Mendosus: A SAN-Based Fault-Injection Test-Bed for the Construction of Highly Available Network Services*

Xiaoyan Li, Richard Martin, Kiran Nagaraja, Thu D. Nguyen, Bin Zhang
{xili, rmartin, knagaraj, tdnguyen, binzhang}@cs.rutgers.edu

Department of Computer Science, Rutgers University
110 Frelinghuysen Rd, Piscataway, NJ 08854

Appears in Proceedings of the 1st Workshop on Novel Uses of System Area Networks (SAN-1), 2002

Abstract

We describe Mendosus, a SAN-based fault-injection test-bed that supports the emulation of LAN-connected systems. Our goal is to provide a test-bed for service designers to systematically assess the impact of end-to-end design decisions on the availability and reliability of network services in the presence of realistic fault scenarios. Mendosus currently supports the emulation of a cluster of PCs connected by a virtual network comprised of arbitrary configurations of Ethernet switches, hubs, and NICs. The application to be tested runs directly on the PCs, communicating over the virtual network. Mendosus injects faults in real-time, allowing the application to be tested against a variety of network and end-node faults. We are working to extend Mendosus to support emulation of a richer set of networks, including wireless systems and WAN-connected systems. Mendosus relies critically on the high performance of the underlying SAN to provide a platform that looks and performs as similar to the emulated platform as possible. Mendosus currently runs on clusters of PCs connected by Giganet VIA networks. Measurements show that this implementation should be able to easily emulate clusters interconnected by 100 Mb/s and/or 1 Gb/s Ethernet LANs.

1 Introduction

Progress in hardware and networking technologies are fundamentally changing the nature of computing. Users are no longer constrained to the desktop or static points of entry to monolithic systems. Rather, users carry multiple devices expecting to be surrounded by a rich and networked computing environment which provides a wealth of services. While these services promise unprecedented computing power and connectivity, their construction is a challenging exercise—examples of costly failures abound (e.g., [6, 17, 20]). As such services become woven into our everyday life, users will demand ultra reliability. An excellent example is the wired telephone system. Today, we expect the telephone to always be available and working correctly; we have little patience with service failures.

Against this backdrop of increasing reliability demands by end users, today's services rely on a plethora of systems, including users' devices such as PCs or PDAs and service providers' clusters [1]. Connecting these systems together is an equally complex set of interconnection networks: the user's dial-up or wireless link, the ISP's WANs, and the service providers' LANs and SANs. In this universe of many interconnected heterogeneous systems and networks, there is a high probability of multiple component failures. Service designers know this, and so today's services are implemented to tolerate faults at many levels (e.g., [7, 18, 13]).

Unfortunately, it is currently difficult to validate end-to-end design choices for tolerating network outages and/or other fault conditions. Not only must the tester put together a sufficiently representative test-bed, he must also find a way of reproducing realistic fault scenarios. The lack of an appropriate testing infrastructure often forces designers to resort to simplistic and gross fault-injection techniques such as "unplugging" components by hand [12]. Such ad-hoc techniques are undesirable because they can lead to either a false sense of confidence in the system's robustness or wasted resources in over-engineering systems to tolerate faults.

We are thus motivated to build a comprehensive emulation-based, fault-injection infrastructure that will allow service designers to systematically assess the im-

*Mendosus was supported in part by NSF grants EIA-0103722 and EIA-9986046.

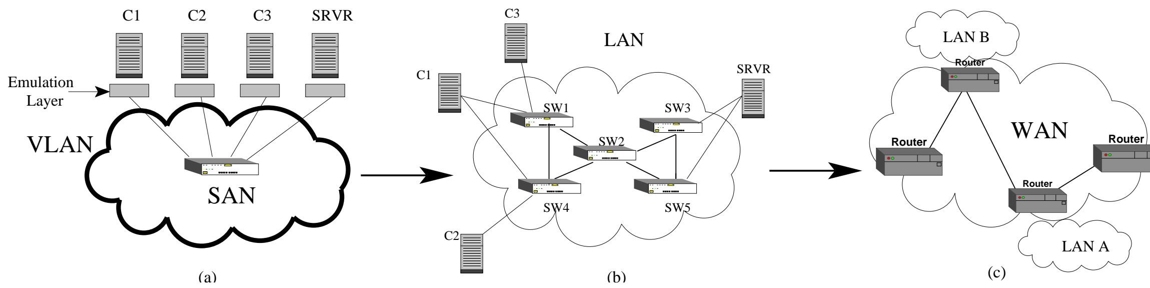

Figure 1: *Our goal is to use a SAN-connected cluster, shown in (a), to emulate a variety of interconnected systems. (c) shows our ultimate goal of emulating complex systems, possibly comprised of multiple different LANs, including wireless, interconnected by a WAN. (b) shows the current status of our prototype toolkit Mendosus, which supports the emulation of clusters interconnected by Ethernet LANs.*

pact of faults on service availability and reliability. We chose emulation, as opposed to simulation or analytic modeling, because it provides the most appropriate balance between three conflicting needs: ease of obtaining and deploying the testing infrastructure, observation of subtle system interactions, and sufficient flexibility to investigate the design space.

Figure 1 shows our vision of using a SAN-connected cluster to emulate systems interconnected by a variety of *virtual networks*. Our eventual goal is to support the emulation of a wide range of interconnected systems, including wireless, LAN, and WAN-connected systems. The use of a SAN is critical because the real network supporting the emulation must have higher or matching performance in all dimensions than the emulated networks.

Hand-in-hand with the capability of emulating a variety of interconnected systems is our goal of providing a comprehensive fault-injection infrastructure. This includes network delays and outages as well as failure of components in devices and servers. For example, links can fail, switches can drop packets, routers can get congested, node components (e.g., disk and NIC) can fail, etc. This infrastructure will allow designers to test services against a variety of faults, separately and in combination, to better assess or validate their availability and reliability. (Data on components' fault behaviors are starting to become available [22, 9]. Service designers can use this data as input to the fault-injection system.)

In this paper, we describe our initial test-bed: a prototype toolkit called Mendosus that supports the emulation of a cluster of PCs connected by arbitrary configurations of Ethernet switches, hubs, and NICs. Mendosus provides the investigator with a virtual network with tuneable performance and fault characteristics. Applications to be tested run directly on Mendosus's PC nodes, communicating through the virtual network. Mendosus injects faults in real-time, allowing a running application to be tested against a variety of fault scenarios. Mendosus can inject faults for a number of components using detailed fault traces or well-known random distributions such as exponential, constant, and weibull.

Currently, Mendosus runs on PC clusters connected by Giganet VIA networks. Mendosus relies on the VIA network's low latency, high bandwidth, and in-order delivery to allow for the emulation of the virtual network in real-time. Our measurements show that Mendosus should easily be able to emulate clusters interconnected by 100 Mb/s and/or 1 Gb/s Ethernet LANs.

The remainder of the paper is organized as follows. Section 2 describes Mendosus's overall architecture. Sections 3 and 4 describe the major components of Mendosus in more detail. Section 5 presents the results from several micro-benchmarks to assess Mendosus's efficiency, and thereby its ability to emulate different networks. Section 6 presents two case studies, showing how Mendosus can be useful for architecting highly available systems. Section 7 discusses related work. Section 8 discusses on-going work and concludes the paper.

2 Architecture

Mendosus is comprised of three software components running on a cluster of PCs interconnected by a Giganet VIA network: (1) a global controller, (2) a user-level daemon running on each node, and (3) a LAN emulator embedded in an emulated Ethernet driver running on each node. Figure 2 illustrates the overall architecture.

The global controller injects faults and maintains a consistent view of the entire network. When emulation starts, the controller parses a configuration file that de-

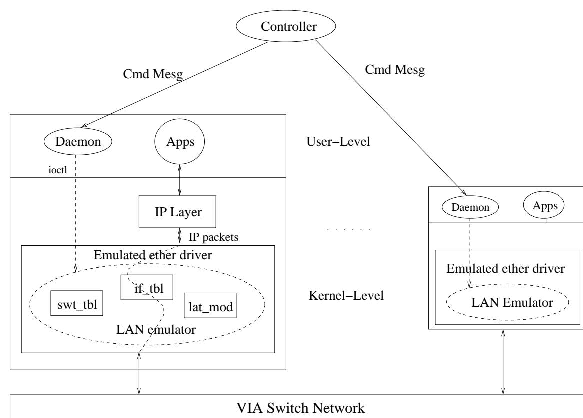

Figure 2: *Mendosus is comprised of three major components: a global controller, a daemon running on each node, and a LAN emulator embedded inside an Ethernet driver running on each node.*

scribes the network to be emulated and components' fault profiles. It forwards the network configuration to the daemon running at each node of the cluster. Then, as the emulation progresses, it uses the fault profiles to decide what faults to inject and when they should be injected. It communicates with the per-node daemons as necessary to affect the faults (and subsequent recovery). Note that while there is one global controller per emulated system, it does not limit the Mendosus's scalability: the controller only deals with faults and does not participate in any per message operations[1].

The per-node daemon serves mainly as an information conduit between the controller and the emulation module on each node. On receiving the network configuration from the controller at the start of an emulation run, the daemon sets up the `/etc/hosts` file so that applications can resolve node names to IP addresses on the emulated network. The daemon also passes the network topology to the emulation module. Then, as the emulation progresses, the controller informs the emulation module (which resides in the kernel) on each node about fault events by sending messages to the daemon, which in turns uses an iotcl interface to talk to the emulation module.

The per-node emulation module maintains the topology and status of the virtual network to route messages. To emulate routing in Ethernet networks, a spanning tree is computed for the virtual network. Each emulated NIC is presented as an Ethernet device; a node may have multiple emulated NICs. When a packet is handed to the Ethernet driver from the IP layer, the driver invokes the emulation module to determine whether the packet should be forwarded over the real network (and which node it should be forwarded to). The emulation module determines the emulated route that would be taken by the packet. If any of the component in the route is down, it drops the packet. If any component in the route is in an intermittent error state, then it determines whether the packet should be dropped. Otherwise, the packet is forwarded to the destination over the underlying SAN. The emulation module uses multiple point-to-point messages to emulate Ethernet multicast and broadcast.

At the receiving end, the packet may be buffered for some time, depending on what the "actual" delay should have been in the virtual network, before being passed to the IP layer. Of course, it is likely difficult or impossible to emulate delays that differ from the real delay by less than order of milliseconds (because of scheduling delays and overheads). We are exploring the implementation of "token buckets" to emulate networks with lower bandwidth than that possible on Mendosus.

[1]One possible limit to scalability is that the controller currently communicates with all nodes about each fault. This is because each node is maintaining the status of the entire virtual LAN to maximize performance. As we move on to emulate more complex interconnection systems such as a WAN, this design decision will almost certainly not propagate. Likely, each node will only maintain the status of components in the immediate network that it is connected to.

3 Controller

The controller is responsible for starting the LAN emulation based on the initial topology specification, injecting faults, and managing topology changes during an emulation run. It maintains a high level consistent view of the entire emulated LAN and can contact the per-node daemons to perform fault injections or real-time topology changes.

Configuration Specification Language. Currently, Mendosus can emulate LANs comprised of the following components: switches, hubs, ports, links, and NICs. The configuration of an emulated LAN is specified using a simple language similar to that used in ns-2 [24]. The configuration must also list the available real resources that are to be used to perform the emulation. We assume that the underlying platform is a fully connected cluster, with the interconnect being a low-latency high-bandwidth SAN.

The following example configuration specifies the network topology shown in Figure 3; the resulting emulation would be run on the real machines a.rutgers.edu and b.rutgers.edu.

set_host Host1

Fault Type	Corresponding Component	Corresponding Fault	Fault parameters
ft_nic_down	Network Interface	NIC temporarily down (e.g. driver hangs)	down time
ft_conn_down	NIC to Switch connection	Connection temporarily down (e.g. unplug)	down time
ft_sw_down	Switch	Switch temporarily down (e.g. power failure)	down time
ft_sw_port_down	Switch	One port temporarily down (e.g. duplex mismatch)	down time
ft_link_down	Switch to Switch Link	Link temporarily down (e.g. faulty cable)	down time
ft_sw_pktdrop	Switch	Switch drops packet (e.g. queue overflow)	drop rate
ft_sw_pktloss	Switch	Switch losses packet (e.g. routing error)	loss rate
ft_link_pktloss	Switch to Switch Link	Link drops packet (e.g. unprotected cables)	loss rate
ft_link_delay	Switch to Switch Link	Link introduces packet delay (e.g. congestion)	loss rate

Table 1: *Types of faults that Mendosus can currently inject.*

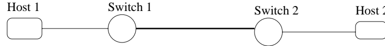

Figure 3: *Example Topology.*

```
set_host Host2
set_switch Switch1
set_switch Switch2
set_nic Host1 166.111.4.1 martin.my.net
set_nic Host2 166.111.4.2 kevin.my.net
set_link Switch1 Switch2 Link1
set_connection martin.my.net Switch1 Conn1
set_connection kevin.my.net Switch2 Conn2
set_addr Host1 a.rutgers.edu
set_addr Host2 b.rutgers.edu
```

Fault Model. Each component in the emulated LAN can have an associated fault profile. Currently, the fault profile can be one of several known distributions, such as exponential, weibull, and constant. The fault profile can also be a trace file, specifying when a component should fail, how it should fail, and when it should be up again. In the future, we plan to build a Java interface to allow for the use of arbitrary code to determine the fault profile of a component. Table 1 shows the current supported fault types. For example, the following line can be used to specify that Switch1 in the above topology will repeatedly go down for 2 seconds following an exponential inter-arrival distribution with an average rate of 0.05, which stands for 0.05 events per second.

```
set_fault Switch1 ft_sw_down 2 poisson 0.05
```

During the emulation, the controller is responsible for scheduling faults and recoveries according to the given fault profiles. When an event occurs, e.g, a NIC is inadvertently disconnected, the controller contacts the daemon running on the appropriate nodes and informs them about the event. In turn, the daemon performs an `ioctl` call to the driver with the event. The driver will set the status of the affected emulated component appropriately.

The controller is responsible for choosing the root of the spanning tree for the virtual Ethernet LAN. It is also responsible for determining when a set of faults leads to network partition and choosing a root for each partition for the spanning tree algorithm.

Real-Time Topology Changes. Change in the underlying platform is an unavoidable fact of life. As highly available services continue to operate across spans of years, components will need to be changed and upgraded. Further, the platform may need to be extended or reconfigured to better meet the changing usage patterns of clients. Thus, service providers must be able to assess the impact of changing the computing platform while the service remains on-line.

Mendosus provides support for service designers to change the topology of the emulated system in an ongoing emulation study. Table 2 lists the supported operations. Currently, changes in topology are specified in the configuration file along with a time at which they should be enacted. We are working on an interactive interface that would allow the user to modify the topology in real-time.

4 Per-Node Daemon and Device Driver

Much of the per-node implementation of Mendosus resides inside our LAN emulator, which is embedded in an emulated Ethernet driver. Currently, our LAN emulator is embedded inside the emulated Ethernet driver provided by Giganet as part of the cLAN driver. Our implementation runs on Linux 2.2.14 with IP aliasing and Giganet cLAN 1.3.0.

Event Type	Corresponding Component	Corresponding Event	Event parameters
topo_nic_rm	Network Interface	NIC removed from the host	N/A
topo_nic_add	LAN Interface	New NIC added	host and virtual-IP
topo_conn_rm	NIC to Switch Connection	Connection removed from the network	N/A
topo_conn_add	NIC to Switch Connection	Connection added to the network	NIC,switch-id and connection-id
topo_sw_rm	Switch	Switch removed from the network	N/A
topo_sw_add	LAN	New switch added to network	switch-id
topo_link_rm	Switch to Switch Link	Link removed from the network	N/A
topo_link_add	LAN	New Link added	both switches and link-id
topo_reconfig	LAN	Switches should reconfigure the spanning tree	N/A

Table 2: *Types of topology changes that Mendosus currently support.*

4.1 Components and Data Structures

A virtual network to be emulated by Mendosus can be comprised of four types of components:

1. **Switch/Hub:** A switch/hub is comprised of a group of ports.
2. **Port:** A port belongs to one switch/hub and is one type of end-termination for links.
3. **NIC:** A NIC is associated with one unique IP address and is a second type of end-termination for links.
4. **Connection Line:** A connection line is a link between two switches/hubs or one switch/hub and one NIC. One end of a connection line is a port, and the other end is either a port or a NIC.

The status of the components of a virtual network is maintained in three tables by the LAN emulator:

1. **Switch Table:** The switch table is a list of all switches and hubs in the virtual network. Each switch/hub has a unique ID, which is used to index this table. The generated spanning tree is also stored in this table, with the entry for each switch/hub containing a pointer to the switch/hub's parent in the spanning tree.
2. **NIC Table:** The NIC table contains all NICs in the virtual network. This is a hash table, and each NIC entry is accessible by its IP address.
3. **Connection Line Table:** The connection line table is a list of all connection lines.

4.2 Implementation

Routing. The LAN emulator uses the above three tables together with a root provided by the controller to generate a breadth-first spanning tree for routing. The spanning tree is kept as part of the switch table: that is, as already mentioned, each entry for a switch or hub has a pointer to the parent switch/hub in the spanning tree. When a packet is to be sent over the emulated LAN, we walk up the spanning tree from the given source and destination IP addresses to find the least common ancestor. This determines the route that the packet will take from the source to the destination in the emulated LAN. If any component in this path is in a faulty state, the packet is dropped (if the component is in an intermittent error state, then the packet may be dropped according to some random distribution).

Although the spanning tree is computed locally at each node, all nodes should compute the same spanning tree as all information should be the same (minus inconsistencies when the controller is in the middle of effecting a fault, which requires the sequential communication with multiple nodes). When the network is partitioned, however, then nodes in different partitions will compute different spanning trees.

Multicast and Broadcast. The LAN emulator emulates Ethernet hardware broadcast by sending a copy of the message to each node in the cluster. Multicast is implemented as a broadcast. When a node receives extraneous multicast packets, it simply drops it.

NIC Failover. The LAN emulator supports multiple virtual network interfaces on each host, which is implemented with IP aliases. Thus, our emulator can emulate network interface failover. That is, when one virtual network interface goes down, the virtual IP address associated with this interface is automatically remapped to another active virtual interface on the same host.

Ioctl Interface. The LAN emulator provides an ioctl-based interface for user-level manipulation of these tables. The main purpose of the per-node daemon is to serve as a conduit for information to be passed from the controller to the driver via this ioctl interface.

	Number of Switches	Throughput (MB/sec)	RTT Latency (usec)
Lan_emu 1	1	79.6	53.4
Lan_emu 2	8	79.1	54.8
Fast Ethernet	1	11.8	88.9
Gigabit Ethernet	0	66.0	130.0

Table 3: *Mendosus's base throughput and latency compared to that achievable on Fast and Gigabit Ethernet. Throughput was measured using netperf over TCP with 65 KB messages. Latency was measured using netperf over UDP with 1 byte messages. All experiments were run on a cluster of 800 MHz Pentium III PCs connected by an HP ProCurve (Fast Ethernet) and a Giganet VIA SAN. Measurements for the Gigabit Ethernet were run using a pair of Alteon cards connected back-to-back.*

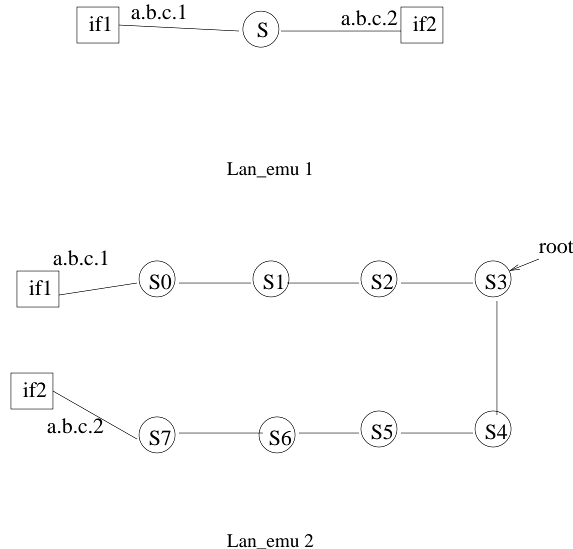

Figure 4: *Emulated systems used for measuring the capability of our emulation network. In Lan_emu 1, messages are being sent from if1 to if2 through 1 intervening switch. In Lan_emu 2, messages are being sent from if1 to if2 through 8 intervening switches.*

5 Performance

We assess Mendosus's performance by running netperf (version 2.2alpha) over the two emulated LANs shown in Figure 4. These measurements are meant to give approximate performance upper bounds on the interconnection systems that Mendosus can emulate. That is, Mendosus should be useful in emulating any networking system that has matching or higher latency and lower throughput that it can achieve. As the performance of the emulated network outstrips what is possible on Mendosus, the emulation will become less accurate. In some instances, Mendosus may still be useful; for example, when the investigator would like to test the correctness of a protocol under non-saturated conditions. Note that aggregate bandwidth is currently not a concern because we are not assuming emulation of end nodes. That is, each PC in the cluster would represent a real machine on a deployed system; only the virtual network is emulated.

Table 3 shows Mendosus's throughput and latency; throughput and latency are also given for Fast (100 Mb/s) and Gigabit Ethernet LANs for comparison. Observe that Mendosus outperforms both LANs, and so should be suitable for emulating systems interconnected by these networks (although Mendosus cannot emulate layer 3 & 4 functionalities that are implemented by some Gigabit Ethernet switches)[2]. Further, note that there is very little performance degradation with increasing number of switches that must be traversed in the emulated LAN. Thus, we should be able to emulate virtual networks comprised of large numbers of switches and hubs.

We also measured the latency of broadcasting over Mendosus. Table 4 shows Mendosus's performance compared to Fast Ethernet. The experiment starts when a sender sends a multicast or broadcast message to a set of receivers. The experiment ends when all replies have been received by the sender. While it seems as though Mendosus outperforms Ethernet, this is not entirely true; since our numbers include the time to collect replies from all receivers, the Ethernet is at a disadvantage because of its higher per-message overheads. What we can conclude from these numbers, however, is that the emulated broadcast is reasonably comparable to the hardware broadcast of Ethernet, and so does not introduce significant inaccuracy into the emulation.

Finally, Figure 5 shows the measured system-wide fault injection delay vs number of daemons to contact. This delay may be important because it represents a window of time when the views of the network among the nodes

[2]The Alteon LAN may outperform Mendosus with respect to bandwidth when jumbo frames are used; we did not report this number, however, which is on order of 100 MB/sec, because we were unable to get stable netperf results in the time available.

Number of Nodes	Lan_emu Broadcast (usec)	Ethernet Multicast (usec)	Ethernet Broadcast (usec)
2	96.96	98.97	99.58
4	105.99	117.13	117.85
6	114.07	135.37	135.81
8	117.85	153.43	154.16
10	125.41	172.90	172.68
12	133.76	189.80	190.60

Table 4: *Mendosus broadcast latency compared to that of Fast Ethernet. All experiments were run on a cluster of 800 MHz Pentium III PCs connected by an HP ProCurve and a Giganet VIA SAN.*

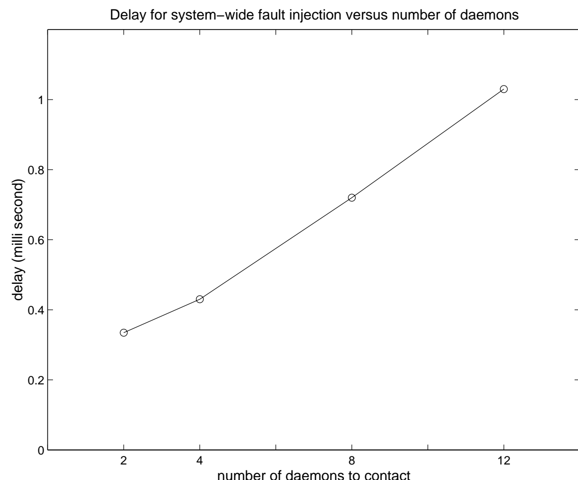

Figure 5: *System-wide fault injection delay vs number of daemons to contact. Delay is computed as the time from when the controller decides to inject a fault until all the daemons have acted on the fault and confirmed with the controller.*

are inconsistent. Observe that while the delay increases linearly with the size of the cluster, the absolute magnitude is on the order of 1 ms, which is too small to matter in most cases.

6 Case Studies

In this section, we show the usefulness and flexibility of Mendosus by examining two case studies. In the first, we use Mendosus to test the correctness of the implementation of a membership protocol; this implementation is part of a toolkit of distributed data structures being designed and implemented to ease the task of building highly available Internet services. In the second, we use Mendosus to measure the increase in availability of a file transfer service as redundant resources are added to the interconnection network.

6.1 Group Membership

The membership protocol we have implemented, called MTRM, is a variation of the Three Round Membership (TRM) [2]. While TRM is targeted to fast and reliable LANs, it has also been designed to be highly robust to potential failures. In this case study, we use Mendosus to inject faults into a cluster running our membership protocol to test whether our implementation gives the expected behavior. Prior to the implementation of Mendosus, testing MTRM was a very difficult and tedious task.

Protocol. Under MTRM, participating nodes are arranged into a logical ring. Each node monitors the status of its upstream and downstream neighbors using point-to-point ALIVE messages. Dynamic node addition is initiated by a new node requesting a current ring member to act as a coordinator for its inclusion. The chosen coordinator then follows a two-phase commit protocol with the current members of the group. The removal of a node proceeds similarly and is initiated when a node does not receive two (*tolerance threshold T*) consecutive ALIVE messages from it neighbor.

When two or more faults occur close together, a coordinator will not be able to complete the two-phase commit necessary to remove a faulty member. To handle these events, MTRM implements a Fast Convergence Mode (FCM). MTRM enters FCM after N_f rounds (of two-phase commit) have failed to achieve consensus about the remaining non-faulty members. In FCM, each node broadcasts an FCM_QUERY message, requesting nodes to report their liveness. Once the group of live nodes has been compiled, the node sends out a new membership message containing a list of all nodes that replied. This leads to fast convergence and once the membership stabilizes, nodes switch back to normal mode.

Fault Injection. Figure 6 gives the (simple) topology

Fault and Recovery Events	Effect on Membership
1.) N/W interface at B down	B is removed from G = {A, C, D}
2.) Link S1-S2 down	Network partition G = {A} G' = {C, D}
3.) Node B up	B joins G = {A, B}
4.) 2 consecutive packet drops at interface A	B removes A from G={B} Ensuing steady state: A & B merge into G = {A, B}
5.) link S1-S2 up	G' and G merge into G = {A, B, C, D}

Table 5: *Faults injected during an emulation run.*

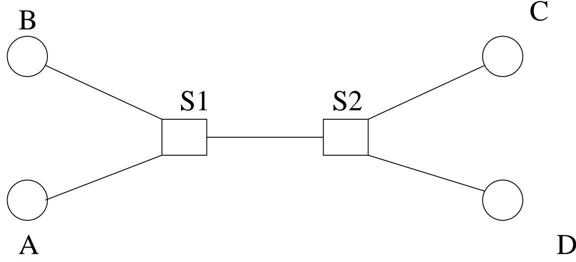

Figure 6: *Topology for testing membership protocol.*

that we use to test MTRM for this paper. It contains four nodes (A, B, C, and D) connected by two switches ($SW1$ and $SW2$), with each node running an instance of MTRM. Table 5 lists the faults that were sequentially introduced during a single emulation run. Alongside each fault, we state the expected behavior that should be validated by the emulation run if our implementation is correct.

Figure 7 presents the results of the emulation run with the fault and recovery events indicated by the dotted lines. Neighboring nodes are monitored for liveness every 10 seconds and N_f is set to 5 rounds.

The first fault brings down the network interface of B and prevents any communication between B and the rest of the group. This may be the result of either a software or hardware fault at the interface. A new group with A, C and D is created after consistently removing B. On the other hand, node B does not receive ALIVE messages and, after N_f rounds to remove its neighbors fail, enters the FCM and converges to group G' with itself as the sole member.

The second fault brings down the link between the 2 switches causing a partition that isolates A from C and D. While C and D agree that A is down, node A switches to FCM mode after N_f failures to determine membership and creates a new group comprised of only itself. We note that the time for membership to converge after the partition (event 2 to when A sees only itself on the group) is 50 secs as expected. This is primarily

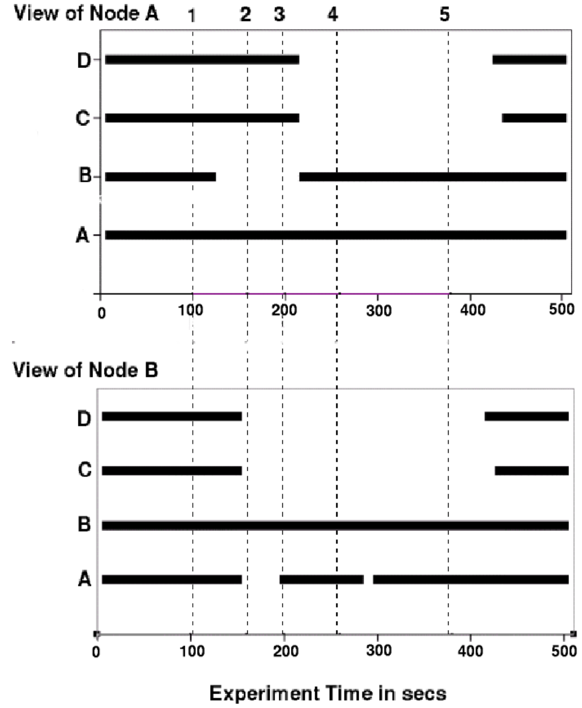

Figure 7: *Membership views of nodes A and B, showing that our implementation of MTRM indeed gives the expected behavior in response to the sequence of faults introduced.*

dictated by N_f and can be configured as desired.

To handle inconsistencies after network partitions, the lowest ID member of a group periodically broadcasts the membership through a CURR_GRP_MEM message. If there exist multiple exclusive groups, then the groups with smaller numbers of members renounce their current membership to join the largest group. In the experiment, after the interface on node B recovers (event 3), B joins the group with A as described above. The recovery from the network partition (event 5) is similar and the membership takes 60 seconds to converge and could

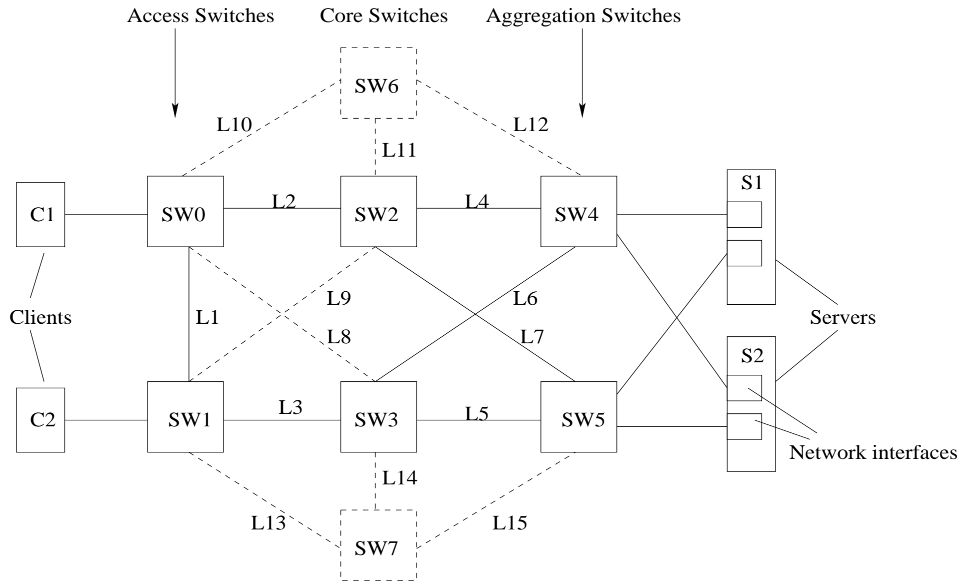

Figure 8: *Topology of a multi-level switch network.*

be altered by changing the delay between consecutive membership advertisements.

The third fault (event 4) gives an example of the usefulness of the emulator in visualizing subtle interactions in distributed protocols. The consecutive packet drops could be the result of network buffer overflows due to unusual amount of traffic. This fault leads B to assume that A is no longer alive. B then removes A from the group, causing A to not receive any ALIVE messages. This should lead A to remove B, forming its own group. *This did not happen.* By observing the logs of A, we found that A at this point, being the lowest ID member of the group {A, B}, had sent out a CURR_GRP_MEM message and B, on receiving it, disbanded its singleton group to join A.

6.2 Multi-Level Switch Network

A typical enterprise LAN is complex with a large number of components support by multiple redundancies to provide high availability. Such a network is comprised of multiple layers of switches, which aggregate and distribute traffic at the edges of the server-client model. The intermediate switches execute intelligent routing protocols such as Hot Standby Routing Protocol (HSRP) and other load balancing schemes. Figure 8 shows the topology we use for our second case study, which is a relatively simple version of the multi-level switch network described above. Our emulator allows a network designer to inject arbitrary faults and determine if the topology is robust to such failures. In the current implementation, the switches are simple and do not implement any fancy routing protocols. We utilize a simple set of file transfer and transaction applications which monitor the network connectivity (between servers and clients) to give an idea of the robustness of the network. The emulator allows the designer to experiment with different configurations to study the benefits that redundant resources may bring.

In Figure 8, the solid components comprise the basic topology and the dashed components are added redundancy for the availability analysis. In the first test, clients ($C1$, $C2$) accessing the file service monitor the bandwidth of data received from the servers ($S1$, $S2$). The faults introduced in this experiment included failure of switches $SW2$ and $SW3$ and links $L2$ and $L3$. The failure of any one of these components is not fatal, since traffic can be routed around the fault. This is achieved by the reconfiguration of the network when the switches run the spanning tree protocol to determine live routes. However, the failure of either combinations of $SW2$ and $SW3$, $SW3$ and $L2$, or $SW2$ and $L3$ cripple the network as seen in the results in Figure 9 .

Availability Analysis. To measure system availability, we allow faults to be injected for every component of the network as described in Table 6. The average rate (failures/sec) of failures have been scaled up to limit the time to convergence of the experimental results. The clients in this experiment were aware of all the servers in the service cluster. This mimics the practical mechanism in which clients on looking up the service name gets dif-

Fault	Details
1.) Switch down	Poisson process; rate = 0.005; recover after 30s
2.) Link down (Links between switches)	Poisson process; rate= 0.005; recover after 30s
3.) Network Interface Down	NICs on $S1$ and $S2$; Poisson process, rate = 0.005; recover after 30s

Table 6: *Faults injected in Multi-level switched network*

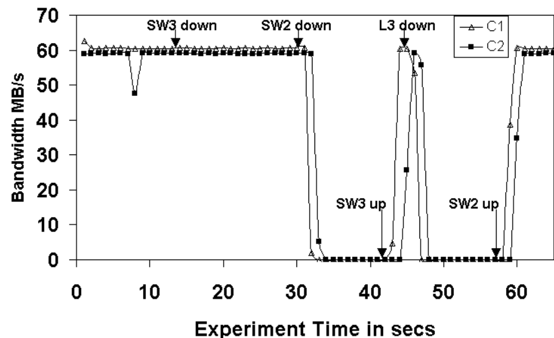

Figure 9: *Bandwidth measured at each client over time as faults are injected into the interconnection network.*

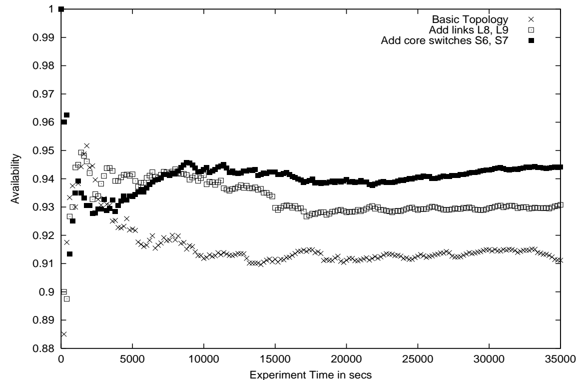

Figure 10: *Availability with increased redundancy*

ferent network endpoints at different times. The clients send out transaction requests through a UDP message to a server and the server replies back with result of the transaction. The clients timeout after a period of 1 second, which is sufficient time for a server to reply if it is online and there exists a path between the client and the server. On failed attempts, the clients cycle through the servers they are aware of. Availability of the service is defined as the ratio of number of successfully satisfied requests to the total number of attempts.

We investigated three topologies, each time increasing the redundancy of the network components within the basic network shown in Figure 8. The second (enhanced) topology included addition of links $L8$ and $L9$, while in the third, extra resources were added within the core network—$S6$, $S7$ and corresponding links. The experiments were run for a duration long enough for the availability as recorded by the clients to stabilize within a margin of 1 percent. Figure 10 shows the cumulative average of the availability of the service cluster. We observe that redundancy increases the overall availability of the service and for the final topology we achieve an increase of 3.5 percent over the basic topology. A system designer could use this data to decide whether the added availability justifies the added expense of buying, operating, and maintaining the redundant components.

7 Related Work

Our work primarily relates to contributions in the fields of network emulation and fault injection, though we borrow ideas from simulation work such as ns-2 [24]. The work at Utah, Emulab [23], a configurable Internet emulator of 330 nodes, offers a testbed for distributed applications. A fraction of the nodes, can be configured as routers, traffic generators or shaper nodes which allow control over packet delays, re-orderings and losses. However, they do not consider failure of individual network components and its affect on routing, connectivity and application behavior. DelayLine[10], a user level, WAN emulator library that allows variation in network performance, provides a socket interface and involves minimal additions and recompilation of the applications. While our implementation is at the kernel level and involves no modifications to the application, our architecture is similar to DelayLine in that we both maintain a central configurator to provide all nodes with a uniform view of the network topology.

Unlike our distributed emulation model, the Lancaster emulator[4] uses a centralized process which moderates all traffic flow in the network. Similarly NISTNET [15] emulates a single configurable router which routes all traffic between the end-points in WAN-based systems and implements delay distributions, packet re-orderings, packet drops and duplication. The objective of these works is to test the effect of network performance dy-

namics on sensitive applications and not failure of network components in general. Our emulation toolkit, while not limited to, allows application performance evaluations similar to the above works and the logP [14] work which studies the communication performance of parallel applications to variations in the latency, overhead and bandwidth of the underlying network. Similar to the trace-based emulation work at CMU [16] which creates a repeatable network environment using empirical parameters derived from collected traces, we support the inclusion of real world fault and performance data in our emulator along with the regular scripts.

In fault-injection research there are numerous projects that look at both hardware and software based fault-injection techniques [5, 8, 11, 19]. Our approach is similar to Orchestra [5], which uses a software based, script-driven probing and fault-injection method to test distributed protocols by introducing a layer in the protocol stack. Another dependability study [21] embeds the fault injection layer in the NIC and studies how the application is affected by packet drops, corrupted messages, delays, interface resets and hangs. All of the above introduce faults locally and are useful for testing application behavior locally. Though our injection technique is fundamentally similar, we introduce network component failures consistently across entire network and hence observe the effects on a global level. Loki [3] a fault injector from UIUC injects correlated faults by maintaining partial view of global state of the system. Our focus currently is on hardware faults and extending it to inject software correlated faults will be possible by using our central fault injection engine for consistent global state.

8 Conclusions and Future Work

As availability becomes an increasingly important "quality" metric for network services, the need for a realistic, emulation-based test-bed is rapidly emerging. In this paper, we have presented Mendosus, a SAN-based fault-injection test-bed that has been designed to allow for emulation of LAN-based systems. Our goal is to provide a test-bed for service designers to systematically assess the impact of end-to-end design decisions on the availability and reliability of network services in the presence of realistic fault scenarios. Although our current implementation is only a first step toward this vision, we demonstrate its flexibility and usefulness through two case studies. In one, we test the correctness of a membership protocol critical to the implementation of many highly available cluster-based services. In the second, we show how a system designer could use Mendosus to assess the gain in availability of a system against the cost of adding redundant resources.

Future work will extend Mendosus to emulate more classes of networks. In particular, our next objectives will be directed toward wireless LAN's and WANs. The challenges in the wireless space include incorporating realistic fault models, managing the extreme performance mismatch between the wireless network and the SAN, and properly emulating the broadcast nature of the wireless network over point-to-point SAN links.

Future work on emulating WAN scenarios includes managing the large volume of traffic through the routers and long-delay links. In addition, correctly emulating WAN router behavior, with their myriad of congestion control algorithms, presents an especially significant challenge.

References

[1] E. Brewer. Lession Learned from Internet Services: ACID vs BASE. Talk presented at the National Science Foundation sponsored workshop on Industrial/Academic Cooperation in Database Systems, Oct. 1998.

[2] F. Cristian and F. Schmuck. Agreeing on Processor Group Membership in Timed Asynchronous Distributed Systems. 1995.

[3] M. Cukier, R. Chandra, D. Henke, J. Pistole, and W. H. Sanders. Fault injection based on a partial view of the global state of a distributed system. In *Symposium on Reliable Distributed Systems*, pages 168–177, 1999.

[4] N. Davies, G. S. Blair, K. Cheverst, and A. Friday. A network emulator to support the development of adaptive applications. Technical report, 1995.

[5] S. Dawson, F. Jahanian, and T. Mitton. ORCHESTRA: A fault injection environment for distributed systems. Technical Report CSE-TR-318-96, 26 1996.

[6] Electronic News. Ebay Outages Cast Clouds on Sun. http://www.electronicnews.com/enews/Issue/1999/06211999/25ebayah.asp, June 1999.

[7] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, 2000.

[8] S. Han, K. Shin, and H. Rosenberg. Doctor: An integrated software fault injection environment for distributed real-time systems, 1995.

[9] T. Heath, R. Martin, and T. D. Nguyen. The Shape of Failure. In *Proceedings of the First Workshop on Evaluating and Architecting System dependabilitY (EASY)*, July 2001.

[10] D. B. Ingham and G. D. Parrington. Delayline: a wide-area network emulation tool. *USENIX, Computing Systems*, 7(3):313–332, 1994.

[11] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: A tool for the validation of system dependability properties. In *FTCS-22: 22nd International Symposium on Fault Tolerant Computing*, pages 336–344, Boston, Massachusetts, 1992. IEEE Computer Society Press.

[12] Manoj Joshi and Nicholas Brenton, Cisco Systems, Jim Helfrich and Patrick Jones, Network Appliance. High availability for network-attached storage. Technical Report TR 3115, Network Appliance and Cisco Systems, 2000.

[13] R. Martin, K. Nagaraja, and T. D. Nguyen. Using Distributed Data Structures for Constructing Cluster-Based Services. In *Proceedings of the First Workshop on Evaluating and Architecting System dependabilitY (EASY)*, July 2001.

[14] R. Martin, A. Vahdat, D. Culler, and T. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *International Symposium on Computer Architecture*, Jun 1997.

[15] National Institute of Standards and Technology(NIST). NISTNET emulator. Available at http://snad.ncsl.nist.gov/itg/nistnet/.

[16] B. Noble, M. Satyanarayanan, G. T. Nguyen, and R. H. Katz. Trace-based mobile network emulation. In *SIGCOMM*, pages 51–61, 1997.

[17] Reuters. Britannica Competitors Study Net Strategy. http://news.cnet.com/news/0-1005-200-1435449.html, Nov. 1999.

[18] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, Availability and Performance in Porcupine: A Highly Scalable Internet Mail Service. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, Charlston, SC, Dec. 1999.

[19] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, D. Rancey, A. Robinson, and T. Lin. FIAT w fault injection based automated testing environment. In *Proc. 18th Int. Symp. on Fault-Tolerant Computing (FTCS-18)*, pages 102–107, Tokyo, Japan, 1988. IEEE Computer Society Press.

[20] SiliconValley.internet.com. Ebay Outage Twice This Week. http://siliconvalley.internet.com/news/article/0,,3531_435741,00.html, Aug. 2000.

[21] D. T. Stott, G. Ries, M.-C. Hsueh, and R. K. Iyer. Dependability analysis of a high-speed network using software-implemented fault injection and simulated fault injection. *IEEE Transactions on Computers*, 47(1):108–119, 1998.

[22] N. Talagala and D. Patterson. An Analysis of Error Behaviour in a Large Storage System. In *The 1999 Workshop on Fault Tolerance in Parallel and Distributed Systems*, 1999.

[23] University of Utah Flux Research Group. Utah Network Testbed. Available at http://www.emulab.net.

[24] USC/ISI, UCSB, CMU. Network Simulator ns-2. Available at http://www.isi.edu/nsnam/ns/.