Optimizing TCP in a Cluster of Low-End Linux Machines

ABDALLA MAHMOUD, AHMED SAMEH, KHALED HARRAS, TAREK DARWICH
Dept. of Computer Science,
The American University in Cairo,
P.O.Box 2511, Cairo,
EGYPT

Abstract:- In this paper we describe the current enabling Clustering technologies and present the design of the LNP (Legendary Now Project) 8-node low-end Linux machines cluster. The main focus is optimizing the TCP layer in the TCP/IP suite. We have implemented three basic modifications: eliminating redundancy in the computation of the checksums, modifying the queuing of small packets, and directing incoming packets to follow the slow-path by default. Eliminating the redundant computation of the checksum saved computation time on both the sender and receiver sides. Removing of the queuing of small packets accelerated their arrival time, and finally forwarding the incoming packets to follow the slow path by default saved computation time spent on prediction.

Key Words: Clusters, Message Passing, TCP, Checksum, Packet Queuing.

1 -Introduction

Computing power and technology have evolved over the past few years. Standalone machines are now conventionally used for lightweight loads, personal and simple business applications. As users strive for more power along with the rising need for sharing data, information and expensive resources, networking technology has greatly advanced to satisfy those needs. Meanwhile, large databases, complicated graphical applications and scientific applications are requiring even higher computing capabilities that cannot be met except by supercomputers. As a result of current communication speeds in networking, along with the increasing demand for supercomputing power with lower costs, networks of workstations that act as clusters have been developed to satisfy these growing needs. The current exponential advancement in computing power and capabilities has led to a visible disadvantage; computers nowadays are forced to retire at an early age when they are still capable of giving and functioning properly. As a result of this, we have decided to recycle these retired machines by combining them into a network of workstations that would ultimately provide an overall increase in performance using components that were "sentenced to death".

2 -Configuration Issues

There are many ways of configuring each component or layer in a cluster [2]. Configuration issues relate to: Hardware, Operating systems, Network setup and Communication software. The choices depend mainly on the amount of money available for buying components, the type of work that needs to done on the cluster, and finally the issues of compatibility.

Hardware which consists mainly of PC's, varies according to the processor type like Intel based architectures: the x86 Family in general (especially Pentium, PPro, PII, PIII) and different AMD Processors, Sparc, Alpha or Macs. For Operating systems, there are many operating systems available and may all support cluster processing. The following are just a few to name: Solaris, Windows NT, FreeBSD (UNIX based operating system for Intel PCs), NetBSD (extended FreeBSD: available for more architectures) and Linux. The networking configuration is one of the hardest choices where an ever-increasing range of networking technologies and products are being developed, and most are available in forms that could be applied to make a cluster out of a group of machines. Some networking technologies that vary highly in cost, bandwidth and performance are: ATM, Ethernet (normal, fast and gigabit), Myrinet, SCI (Scalable Coherent Interconnect), SLIP (Serial Line Interface Protocol) and the USB (Universal Serial Bus).

Finally, we come to the Communication Software, also called Message Passing Systems, which is the layer that is responsible for distributing messages between different nodes. The most famous message passing systems are PVM and MPI [3]. PVM (Parallel Virtual Machine) is a freely available, portable, message-passing library generally implemented on top of sockets. It is clearly established as the standard for message-passing cluster parallel computing. PVM supports single-processor and SMP Linux machines, as well as clusters of Linux machines linked by socket-capable networks (SLIP, PLIP, Ethernet, ATM, etc.). The great advantage of PVM is that it can work across groups of heterogeneous nodes, or in other words clusters built up of machines with different types of hardware (processors, network cards) and different configurations. Nevertheless, PVM requires a specific Operating System, which is Linux. This concept of heterogeneity could also be expanded, in theory, to the point where the machines linked together through the Internet act as a single, huge parallel cluster. Another important advantage is its free availability and ease of download. And this has naturally led to the development of many programming language compilers, various application libraries, debugging tools, etc. specifically designed to work on top of the PVM platform. However, there are some significant disadvantages related to the use of the PVM as a communication software. First, message-passing calls generally add an important overhead to standard socket communication operations, which already had high latency. Second, it is very difficult to program using PVM calls, which require relatively highly skilled programmers to handle them. In other words, PVM does not provide a user-friendly environment for programmers.

The other, equally popular, communication software is MPI, or Message Passing Interface [4]. MPI is known as the official new standard, and is largely based on the old PVM. PVM has a unique execution control environment, and hence making a program execution done the same way everywhere. On the other hand, MPI is highly dependent on the implementation being used, and thus does not specify how the control environment is being implemented. The second issue is that of heterogeneity. PVM was designed in the first place to support clusters of heterogeneous machines. But MPI assumes a high degree of parallelism across workstations, or to MPP (Massively Parallel Processor) architectures. Third, MPI provides much more functionality than simple message passing techniques, such as Remote Memory Access (RMA) and parallel file I/O [5]. So PVM is more specialized than the generalized MPI message passing system.

3 Cluster Design

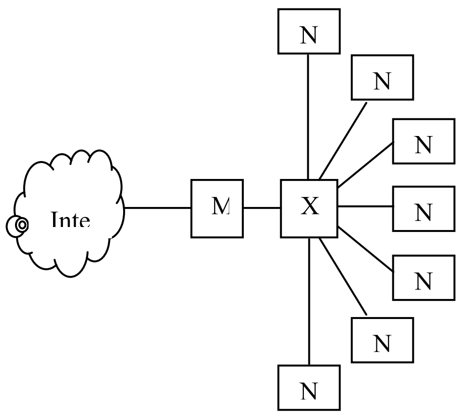

Figure1. The LNP Cluster Design

Figure 1 shows the layout of the LNP 8-node cluster. This design emphasizes the fact that the external world (the internet) does not see the cluster; it only sees the main node. The same happens on the other side, the cluster (nodes) cannot see except themselves and their server, but know nothing about the outside world. In this case the cluster behaves and acts as one entity. All the configuration issues mentioned in the previous section were decided upon in the LNP Cluster. The hardware used, are Intel based machines varying from 486 to Pentium-I (233 MHz), the OS is Linux 2.2.12 kernel, the network configuration is a Xyplex switch with 10 Mbps Ethernet cards, star topology and finally, the virtual machine used was PVM 3.4.3. Security is eliminated between the nodes of the cluster to enable remote access (rlogin, rsh… etc) from other internal nodes. All the nodes are connected together via a 10 Mbps switch, each having what we call a virtual IP. The main node is connected to the cluster and the Internet via two different Ethernet cards. This means that the main node has one main IP and another virtual IP.

4 TCP & Reverse Engineering

After introducing the cluster design and various configuration issues, we now move to the second phase that is optimizing TCP. After narrowing down our study to the TCP layer in the TCP/IP suite, we now discuss the reverse engineering of the TCP suite under Linux [6].

Working with a legacy system like TCP/IP Suite, we had to answer the question of what are the different approaches we could adopt to analyze this Suite (especially the TCP layer within the suite). Before we decide on the possible changes to make in the code, we had to carry out an extensive analysis of the code, find the different links between functions and modules, understand what each function does, and most important of all, identify the traditional paths in the communication process under TCP: namely establishing a connection, sending and receiving a packet, and terminating the connection. In other words, we used the reverse engineering technique to derive the design and specification of the system from its source code.

Following this, we ended up having a physical chart of code dependencies and intercommunications. Of course, reverse engineering to develop a better understanding of a system is a part of the reengineering process. One of the tools used was a program called Portable Book Shelves (PBS) for visualization of the code [7]. Along with the LXR Web Page [8] to follow functions and variable dependencies, we traced dependencies and definitions of variable and functions. Thus, having a physical chart mapping theory of the code to reality, we ended up optimizing the TCP layer as presented in the next section.

5-Optimizing TCP

It is known that packets sent are check-summed and a specific value is then stored in an appropriate field in the TCP header. Also, all incoming packets are check-summed to verify that they are correct, and that the value in the header matches the real, computed checksum on the packet. If these values match, then the packet is considered to be correct; if not, the packet is faulty and a retransmission request is issued. This is the way TCP deals with error control.

After thorough analysis of "layers" below the TCP layer, we discovered that each layer has some way of error control, built in it. For example, the IP layer does error control in its header; the Ethernet layer does low-level error control too [9]. We need to go back in history to understand why this was done in the first place. When the communication protocols were first invented, the hardware used at that time was not as reliable as today's hardware: error rates were high, even on short distances, and within the same machine. In addition to that, because of the idea of the "universality" of TCP, in other words the ability to use it over very large distances, the assumption was that when distances increase, the probability of error increases. Another issue behind this multi-layer error control is the modularity of each layer involved: each one has its "own" environment to deal with. Therefore, the error control was done at different levels. So the idea that we had was to remove the error checking from within the TCP layer, or at least reduce it to the minimum, so that computation time needed for check summing is saved, and thus latency is reduced. It is important to note that the reliability of our system was not compromised for saving time: error control was still done at other layers, and in addition we conducted a number of tests to monitor the error rate on our cluster, which we found to be zero at all times.

In an attempt to improve efficiency in standard TCP, the designers of this communication layer introduced the idea of queuing small packets before they are sent, so that a big packet of small packets is built and sent one time. The way it is implemented in TCP is that packets are checked for their size, and if a packet is "too" small to be sent right away, or in other words its size is less than the maximum segment size agreed upon between communicating ends, this packet is queued in a specific queue so that other small packets that could possibly fit in are assembled together into one big packet and sent in one time, instead of doing it many times. Timers are specified to monitor this, and to put the packets that have been queued for a "long" time on the line and transmit them [10].

When the designers of standard TCP thought of this idea, they had in mind the transmission time over long distances, and the overhead of adding headers on packets that are too small, especially when these are sent to the same destination. For example, when a packet is sent overseas, the transmission time is to be considered, not only because of the distance, but also because of the different network problems that could occur on the way, such as congestion for instance. This is not true in our case, where we have very short distances, no congestion at all, transmission time is negligible, and therefore problems related to network are non-existent. In addition to the mentioned facts, cluster computing is heavily based on the fast exchange of small-size messages, and this was the reason of thinking of eliminating the queuing of small packets all together. So the time the packets, which are small in size in most of the cases in clustering environments, wait before being sent is large enough compared to the time it would take to transmit physically the packet, and to the overhead added by the header on each one of them. This approach is wasteful, thus we thought of modifying it, and allowing small packets to be sent as they are built.

In order to speed up the processing time spent when receiving a packet, standard TCP defines a "header prediction" mechanism, which is responsible for "guessing" what will be in the packet itself from the first few fields of the header, and by recording the link identification between the source and destination [11]. According to this header prediction, a received packet follows one of two paths in the receiving process: the fast path or the standard, slow path. The fast path really cuts down some of the computation time when the packet satisfies one of the conditions known through the prediction mechanism. However, it works only when there are pure senders or pure receivers, that is, one end is always sending and the other one acknowledging the received packets; in this situation, either the sequence number or the ACK value must stay constant. When these conditions are not satisfied, the packet drops into a standard receive procedure that handles all cases: this is what is called the "slow path".

6-Experimentation and Results

A procedure was set to measure the performance of the cluster in general and the results of different modifications, as we moved on. Since our focus was on elements inside the Linux Kernel communication suite, we had to devise a flexible testing strategy to be applied on an "ongoing" basis. First, we had to make extensive measures on the standard version of the Kernel, so that we refer to these (use them as reference) to evaluate the impact of our *changes*. After each specific modification within the code, we reevaluated the performance of our cluster and compared it relative to the standard set or to other different results we obtained in other attempts. Finally, we evaluated all the results we had at our disposition, rejected the "useless" ones, and combined the attempts that had a considerable improvement.

To measure the performance of the communication suite in our cluster, we used several benchmarking tools, mainly Netpipe [12], Netperf [13], and Povray [14]. By taking the end-to-end application view of a network, Netpipe clearly shows the overhead associated with different protocol layers. Netpipe really helps determining the time needed to transmit a data block of a specific size, the maximum throughput and saturation level, and comparing performance between different protocol layers, namely TCP and the virtual machine. Netperf is a benchmarking tool that can be used to measure various aspects of networking performance. Its primary focus is on bulk data transfer and request/response performance using either TCP or UDP. Povray is a little bit different. It is a 3D-Image rendering application that makes use of parallelism, specifically designed to give information about the performance of parallel and distributed systems. It tells about the time needed to render an image, load balancing within the system, and other connection-related data. After having implemented the three changes we mentioned above, we tested the new TCP, through the benchmarks, to measure the improvement. The results were good and were "constant" in a sense: we were able to record an improvement of 5.7% on a packet size of 1 byte. The improvement we obtained for packets less than 128 bytes was always greater than 2.6%, for the range of 128 bytes to 259 bytes it was above 1%, and for the rest an improvement of less than 1%. The following graph depicts this, it shows the time needed to transmit a packet of a specified size. We were able to shift the curve down:

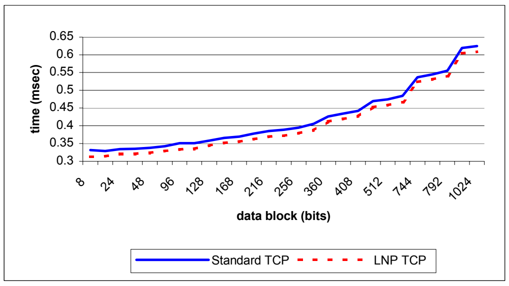

Figure2. Zoomed Transmission Time VS Data Blocks

We also have undertaken a series of tests on the PVM using our modified TCP. The ranges of results were very similar to the ranges for the TCP tests, which is very natural. However, the upper limit for the improvement was less than that measured when testing TCP only: for one byte, the improvement was about 3.8%. But in this case, the distribution of the performance on different ranges was more normal than in the case of TCP; in other words, we measured an improvement of more than 1% on all packets with size less than 1K. The graph below illustrates this (the right side of the graph):

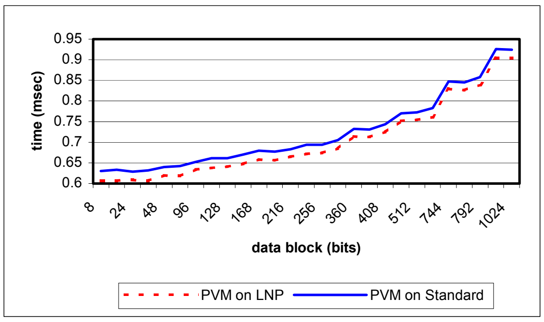

Figure3. Zoomed Time of PVM on

We conducted three sets of tests using Povray: always keeping the same size of the image, we tested rendering the same image using different block sizes. The first test was on 4X4 pixels, in this case, more traffic exists in the network because we have more frames to exchange. The results conformed to our previous results; we were able to render the image, on two nodes, in 355 seconds on the modified TCP, whereas it took 359 seconds on the standard TCP. This diagram illustrates this:

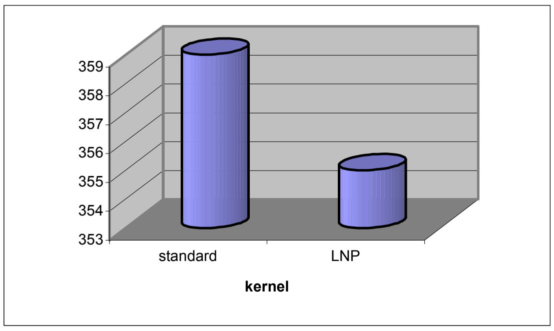

Figure4. Time To Render Image (divided by 4x4 pixels)

We tested rendering an image divided into 16X16 pixels frames, and 32X32 pixels frames. And as we expected, the bigger the block size, the less visible the improvement is. In the former, we saved 3 seconds; in the latter, we saved 1 second. The reason for the decreasing performance along with the increasing block size is easy to understand. When packets are large enough, the computation time (on headers, etc…) is much less important relative to the total size, than when the same computations are done on small packets with *com*

less data. Another point is that the elimination of the queuing, which originally took place in the case of small packets, had an effect on this sort of packets only. In other words, the smaller the packets, the more queuing was done in the standard TCP, whereas this is not true anymore in the modified TCP.

7 Conclusion

Currently, we have our TCP working only on our cluster environment; in other words, when we are disconnected from the external world. The need to maintain the connection with the external world is necessary, and based on this, the TCP could be future enhanced to be able to dynamically switch between two paths of computations that it would go through: The first path is the standard TCP, and the second is our optimized cluster based TCP. This could actually be done by using the reserved bits in the TCP header, that designers long ago left them for any possible future uses.

9 References

[1]http://www.dgs.monash.edu.au/~rajkumar/cluster/index.html
[2] http://www.parasys.co.uk/install.html
[3] http://www.epm.ornl.gov/pvm/
[4] http://www.isi.edu/atomic2/gbn95/user-bw.html
[5]http://cch.loria.fr/documentation/local/PvmBook/node89.html
[6] Sommerville, Ian. Software Engineering. Addison Wesley, London 1998.
[7]http://www.grad.math.uwaterloo.ca/~toparry/pbs/common/html/pbs.html
[8] http://www.lxr.linux.no
[9] Stevens, by W. Richard .TCP/IP Illustrated, Volume 1: The Protocols. Addison Wesley, London 1994.
[10] Beck, Michael, et al. Linux Kernel Internals. Addison Wesley, London 1998.
[11] Wright, Gary R., et al. TCP/IP Illustrated, Volume 2: The Implementation.
Addison Wesley, London 1995.
[12] http://www.icase.edu/coral/LinuxTCP2.html
[13] www.beowulf-underground.org/software.html
[14]www.dmoz.org/Computers/Internet/Protocols/Transmission_Protocols/Networking_Resourc -es/Addressing/
[15]http://www.protocols.