapshot Structure Member Descriptions \(struct tpm\\_entry\\_description\)](#)
  - [State-Keeping Variables](#)
  - [Summary](#)
- [Appendix](#)
  - [kvm](#)
  - [Details of the application header record](#)
  - [Where to obtain the software](#)
- [References](#)

## 1. Introduction

An early version of the PSC kernel monitor was used to verify the operation of the TCP autotuning implementation [[PSC98](#)], and was used to produce many of the graphs for the Sigcomm98 "Automatic TCP Buffer Tuning" paper [[SMM98](#)].

The PSC kernel monitoring package allows the visualization of kernel networking variables, as illustrated in the example below.

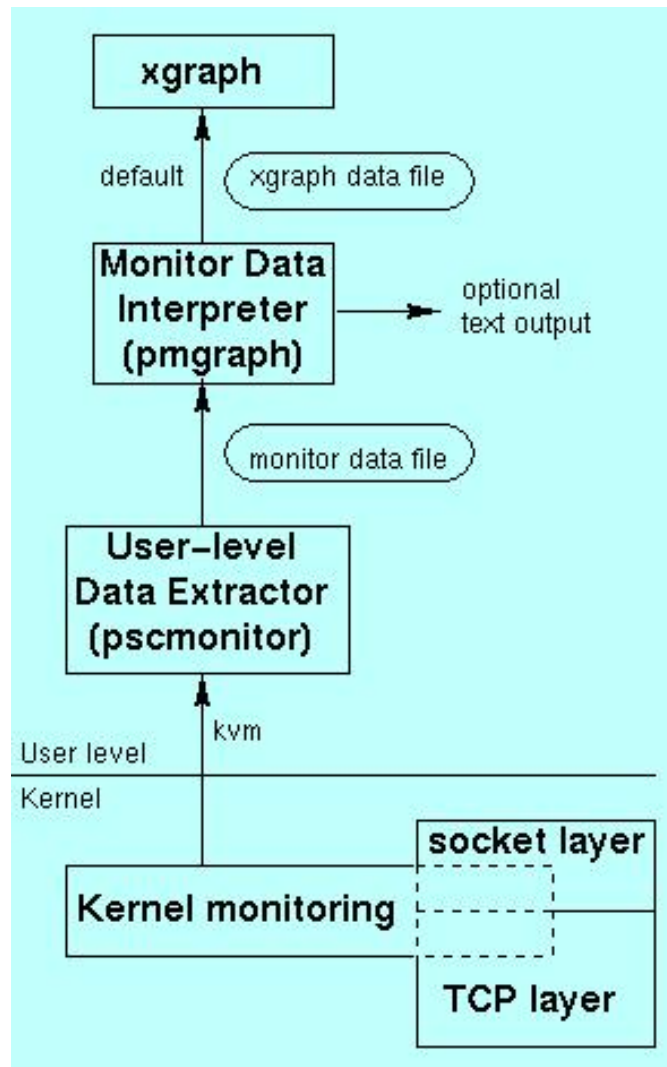


The PSC kernel monitor was designed with the following goals in mind:

- It should allow observation of TCP and socket variables, allowing a deeper understanding of TCP internals.
- It should allow debugging of TCP modifications.
- It should put as little load upon the system as possible.
- It should be easily extensible to allow the monitoring of additional kernel variables.
- Newer versions of the application tools should be able to read older data files.
- A single version of an application tool should be able to run on any of the PSC research kernels, regardless of which features have been compiled into the kernel.
- Though designed for NetBSD, the ideas should be generally applicable to other operating systems.

In order to achieve the goals of backwards-compatibility and the mixing and matching of kernels with a single application program, the kernel monitoring data sets are self-describing. This allows newer versions of the monitor graphing tools to read old data sets, as well as allowing users to exchange data sets that were collected in kernels with their own modifications.

## 2. Architecture



**PSC TCP Kernel Monitor Architecture**

The primary component of the project is an in-kernel monitor which uses macros placed in the source code of the TCP stack to record the values of a set of TCP and socket variables into a circular table in the kernel. The macros are placed around certain "events" in the TCP stack, such as a change in the congestion window size, the discovery of a segment loss, or the occurrence of a retransmission timeout. The table can be read by applications with [kvm](#). Another important component is the set of application tools that save the kernel monitor table to a file, or that process the data files.

The **pscmonitor** application reads the saved "snapshots" from the kernel's circular table using [kvm](#), and saves the data to a binary file. The file is then read by **pmgraph** which graphs the specified parameters using **xgraph** [X99].

**Pmgraph** is also able to convert the binary monitor data file into an ASCII file containing the specified parameters.

In this paper, the layers will be described starting at the top (the display application) and working down to the kernel data structures.

### 3. Application tools

There is a small suite of applications that are used with the kernel monitor.

#### **pmgraph**

Produces graphs or text output from kernel monitor files for each connection in the file.

#### **pscmonitor**

periodically extracts the monitoring information from the kernel and saves it to a file.

## 1. pmgraph

The **pmgraph** application reads kernel monitor files obtained with **psmonitor** and converts them either into text format or into graphical format, using **xgraph** [X99]. When graphing, all specified parameters can be displayed for a given connection (to see the relation of the parameters to each other). Alternatively, a specified parameter can be displayed for all named connections (to see the relation of the connections upon each other in terms of that parameter).

Pmgraph usage is described as:

```
Usage: pmgraph [options]
Options:      (default is show everything in a graph)
  -b          show so_snd.sb_cc
  -c          show cwnd
  -d          print verbose debugging information
  -D          output gaps in the data to stdout
               This allows the detection of wrapping of the table
               between readings by the psmonitor
  -f          show snd_fack
  -F          show hiwat_fair_share
  -h          print this help message
  -H          show so_snd.sb_hiwat
  -l          show calling location
               Unique identifiers for each snapshot (data sample,
               or set of concurrent datapoints in a graph), to allow
               tracing through the TCP stack
  -m          show mbuf clusters in use (in bytes)
  -M          print maximum amount of system memory used in trace
               If used without any other params, no other output will be
               produced
  -n          show so_snd.sb_net_target
  -p lport.rport  only include connections specified by local and
               remote ports
               More than one connection can be specified by using the
               "-p" option multiple times in the command line.
               Irrelevant connections can be ignored this way, or
               individual connections can be studied out of a group
  -P          show graphs of one parameter for all connections
               (default is to show graphs of one connection with all
               parameters)
  -r          show so_rcv.sb_cc
  -s          show snd_max
  -S #c       scale option c by floating point #
               Allows parameters of smaller magnitude to be graphed
               with a similar magnitude to other parameters on a
               single graph. (ex. -S 1000l graphs the call location
               with 1000 times the recorded value
  -t          show so_snd.sb_mem_target
  -T          do not graph, but convert binary input file into text sent
               to stdout
               Essentially, perform a binary-to-text conversion of the
               data file, but also support filtering of connections
               and specification of relevant parameters
  -v          show calling values
```

-x produce xgraph files without calling the xgraph program

Note that some of the options in **pmgraph** are for the display of variables used in experimental TCP implementations.

Samples of the graphical output of **pmgraph** are included below.



This graph shows the congestion window and the send socket buffer's high water mark plotted for a single connection. This graph was generated with the command `pmgraph na.9903051452.pm -p 1055.5050 -c -H`. (This TCP connection uses an early version of Rate Halving [MM97] with Autotuning sender-side socket buffers [SMM98, PSC98].)



This graph shows cwnd plotted for three concurrent connections. It was generated with the command `pmgraph na.9903051452.pm -P -p 1055.5050 -p 1056.5051 -p 1057.5052 -c`. (These TCP connections use an early version of Rate Halving [MM97].)

## 2. pscmonitor

The output file produced by `pscmonitor` is in binary format, to allow faster reads/writes. The first part of the file is a header section that describes the format of the data and the kernel features, while the second part is a sequential dump of the kernel monitoring snapshots, with the entries sorted by their monitoring sequence number.

The header consists of a number of variable-length records (inspired by the format for TCP header options [RFC793]) that look like the following:

### Header Record Format

Kind (2 bytes)	Length (2 bytes)	Data (Variable, Length - 4 bytes)
----------------	------------------	-----------------------------------

The **Length** field indicates the length of the header record. The minimum header record is 4 bytes, for records with no data.

Kind is one of the following:

### Kinds of Header Records

Kind name	Description	Kind Value	Record Length (bytes)	Data Length (bytes)
KIND_END	Indicates the end of the header	0	4	0
KIND_MODS	Contains the 32-bit <code>psc_mods</code> flag in the data field	1	8	4
KIND_VERSION	Contains the <code>psc_version</code> string for the <code>psc_mods</code>	2	12	8
KIND_TABLE_ENTRY_SIZE	Length (in bytes) of one snapshot (from <code>tpm_entry_size</code> )	3	8	4
KIND_DATA_DEFINITION	Contains a <code>struct tpm_entry_description</code> that describes one variable appearing in all snapshots	4	36	32
KIND_ENDIAN	Specifies the byte-order of multi-octet data fields	5	8	4
KIND_MCLSIZE	Size (in bytes) of an mbuf cluster on the monitored machine	6	8	4

A number of header records are concatenated together ending with the `KIND_END` record, which signifies the end of the header. The raw data from the kernel monitor appears after the header. The length of each snapshot appearing in the data is given by the `KIND_TABLE_ENTRY_SIZE` header record, and the format of each snapshot is determined by interpreting all of the `KIND_DATA_DEFINITION` records. The significance of each header record type should become much clearer after reading the section on [Data Structures](#) below. Some specific details that merit mentioning are:

#### KIND\_MODS

A record representing the features present in the kernel from which the trace was collected. If the `KIND_MODS` record is missing, the mods are assumed to be `PSC_RENO`. More information about the feature flag appears [below](#).

#### KIND\_ENDIAN

Kernel monitor data that appears in the table itself is stored in the native-endian of the machine it was collected on, rather than in network order, in order to make data collection as fast as possible. Slowing the data collection process could result in lost entries as the monitoring table wraps. Therefore, it is up to the applications that read the data file to convert the data to the appropriate format for that platform. The values for `KIND_ENDIAN` are:

1. `PM_LITTLE_ENDIAN` (= 0)
2. `PM_BIG_ENDIAN` (= 1)
3. `PM_PDP_ENDIAN` (= 2)

If the `KIND_ENDIAN` record is missing, the data set is assumed to be `LITTLE_ENDIAN`.

All multi-octet values in the file **header** are in network order.

[More information](#) on the **header** can be found in the appendix.

# 4. Data Structures / Kernel Monitor Internals

- [Kernel Features Flags \(psc\\_mods\)](#)
- [Table of Snapshots \(struct tpm\\_entry\)](#)
- [Snapshot Structure Member Descriptions \(struct tpm\\_entry\\_description\)](#)
- [State-Keeping Variables](#)
- [Summary](#)

## 1. Kernel Features Flags (psc\_mods)

A minor addition has been made to all PSC-modified NetBSD research kernels. A features flag has been added to the kernel. Applications can use [kvm](#) to read the flags to determine which features exist in the kernel. One application version can work with nearly any research kernel without needing to be recompiled. So far, applications which have been modified to make use of the features flags include not only the kernel monitoring tools, but also an enhanced version of netstat.

The features flags have been implemented as two [kvm](#)-readable variables:

### Variables

name	type	declared in	defined in	kvm-accessible
psc_version	char*	tcp.h	tcp_subr.c	Y
psc_mods	u_int16_t	tcp.h	tcp_subr.c	Y

**psc\_version** is a string containing the PSC source code modification version, such as "0.9".

**psc\_mods** is a bit mask of the features included in the kernel. The bitmask is currently defined in tcp.h as:

### Bits

name	value	description
PSC_RENO	1	PSC Common kernel without SACK or FACK
PSC_SACK	2	Selective Acknowledgments
PSC_FACK	4	Forward Acknowledgements
PSC_AUTO	8	Autotuning socket buffers

## 2. Table of Snapshots (struct tpm\_entry)

The kernel monitor consists of an array of structures (hereby known as the **table**). Each structure represents a snapshot of the system at a particular time. (Think of each structure in the array as a row in the table. Each structure member forms a column of the table.)

The table is an array of struct tpm\_entry (which is defined in tcp\_pscmonitor.h):

```
struct tpm_entry {  
  
    /* general */  
    u_long          seq_no;           /* tpm entry number */  
    struct timeval  time;            /* time of entry */  
    u_long          callvalue;       /* used for debugging */  
    u_long          location;        /* unique value identifying  
                                     triggering code point */  
  
    /* Connection Specific */  
    /* Reno */
```

```

u_int16_t      lport;          /* local port number */
u_int16_t      rport;          /* remote port number */
u_long         snd_cwnd;        /* congestion window */
tcp_seq        snd_max;        /* highest sequence number sent */

/*      FACK */
tcp_seq        snd_fack;        /* highest sequence number
                                (s)acked*/

/*      Socket */
/*      Socket: Reno */
u_long         sb_hiwat;        /* send socket buffer hi water
                                mark */
u_long         snd_sb_cc;        /* space used in send socket buf */
u_long         rcv_sb_cc;        /* space used in rcv socket buf */

/*      Socket: Autotuning */
u_long         sb_target_hiwat; /*      sb_mem_target for same */
u_long         sb_net_target;   /*      sb_net_target for same */
u_long         hiwat_fair_share; /* per-connection buffer fair
                                share */

/* System-wide */
u_long         m_clused;        /* m_clusters - m_clfree */
} tpm_table[TPM_ENTRIES];

```

The entire table may be read by user-level applications using [kvm](#). Some notes on members of the structure that may not be completely obvious are discussed below.

### general

This portion of the structure contains fields that are used for operation of the monitor itself, as well as debugging.

#### seq\_no

The unique monitoring sequence number for this snapshot, used to identify previously-read snapshots, and to detect wrapping of the table.

#### time

The time that the snapshot was recorded.

#### callvalue

A value that was passed in the macro that caused this snapshot to be recorded. This can be used to record flags or state that are not relevant to all snapshot locations.

#### location

A unique value used to identify where in the code this snapshot was taken.

## Connection Specific

This portion of the structure contains fields that are specific to individual connections.

### Socket

These fields are more appropriately considered socket layer values, rather than TCP layer values.

## System-wide

This portion of the structure contains fields that are not connection-specific, such as total available system resources.

Each atomic snapshot taken by any of the connections in the system is interleaved in a single table with

snapshots taken by other connections. The order in which they were recorded can be determined by the **seq\_no** of each entry, and the relation of time between the entries can be found from the **time** field. As a practical matter, all the entries from a single connection can be associated with each other using the **rport** and **lport** fields. IP addresses are not currently included in the structure, since each connection is identifiable using only the port numbers in nearly all cases. (There are a few obscure cases in which the port numbers alone are not sufficient for distinguishing between different TCP connections, but we do not encounter any of these cases in the experiments that we are interested in. The extensible nature of the kernel monitor architecture should allow the addition of IP addresses, if necessary for other research projects.)

### 3. Snapshot Structure Member Descriptions (struct tpm\_entry\_description)

Since the uses of the kernel monitor are expected to grow and change, a description of each column in the table is stored in the kernel also, to be read by user-level applications and stored with the tables in all data files. This gives the monitor and the data files the property of being self-describing. The descriptions are stored in an array of struct tpm\_entry\_description, which is declared in tcp\_pscmonitor.h:

```
struct tpm_entry_description {      /* Describe the current TPM entry */
#define TPM_STRLEN 24              /* Do not change, if to remain
                                   compatible with other versions */
    char        name_string[TPM_STRLEN]; /* name of param */
#define TPM_STR_SEQ_NO           "seq_no"
#define TPM_STR_TIME             "time"
#define TPM_STR_CALLVALUE       "callvalue"
#define TPM_STR_LOCATION        "location"
#define TPM_STR_LPORT           "lport"
#define TPM_STR_RPORT           "rport"
#define TPM_STR_SND_CWND        "snd_cwnd"
#define TPM_STR_SND_MAX         "snd_max"
#define TPM_STR_SND_FACK        "snd_fack"
#define TPM_STR_SB_HIWAT        "sb_hiwat"
#define TPM_STR_SND_SB_CC       "snd_sb_cc"
#define TPM_STR_RCV_SB_CC       "rcv_sb_cc"
#define TPM_STR_SB_TARGET_HIWAT "sb_target_hiwat"
#define TPM_STR_M_CLOSED        "m_closed"
#define TPM_STR_LAST            ""
    u_int16_t  offset;            /* offset of param into tpm_entry */
    u_char     length;           /* length of param */
    u_char     scope;           /* scope of relevance of param */
#define TPM_SCOPE_PM            0 /* PSC monitor value */
#define TPM_SCOPE_SYS          1 /* System-wide variable */
#define TPM_SCOPE_INDIV        2 /* relevant only to indiv. conn. */
    u_int16_t  mask;            /* Which bits in psc_mods this param
                                   relates to
                                   0 = all */
    u_int16_t  flags;           /* Attributes */
#define TPM_F_INT_HOST         0 /* Integer in host order */
#define TPM_F_INT_NET          1 /* Integer in net order, ie. ports*/
#define TPM_F_STRING           2 /*raw bits, such as string or float*/
};
```

#### **name\_string**

A string value used to identify a particular member in the structure (column in the table). The last element in the array of struct tpm\_entry\_description should use a name string of TPM\_STR\_LAST.

#### **offset**

The number of octets into the struct `tpm_entry` where this member field begins.

### **length**

The number of bytes in struct `tpm_entry` that are used by this member field. **The length should be zero if this field is not used by the current kernel.** For example, the `snd_fack` field would have a length of zero in a Reno kernel. The `pscmmonitor` application uses this to determine which fields should be ignored.

### **scope**

Indicates how broadly the value of this field is relevant. For example, if the scope is `TPM_SCOPE_INDIV`, such as `snd_max`, then this parameter only has relevance for the connection that logged it in the table. On the other hand, if the scope is `TPM_SCOPE_SYS`, such as `m_clused`, then this parameter has relevance to the whole system, not just the connection that recorded it in the table. Finally, if the scope is `TPM_SCOPE_PM`, the value is used for debugging only, and does not have relevance to a connection or the system.

### **mask**

The mask word contains a bit map that describes for which kernel types the parameter is relevant. The bits are the same as used for [psc\\_mods](#). For example, if the `PSC_FACK` bit is set in the mask, then the field is relevant to a FACK kernel. If that bit is not set in the mask, then the field has no meaning to a FACK kernel. If the mask is zero, then the parameter applies to all kernels.

### **flags**

The flags mark attributes, such as whether the data is a number in host or network order, or a bit string which is not to be manipulated

The description structure is made available to applications with [kvm](#), and is stored with all data files, to allow the data files to be read later by applications that are able to deal with newer table formats, as well as old.

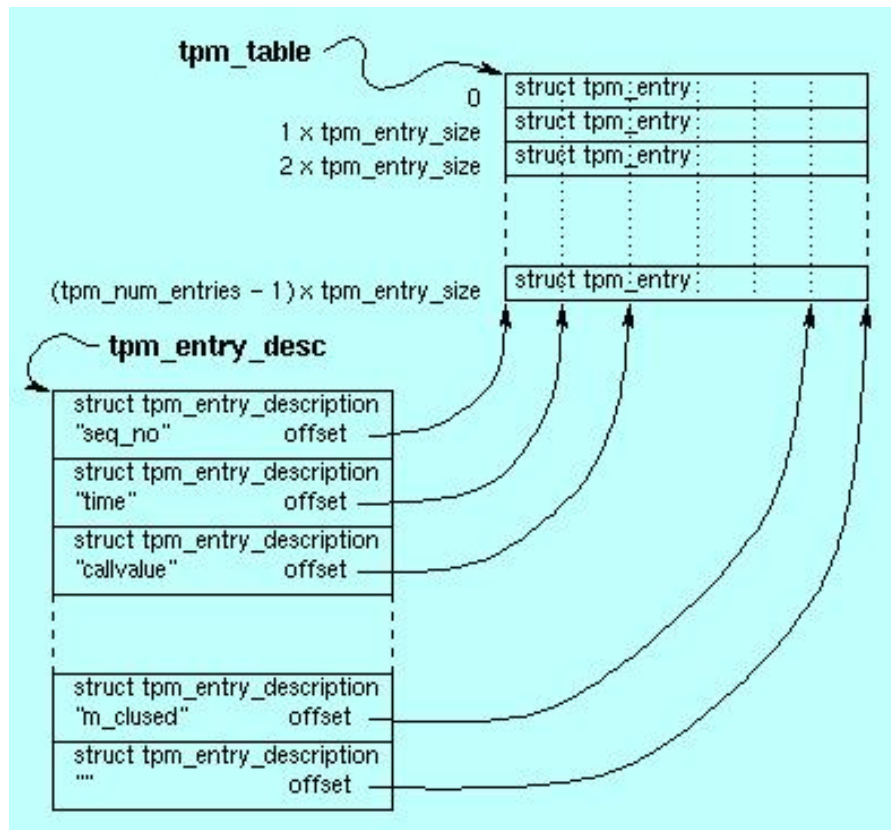
The relation between the table and the snapshot structure member descriptions are described below in the [Summary](#).

## 4. **State-Keeping Variables**

There are several state keeping variables described in the next section which are used for the general upkeep of the monitoring system. These variables include the current monitoring sequence number, a pointer to the current snapshot in the table, the size (in octets) of a snapshot, and the maximum number of snapshots in the table.

## 5. **Summary**

### **Relation of Data Structures**



The above diagram illustrates the relation between the structures described in the previous sections. **Tpm\_table** points to the array of structures (`struct tpm_entry`) that were described in the [Table](#) section above. Each row is one structure (the fields are layed out horizontally) and is filled in each time a snapshot is recorded. The byte offsets are listed down the left side of the array. **Tpm\_entry\_desc** points to a second array of structures (`struct tpm_entry_description`) that were described in the [Snapshot Structure Member Descriptions](#) section above. This array gives the monitoring system its self-describing quality. Each structure in the second array describes one snapshot member (column) in the **tpm\_table**, including an offset into the structure.

### Types

name	defined in
<code>struct tpm_entry</code>	<code>tcp_pscmonitor.h</code>
<code>struct tpm_entry_description</code>	<code>tcp_pscmonitor.h</code>

### Kernel variables

name	type	declared in	defined in	kvm	desc
<code>tpm_table</code>	<code>struct tpm_entry*</code>	<code>tcp_pscmonitor.h</code>	<code>tcp_pscmonitor.h</code>	Y	table of snapshots
<code>tpm_entry_desc</code>	<code>struct tpm_entry_description*</code>	--	<code>tcp_subr.c</code>	Y	description of columns
<code>tpm_index</code>	<code>struct tpm_entry*</code>	<code>tcp_pscmonitor.h</code>	<code>tcp_subr.c</code>	N	next row to fill
<code>tpm_seq</code>	<code>u_long</code>	<code>tcp_pscmonitor.h</code>	<code>tcp_subr.c</code>	N	next seq_no to use
<code>tpm_num_entries</code>	<code>u_long</code>	<code>tcp_pscmonitor.h</code>	<code>tcp_subr.c</code>	Y	num of rows in table
<code>tpm_entry_size</code>	<code>u_char</code>	<code>tcp_pscmonitor.h</code>	<code>tcp_subr.c</code>	Y	octets per row

### Macros

name	defined in	description
<code>TPM_INDEX_INCR</code>	<code>tcp_pscmonitor.h</code>	Move to next row of table
<code>TPM_SNAPSHOT</code>	<code>tcp_pscmonitor.h</code>	Take snapshot and put in table

## 5. Appendix

### 1. kvm

NetBSD supports a mechanism called *kvm* for applications with setuid root to read data structures from the kernel. Access to the data structures is implemented with a set of system calls. There must be kernel support for the particular data structures to be read (in other words, the only data structures that can be read from the kernel are the ones that the kernel allows).

### 2. Details of application header records

From pscmonitor.h:

```
struct _pm_mods {          /* make structure a word so subsequent records
                           are word aligned */
    u_int16_t mods;
    u_int16_t pad;
};

struct _pm_tes {          /* make structure a word so subsequent records
                           are word aligned */
    u_int16_t table_entry_size;
    u_int16_t pad;
};

struct header_entry {
    u_int16_t kind;       /* designed so that data begins on a word
                           boundary */
    u_int16_t length;    /* designed so that data begins on a word
                           boundary */
    union {
        struct _pm_mods mods;
        struct _pm_tes tes;
        char version[VERSION_LENGTH];
        struct tpm_entry_description desc;
    } data;
};
#define pm_mods          data.mods.mods
#define pm_table_entry_size  data.tes.table_entry_size
```

### 3. Where to obtain the software

The version of the PSC kernel monitor described in this report is available from [http://www.psc.edu/networking/ftp/tools/netbsd132\\_rh\\_10.tgz](http://www.psc.edu/networking/ftp/tools/netbsd132_rh_10.tgz), which is the *NetBSD 1.3.2 SACK/RH/AUTO/ECN* implementation on the <http://www.psc.edu/networking/tcp.html> web page.

## 6. References

- [MM97] Matt Mathis and Jamshid Mahdavi. "TCP Rate-Halving with Bounding Parameters," <http://www.psc.edu/networking/papers/FACKnotes/current/>, Pittsburgh Supercomputing Center, 1997.

- [PSC98] "Automatic TCP Buffer Tuning Research" web page, Pittsburgh Supercomputing Center, 1998. Available at:  
<http://www.psc.edu/networking/auto.html>
- [RFC793] "Transmission Control Protocol." IETF RFC 793, September 1981. Available from: <http://www.rfc-editor.org/rfc/rfc793.txt>.
- [SMM98] J. Semke, J. Mahdavi, and M. Mathis. "[Automatic TCP Buffer Tuning](#)," *ACM Sigcomm '98/Computer Communication Review*, Volume 28, Number 4, October 1998.
- [X99] xgraph plotting and graphing tool.  
<http://www-mash.cs.berkeley.edu/xgraph/>, U.C.Berkeley, 1999.