RECONSTRUCTION OF APPLICATION LAYER MESSAGE SEQUENCES BY NETWORK MONITORING

Amitoj Singh

Fermi National Accelerator Laboratory
Batavia, IL, 60510
USA

Jaspal Subhlok

Department of Computer Science, University of Houston
Houston, TX, 77204
USA

Abstract

Monitoring communication is central to the development and tuning of parallel and distributed applications. Available tools for network monitoring typically capture the network traffic at TCP or IP layers, but a software developer is most interested in the message exchange sequence between pairs of nodes executing the application. However, capturing application level traffic requires instrumenting the application code, which is cumbersome and not even an option in most cases. We present a procedure that reconstructs the application layer message sequences by analyzing TCP layer traffic. The basic idea is that, since TCP traffic is constructed with a well defined procedure from application traffic, it should be possible to reconstruct application message exchanges from TCP segments. We demonstrate that the procedure is effective, although not perfect, by extensive experimentation with NASA's NAS parallel benchmarks.

Keywords

TCP, network monitoring, NAS benchmarks, grid computing

1 Introduction

Monitoring the communication behavior of distributed applications is important for resource management and performance debugging aspects of application development. A number of tools are available to monitor network traffic, some examples being tcpdump, iptraf, and SNMP based probes, and more advanced tools built with these and intrusive network measurements [8,14]. While these tools measure network traffic, often the user's real intent is to capture the messages exchanged by application processes. The two are not the same since application traffic is typically segmented into fixed size packets by the network protocol layers. In general, it is not possible to directly capture the application message sequences without instrumenting the application itself, which is generally not an option. The goal of this research is to reconstruct the application layer communication from the network communication captured at the TCP level.

The resultant approach allows the discovery of message sequences between application processes using any application level protocol without any knowledge of the application and without access to the application code.

This research is motivated by the resource selection and management in grid computing [2,5,6,7,11]. The basic question is as follows: On what set of nodes will a parallel or distributed application perform the best ? Our approach to solving this problem consists of the following steps:

Application Characterization: Development of an application performance profile that captures the resource needs of an application and models its performance under different network conditions.

Network characterization: Measure the existing network conditions such as CPU loads and available bandwidth on network links.

Node selection: Estimate the application performance on different sets of nodes on the network under existing network conditions using the application profile and select the best nodes for execution.

The research presented in this paper is on the subject of constructing an application profile. To achieve this, it is necessary to know the precise pattern of message exchanges between application processes so that the performance of message exchanges and the application can be estimated under different network conditions. Rest of this paper describes and validates a scheme to infer the application level message exchanges from network traffic.

The paper specifically focuses on constructing user level messages based on the TCP segment stream between pairs of nodes executing an application. Our goal is to be able to generate the sequence of messages with the size in bytes of each message between communi-

cating nodes. The experiments presented in this paper are for parallel applications using the MPI communication library for message passing. The basic procedure is to first collect the sequence of TCP segments at nodes or intermediate routers using a utility such as TCPDump. Since the construction of TCP segments from application messages follows a well-defined process, we attempt to "reverse engineer" the process to obtain the size and sequence of application layer messages. The results are validated by capturing this information for NAS benchmarks [1] executing on a cluster of workstations and comparing it with the actual message exchange sequence for NAS benchmarks measured with program instrumentation.

2 Principles of message reconstruction

Let us first examine how the TCP segments are constructed from application messages. When a new message is received from the application layer for transmission, it is entered into a new TCP segment [9]. If the message is smaller than the maximum segment size (MSS), the entire message fits into a single segment. Otherwise, a sequence of TCP segments are composed and transmitted until the complete application layer message is processed. The key information that we exploit for application layer message reconstruction is as follows:

An application message is typically fragmented into a consecutive sequence of TCP segments and all except the last segment is of size MSS.

Based on this, we can simply locate the TCP segments of size less than MSS and reconstruct the size of the application layer message. While this is our guiding principle, there are several factors that make the process complex:

- When an application sends two or more messages rapidly, they often get combined. That is, the last TCP segment for the first message also includes the beginning of the next message.
- In some cases, an incomplete TCP segment gets transmitted even though the application level message it is carrying as payload is not finished. That is, a TCP segment containing the middle of an application message may be of size smaller than MSS indicating that the message has finished when in fact it has not.

Above occurrences are unpredictable and depend on the specific system activity sequence each time a program is executed. We use heuristics to minimize the impact of these factors and show that message sequences can be captured with very good, although imperfect, accuracy with our procedure.

3 Procedure for message reconstruction

The procedure for reconstructing the application layer message sequence from the TCP stream detected on the network is divided into the following phases:

First phase: Separate TCP streams.

Second phase: Sanitize a TCP stream.

Third phase: Reconstruct application layer messages.

Fourth phase: Error minimization by applying "best-of-three" technique.

3.1 Separating TCP streams

A typical network link is traversed by several TCP streams, each consisting of a series of TCP segments. We are interested in each TCP stream containing data that was exchanged by the application of interest.

There are two approaches to separating the TCP streams. First is based on the fact that every TCP stream transfers data segments between unique pairs of port numbers and IP addresses. The second approach is based on the fact that each TCP stream normally spans a unique series of sequence numbers. Both approaches can fail under pathological conditions but are effective in practice. In our example we demonstrate the second approach as it is related to the later phases of analysis.

3.2 Sanitizing a TCP stream

TCP treats a data transfer as a stream of bytes with an assigned sequence number for each byte. TCP segments can get lost or arrive out of order. TCP acknowledges received segments and retransmits lost segments. Further, the tcpdump mechanism can occasionally not register a segment in heavy traffic.

We need to detect missing segments and filter out duplicate transmissions. Duplicate transmissions are easy to detect and are simply discarded. If a fixed number of bytes are missing in the TCP stream, it is assumed that a TCP segment(s) of corresponding size was lost for some reason, and a segment of that size is "inserted" in the stream. At the end of this phase, we have a sequential series of segments spanning from the first to the last byte of data that was sent.

3.3 Reconstructing application layer message stream

As stated earlier, the central idea of the reconstruction procedure is that an application message is typically fragmented into a consecutive sequence of TCP segments and all except the last segment is of size MSS. Hence, the last TCP segment corresponding to an application level message is the only one that is smaller than MSS. Note that for short messages, the last seg-

ment is also the first segment. In general, we assume that a new application level message starts after a segment that is smaller than MSS and ends at the next segment that is smaller than MSS.

3.3 "Best of three" error minimization

When an application "pauses" execution during the transfer of a message to the transport layer (e.g., when the CPU has to be relinquished to service an interrupt), a TCP segment smaller than MSS may be sent before the message is finished. This would lead to incorrect identification of messages with our procedure. However, it is extremely unlikely that the same message will be split in the same way on two different runs. Similarly, it is possible that two short messages may be packed in the same segment, but it is unlikely that identical packing will happen on consecutive runs.

This motivates the best-of-three technique that we use to minimize errors. We run the same application three times with the same input and use the previous phases to generate three output files containing the sequence of application messages that were generated. The true application level message sequence should be identical for the three runs. At places in the series where the 3 constructed sequences do not agree, we look for the majority to agree and take that as the correct subsequence. In almost all cases, two or all of the runs agree. If not, a random choice is made, but it is also possible to have a scheme where additional runs may be used to increase accuracy further when needed.

4 Example message reconstruction

In the following example each TCP packet will be represented as shown below.

source > destination : s : e (d)
node1 > node2 : 8345 **:** 9793 (1448)

where

s - starting sequence number
e - ending sequence number
d - data size in bytes in the TCP segment

the MSS = 1448 for our setup.

The following is a simplified snapshot of a TCP communication sequence captured while running a sample parallel application. We will analyze the output and try to reconstruct the parallel application message.

1. node1 > node2: 88801**:**88833(**32**)
2. node1 > node2: 1**:**1449(**1448**)
3. node1 > node2: 88833**:**88897(**64**)
4. node1 > node2: 1449**:**2897(**1448**)
5. node1 > node2: 2897**:**4345(**1448**)
6. node1 > node2: . 7241:8238(**997**)

From the range of sequence numbers, it is clear that line numbers 1 and 3 correspond to one TCP sequence and the remaining lines correspond to another TCP sequence. Separating the two streams, we have :

stream1:
1. node1 > node2: 88801**:**88833(**32**)
3. node 1 > node2: 88833**:**88897**(64)**

stream 2:
2. node1 > node2: 1**:**1449(**1448**)
4. node1 > node2: 1449**:**2897(**1448**)
5. node1 > node2: 2897**:**4345(**1448**)
6. node1 > node2: . 7241:8238(**997**)

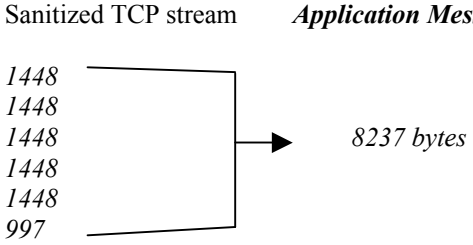
Stream1 clearly consists of two messages of size 32 and 64 respectively, and need not be analyzed further. We now focus on stream2. We first renumber the segments in order :

1. node1 > node2: 1**:**1449(**1448**)
2. node1 > node2: 1449**:**2897(**1448**)
3. node1 > node2: 2897**:**4345(**1448**)
4. node1 > node2: . 7241:8238(**997**)

A simple analysis of the stream shows that bytes with sequence numbers between 4345 and 7241 are unaccounted for. Now, 7241 – 4345 = 2896, which is double of 1448, our MSS (Maximum Segment Size). Our analysis concludes that two segments of size MSS were not reported, and therefore we "insert" the missing segments. The corrected output file is as follows:

1. node1 > node2: 1**:**1449(**1448**)
2. node1 > node2: 1449**:**2897(**1448**)
3. node1 > node2: 2897**:**4345(**1448**)
3.1 node1 > node2: 4345 :5793(**1448**) (inserted)
3.2 node1 > node2: 5793**:**7241(**1448**) (inserted)
4. node1 > node2: . 7241:8238(**997**)

This stream corresponds to a single application layer message that finished with the segment shown in line 4 since that is the first segment smaller than 1448. It is evident that an application layer message of 8237 bytes was split into five 1448 and one 997 byte data segments as illustrated below:



The above corresponds to a section of the TCP stream for the test application, which was a small parallel program. When this program is run multiple times, the application layer message stream generated by the above procedure is not always identical, although the total number of bytes remains the same. We show three sample results from reconstruction from different runs of the program:

Reconstructed application layer message stream:

run 1	*run 2*	*run 3*
1. 8237	8237	8269
2. 32	32	64
3. 64	7336	7272
4. 7272		
15605	15605	15605

We now illustrate the "best of three" heuristic we used to reconcile the differences between runs to produce the most probable true sequence of application messages. The basic approach is as follows. Each run corresponds to a sequence of cuts that divide the stream of 15605 bytes into messages. We illustrate the cuts corresponding to each run as follows:

	Cut at	*run1*	*run2*	*run3*
1.	8237	Yes	Yes	No
2.	8237+32=8269	Yes	Yes	Yes
3.	8269+64=8333	Yes	No	Yes
4.	8333+7336=15605	Yes	Yes	Yes

We only accept the cuts that occur in majority of the runs and ignore the others. This gives us the following final result:

Message Sequence = 8237, 32 , 64 , 7272

Total bytes = 15605

Please note that the sample inputs were chosen to illustrate the problems associated with this approach. In practice, most messages are detected exactly without the complications discussed here.

5 Experiments and results

In order to validate this approach to application level message reconstruction, we performed a set of experiments with the NAS parallel benchmarks 1,13]. The codes used are BT (Block Tridiagonal solver), CG (Conjugate Gradient), IS (Integer Sort), LU (LU solver), MG (Multigrid), and SP (Pentadiagonal solver). All programs are in Fortran 77, except IS, which is a C program. All benchmarks were compiled with class A size for 4 nodes and executed on 4 nodes. g77 and gcc (Fortran 77 and C compilers from GNU) were used for compilation. The programs were executed on a small 100Mbps Ethernet based cluster of 4 Pentium workstations. The NAS benchmarks are MIMD parallel programs with extensive and frequent communication, and hence challenging test cases for message reconstruction

The MPI implementation of the NAS benchmarks was used. MPI is a portable message passing library, and the implementation used runs on top of TCP. Hence the MPI messages are broken into TCP segments for transmission. The benchmark applications were executed and tcpdump was employed to capture TCP headers. The procedure in this paper was employed to reconstruct the application level MPI message exchanges. The results were compared against the exact message sequence captured separately by profiling the communication calls in these programs and re-executing.

Before we present the results, we need to discuss one detail of MPI communication. MPI introduces additional control messages in addition to the application communication. There are a large number of these messages but they are very small (typically around 32 bytes) and generate well under 1% of the total traffic in all of our example applications. Since there is no way to verify what the MPI control traffic is supposed to be, we shall limit our discussion to application data segments only except that we shall note where the application traffic is not captured accurately because of the control traffic.

The results of our experiments are presented in Table 1 and also illustrated in Figure 1. We list the main observations:

- *Almost all application messages were captured either precisely or approximately:* While the majority of the application level messages are captured precisely, a significant fraction of the messages are not. However almost all of the remaining messages are constructed approximately – either the number of bytes in the reconstructed message was off by a small number (less than 100) of bytes due to mixing with control traffic or two consecutive messages are reconstructed as a single message.
- *The accuracy of the procedure varies widely among benchmarks:* While all messages in the IS benchmark are constructed accurately, the reconstruction of almost all messages in LU was approximate.
- *Reconstruction is poor for small messages:* The major difference between the benchmarks IS and LU, the cases that demonstrate the most accurate and the least accurate reconstruction, respectively,

is that IS sends a few very large messages, while LU sends a large number of small messages. The conclusion is that the reconstruction procedure captures most of the large messages accurately but the small messages only approximately.

Benchmark Application Description			**Messages detected exactly correctly (EXACT MATCH)**		**Messages detected exactly OR combined with upto 100 bytes of system data**		**Messages detected exactly OR combined with upto 100 bytes of system data AND/OR Combined with one other application message (APPROX MATCH)**		**Messages identified completely incorrectly (NO MATCH)**	
Name	**Number of Messages**	**Average message size (Bytes)**	**Number**	**%-age**	**Number**	**%-age**	**Number**	**%-age**	**Number**	**%-age**
BT	620	150,620	384	**62**	434	**70**	620	**100**	0	**0**
CG	416	56,001	383	**92**	416	**100**	416	**100**	0	**0**
IS	11	2,117,547	11	**100**	11	**100**	11	**100**	0	**0**
LU	15,324	3,948	252	**2**	15,032	**98**	15,324	**100**	0	**0**
MG	154	81,422	82	**53**	142	**92**	150	**97**	4	**3**
SP	1,458	112,836	1,088	**75**	1,310	**90**	1458	**100**	0	**0**

Table 1. Reconstruction of application level messages for NAS benchmarks

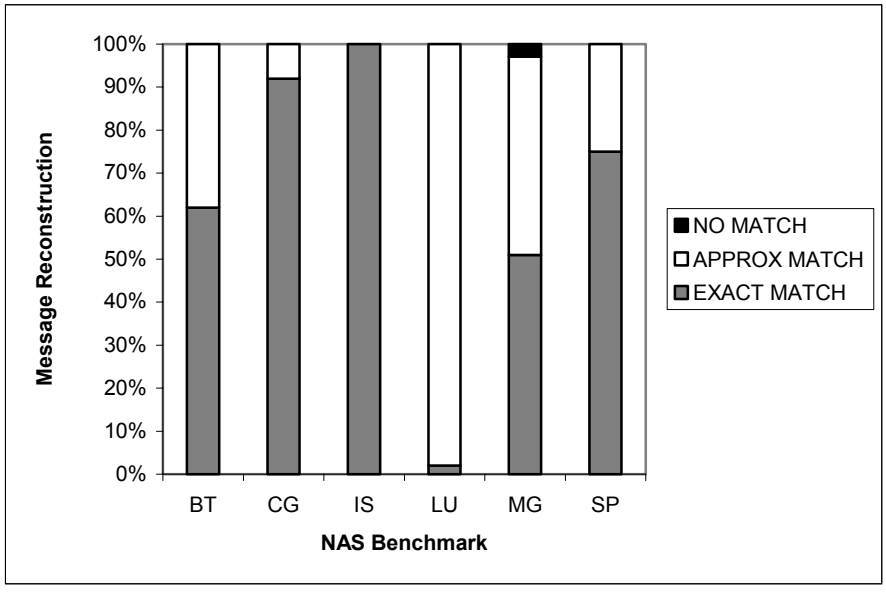


Figure 1. Accuracy of application level message reconstruction for NAS benchmarks

6 Conclusions

We have introduced and validated a simple framework to capture the application level message exchanges in a distributed application by monitoring and analyzing TCP traffic. The procedure works well for large messages, but pairs of distinct small messages often appear as a single message after reconstruction. While the procedure is not perfect, it is sufficient for the purpose for which it is designed, which is, characterizing an application's communication behavior for resource selection in grid environments and understanding the communication patterns. The procedure is entirely based on system and network measurements, and therefore no access to source code or libraries is necessary.

7 Acknowledgements

This research was sponsored by the Los Alamos Computer Science Institute (LACSI) through Los Alamos National Lab contract number 03891-99-23 as part of the prime contract (W-7405-ENG-36) between the DOE and the Regents of the University of California. Support was also provided by the Texas Advanced Technology Program under grant number 003652-0424, and University of Houston's Texas Learning and Computation Center.

We also wish to thank other members of our research group, in particular Shreenivasa Venkataramaiah and Srikanth Goteti, for their valuable contributions to this research.

References

[1] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, & M. Yarrow, The NAS Parallel Benchmarks 2.0, *Tech. Rep. 95-020,* NASA Ames Research Center, December 1995.

[2] F. Berman, R. Wolski, S. Figueira, J. Schopf, & G. Shao, Application-level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing '96,* Pittsburgh, PA, November 1996.

[3] P. Bhatt, V. Prasanna, & C. Raghavendra, Adaptive communication algorithms for distributed heterogeneous systems. In *Seventh IEEE Symposium on High-Performance Distributed Computing,* Chicago, IL, July 1998.

[4] P. Dinda, Statistical properties of host load in a distributed environment. In *Fourth Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers,* Pittsburgh, PA, May 1998.

[5] I. Foster, & K. Kesselman, Globus: A metacomputing infrastructure toolkit. *Journal of Supercomputer Applications 11*, 2 (1997), 115--128.

[6] A. Grimshaw, & W. Wulf, The Legion vision of a worldwide virtual computer. *Communications of the ACM 40*, 1 (January 1997).

[7] M. Litzkow, M. Livny, & M. Mutka, Condor --- A hunter of idle workstations. In *Proceedings of the Eighth Conference on Distributed Computing Systems,* San Jose, California, June 1988.

[8] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, & J. Subhlok, A resource query interface for network-aware applications. In *Seventh IEEE Symposium on High-Performance Distributed Computing*, Chicago, IL, July 1998.

[9] J. Postel, (ed.), Transmission Control Protocol-DARPA Internet Program Protocol Specification, *RFC 793*, USC/Information Sciences Institute, Sept. 1981.

[10] M. Stemm, S. Seshan, & R. Katz, Spand: Shared passive network performance discovery. In *USENIX Symposium on Internet Technologies and Systems,* Monterey, CA, June 1997.

[11] J. Subhlok, P. Lieu, & B. Lowekamp, Automatic node selection for high performance applications on networks. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* Atlanta, GA, May 1999, pp. 163--172.

[12] J. Subhlok, & G. Vondran, Optimal latency--throughput tradeoffs for data parallel pipelines. In *Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, Padua, Italy, June 1996, pp. 62--71.

[13] T. Tabe, & Q. Stout, The use of the MPI communication library in the NAS Parallel Benchmark, *Tech. Rep. CSE-TR-386-99*, Department of Computer Science, University of Michigan, Nov 1999.

[14] R. Wolski, N. Spring, & C. Peterson, Implementing a performance forecasting system for metacomputing: The Network Weather Service. In *Proceedings of Supercomputing '97*, San Jose, CA, Nov 1997.