Accurate Network Load Monitoring: Loadman

James Mercer, Craig A. Farrell & Mike Schulze
Department of Computer Science
Curtin University of Technology
PO Box U1987, Perth 6001
Australia
email:{jamie,craig,mike}@cs.curtin.edu.au

January 7th, 1994

Abstract

Maintaining accurate performance statistics for computer networks is a fundamental problem for network administrators. A number of tools exist that purport to produce accurate network load figures, however in practice very few actually do. In this paper we discuss the design of an Ethernet LAN load monitor, Loadman, which accurately reports network load. Loadman's design is based upon the loadring algorithm published by Jeff Mogul at DEC Western Research Labs in 1990, which in the absence of dropped packets guarantees to report accurate instantaneous load statistics. Loadman is a passive network monitor which has no impact on network load during use. We believe the use of the loadring algorithm guarantees Loadman to be one of the most accurate workstation based load monitors available.

1.0 Introduction

As distributed systems on Local Area Networks (LANs) become more common it is important to develop network monitoring tools that return statistics that are both useful and accurate. Although special purpose plug-in monitoring tools are available, they are usually expensive. Workstations are increasing in power while their cost remains relatively constant, and if they are part of an existing network, it is much more cost effective to monitor the network using an existing workstation [6]. Generally, the network hardware ignores all packets from the network that are not addressed to its own address. Most modern UNIX based workstations however also support a "promiscuous" network monitoring mode. This allows a workstation to receive all the network traffic which passes through the section of the LAN that the monitoring workstation is connected to. If a simple history of network traffic is maintained using the information supplied by a workstation's packetfilter, then the exact load figure, for this interval, can be calculated from the history data.

2.0 Background

For any network load monitor to be effective it must firstly be able to promiscuously receive network data. The mechanism provided on many workstations to do this is called the packet filter. A number of different packetfilters exist within most vendor supplied UNIX implementations.

It is important to remember that while a packetfilter is able to receive data-link level traffic, physical transmissions are shielded by the link level drivers. This means network errors such as collisions cannot be detected by a packetfilter. This is one of the negative aspects of using a workstation to monitor network traffic.

2.1 Available Packet Filters

2.1.1 Snoop - Irix

Snoop is the packetfilter which is provided as part of the IRIX operating system from Silicon Graphics. *Snoop* supports packet arrival testing as well as promiscuous network access. Promiscuous access allows all packets to be read from the network, regardless of their destination address. The *Snoop* packetfilter does not support batched reads, and can potentially incur large system call overheads as a result of having to perform a read for each packet.

2.1.2 Network Interface Tap - SunOS

The SunOS *Network Interface Tap* (*NIT*) supports packet arrival testing (`FIONREAD`), as well as promiscuous network monitoring. *NIT* also supports batched reads which results in a large cutdown in system call overheads. A major problem with *NIT* is that it only monitors the incoming network traffic, meaning it cannot "see" any of the transmissions from the monitoring host. This is a problem if the workstation is the source of a significant proportion of network traffic.

2.1.3 Packet Filter - Ultrix

The *Ultrix Packet Filter* can be configured so that any user may promiscuously receive packets. The *Ultrix Packet Filter* is superior to the *NIT* interface because it has the ability to monitor both incoming and outgoing Ethernet transmissions, allowing statistics to be gathered on the total visible Ethernet traffic. This packetfilter supports batched reads of up to 255 packets.

2.1.4 BPF Packet Filter - BSD

The *BPF* packetfilter is one the most efficient available [3]. It offers buffered reads, promiscuous network access and packet waiting tests. *BPF* uses a combined tap and filter that provides "in situ" filtering, resulting in a dramatic speedup in packet processing times. BPF requires operating system source to allow it to be implemented and used; it is for this reason that we were unable to use *BPF* in this project.

2.2 Sources of Inaccuracy in Load Monitors.

Network monitoring is not as simple as it seems. Mogul [6] articulated many of the sources of inaccuracy in network load monitors. The inaccuracies Mogul discusses include:

- "**Invisible" channel occupancy:** In network systems where many hosts compete to transmit as soon as the network becomes clear, it is often the case that the network is occupied with collisions, and so is not carrying any useful data. These garbled packets are kept hidden from the workstations by the link level drivers, therefore the network load calculations cannot include these figures. The other "Invisible" data on the network is fixed for each packet and so can be included into the calculations. Mogul states that this per-packet invisible data is made up of:

  - 96 bits for Inter-frame spacing
  - 64 bit for preamble
  - 32 bits for Frame Check Sequence

  This makes a total overhead of 192 bits or 24 bytes per packet.

- **Lost Packets:** If the network monitor cannot process packets as fast as the network can deliver them, then packets will be lost. Ethernet has a theoretical maximum packet rate of 14880 packets per second delivered at a bit rate of 10 million bits per second. For a monitor to keep up with the network, even over time intervals in the order of tenths of a seconds, the monitor must be capable of processing 14880, 64 byte packets per second.

- **Inaccurate Timestamping:** Timesharing systems cannot guarantee the regular servicing of active routines. Although this is unimportant in some applications, it generates inaccuracies in time dependant applications like load monitors. Many load monitors use system calls from within the application to find the start and finish time for each interval, so slight variations may occur in the interval length. This varying length skews both the instantaneous load figures, and the average load figures, as well as making it impossible to compare the figures between two machines. Fixed sampling intervals can also split a peak load between two intervals (See Figure 1) causing peaks to be missed. The use of a sliding interval ensures that all peaks are captured.

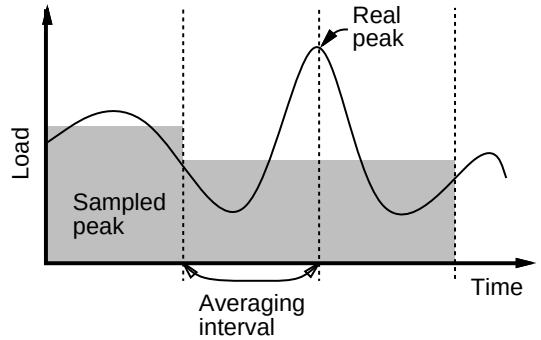

Figure 1:
Poor Interval Hides Real Peaks (Mogul 1990).

2.3 Existing Tools

Of the network load monitors available, there are a number of public domain implementations that are distributed as source code. Although we chose to develop our own code, examination of the publicly available network monitors revealed a number of potential shortcomings.

2.3.1 Xnetload

Xnetload was developed by Roger Smith from Sterling Software at the NASA-Ames Research Centre. This system is designed under SunOS 4.x using Remote Procedure Calls (RPC). The user must first install the *etherd* RPC daemon. *Etherd* gathers network statistics using the SunOS *Network Interface Tap* (*NIT*). There are two problems that occur through this system. Firstly, *NIT* only monitors the incoming queue from the network. This means that it cannot count the packets that originate from the local protocol stack. Secondly, if *etherd* is not running on the local machine then the client must put an RPC call across the network to the remote daemon, adding to the network load and possible inaccuracies.

The source code for *xnetload* reveals several noticeable points that generate inaccuracies in its load calculation. The most obvious error occurs in this section of code:

```
nload = (float) (((double)bits / TimeDelta)
      / ((double) (10*1024*1024))) * 100.0);
```

The 10 x 1024 x 1024, which results in 10 485 760, is supposed to represent the maximum bit rate for Ethernet. The correct figure, as pointed out by Boggs *et al* [1] is 10 million bits per second. The program uses the *timeofday()* system call to calculate the interval, and as mentioned earlier, the use of system calls carries large overheads. *Etherd* gives no indication of support for any statistics on dropped or missed packets. This means that for each dropped packet, an inaccuracy is introduced into the load calculation.

The fixed, per-packet overhead (i.e. Inter-frame Gap, Preamble and Frame Check Sequence) have been ignored. This overhead represents 24 bytes per packet, so ignoring these values will affect the result of load calculations, especially if the network is heavily loaded.

2.3.2 Xnetmon

Xnetmon was a monitoring tool developed by the Department of Electrical Engineering and Computer Science at the University of Wisconsin in Milwaukee. *Xnetmon* uses a *time()* system call to calculate the time interval used for the calculation of the average network load. *Xnetmon* also attempts to account for the "Invisible" network data which was ignored in *Xnetload*. *Xnetmon* adds 4 bytes to the size of the packet returned by the packetfilter to account for the Frame Check Sequence. Unfortunately the Inter-frame Gap and Preamble are missed causing each packet to be undersized by 20 bytes. Lost packets are counted by *xnetmon* using the mechanism that is provided by the *Ultrix Packet Filter*. Although the packetfilter is able to indicate that packets have been

dropped, it is unable to tell you the size of the packets dropped. This means that there is no way to maintain an accurate load figure over a time interval if packets are dropped during that interval.

2.3.3 XNetmon

XNetmon was developed at Delft University in The Netherlands. It is a different type of network monitor, because it examines kernel variables instead of promiscuously monitoring the network. These variables reside in kernel memory and are generally unavailable to normal users. The disadvantage of using this type of monitor is that it only provides statistics for the monitoring host.

3.0 The *Loadring* Algorithm

Mogul[6] outlines an algorithm that provides an accurate measure of network load over a time period.

In order to calculate an accurate measure of the network load at any given instance, it is necessary to maintain a set of network statistics for the past time interval. The total number of packets, the total number of bytes and the time interval are the three figures required. The "*loadring*" algorithm also relies on having the system place an accurate timestamp on the packets as they are read from the network. The algorithm uses a ring buffer to maintain a recent history of the network, hence the name *loadring*.

The algorithm comprises of two parts (See Figure 2). The first is the ring buffer, where each node on the ring represents a packet which contains the size of that packet in bytes and the timestamp showing packets arrival time. The second part is a group of counters which maintain the total number of bytes and packets that are in the valid time interval for the ring.

When a packet arrives, its length is added to the total number of bytes in the ring. The time stamp and number of bytes from the packet are stored in the front node of the ring and the number of packets in the ring is incremented. The time stamp is taken from the incoming packet and the timeout interval is subtracted from it. Any packets older than this new time are invalidated. As the packets are inserted in chronological order, the oldest packets are at the rear of the ring, therefore the removal time only has to be compared against the time stamps of the nodes at the

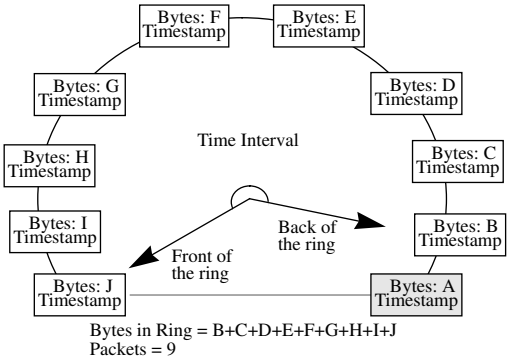

Figure 2: The *Loadring* Structure

rear of the ring. The pointer to the rear of the ring is moved forward until a packet is found that is "younger" than the timeout value. Because the timestamps are taken from the packets, not from a separate time call, the ring always contains data that is relevant to the previous interval. The interval slides rather than jumps, which means that all network peaks, with regards to the network interval, will be caught regardless of the interval size.

The byte and packet counters are stored separately from the ring data. This is an advantage because the load calculations can be made without affecting any of the data within the ring.

The algorithm is also able to cope with packets arriving infrequently. If the network is relatively idle with packets arriving at intervals greater than the history interval, then timestamps (from incoming packets) do not arrive often enough to generate timeouts for the packets already in the ring. This causes the data in the ring to become "stale". We overcame this problem using a "stale" flag which is set when a load calculation is performed and cleared when data is inserted into the ring. This avoids using a costly system call to obtain the correct time. If the stale flag is set when calculating the load, stale packets are invalidated before any calculations are performed.

4.0 Implementation

The *Loadring* algorithm has been implemented in C using the *Ultrix Packet Filter.* The *Ultrix Packet Filter* was chosen because of the advantages it offers over the other common packetfilters. Specifically, the *Ultrix Packet Filter* supports batched reads of up to 255 packets (unlike *Snoop*) , monitors both the input and the output buffers (unlike *NIT*) and allows promiscuous access for non-root users (unlike both *Snoop* and *NIT*).

The interval that the ring maintains has to be large enough so that the maximum packet size could easily be delivered within the interval. Mogul[6] suggests that intervals in the order of tenths of seconds are best, so the initial implementation aims at an interval of one tenth of a second. An Ethernet can deliver up to 14880 packets per second and so for an interval of 0.1 seconds there needs to be room to accommodate 1488 packets. This was rounded up to 1500 to eliminate any chance of pointer overflow.

An array of nodes was declared, with a doubly linked circular list constructed within it. This gave the (array) advantage that the exact number of nodes in the ring was always known (otherwise a table of free nodes would have had to be maintained). While the linked list structure meant that any sub-interval could be traversed from the front pointer working towards the back pointer to calculate load values for a time sub-interval. The array also allowed for the declaration of the structure as a block of contiguously allocated memory, thus reducing the probability of page faults. Each node on the ring contains:

- A pointer to the previous node.
- The number of bytes for this packet.
- A pointer to a character string. This is not used at

the moment but it will allow additional data about the packet to be stored (i.e. destination address, source address, packet type).
- A pointer to the next node in the ring.

The initial implementation of the algorithm was in three parts. The first section dealt with maintaining the ring structure: inserting a packet into the front node of the ring, incrementing the total bytes and total number of packets in the ring. The expiry time for the packets at the back of the ring was calculated using the timestamp from the new packet. Finally the back pointer is advanced until the time stamp of the packet under the back pointer is younger than the expiry time. Each time the pointer to the back node of the ring is advanced, the number of packets in the ring is decremented and the size of the packet at the back pointer is subtracted from the total number of bytes in the ring.

The second part of the code deals with the calculation of network load, peak load and average load. The network load is calculated as the total number of bits across the network during an interval, divided by the total possible number of bits that could of crossed the network in that interval. All figures were maintained as percentages of the total network bandwidth. As mentioned in Section 2 there are some "unseen" figures that need to be added in to the equation. The equation used to calculate the network load is:

Load = ((Bytes x 8) + (Overhead x Packets)) / (Interval x Maximum bit rate)

where: Overhead is 192 bits.
Maximum bit rate is 10 000 000 bps
Bytes, Packets and Interval are implementation specific.

The peak load is updated whenever a load value exceeding the current value is encountered. The average load is calculated using a formula for calculating sliding averages, similar to the one used in the *etherman*[7] package.

Average Load = Average Load + ((1 / (Number Averages + 1)) * ((Load * 100) - Average Load))

where: Load is the current network load and
Number Averages is the number of times the equation has been performed.

The final section is the code to get the packet details from the *Ultrix Packet Filter*. This section of code provides the input data for network load calculations, rapid servicing of this function will reduce the risk of dropping packets.

4.1 Reducing dropped packets

The solution to minimising the number of dropped packets can be found in the way feedback mechanisms operate. Feedback loops come in two varieties; negative and positive feedback loops. Negative feedback loops tend to move a system towards an equilibrium or balanced state. Positive feedback loops magnify their own effects, leading to instability (sometimes called a vicious circle).

In terms of reducing the number of dropped packets we need to employ a negative feedback loop to reduce the per packet processing time when a large number of packets are pending (see *eq*. 1).

$$\mathbf{CPU_required} = \mathbf{N_{packets}} * \mathbf{time_to_process_packet} \quad (eq.\ 1)$$

Using (*eq*. 1), if a large burst of packets were to arrive and the time taken to process a packet remained constant, packets would be lost when CPU_requried went beyond the machine's capabilities. In reality, what tends to happen when a burst of packets occurs is the time taken to process a packet actually increases since more interrupts occur, more data needs to be copied, etc. This results in a positive feedback loop since more time is required to process this batch of packets. Hence a longer time is available for the next batch of packets to arrive, increasing the probability of a overflow occurring before the next batched read occurs.

The solution to this "vicious circle" is to employ a negative feedback mechanism to reduce the per packet processing time in the presence of a large number of pending packets.

Loadman's packet reading mechanism implements a negative feedback capability by exploiting the packetfilter's ability to check if packets are pending during the packet processing (via `FIONREAD`). The original algorithm reads a batch of packets from the packetfilter and processes the entire buffer. When the buffer is empty the next batch of packets is read. The "while no packets are buffered in kernel" [7] algorithm (see figure 3) checks for pending packets during packet processing which, if present are immediately copied into a temporary pseudo-ring buffer. Packet processing is resumed when no packets are pending.

```
while (true) {
    read a batch of packets
    put packets in pseudo-ring buffer
    while (no packets pending){
        remove one packet from the ring
        process packet
    }
}
```

Figure 3: "while no packets buffered in kernel" algorithm

The difference between the two algorithms is that the later method prevents large numbers of packets from building up in the packetfilter's buffer (kernel input queue), increasing the risk of queue overflow. The risk of overflow is greatest when a large number of small packets arrive quickly. This "high burst" network traffic presents the biggest problem on slower machines. The packet waiting test allows the user process to detect whether the queue contains packets to be processed and reduces the probability of packet overflow.

5.0 Loadman

Loadman allows easy visualisation of network traffic, as can be seen in Figure 4. This graph represents the traffic flow over 500 seconds on the Curtin University Computer Science Ethernet backbone. As can be seen

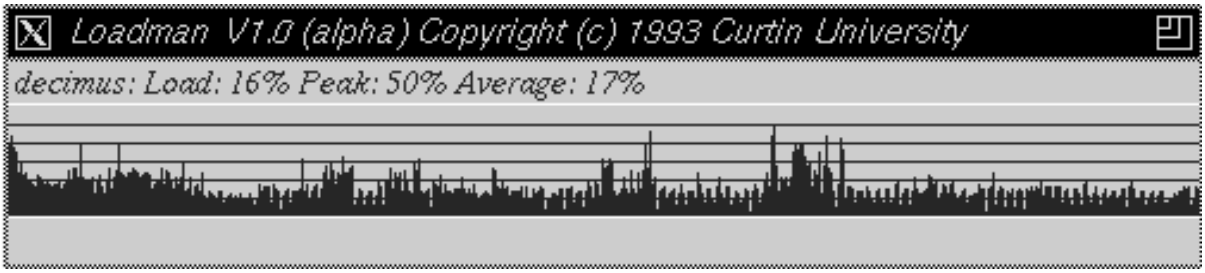

Figure 4: Computer Science Ethernet backbone traffic

the traffic level averages around 17% of total available bandwidth, with occasional bursts of higher traffic (the peak burst during this monitoring period being 50%).

Figure 5 shows the peak load, dropped packets, and average load logged over a three and a quarter hour period for the Computer Science Ethernet backbone. During this session log files were updated at 5 minute intervals. When log files are being generated the *time()* system call is used approximately once every second. This call is used to guarantee that the logfile is updated at correct intervals regardless of network traffic.

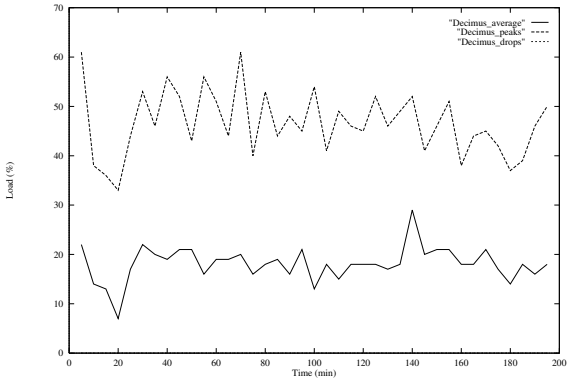

Figure 5: Computer Science Network Load History.

The DECstation 5000/240 running *loadman* did not drop any packets during its logging. Therefore these figures are an accurate representation of load on the Computer Science departmental network during the monitoring period.

6.0 Problems

The only known problems associated with *Loadman* result from the dropping of packets. Obviously, dropping packets is not a problem unique to *Loadman*; in fact any network load monitor that drops packets will return a load value that is inaccurate. There are several reasons why *Loadman* could drop packets:

- The user process could be swapped out when a large batch of packets arrives.
- The kernel could be overrun with interrupts caused by high packet arrival rate.
- There could be too much data copying between user and kernel memory.
- The packetfilter may be incorrectly configured. It is possible to alter the length of the input queue the kernel uses to hold waiting packets. If this queue is too short then an incoming burst of packets may overflow the queue and dropped packets may result. If the queue is too long then the kernel resources may become exhausted resulting in poor performance. A suitable length must be chosen to balance the kernel resources with the rate of incoming network traffic.

Loadman records the number of packets it drops but the *Ultrix Packet Filter* can only supply the number of packets dropped, not their size. The lack of this data prevents the use of an estimate to maintain a correct load figure when packets are lost.

7.0 Future Plans

It is planned to improve *Loadman* so that it can:

- Provide load statistics for a specific host or list of hosts.
- List top ten busiest hosts.
- Provide load between a pair of hosts.
- List load by Ethernet protocol.

These added functions can be implemented through the use of the augmented *loadring* algorithm[6]. The augmented *loadring* maintains additional data in each node on the ring. Data such as packet protocol type, destination address and host address are kept on each node. Additional counters must be maintained to generate the required statistics. The augmented *loadring* requires a much more complex ring maintenance procedure as data is being tested, updated and maintained in several different structures at once.

8.0 Conclusion

Although there are many network specific load monitors, most lack the accuracy afforded by *Loadman*. It is our belief that we have implemented one of the most efficient and accurate workstation based Ethernet load monitors available today. *Loadman* is hardware independent and can be easily ported to any network that can guarantee a fixed bandwidth and that can offer a packetfilter which supports a promiscuous mode of operation. The heart of Loadman is the *loadring* algorithm designed by Jeffrey Mogul at DEC WRL in 1990. This algorithm, provided there are no dropped packets and accurate packet time stamping, guarantees accurate network statistics.

9.0 Availability

Loadman will be made available as part of the *Netman* project from the School of Computer Science at Curtin University of Technology, Perth, Western Australia. *Loadman* will be available via anonymous ftp from host ftp.cs.curtin.edu.au as a binary distribution for DECstations. The file required is loadman-1.0.tar.Z, from the directory /pub/netman/dec-mips.

10.0 Acknowledgements

Many thanks to Jeff Mogul, Steve McCanne, Van Jacobsen, Christopher Kent, David Boggs and all the Internet community for the knowledge they share through their papers and software. Thanks also to those people who made their source code available for examination; Roger Smith for *xnetload*, Louis Weerpass and Wolter Haverhals for *XNetmon* and James Jegers for *xnetmon*.

11.0 References

1. Boggs, D.R., Mogul, J.C. & Kent, C.A (1988) Measured Capacity of an Ethernet: Myths and Reality. Proceedings of the SIGCOMM 1988 Symposium on Communications Architectures and Protocols, ACM SIGCOMM, Stanford, California, August 1988.

2. Jegers, J. Xnetmon (1992) Source Code. Department of Electrical Engineering and Computer Science, University of Wisconsin, Milwaukee.

3. McCane, S. & Jacobson, V. (1993) The BSD Packet Filter: A New Architecture for User-level Packet Capture. Proceedings of the Winter 1993 USENIX Conference, Jan. 25-29 1993, San Dieago, California.

4. Metcalfe, R.M. & Boggs, D.R. (1976) Ethernet: Distributed Packet Switching for Local Computer Networks. Communications of the ACM v19 n7 pp395 - 404

5. Mogul, J., Rashid, R.F. & Accetta, M.J. (1987) The Packet Filter: An Efficient Mechanism fro User-level Network Code. Proceedings of the 11th Symposium on Operating Systems Principles, ACM SIGOPS, Austin, Texas, November 1987.

6. Mogul, J. (1990) Efficient Use of Workstations for Passive Monitoring of Local Area Networks. Proceedings of the ACM SIGCOMM 1990 Symposium on Communications Architectures and Protocols.

7. Mogul, J. (1993) Personal communications.

8. Netman Project (1993) Etherman Source Code, Department of Computer Science, Curtin University of Technology.

9. Smith, R. Xnetload Source Code. Sterling Software at NASA - Ames Research Centre, Mountain View, California.

10. Weerpas, L. Th, & Haverhals, W. A. P. XNetmon Source Code. Delft Technical University.

11. Data Link Provider Interface Specification. Unix International, OSI Work Group, August 1991.