Applications of Passive Message Logging and TCP Stream Reconstruction to Provide Application-Level Fault Tolerance

Sunny Gleason
COM S 717

December 17, 2001

0.1 Introduction

The proliferation of large-scale networks and techniques for distributed computation has resulted in an incredible growth in distributed applications. However, as computation is split among various machines, it becomes vulnerable to failures in any part of the network, as well as failures of the machines themselves. As we become more dependent on these distributed systems for our daily lives, the prevention of and recovery from system failures becomes increasingly important.

A variety of techniques are already available for providing application-level fault-tolerance, such as redundant hardware, software replication, coordinated and uncoordinated checkpointing, and message logging, both optimistic and pessimistic. However, most software mechanisms for fault-tolerance assume that all communications channels are point-to-point, and that the collection of hosts is homogeneous. Subsequently, these techniques tend to allocate resources symmetrically on each machine, for the sole purpose of providing the application-level fault-tolerance mechanisms. This accomplishes the goal of providing fault-tolerance, at the expense of wasting resources on compute servers during failure-free operation.

In contrast to the assumptions listed above, it seems that clusters of heterogeneous workstations are being deployed more and more frequently, especially as new generations of PC's are put into use and older ones are retired from service. In addition, these workstations are commonly connected using ethernet hubs, which have the property that communication is broadcast to all workstations on the hub, not just the intended destination. While the broadcast nature of ethernet is a problem for those seeking to protect the privacy of LAN communications, this property is exploited in our system to provide lightweight message logging services for any class of distributed network application.

In this paper, we examine a technique for reducing the overhead of message logging on clusters of compute servers operating over a LAN or WAN. Our technique uses additional machines on the network running specialized software to take on the burden of message logging; compute servers are unaware of the fault-tolerance mechanism until the time of crash recovery. Upon crash recovery, compute servers negotiate with some set of message logging servers to reconstruct the set of messages which determined their state before crashing. In addition, the message logging task itself is extremely simple; thus machines which are not powerful enough to make progress on the distributed application itself could conceivably be deployed on the system as message logging servers.

In the following section, we describe our target application: the set of programs developed using the MPICH 1.2.2 implementation of the Message Passing Interface standard. In Section 0.3, we outline the basics of network intrusion destection systems, the key technology on which this paper is based. Section

0.4 contains details of our message logging mechanism and experimental apparatus. We present the results of our experiments in Section 0.5, along with some insights that we have gained and directions for future work.

0.2 MPICH

In this project, we focused on applications written using MPI, the Message Passing Interface. MPI itself is an API for parallel and distributed computation using the message-passing model; there are several implementations of MPI available, including MPICH, LAM/MPI, and Chimp. We chose to work with the MPICH distribution because of its open source distribution, and the large number of platforms to which it has been ported.

We wanted to examine the feasibility of application-level fault-tolerance for MPI programs running on clusters of heterogeneous workstations, in contrast to specialized parallel computers. MPICH features the CH_P4 driver, which provides MPI support over standard IP networks using TCP connections. We believe that the heterogeneous cluster model is compelling for a number of reasons.

First, it is a most cost-effective way of building a platform for distributed computation. General-purpose workstations are extremely inexpensive in comparison to massively parallel architectures. The proliferation of commodity hardware, and open-source software and operating systems should further widen this gap in the future.

Second, workstations in a LAN or WAN environment are typically connected via a shared ethernet, at 10 megabit, 100 megabit, or gigabit per-second speeds. LANs connected using ethernet hubs utilize a shared broadcast communiation medium, or "ether". As we will see in the following section, the shared broadcast medium allows us to perform message logging without involving the compute servers themselves.

MPI programs, which may run over any number of hosts connected via an ethernet, typically follow the following steps. In its standard configuration, MPICH encourages the SPMD model, or single program multiple data, where identical copies of the program run on each node in the network (in programs where node specialization is needed, machines may decide which roles to perform based on node id, for example). In the configuration we employed, a single machine coordinates the computation, collects the results and displays those results to the user.

The SPMD program is invoked using *mpirun*, a wrapper script designed to hide the details of a heterogeneous MPI environment. In our experimental configuration, the wrapper script starts a P4 listener process on node 0, then proceeds to invoke the program on other nodes throughout the cluster using the *rsh* remote shell utility. The remote nodes then execute their copy of the program, which connects to the listener process port on the coordinating machine using normal TCP connections.

While the program executes on the remote host, standard error and standard

output continue to flow back to the coordinator through the rsh console. All other MPI data flows through the single TCP connection between the listener process and the "client" workstations. This data includes the initial handshaking and environment setting for initialization of the MPI environment, as well as the marshalled data from MPI function calls.

The ability to passively capture this stream of data is the key idea behind this paper. Since ethernet is a broadcast medium, we can use packet capture techniques to reconstruct the pattern of communication between any nodes on the network. As we'll see in the following section, this opens up several new challenges and opportunities.

0.3 IDS

The ability to capture IP packets on an ethernet network has been around for a long time – the *tcpdump* utility dates to around 1988. For simple datagrams used in RPC (remote procedure call), packet capture could have been a feasible means of providing application-level fault-tolerance. However, the complexity of TCP, due to segmentation and retransmission schemes, has made packet-level capture infeasible for applications using the stream-oriented protocol.

The growth of the Internet has also brought an increase in the number of network-based attacks in corporate environments, prompting the development of intrusion detection systems, or IDS. Typically, intrusion detection systems are broken up into 2 main categories: host-based and network-based.

Host-based IDS typically deal with a single machine only. They include local virus scanners and checksum-based file integrity utilities such as Tripwire. Host-based IDS have the advantage of being completely customizable to the characteristics of a specific machine. However, they are not able to carry out the task of alerting system administrators in the face of all attacks. For the example of local virus scanners, the latest virus signatures must be deployed across all hosts in order to ensure proper coverage.

Network-based IDS detect network attacks and system compromises by monitoring the traffic on a network. For example, a set of TCP SYN packets sent from a remote host to a wide range of ports on a single server could constitute a *port scan*: the attempt of an adversary to gain information about what services are running on a particular machine. Network-based IDS typically have the ability to operate in packet-capture mode as well as alert mode. It is exactly the packet-capture mode which is of interest to us in this project, as we'll see in later sections.

Network-based IDS use signatures (much like virus scanners) in order to detect attacks and system compromises. Like virus scanners, they must be updated frequently in order to provide protection. Unlike local virus scanners, however, they require deployment on only a single machine on a shared ethernet network.

As network applications have grown more complex, network-based IDS have become more sophisticated in order to keep up with the growing complexity of

network attacks. For example, to detect certain classes of HTTP server exploits, IDS must be able to reconstruct the TCP streams (on which HTTP is based) as well as understand elements of the HTTP protocol (such as request types and URL's).

On large networks with thousands of simultaneous requests, the burden on network-based IDS can be quite computation-intensive, since the IDS must examine all requests of a given type in order to provide protection from that class of attacks. Thus, network-based IDS have become extremely efficient, in order to be able to detect a wide variety of attacks.

The IDS which we use in this project is called Snort. We chose Snort because of its wide range of features (such as TCP stream reconstruction using the Stream4 module), its open-source distribution, and the availability of ports of Snort on several different platforms. Additionally, Snort provides performance comparable to several closed-source IDS tools.

This combination of features should allow us to implement application-level fault-tolerance for multiple platforms and a variety of applications. In the following section, we outline our efforts to examine application-level fault-tolerance for MPI applications. These techniques should be applicable to a wide variety of other API's for network computation, such as SOAP or RMI.

0.4 Architecture

The goal of our project is to provide a framework for application-level fault-tolerance for MPI programs, without having to perform drastic modifications to the application code or store vast amounts of data on the computate servers themselves.

By consolidating message logging across a limited number of dedicated hosts, we accomplish two goals. First, we reduce the the resource requirements on compute servers in failure-free operation. By removing the burden of message logging from compute servers, those servers will have more resources to allocate to the problems which they are trying to solve. Second, we facilitate crash recovery by minimizing the number of machines that a server must contact in the process of recovery.

This represents a concentration of responsibility within the network, which would seem to provide a single point of failure. This would be true in the case that the message-logging machines were of the same complexity as our network compute servers. However, the task of TCP stream reconstruction is extremely simple in comparison to the coordination of distributed computation.

For example, it would be quite feasible to create network appliances with no hard drives and vast amounts of memory, providing reliability guarantees comparable to the hardware routers and bridges available today. Since message logs must be cleaned quite frequently when message rates are extremely high, the lack of secondary storage could offer tremendous performance gains. Thus, we are hopeful that the further development of our techniques could lead to a new class of network devices.

Additionally, since our message logging is completely passive in the case of failure-free operation, the addition of multiple message logging machines should have a negligible effect on network performance.

Our experimental apparatus consists of a cluster of six Pentium-II class workstations running Red Hat Linux 7.2 and MPICH version 1.2.2.3. The workstations communicate via a 10mbps ethernet hub. To support the *mpirun* script, we run an rsh server on each workstation. In addition, one host (the *gateway*) has an extra network interface connected to the department network, providing us with the ability to work and observe the system remotely.

In our implementation, a single host on the network (in this case, the gateway machine) runs Snort in packet capture mode, with TCP stream reconstruction and TCP session logging to file. We designate another host on the network to be the coordinator, and configured the mpich listener process to always bind to a specific port on that system. To improve performance and reduce the amount of logged information, we restrict Snort to listen for packets arising only on the 10mbps network. We further restrict Snort to log only TCP sessions which connect to the MPI listener port on the coordinator, sessions which connect to the rsh ports on the other workstations, as well as TCP connections between user ports on the workstations themselves, (since we were running no other network applications between workstations).

The logging of these sessions only is sufficient to reconstruct the entire MPI conversation. The rsh commands originating from the coordinator tell each workstation which application to run. The subsequent connections between the workstations and the coordinator contain MPI initialization data, which is sufficient for deciding which workstations are part of which distributed computations, in the event of multiple simultaneous computations.

We note that this simple architecture is possible only because network communications are unencrypted. For example, if ssh was employed instead of rsh, additional mechanisms would be necessary in order to allow the message logging machines to decrypt the remote sessions. Similarly, the deployment of IPSec networks would make the task of passive messsage logging considerably more difficult. However, since most clusters are located in trusted environments (i.e. behind firewalls), operating these clusters without encryption is entirely feasible.

Despite the simplicity of our network monitoring model, we encountered a number of obstacles to the full development of MPI application-level fault-tolerance using passive message logging, as we note in the following section.

0.5 Results / Conclusion

Our goal was to create a message log server on the gateway machine, such that crashed workstations could contact the server to obtain the full set (or some subset) of messages that they had received before crashing. However, the complexity of the MPICH CH_P4 device driver proved to be too complex to implement this facility in a timely fashion.

The large variety of message and datatypes made "parsing" the recorded TCP sessions extremely difficult. Messages are not of fixed length, requiring the entire stream to be processed in order to recover the final few messages.

In addition, the communication patterns which occur during MPI initialiation are quite complex, making the extraction of relevant data such as node id and host architecture more difficult.

It seems as though we approached this project from the wrong direction. MPI has an extremely rich set of functions and datatypes, thus CH_P4 must be extremely complex in order to support cross-platform operation. It may have been more productive to start with a "toy" protocol, rather than a full-blown package for distributed computation.

The complexity of deciphering communications between a single pair of hosts was more than difficult enough for the project we were undertaking. Thus, the many hours we spent setting up a 6-machine Linux cluster running MPICH could have been put to better use if we had just set up a 2-machine network, or a 1-machine network (utilizing the loopback interface).

For a small-sized test application, we were able to successfully gather MPI sessions between the 6 machines simultaneously. In our configuration, a message logging server with a modest CPU running Snort should be able to keep up with a fully-saturated 100mbps ethernet LAN. We feel that this is an encouraging result; one which is well worth further examination on a higher-capacity network.

Recent articles on the Snort web site have pointed out the difficulty of running network IDS sucessfully on a fully-saturated gigabit ethernet. Since our message-logging application is an extremely simple application of network IDS, it would be interesting to see how much higher load our system can tolerate compared to full-blown IDS configurations. Further, we expect that advances in IDS technology will be immediately applicable to our system.

In light of current trends in technology, we felt that it would be useful to note a couple performance optimizations. Since the price of RAM is decreasing steadily, we would recommend loading the message-logging servers with vast amounts of RAM. By configuring Snort to log to a RAM disk, the reduction in disk activity could yield substantial performance gains, both in crash recovery time, and ability to handle high network loads.

Altogether, we believe that passive message logging is an exciting application of network-based intrusion detection systems. We believe that the idea presents a clever and simple solution to reducing the overhead associated with message logging. We plan on continuing our analysis of the MPI CH_P4 protocol, so that we may better understand what issues should be taken into account when designing protocols for fault-tolerant computing. Subsequent projects may include applications of passive message logging to protocols such as SOAP or RMI.

REFERENCES

MPICH: A Portable Implementation of MPI
http://www-unix.mcs.anl.gov/mpi/mpich/

Snort: The Open Source Network Intrusion Detection System
http://www.snort.org/