

A Middleware-Level Parallel Transfer Technique over Multiple Network Interfaces

Nader Mohamed, Jameela Al-Jaroodi, Hong Jiang, and David Swanson

Department of Computer Science and Engineering
University of Nebraska – Lincoln
Lincoln, NE 68588-0115
[nmohamed, jaljaroo, jiang, dswanson@cse.unl.edu]

Abstract.

Network middleware is a software layer that provides abstract network APIs to hide the low-level technical details from users. Existing network middleware support single network interface and link (channel) message transfers. In this paper, we describe a middleware-level parallel transfer technique that utilizes multiple network interface units that may be connected through multiple networks. A prototype socket called MuniSocket (Multiple Network Interface Socket) has been implemented to provide this functionality. MuniSocket provides parallel message fragmentation and reconstruction mechanisms in addition to load balancing. It operates on any reliable transport protocol such as TCP and transparently provides an expandable high bandwidth solution that (1) reduces message transfer time, (2) provides fault tolerance, and (3) facilitates dynamic load balancing among the underlying multiple networks. The experimental evaluation of MuniSocket illustrates good performance gains, where a peak bandwidth of 187Mbps was achieved on two fast Ethernet networks.

Key Words: Middleware, socket, parallel data transfer, and load balancing

1. Introduction

The growing popularity of clusters [3] and the grid [6] infrastructure have provided huge opportunities for accommodating compute and data intensive applications such as remote satellite observation, distributed data mining, and distributed scientific simulation experiments as in [10]. Such systems can be effortlessly expanded to provide more processing power and storage in a cost-effective manner. However, increasing the bandwidth of the interconnects is not as easy. Although current cluster configurations provide multiple network interfaces per node and multiple interconnections between nodes, current network protocols, software, and APIs such as Sockets are designed for a single physical network interface. This gives rise to the need for protocols, and software services that can seamlessly support multiple physical network interfaces on each node and provide transparent and efficient utilization of these interfaces on a cluster. One important approach to provide such utilization is

the striping technique [1], which can be implemented at different network protocol levels by distributing incoming packets among the available network interfaces. One example is channel bonding [16], which is implemented at the kernel level and provides high throughput for the applications. However, channel bonding is hardware and operating system dependant, and has some limitations (see Section 5).

In [11], we proposed a UDP-based protocol to utilize the existing multiple network interfaces to transfer large messages. To have a reliable transfer service with UDP, the protocol itself included a reliability mechanism in addition to fault tolerance. However, the reliability overhead made the UDP-based multiple networks protocol perform well with large messages while for small messages, the overhead is the dominant contributor to message latency. In this paper, we propose some techniques that utilize TCP's existing functions such as reliability and flow and congestion control to implement a more efficient TCP-based MuniSocket. These techniques make the transfer services efficient for both small and large messages. We propose a user-level socket model (MuniSocket) that utilizes multiple physical network connections on a cluster in a way that is transparent to the user application. We also describe a prototype implementation of MuniSocket. In MuniSocket, user messages are partitioned into uniformly sized fragments which are transferred in parallel, through multiple network interfaces, via one or more physical networks, to the destination again through multiple network interfaces. At the destination, fragments are reconstructed back to the original message (see Figure 1). The main difference between MuniSocket and the standard Socket [14] is that MuniSocket processes and transfers user messages in parallel, fully utilizing the existing multiple network interconnects, while the standard Socket processes and transfers the user messages sequentially through a single network interface. MuniSocket also handles the multiple interfaces at the application level, thus providing more effective services and better control over the system. Furthermore, the processing is done in low granularity by processing the application messages, thus reducing the overhead incurred by the protocol. Moreover, the technique provides fault tolerance and dynamic load balancing to achieve the maximum possible bandwidth. The experimental results confirmed the initial expectations and showed good performance enhancements that will benefit data-intensive applications running on clusters connected by multiple interconnection networks, by taking advantage of the physical and logical parallelism and the redundancy in network interconnects. The results obtained show that MuniSocket is useful in applications that need high and scalable bandwidth communications, but can afford to sacrifice some processing time in return. Examples of these applications are ones that use blocking (synchronous) data transfers, data replication, file transfer, and bulk data transfers.

In the rest of this paper, we first discuss the proposed technique and the prototype implementation, MuniSocket, in Section 2. Section 3 describes the enhancement of the model to dynamically select the transmission method. Then, in Section 4 we introduce a load balancing mechanism to optimize the performance for transmitting large messages over loaded and heterogeneous networks in terms of message transfer latency and effective bandwidth. Experimental measurements and system evaluations are included in sections 2 through 4. Section 5 gives a comparative perspective of the proposed technique and other approaches such as parallel sockets and channel bonding. Finally, Section 6 concludes the paper.

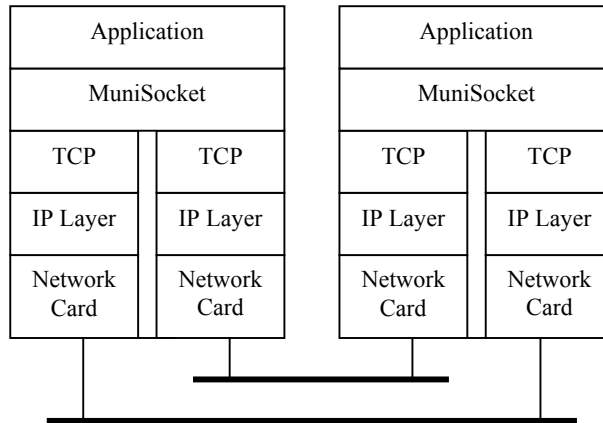


Fig. 1. Architecture of multiple network connections with MuniSocket.

2. The MuniSocket Model and Implementation

Although current technologies made acquiring high performance components at reasonable costs rather simple, the available software technologies are not yet adequate to seamlessly handle the different possible configurations of hardware components. For example, most cluster system configurations include two or more network interface cards (NIC), but the applications cannot seamlessly utilize these NICs simultaneously. Conventionally, each NIC on a machine is assigned an IP address; therefore, each machine will have multiple IP addresses. Most of the current distributed applications use TCP/IP protocols, thus requiring the use of a fixed IP address to identify the machine. This leads to the necessary requirement that a given application use a single IP address on each node to establish connections with other nodes, thus preventing the application from utilizing other available networks. This means that the bandwidth available for communication to any application is bounded by that available on the single network interface associated with the assigned IP address. Thus, even if there are multiple networks connecting the nodes of the cluster, only one of them will be utilized for any given application at any point of time. To satisfy the application's demand for higher bandwidth, with the current technology, the existing networks have to be replaced by more advanced/faster networks, which is a costly and non-scalable solution.

The proposed MuniSocket model provides an extendable solution to increase network bandwidth using multiple low bandwidth network interfaces. MuniSocket middleware resides above the TCP layer and interacts directly with the application. Any message generated by the application is processed by MuniSocket instead of the standard Socket. The message is fragmented into smaller fragments and then transmitted in parallel through the available interfaces to the receiving end. At the receiving

end, the message is reconstructed from the fragments and passed to the application. This process is performed in parallel utilizing the available multiple processors on each of the cluster nodes.

2.1 Basic MuniSocket Implementation

MuniSocket is a prototype middleware layer that resides between the applications and the available network resources (see Fig. 1). It provides primitives to transparently transfer large messages through multiple physical networks. TCP/IP is used for implementing MuniSocket for the following reasons:

1. TCP is readily available for all operating systems and the majority of network types. Adopting TCP facilitates porting and using MuniSocket on heterogeneous systems connected by heterogeneous networks.
2. TCP/IP is part of the current Internet infrastructure such as Internet routers. As a result, MuniSocket can be used on nodes connected over the Internet.
3. TCP provides efficient multiplexing among multiple logical communication streams, which is needed by any transport or middleware layer protocol.
4. Many companies and researchers are working on reducing the overhead of IP and TCP and providing network API implementations that utilize specific advanced types of networks [5][9], thus MuniSocket can benefit from these advances.

Using multiple networks to transfer messages requires addressing many issues to provide a complete and efficient solution. Such issues include, but are not limited to the following:

1. The main problem with fragmenting the message and sending the fragments through multiple networks is the out-of-order arrivals of fragments at the destination. A mechanism to reorder the received fragments is needed.
2. The load on each network varies over time. To approach peak bandwidth performance, it is critical to have a dynamic load balancing mechanism to distribute the workload among available networks based on current available bandwidth.
3. Each network may have different latency and bandwidth values. Thus, it is essential to design an efficient algorithm to deal with this type of heterogeneity in a transparent way.
4. The above requirements need to be implemented using very efficient algorithms to avoid introducing high overhead with the solution.

The components of each MuniSocket include one sending thread per NIC and one receiving thread per NIC (see Fig. 2). Each message is divided into two or more equal fragments (depending on the number of available network interfaces) that are then sent using TCP. Multiple threads are used to process the message and prepare the fragments for transmission. Each sending thread uses a different TCP port and buffer. On the other end, the receiving threads accept the fragments and combine them in the receiving buffer provided by the user. In this approach, there are no extra data copies other than those made by TCP operations for both the sending and receiving sides.

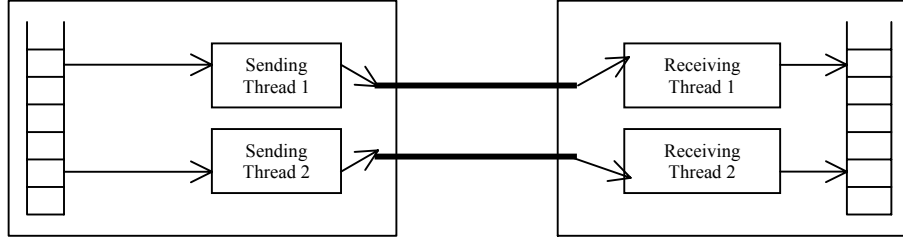


Fig. 2. The MuniSocket Architecture: the sender components are on the left and the receiver components are on the right. The sender and receiver are connected by two physical networks.

2.2 Performance of MuniSocket

To evaluate the performance gains of the basic implementation of the proposed model, MuniSocket, a number of experiments were conducted. All experiments were executed on Sandhills, a 24 dual processor node cluster. Each node contains two AthlonMP 1.2GHz processors with 256KB cache per processor and a total of 1GB RAM per node. Two nodes were equipped with two Fast Ethernet cards and two Gigabit Ethernet cards. The experiments were designed to measure the round trip time (RTT) so that the effective bandwidth for the single network Socket and for the MuniSocket using two Fast Ethernet networks and two Gigabit networks, can be derived indirectly from the RTT as follows:

$$Bandwidth (Mbps) = (8 * Message\ size / 10^6) / (RTT / 2)$$

The communication performance in terms of round trip time for TCP over fast Ethernet (TCP 100), TCP over Gigabit Ethernet (TCP 1000), MuniSocket over two fast Ethernet cards (MuniSocket 2*100), and MuniSocket over two Gigabit Ethernet cards (MuniSocket 2*1000) is shown in Figure 3 (Message sizes increases in power of two increments). The results show high RTT values for MuniSocket 2*100, as compared to TCP 100, for messages smaller than 1KB. However, as the message size increases the gain (reduction in RTT) for MuniSocket 2*100 becomes evident and clearly shows the benefits of this method. Similarly for the Gigabit networks, the TCP 1000 is better than MuniSocket 2*1000 for small messages, but it gets better with messages larger than 32KB. Figures 4 and 5 show the bandwidth gain incurred by MuniSocket for the fast and Gigabit Ethernet networks, respectively. In both cases, the benefit of using MuniSocket is evident, especially for the fast Ethernet, which shows over 90 percent efficiency for messages larger than 32KB and reaching up to 99.4 percent for a 2MB message. Where

$$Efficiency = (BW(MuniSocket) / BW(TCP) / No. Interfaces\ used) * 100$$

On the other hand, the overhead imposed by MuniSocket was measured during several experiments. The results show that this overhead is fairly low and grows slowly with increasing message sizes. However, this overhead is negligible with large

messages. Table 1 gives a sample of the results obtained on the dual processor nodes for TCP 100, TCP 1000, MuniSocket 2*100 and MuniSocket 2*1000 for a message of size 16MB. The CPU utilization, per processor, includes all incurred processing including both the TCP overhead and the MuniSocket overhead. In general, this overhead is acceptable given the high gain in bandwidth especially for the Fast Ethernet networks.

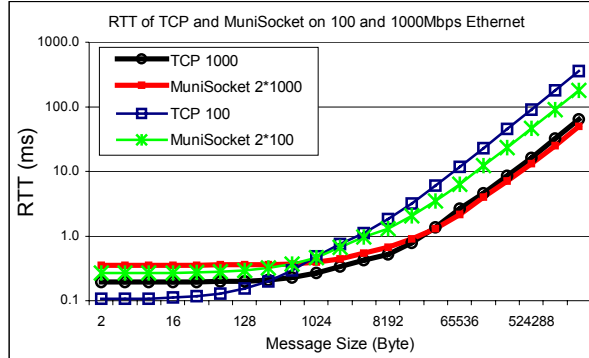


Fig. 3. Return trip time measurements of TCP and MuniSocket over fast and Gigabit Ethernet.

Table 1. CPU utilization for ping-pong program using TCP and MuniSocket on fast and gigabit Ethernet

Method used	TCP 100	MuniSocket 2*100	TCP 1000	MuniSocket 2*1000
Percent CPU Utilization per processor	12.5%	20%	37.8%	48.4%

Generally, the results obtained show good speedup and efficiency performances for MuniSocket when applied to multiple networks. The overall efficiency decreases with smaller messages as latency due to overhead, which is relatively message size-independent, becomes more significant compared to the round-trip time. Basic MuniSocket can provide high bandwidth and shorter message transfer times for large messages. However, its main weakness is that it does not provide load balancing among the network interfaces. In addition, the overhead is considered high for small messages. The next two sections discuss solutions to the mentioned weaknesses. Another superficial problem is that the current implementation of MuniSocket is in Java. We expect to gain better performance using other compiled language such as C, especially with high-performance networks such as gigabit Ethernet. Nevertheless, using Java makes the implementation portable and supports different hardware/software configurations. In addition, the advancements in hardware technology and the Java virtual machine (JVM) performance will eventually make this problem insignificant.

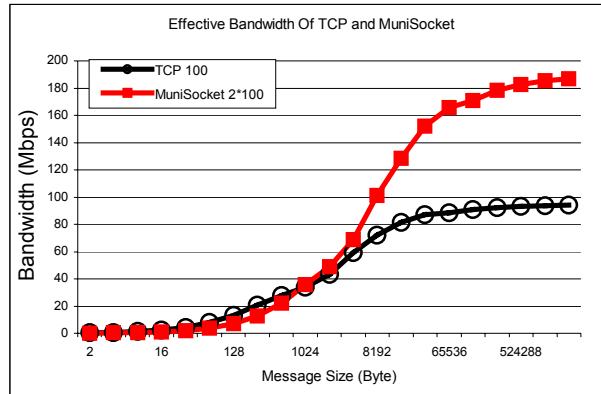


Fig. 4. Bandwidth using fast Ethernet

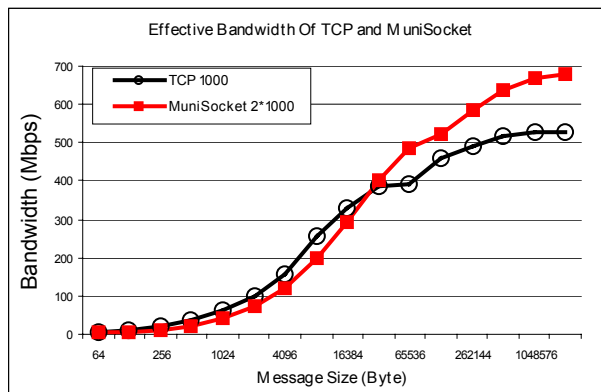


Fig. 5. Bandwidth using Gigabit Ethernet

3. Enhancing MuniSocket Performance

The basic model and prototype implementation, as described in Section 2, provide an efficient mechanism to utilize multiple homogeneous interfaces to achieve higher bandwidth. However, this mechanism does not give better performance at all message sizes. Using fast Ethernet, the experiments show some overhead for messages smaller than 1KB, but after that the performance improves dramatically giving much better results than TCP. To minimize the overhead relative to small messages, a dynamic decision has to be made when a message is received for transmission. The system should decide whether to divide the message or send it as it is on a single network interface based on the message size and a pre-determined cut-off value. To demonstrate this mechanism, assume we have two threads linked to two network interfaces. Let RTT_i be the return trip time of a message M on a single network and

RTT_2 be for two networks, which includes the threads overhead T_{oh} . The thread overhead T_{oh} is slightly dependent on the message size; therefore, we calculate the message size at the intersection point where $RTT_1(M) = RTT_2(M)$ (for example, the intersection point between TCP 100 and MuniSocket 2*100 in Figure 3). This intersection point represents the desired message size that can be used as a cut-off to transition from using a single network interface to utilizing two network interfaces.

Assuming dedicated networks, the system registers transmission times for both a single interface and two interfaces at start time. Using these determined values the cut-off message size can be calculated using different methods such as;

1. Mathematical models for curve fitting and calculating the message size at the intersection point.
2. Experimental search methods, which can be conducted at start time. A binary search algorithm can be used, which can be further optimized by estimating rough bounds for the search.

Using either method is considered feasible since it is done once at startup time. With the cut-off message size known, the system is able to decide whether to send the message on a single interface (when $M < cut_off$) or use two interfaces (when $M \geq cut_off$). Experiments were conducted to evaluate the performance of the enhanced MuniSocket over fast and Gigabit Ethernet networks. With this mechanism in MuniSocket, the overall performance improved for both the fast and Gigabit Ethernet networks (see Figures 6 and 7). From the experimental results we see that the enhanced MuniSocket performs well with all message sizes by combining the single and multiple interfaces utilization. In addition, this method applies to two or more interfaces in a similar manner, thus optimizing the overall performance.

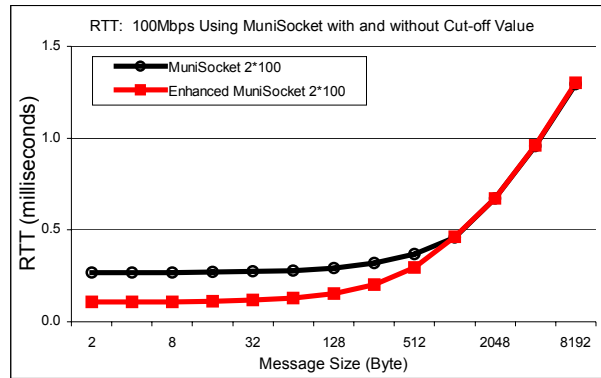


Fig. 6. Enhanced Vs. basic MuniSocket for Fast Ethernet.

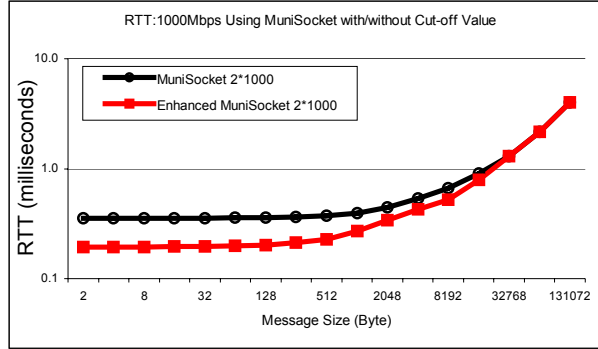


Fig. 7. Enhanced Vs. basic MuniSocket for Gigabit Ethernet

4. Load Balancing and Fault Tolerance

The basic MuniSocket implementation partitions the original message into two or more equal sized fragments to be transmitted over the available interfaces. However, if any of the interfaces has different bandwidth or latency than the others, the transmission time will be proportional to the slowest network, thus minimizing the gain. To solve this problem a dynamic load-balancing algorithm was introduced in MuniSocket to enhance the performance on loaded and heterogeneous networks. This method is devised to balance the load for large messages such that slower networks will not slow down the whole collection. The main idea is to fragment the message into a large number of small fragments that are transmitted over the available interfaces. Load balancing is achieved by having threads connected to unloaded networks process more fragments while other threads connected to loaded networks are blocked for longer periods by the congestion and flow control mechanisms of TCP.

The fragment status vectors maintain information about the packets status and provide a fragment sequence number to the sending threads. During message transfer, each fragment can be in one of three states on the sender (see figure 8). These states are maintained in the fragment status vector by the socket threads. On the sender side, a fragment can be either *ready*, *in-transit*, or *acknowledged*. When a sending thread starts to process a fragment, the fragment state changes from *ready* to *in-transit*. As soon as a fragment is correctly sent, the sender thread changes its state to *acknowledged*. Since TCP is used, the sending thread is able to know if the packet has arrived at the receiving end or not. The counter provides a sequence number that represents a fragment in the user message. If the fragment size is F bytes and message size is M bytes, then the counter provides a sequence number between 0 and $\lceil M/F \rceil - 1$. Each sending thread tries to take a number, i , from the counter, and prepare a header that contains the sequence number i . The thread sends the header followed by the corresponding message fragment from position $F*i$ to $F*(i+1)-1$ in the sender's memory. On the other end, the receiving threads accept the fragments and, based on the se-

quence number, copy the fragments to their proper place in the receiving buffer provided by the user. All sender and receiver threads perform their tasks in parallel.

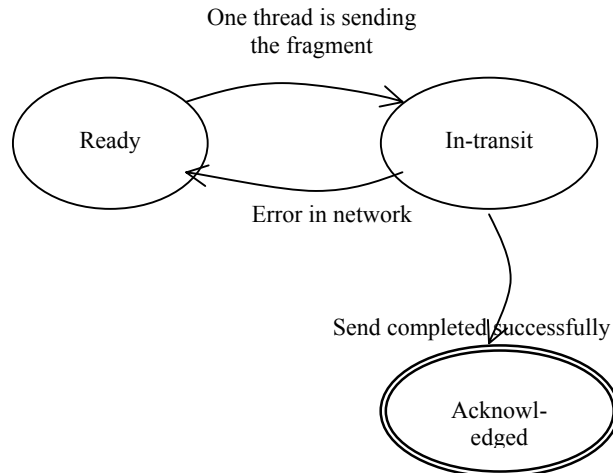


Fig. 8. State Diagrams of the Reliable Packet Transfer Protocol from sender side

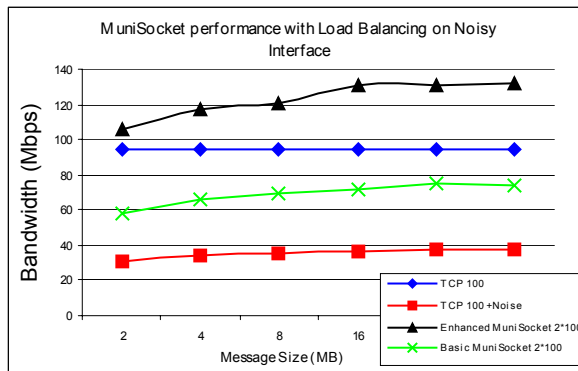


Fig. 9. Effects of load balancing on performance

Experiments were conducted using the dynamic load balancing technique in MuniSocket for large messages. Each message is divided into fragments of size 64KB. Using a separate application to simulate load on one of the interfaces, ping-pong was used to measure the performance of TCP and MuniSocket. Figure 9 shows four different parts of the experiment. TCP 100 represents the bandwidth on one un-loaded interface, while TCP 100+noise represents the effective bandwidth the other loaded Ethernet interface. Basic MuniSocket 2*100 represents the effective bandwidth when using both the loaded and unloaded interfaces at the same time without the dynamic load balancing mechanism. Enhanced MuniSocket 2*100 shows the effect of the dynamic load balancing mechanism on the same two networks. The results show that although one network can provide less than 40 percent of its peak bandwidth, we were

still able to achieve high bandwidth gain using the dynamic load balancing technique. In other words, without using MuniSocket we obtain an average of 64.21 Mbps using either network. However, using the enhanced MuniSocket we can achieve an average of 118.95Mbps using both interfaces simultaneously.

Fault tolerance is an additional advantage of using MuniSocket. It is achieved by allowing any sender thread connected to a stable network to continue sending fragments while making any sender thread connected to an unstable or disconnected network monitor the network status. When a network connection fails, the associated sender thread marks the fragment in the state *in-send as ready* such that another sending thread can process the fragment. The result will be a continuing connection, but with a lower capacity. Another necessary mechanism is added to provide fault recovery such that the system would include the interface if it regains its connection. This is done by associating sender thread probes to the interfaces that failed earlier to see if they are back online. When an interface is reconnected, the sender thread resumes normal operation and starts processing more fragments.

To summarize, although MuniSocket is efficient and produces good performance gains for medium to large messages, it still has some dominant factors that make it very difficult (if not impossible) to enhance its performance further for small messages. Similar to parallel processing where parallelism benefits mostly medium to large size computational problems but not small (or inherently sequential) problems, network striping can be utilized to enhance performance of medium and large messages. Similarly, just as the performance of a small or inherently sequential computational problem can only be enhanced by upgrading the processor, the performance of transferring small messages can best be enhanced by upgrading the network.

To illustrate the benefits of MuniSocket, a prototype implementation for a file transfer tool was created using MuniSocket to transfer files or parts of files between local hard disks of cluster nodes. The transfer times for 16MB and 32MB are shown in table 2. These basic functions are the basis for many applications that require file and data transfer among cluster nodes. Two famous examples are the file transfer protocol (FTP) and the distributed network file system (DNFS), where files need to be replicated and moved among different nodes based on user requests or current loads. As illustrated, MuniSocket provides large gains in the transfer time for such applications; therefore, using it will be advantageous.

Table 2. Measurement times for partial file transfer between two cluster nodes.

Part Size (MB)	Single Fast Ethernet Network		Dual Fast Ethernet Networks	
	Put time	Get time	Put time	Get time
16	1.511	1.521	0.834	0.847
32	2.924	2.940	1.544	1.563

5. A Comparative Perspective

The current research for providing high bandwidth solutions generally implies two directions, namely, virtual TCP connections and using the striping technique. Parallel

transfer at the socket and transport layer levels has been investigated by some research projects such as multiple TCP connections [12], multiple streams [4], and parallel socket [13]. However, these studies focus on achieving optimal bandwidth in wide-area networks by overcoming the limitations imposed by the TCP window size in a single network connection by means of parallel logical flows. This approach creates multiple virtual TCP connections for applications to saturate the single network connection and achieve peak bandwidth. One of the examples of multiple logical streams technique is found in GridFTP [7]. GridFTP uses multiple TCP streams to improve aggregate bandwidth over using a single TCP stream in wide area networks. In addition, GridFTP can transfer data from multiple servers where data is partitioned to improve performance.

The second approach is using striping [2] at different levels of the communication protocol stack to utilize multiple network interfaces. This technique is well-known in conjunction with storage systems such as in the redundant arrays of inexpensive disk (RAID) architecture [8]. However, in networks striping is used to describe the aggregation of multiple networks to achieve higher bandwidth and hence higher throughput. A number of research projects use the striping methods, such as the IBM project for striping PTM packets [15], and the stripe, which is an IP packets scalable striping protocol [1]. Another well-known implementation of the striping technique is the Ethernet channel bonding [16], which transparently provides a high bandwidth NIC comprised of multiple lower bandwidth NICs. The input of the striping algorithms is a set of packets or frames with varying lengths generated from different higher-level layers or different applications. The aim of the striping algorithms is to distribute these varying sized units into multiple available connections. In addition, the striping algorithms try to achieve load balancing for multiple networks and fairness in serving packets and frames coming from higher layers.

Our contribution, as demonstrated by MuniSocket, has a number of significant differences from the above-mentioned approaches. Our approach provides a middleware solution that is close to the application. This resembles the functions of the multiple virtual interfaces, but it utilizes multiple NICs for transmission rather than pooling all streams into a single NIC. Our approach also differs from the multiple logical streams by providing the basis for sophisticated reliability and fault tolerance mechanisms.

On the other hand, our approach is similar to the striping technique in terms of using multiple NICs to achieve higher bandwidth. However, it aims for lowering response time as opposed to higher throughput in the striping techniques. Another difference from the striping technique is that it distributes incoming packets from different applications among the available interface(s), while our approach divides the messages generated by the application into fragments that are then transmitted using TCP. As discussed in [2], striping can be done at different levels of the communication protocol stack. One example is channel bonding [16], which is implemented as a Linux patch that links NICs together at the transport queue level. This approach provides relatively good bandwidth improvements. An example benchmark on a cluster called JAZZ [17] shows an increase in bandwidth from 86.8Mbps to 160.4Mbps for a 1MB message and from 86.88Mbps to 162.08Mbps for a 4MB message on two fast Ethernet networks. Figure 10 shows the calculated speedup on two fast Ethernet networks achieved in channel bonding (using the JAZZ benchmark) compared to the speedup achieved by MuniSocket. Apparently, MuniSocket has better performance.

Furthermore, channel bonding is hardware and operating system dependant, is inflexible, and does not support heterogeneous interfaces. Some of the limitations of channel bonding are:

1. All NICs on a node must have the same MAC address.
2. Requires separate switches to connect the different NICs on the nodes (to avoid MAC confusion).
3. Non-bonded nodes to bonded nodes communication is too slow, thus all nodes on a cluster must be bonded.
4. Bonded NICs cannot be accessed individually, thus limiting their utilization.

These limitations are mostly imposed by the low level implementation of channel bonding. This is avoided in our approach, where the implementation is immediately below the application level. This provides more flexibility in handling application demands and is independent of the hardware and protocols used in the lower layers. In addition, our approach allows for using the different NICs as one unit or as independent units and allows for different configurations and hardware components to be used transparently.

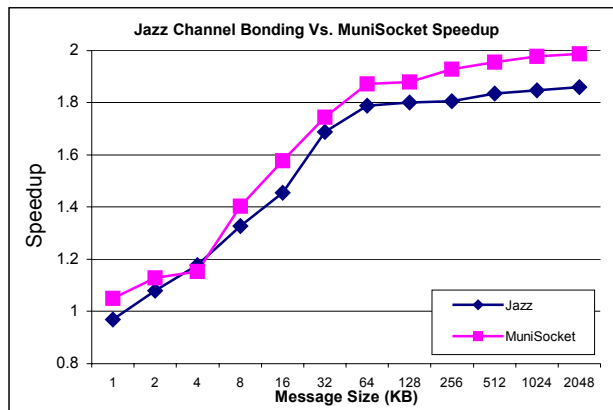


Fig. 10. Comparing Speedup of channel bonding and MuniSocket on two fast Ethernets.

We further distinguish our approach by utilizing the available multiple processors to perform the fragmentation and assembly of messages in parallel. Message processing is performed with coarse granularity, thus lowering the effective overhead by reducing the frequency of fragmentation and reconstruction. Another difference is that this method generates fragments of varying sizes to optimize the utilization of the available bandwidth and compensate for load imbalance. In addition, implementing our approach as a middleware over TCP provides a portable framework to be used on different platforms and on heterogeneous systems. This provides a cost-effective and scalable solution to providing high bandwidth, load balancing among the interfaces, and fault tolerance for data-intensive distributed applications.

6. Conclusion

In this paper, a middleware-level technique to utilize the existing multiple network interfaces on clusters was designed and discussed. This model, represented by MuniSocket, provides expandability, high bandwidth, fault tolerance, and load balancing using the available multiple network interfaces. In MuniSocket, message fragmentation, transmission and reconstruction are performed in parallel transparently from the user applications. MuniSocket is based on reliable transfer protocols such as TCP and it was implemented and evaluated using TCP in a basic form for dedicated homogeneous networks. Then the cut-off decision enhancement, to decide whether to use one or more interfaces was added and evaluated. Finally, a load balancing mechanism to compensate for slow and loaded interfaces was added and further evaluated. The performance results were very good especially with fast Ethernet networks, where peak bandwidth of 187Mbps was achieved on two interfaces. In addition, the cut-off enhancement and load balancing mechanisms further improved this performance and allowed for compensation in loaded networks. Moreover, the protocol automatically compensates for broken links by distributing the load to the other working network connections. In addition, further enhancements and optimizations of this model are currently under investigation.

Acknowledgements

This project was partially supported by a National Science Foundation grant (EPS-0091900) and a Nebraska University Foundation grant, for which we are grateful. We would also like to thank the members of the secure distributed information (SDI) group and the research computing facility (RCF) at the University of Nebraska-Lincoln for their continuous help and support.

References

- [1] Adishesu, H., Parulkar, G., and Vargese G., A Reliable and Scalable Striping Protocol, *Computer Communication Review*, volume 26, page 131-141, October 1996.
- [2] Brendan, C., Traw, S., and Smith J., Striping Within the Network Subsystem, *IEEE Network*, pages 22-29, July/August 1995
- [3] Buyya, R., editor. *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall, Inc., 1999.
- [4] Chen, J, Akers, W., Chen, Y., and Watson W., Java Parallel Secure Stream for Grid Computing, *Computing in High Energy and nuclear physics*, CHEP 01 Beijing China, Sept 2001.
- [5] Fischer M. GMSOCKS – A Direct Sockets Implementation on Myrinet. *CLUSTER'01*, 2001.
- [6] Foster, I., C. Kesselman, C., editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc. 1999.
- [7] GridFTP: Universal Data Transfer for the Grid, The Globus Project white paper. 2000. The University of Chicago and the University of Southern California. At

- <http://www.globus.org/datagrid/deliverables/C2WPdraft3.pdf>, and GridFTP Update. 2002. At <http://www.globus.org/datagrid/deliverables/GridFTP-Overview-200201.pdf>
- [8] Katz, R., Gibson, G., and Patterson, D., Disk System Architectures for High Performance Computing, in Proc. of the IEEE, vol.77, no. 12, Dec. 1989
 - [9] Kim, J. , Kim, K., and Jung S. SOVIA: A User-level Sockets Layer over Virtual Interface Architecture. CLUSTER'01, 2001.
 - [10] Marzullo, K., Ogg, M., Ricciardi, A., Amoroso, A., Calkins, F., and Rothfus, E. NILE: Wide-area computing for high energy physics. Proc. 1996 SIGOPS Conf. New York: ACM, 1996.
 - [11] Mohamed, N., Al-Jaroodi, J., Jiang, H., and Swanson, D., A User-level Socket Layer over Multiple Physical Network Interfaces, PDCS2002, 14th IASTED International Conference on Parallel and Distributed Computing and Systems, Cambridge, USA, November 2002.
 - [12] Ostermann, S., Allman, M., and H. Kruse, 1996, An Application-Level solution to TCP's Satellite Inefficiencies, Workshop on Satellite-based Information Services (WOSBIS), November, 1996.
 - [13] Sivakumar, H., Bailey S., and Grossman, R., Pockets: The Case for Application-level Network Striping for Data Intensive Applications using high Speed Wide Area Networks, SC2000: High-Performance Network and Computing Conference, Dallas, TX, 11/00.
 - [14] Stevens, W. Unix network programming: Networking APIs: Socket and XTI, volume 1. Prentice-Hall PTR, Upper Saddle River, NJ 07458, USA, second edition, 1998.
 - [15] Theoharakis V. and Guerin R., SONET OD-12 Interface for Variable Length Packets, Proc. Second Int'l Conf. On Computer Communication and Networks, San Diego, CA, June 28-30, 1993.
 - [16] Web page for Beowulf Ethernet Channel bonding:
<http://www.beowulf.org/software/bonding.html>, June 2002.
 - [17] Web page for Channel bonding benchmark results:
<http://www.fos.su.se/compchem/jazz/bond.html>, June 2002.