Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Systems and Software 78 (2005) 270–281
The Journal of Systems and Software
www.elsevier.com/locate/jss

HANet: a framework toward ultimately reliable network services

Ying-Jie Jiang, Da-Wei Chang *, Ruei-Chuan Chang

Department of Computer and Information Science, National Chiao-Tung University, 1001 Ta Hsueh Road, HsinChu, Taiwan 30050, ROC

Received 16 December 2003; received in revised form 11 October 2004; accepted 21 October 2004
Available online 26 November 2004

Abstract

High availability is becoming an essential part of network services because even a little downtime may lead to a great loss of money. According to previous research, network failure is one of the major causes of system unavailability. In this paper, we propose a framework called HANet for building highly available network services. The main goal of HANet is to allow a server to continue providing services when all its network interfaces to the outside world (i.e., public interfaces) have failed. This is achieved by two techniques. First, a network interface can be backed up not only by other public network interfaces, but also by other inter-server I/O communication interfaces (i.e., private interfaces) such as Ethernet, USB, RS232, etc. Therefore, IP packets can still be transmitted and received via these I/O links, even when all of the public network interfaces have failed. Second, HANet allows a server to take over the packet transmission job of another network-failed server.

The benefit of HANet is that a network-failed server will not lose any requests which are being processed. And, it is efficient since no synchronization overhead or replaying process is required. Moreover, it is totally transparent to server applications and clients. To demonstrate the feasibility of HANet, we implemented it in the Linux kernel. According to the performance results, using a private Fast Ethernet interface for data communication leads to only 1% overhead in user-perceived latency when the public Fast Ethernet interface of the server has failed. This indicates that HANet is efficient, and hence is feasible for commercial network services.

Keywords: High availability; Fault tolerant; Operating system; Network services

1. Introduction

In recent years, the development of E-commerce has led to the emergence of many business websites. However, the services they provide may not always available due to the hardware or software component errors. According to previous research (Performance Technologies, 2001), a few minutes of downtime for a website can lead to a great loss of money for the business.

Therefore, much research effort has been addressed to the problem of how to improve the availability of a network service. Clusters (Cristian, 1991) use machine redundancy to achieve high availability, so when one machine fails, another machine takes over the job. However, the requests being processed in the failed machine will be discarded, and in most of the cases, the requests will be issued again. With connection migration (Snoeren et al., 2001), a request can be migrated or recovered. However, a migrated request still has to be replayed again from the beginning, which increases the request-serving delay and may cause the client to time-out. The problem of the existing clusters is that they discard the request states in the failed machine. However, a previous study (Oppenheimer et al., 2003) showed that network failure is one of the major reasons for the service errors. If a system failure is caused by the network problem, the server process in the failed machine will

* Corresponding author. Tel.: +886 3 5712121x56656; fax: +886 3 5721490.
E-mail addresses: chiangyj@csie.nctu.edu.tw (Y.-J. Jiang), david@os.nctu.edu.tw (D.-W. Chang), rc@cc.nctu.edu.tw (R.-C. Chang).

0164-1212/$ - see front matter © 2004 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2004.10.013

still work and the request states will remain correct. In this case, the service will still be available as long the request states can be obtained from the failed server.

Another technique that improves system availability is component redundancy (Intel Corporation, 2003; Jann et al., 2003; Patterson et al., 1989). Since network cards are one of the most important components of a network service, many network systems use network card redundancy to improve their availability. Thus, the systems are available as long as one network card works correctly. However, this approach has some limitations. First, most computers have very limited slots (i.e., usually 2–4) for network cards, which prevents them from providing extremely reliable network services. Second, according to previous research (Engler et al., 2000), drivers are the most error-prone part of an operating system. If the same driver is used for all of the network cards, a problem in the driver may cause all of the cards to stop working at the same time.

In this paper, we propose a framework named HANet for building highly available network services. Based on a cluster architecture, HANet allows a server to continue providing services while all its network interfaces to the outside world (i.e., public interfaces) have failed. It consists of two techniques: Packet Transmission Agent (PTA) and Uniform Communication Channel (UCC). The former allows a server to take over the packet transmission job of another network-failed server. The latter extends the network interface redundancy mechanism. It allows a public network interface to be backed up by not only other public interfaces, but also other inter-server communication links (i.e., private interfaces) such as USB, RS232, wireless, etc. These two techniques together help service providers to provide extremely highly available network services. Moreover, the techniques are transparent to both client-side systems and server-side applications. In order to demonstrate the feasibility of the proposed framework, we implemented it in the Linux kernel. According to the performance results, the system can achieve the desired functionality with little overhead.

The rest of this paper is organized as follows. In Section 2, we describe the design of the framework. Section 3 presents the implementation details, which are followed by the performance evaluation shown in Section 4. In Section 5, we discuss some extensions to the current HANet implementation. Section 6 presents some works related to ours, which is followed by the conclusions in Section 7.

2. Design of HANet

In this section we first give an overview of HANet, which is followed by the description of the uniform communication channel and the packet transmission agent techniques.

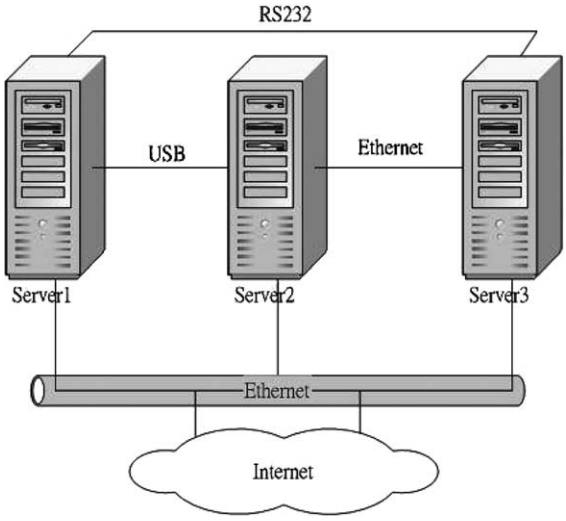

Fig. 1. Overview of HANet.

2.1. Overview of HANet

Fig. 1 illustrates the system architecture of HANet. It consists of a cluster of server machines, which are connected via more than one communication interface. Some interfaces (i.e., public interfaces) connect directly to the public network, while the others (i.e., private interfaces) are for inter-server communication. The unique feature of HANet is that it uses various kinds of I/O communication channels, [1] instead of using network interface cards (NIC) only, for packet transmission. In Fig. 1, server1 originally serves the clients via the Ethernet interface. If the interface suddenly fails, the server will select a sibling, say server2, to handle the packet transmission job for it. As a result, packets sent by server1 to the client will reach server2 first (via the USB interface), and then server2 will forward the packets through its Ethernet interface. Incoming packets are sent to server2, which forwards the packets back to server1.

It is significant that server1 does not lose any requests or connections while its NIC fails. Therefore, there is no need to re-construct any connections or re-dispatch any requests. Moreover, this approach is transparent to both clients and server applications.

2.2. Uniform communication channel

As we have mentioned above, the UCC architecture allows a network interface to be backed up by not only

[1] In current implementation, the I/O channels include Ethernet, RS232, and USB links.

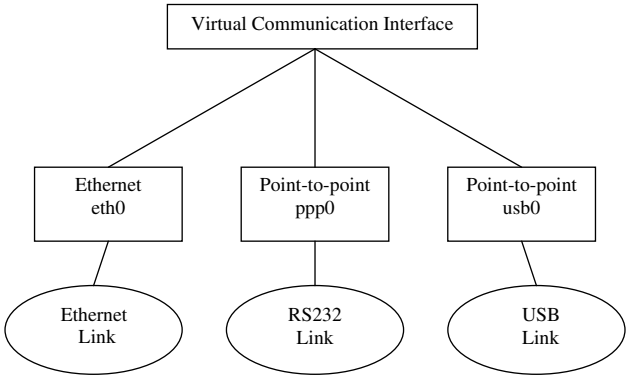

Fig. 2. UCC architecture.

other public network interfaces, but also other private communication links such as USB, RS232, wireless, etc. In the UCC architecture, a *virtual* communication channel is mapped to multiple *physical* links with different types (e.g., Ethernet, USB, and RS232), which are responsible for packet transmission and reception. The channel is available as long as at least one physical link works correctly. This approach can improve the availability of a network system greatly.

Fig. 2 shows the concept of the UCC. As shown in the figure, we add a virtual communication interface (VCI) layer on top of the link layer. This layer is responsible for managing multiple physical interfaces of different types. At any time, only one of the physical interfaces is active, while others act as backups. If the active interface fails, the VCI layer will detect the error and activate another backup interface.

There are two components, namely the fault detector and the interface manager, in the VCI layer. Below we give detailed descriptions on them.

2.2.1. Fault detector

The fault detector is responsible for detecting failures and maintaining the status of different I/O communication devices. We refer to these devices as network devices in the rest of the paper since they can be used for packet transmission. The fault detector executes the fault detection routines periodically, and records the status of the managed network devices.

The status of a network device includes two flags: ACTIVE and GOOD. If the ACTIVE flag is set, the network device is currently the active interface. That is, the device is currently responsible for transmitting/receiving packets to/from the outside world. The GOOD flag indicates that the device is alive and can be used as an active interface. Note that only one network interface can be active at any time.

After updating the status of the network interfaces according to the results of fault detection routines, the fault detector will decide whether or not to change the active network interface. If a change is needed, it will ask the interface manager to determine which network interface is the next active one.

2.2.2. Interface manager

The interface manager determines the next active interface (from the current GOOD interfaces) according to the following rules. First, high bandwidth links are preferred. For example, a 100 Mbps Fast Ethernet link is better than a 12 Mbps USB 1.1 link, which in turn is better than a 115.2 kbps RS232 link. Second, public interfaces are preferred. If no public interfaces are available, a private interface will be selected. As we describe in the next section, a network packet from a server can be transmitted to the client via the private interfaces. However, transmitting packets via private interfaces has more overhead since the packets must be routed to the sibling first. Therefore, a server will try its best sending packets via its public interfaces unless all of them have failed.

When the next active network interface is determined, the interface manager must set the attributes of the virtual network interface according to the link-layer type of the active interface. There are two link-layer types, broadcasting (e.g., Ethernet) and point-to-point (e.g., USB, and RS232). The attributes include the hardware header length, MAC address length, etc. After the attributes are set, packets can be transmitted and received via the new active interface.

2.3. Packet transmission agent

The PTA technique allows a server to take over the packet transmission job for its sibling while the latter can not access the public network. As we mentioned above, if the active interface of server1 fails, and there are no public interfaces available, the interface manager of server1 will select a private interface. Then, it will ask the other end of the private interface (i.e., the sibling server) to perform packet transmission and reception for it. To take over the packet transmission job, the sibling server must configure itself to be able to forward packets sent by the server1 as well as deliver packets to the server1. After the configuration is done, server1 can send and receive packets through the sibling computer. Fig. 3 illustrates this procedure and the resulting packet flow. Note that, in this figure, when the Ethernet adapter of the server revives, the active network interface will be set back to eth0 for performance consideration.

Functionality of the PTA technique is achieved by the cooperation of the two components: the transmission agent requester on the network-failed server, and the transmission agent responder on the sibling server.

Transmission agent requester. The responsibility of the transmission agent requester is to ask the sibling

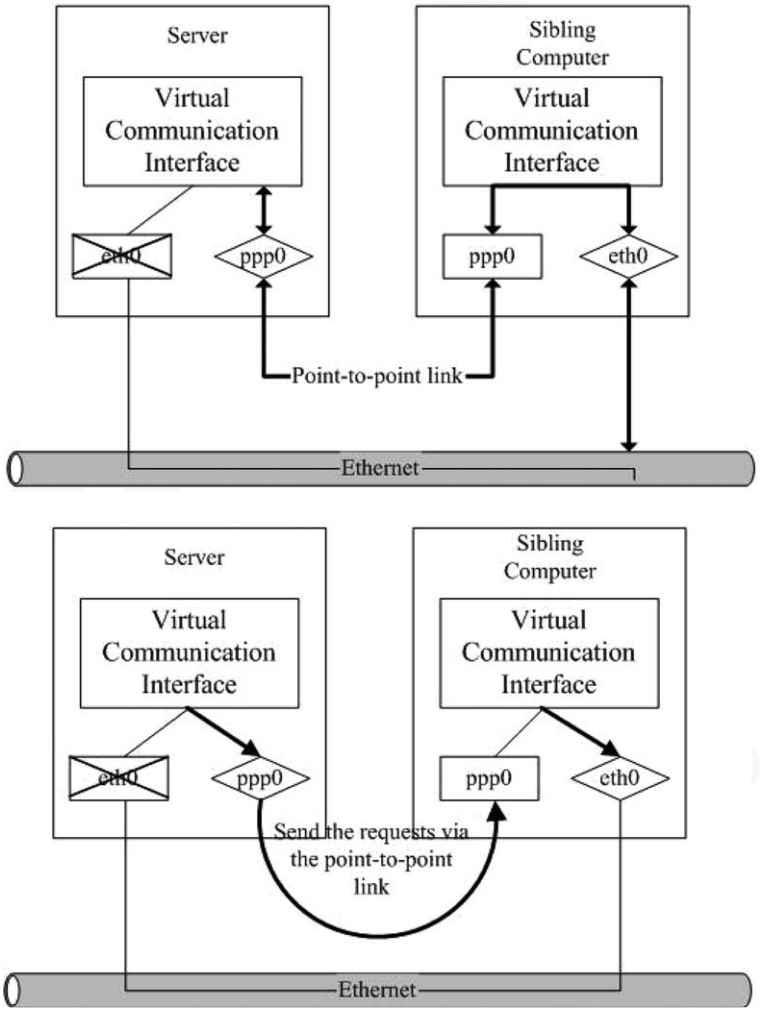

Fig. 3. Packet transmission take-over and the resulting packet flow.

computer to send and receive packets for it. It tells the sibling computer which network interface is the new active communication channel, so the sibling computer can route packets to the server through the new active channel instead of the failed one.

Transmission agent responder. The transmission agent responder on the sibling computer receives the request, and performs reconfiguration to take over the packet transmission job. Specifically, the reconfiguration includes enabling IP forwarding, enabling proxy ARP, adding a routing path that reaches the server via the new active link, and invalidating its routing cache entries. After the reconfiguration is complete, it broadcasts a gratuitous ARP packet through one of its public interfaces in order to update the ARP cache entries of the other computers on the same LAN. This allows all the IP packets to the server to be sent to the sibling. Therefore, the server can send and receive packets through the sibling computer.

3. Implementation of HANet

To demonstrate the feasibility of HANet, we implemented it in the Linux kernel. The implementation is based on the Linux bonding driver (Davis, 2003), which has the ability to detect Ethernet link failures and reroute network traffic to another Ethernet link in a manner that is transparent to applications. We extended the Linux bonding driver so that it has the following new capabilities. First, it can manage not only Ethernet links but also other types of I/O interfaces. Second, a new fault detection method is used to detect errors of all the communication links. In the following, we describe the position of HANet in the Linux kernel and the implementation details of HANet.

HANet sits between the TCP/IP and link layers, as shown Fig. 4. All the packets from the TCP/IP layer will reach the VCI (Virtual Communication Interface) layer first and be transmitted using the active communication channel in the link layer.

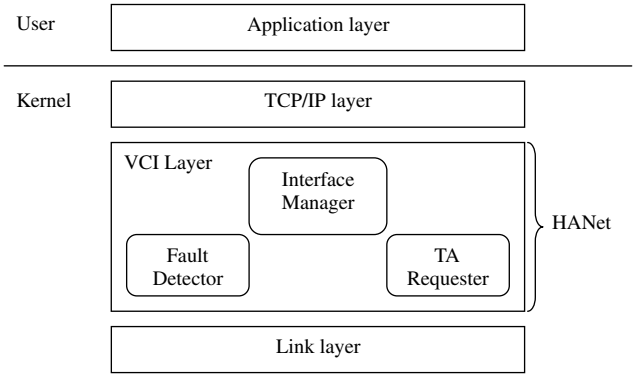

Fig. 4. Position of HANet.

3.1. Supporting multiple link types

In the Linux bonding driver, a virtual device named bonding device enslaves one or more Ethernet devices. We extend the bonding driver implementation so that a bonding device can enslave not only Ethernet devices, but also RS232 and USB devices. Since a bonding device is viewed by the upper layer kernel code as an ordinary network device, it provides a *net_device* structure for communication with the kernel. And, each slave of the bonding device also has a corresponding net_device structure. When the kernel wants to send a packet to the bonding device, it invokes the hard_start_xmit( ) function in the net_device structure of the bonding device. This function finds out the current active slave of the bonding device, and invokes the hard_start_xmit( ) function in the net_device structure of that slave, which actually sends the packet out.

In order to support other kinds of I/O links in the bonding device, we must construct a net_device structure for each I/O device. For RS232 links, we start a PPP daemon (Mackerras, 2004) on the ttyS0 serial device (COM1). This causes two modules to be loaded into the kernel, ppp_generic and ppp_async. The former creates a net_device interface for the serial device, while the latter is responsible for transmitting and receiving data on the serial link. For USB links, we use the USB host-to-host link driver, which is also called the USBnet driver (Brownell, 2002). The driver is used for transmitting and receiving network packets on USB links. Therefore, it creates a net_device structure for the USB device.

We do not change the formats of the packets that are transferred on the private links. Instead, we follow the packet formats used by the link-layer drivers. For example, a packet on the RS232 link does not contain an Ethernet header. Only a 4-byte header (which contains the protocol number) is put in front of the IP header. In contrast, a packet on the USB link does contain an Ethernet header because the USBnet driver puts Ethernet headers on their packets.

Note that some I/O links may not have corresponding drivers that can provide the net_device interface. Integrating these links into the VCI layer requires modifications to the corresponding drivers to create and manage the net_device structures, which needs more effort.

3.2. Fault detector

Originally, the Linux bonding driver supports two kinds of fault detection methods: ARP monitoring and MII link status monitoring. Since both of them have some limitations, we added another method: host ping file, as proposed in (Milz, 1998). Moreover, our fault detector is extensible in that it allows system designers to add their own fault detection methods. In the following, we describe each of the methods.

ARP monitoring. In this method, a host tests the status of its links by periodically sending ARP requests to a target host on the same LAN. If no ARP reply is received, the link will be regarded as bad. This method has some limitations. First, it can only be used in a LAN environment. Second, a false positive will occur if the target host has failed or powered off. Finally, this method can not be used in point-to-point links such as RS232, USB, parallel, etc. since it is useful only for broadcast type links.

MII link status monitoring (Scyld Computing Corporation, 2003): In this method, the fault detector detects the link status by periodically polling the MII status register of the Ethernet network adapters. The drawback of this method is that it is only suitable for Ethernet cards. Many other kinds of links are not equipped with such registers.

Host ping file. In this method, the server periodically pings a list of other computers listed in a file, and the network interface is regarded as bad if the server does not get any replies. Different from the other two methods, this approach does not require special hardware support, and it can be used in both broadcast and point-to-point type links. The disadvantage of this approach is that it has higher fault-detection time.

It is worth mentioning that, in addition to the fault detection approaches described above, our fault detector allows system designers to register their own fault detection methods for some specific network devices. This makes our fault detector extensible for future use.

3.3. Deciding the active interface

As we mentioned before, the interface manager prefers public and high-speed interfaces when choosing a new active device. Although it may be possible for the interface manager to figure out the speed of each interface, the interface manager does not know whether a given interface is public or not. We solve this problem

by allowing the system administrator to send such information to the VCI layer by using the *proc* file system interface. For each slave interface, the administrator can transmit to the VCI layer the following information: whether the interface is public or not, and the speed of the interface. Therefore, the interface manager can select the next active interface accordingly when the current one has failed.

3.4. Interface switching

As we mentioned above, the interface manager will reconfigure the attributes of the VCI when the active network interface is changed. The attributes represent the information stored in the net_device structure. This reconfiguration is required since the Linux network subsystem uses these attributes for packet construction before it transmits the packet to the bonding device. For example, the field hard_header_len specifies the number of bytes that should be reserved for the hardware MAC header, and the field hard_header is a function pointer for constructing the hardware header. Therefore, the attributes of the bonding device should reflect the attributes of its current active slave. Since different I/O links have different attributes, a reconfiguration is needed when the type of the active slave changes. Note that the reconfiguration is not required for the original bonding driver implementation since it manages only Ethernet devices.

```
Switching to a RS232 Interface:
    bond->device->hard_header_len          = PPP_HDRLEN;     /* = 4 */
    bond->device->mtu                      = 1500;
    bond->device->addr_len                 = 0;
    bond->device->type                     = ARPHRD_PPP;
    bond->device->change_mtu               = 0;
    bond->device->hard_header              = 0;
    bond->device->rebuild_header           = 0;
    bond->device->set_mac_address          = 0;
    bond->device->hard_header_cache        = 0;
    bond->device->header_cache_update      = 0;
    bond->device->hard_header_parse        = 0;
    bond->device->flags=    IFF_UP | IFF_POINTOPOINT | IFF_NOARP |
                            IFF_MULTICAST | IFF_MASTER;

Switching to an USB Interface:
    slave_t* usbslave = newslave;
    ether_setup(bond->device);
    bond->device->tx_queue_len    = 0;
      bond->device->change_mtu          = usbslave->dev->change_mtu;
      bond->device->watchdog_timeo      = usb_slave->dev->watchdog_timeo;
      bond->device->tx_timeout          = usb_slave->dev->tx_timeout;
      bond->device->flags               |= IFF_UP | IFF_MASTER;

Switching to an Ethernet Interface:
    ether_setup(bond->device);
    bond->device->tx_queue_len          = 0;
    bond->device->flags                 |= IFF_UP|IFF_MASTER;
```

Fig. 5. Code for VCI reconfiguration.

Fig. 5 shows the reconfiguration code that will be executed when switching to different kinds of active interfaces. The code is derived from the original driver implementation for the corresponding links. For example, the code for switching to a RS232 link is derived from the generic PPP driver, while the code for switching to an USB link is derived from the USBnet driver. Before describing the code, it is worth emphasizing that the code will not modify the attributes of the slaves. It only sets the attributes of the bonding device according to those of the current active slave.

From Fig. 5 we can see that, when switching to a RS232 link, the hard_header_len field of the bonding device is set to 4. This makes the kernel reserve a 4-byte room for the hardware header before it sends the packet to the bonding device. The room will be filled by the hard_start_xmit( ) function of the generic PPP driver. The flag setting follows that in the generic PPP driver except for the IFF_MASTER flag. This flag is set since a bonding device is a master device. The remaining fields are all the same with those in the generic PPP driver.

When switching to an USB interface, the ether_setup( ) function is invoked. The function will set the attributes of the bonding device according the Ethernet information. The USBnet driver calls this function because it uses Ethernet header as the hardware headers for its packets. Because a bonding device is a virtual device without a TX queue, the tx_queue_len should be set as 0. However, the ether_setup( ) function will set the value as 100. Therefore, the value should be reset to 0 after the invocation of the ether_setup( ) function.

The code for switching to an Ethernet interface is quite simple. It just invokes the ether_setup( ) function, and resets the tx_queue_len and the flags.

This description shows that the general principle for writing the reconfiguration code is to follow the code of the original driver implementation. Therefore, adding a new I/O link to the VCI layer is easy providing that the original driver for the new I/O link has managed the net_device structure.

3.5. Packet transmission takeover

In order to take over the packet transmission job, the sibling computer must perform the following tasks. First, the sibling computer should turn on the IP forwarding option, which allows it to forward packets from the server to the destination hosts. This is done by setting the global kernel variable for the IP forwarding option (i.e., ipv4_devconf.forwarding) as 1 and invoking the kernel function inet_forward_change( ) to reflect the changes of the variable. Second, it has to add a routing path to the server, which enables it to forward packets to the server. This is done by calling the kernel function ip_rt_ioctl( ) with the SIOCADDRT command and the

new route path as the parameters. The SIOCADDRT command is used to add the given route path into the host. Third, it should add a proxy ARP entry so that it can answer ARP requests for the server. This is achieved by calling the kernel function inet_dgram_ops.ioctl( ) with the SIOCSARP command and an ARP mapping as the parameters. The SIOCSARP command is used for adding an ARP mapping entry to the local host, and the ARP mapping contains the IP address of the server and the MAC address of the sibling. As a result, the sibling computer will reply with its own MAC address when an ARP request with the server's IP address is seen. Fourth, the sibling computer should invalidate its routing cache entries by calling the rt_ cache_flush( ) kernel function. Finally, it has to invalidate or update the ARP cache entries of the other servers on the same LAN by broadcasting a gratuitous ARP (Stevens, 1994) packet (Horman, 2000). This packet is constructed by hand and sent through one of the public interfaces of the sibling computer. After all these operations are finished, the sibling computer can perform packet transmission and reception for the server.

3.6. Transmission agent responder

The transmission agent responder is responsible for accepting requests from the server and taking over the packet transmission/reception job for the server. We implemented it as an in-kernel UDP server. The reason why we use UDP instead of TCP is that the latter requires a 3-way handshake procedure during the connection setup, which causes the following problem. When the active link switches from a public interface to a private one, the server will open a TCP connection and send a request to the sibling computer via the new active link. However, the SYN/ACK of the 3-way handshaking from the sibling computer will be transmitted via the failed routing path to the server. Therefore, the server can not receive the SYN/ACK, so the connection can not be established. By using UDP, the server can send a request (containing the name of the new active link) to the sibling computer. After knowing the new active link, the sibling computer can update the routing path to the server accordingly.

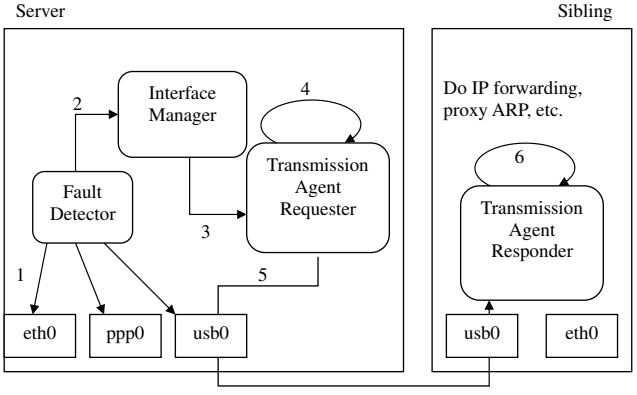

Fig. 6. Flow of changing the active link from eth0 to usb0.

3.7. Control flow for interface switching

In the last part of this section, we show the overall control flow of HANet when the active interface changes from a public one, say eth0, to a private one, say usb0. Fig. 6 illustrates this flow.

1. The fault detector detects an error on eth0 and changes the status of eth0 to bad.
2. The fault detector consults the interface manager to determine the new active channel. The interface manager then chooses usb0 as the new active channel and switches the active channel to usb0.
3. The interface manager notifies the transmission agent requester.
4. The transmission agent requester adds ARP entries into the server to direct all the output packets to the sibling. Then, it flushes the routing cache entries since the output interface is changed.
5. The transmission agent requester asks the sibling computer via the new active channel (usb0) to take over its packet transmission job through usb0.
6. The transmission agent responder (i.e., the in-kernel UDP server) receives the request and takes over the packet transmission/reception job for the server. From this point on, the server can send and receive packets through the sibling computer.

4. Performance evaluation

To measure the performance of HANet, we setup an experimental WWW document-serving testbed, which consists of server and client machines. The major goal of the experiment is to show that even when the public network of a server fails, the WWW server that runs on top of it can still provide services (with little performance degradation) via the network channels of the sibling computers.

The testbed consists of two servers (each of which acts as the sibling of the other), and five clients. All of the machines are connected to a 100 Mbps Fast Ethernet switch. In addition, there are two private channels: an USB link (USB 1.1) and a Fast Ethernet link, [2] between

[2] The private Ethernet link is used for measuring the overhead of transmitting/receiving packets via a private interface. Since the link type does not change after switching to the private interface, the performance degradation after the interface switching reflects the overhead of using private links for data transmission.

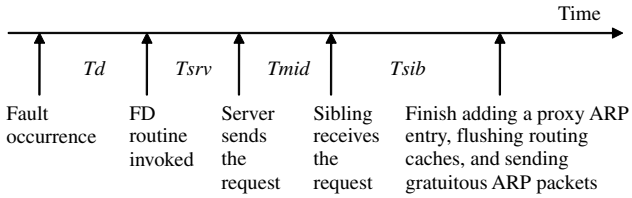

Fig. 7. Different parts of the failover time.

the servers. Each machine is equipped with a 1.6 GHz Pentium 4 CPU and 256 MB DRAM. The operating system is Linux (kernel version 2.4.18–18.8.0), and the HTTP server on the server machine is Apache (version 2.0.40).

In the first experiment, we measure the time required by different parts of the failover procedure. We assume that the original active interface of the server, say eth0, fails, and the next active interface is the private Ethernet link. As shown in Fig. 7, the failover time includes the following components. *Td* is the time between the fault occurs and the invocation of the fault detection routine, which depends mainly on the invocation frequency of the fault detection routine. Since fault detection is triggered every 100 ms in HANet, the average value of *Td* can be regarded as 50 ms. *Tsrv* is the time that the server spends in detecting failure, changing the active network interface, flushing the routing cache, and sending a request to the sibling computer. *Tmid* is the time spent in transmitting the request on the network media, invoking the interrupt service routine in the sibling computer for receiving the request packet, and scheduling the execution of the transmission agent responder. Finally, *Tsib* is the time that the transmission agent responder spends in parsing the request, adding a proxy ARP entry, flushing the routing caches, and sending the gratuitous ARP packets.

Note that the time *Tmid* involves both the server and the sibling hosts that are not time-synchronized, so it can not be measured directly. In order to get an accurate result, we use the same machine for the server and the sibling when measuring *Tmid*. That is, the server sends requests from one network interface to another on the same machine via a private link. In this way, we can ignore the timing synchronization problem between the server and the sibling. Table 1 shows the results, which are measured by using the Pentium Timestamp Counter (Rubini, 2000). This table shows that, the total failover time is about 0.66us plus *Td*, which is 50 ms in average in the current implementation. Therefore, in our current implementation, the failover time is dominated by the fault detection time.

Table 1
Results of different parts of the failover time

Tsvr (ns)	Tmid (ns)	Tsib (ns)	Ttotal (ns)
293.5	69.28	304.55	667.33

In the second experiment, we measure the overhead of HANet. Specifically, we compare the performance of a HANet server with the original server under the condition that the public interfaces are alive. Therefore, they serve HTTP requests through their public interfaces. In this experiment, the five client machines are used to simulate the web users. The workload is obtained from the Surge benchmark (Barford and Crovella, 1998). The experiment time for each round is 1 h.

Fig. 8 shows the throughput results. This figure shows that, adding HANet code into the Linux kernel does not incur visible performance degradation. This is not surprising since the major overhead under this condition is the invocation of the fault detection routines, which consumes little time. Fig. 9 shows the average request processing latency perceived by the users. Similar to Fig. 8, the user-perceived latency is almost the same for the HANet server and the original one, which implies the overhead of HANet is extremely little and can be ignored.

In the third experiment, we measure the server performance under the condition that the public interfaces of the server have failed. In this case, packets go through the private interface. Two private interfaces are used in this experiment, namely a 12 Mbps USB interface and a 100 Mbps Fast Ethernet interface. Fig. 10 show the throughput results. From the figure we can see that, except for the second case, the server throughputs are almost the same. It is because that the bandwidth of USB link is only 12 Mbps, which limits the server throughput. The third case shows that the server using a Fast Ethernet link does not incur throughput degradation since the speed of the Fast Ethernet is sufficiently high.

Fig. 11 shows the latency results. This figure shows that, owing to the low bandwidth, the latency of the

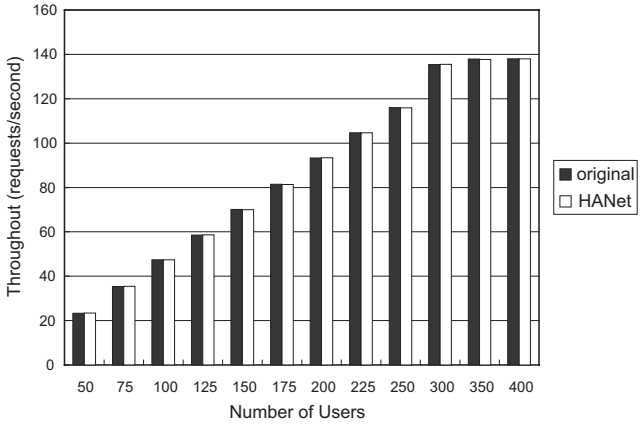

Fig. 8. Throughput of the HANet server with public interface.

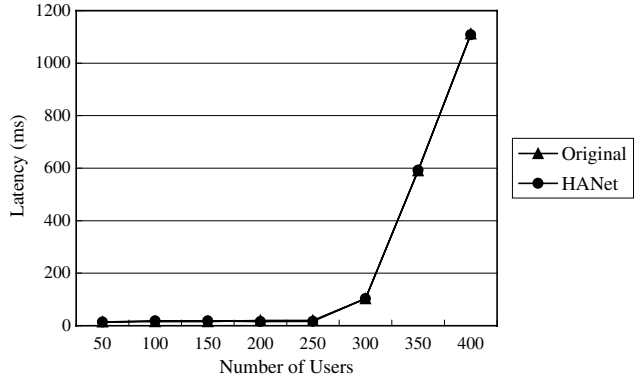

Fig. 9. User-perceived latency of the HANet server with public interface.

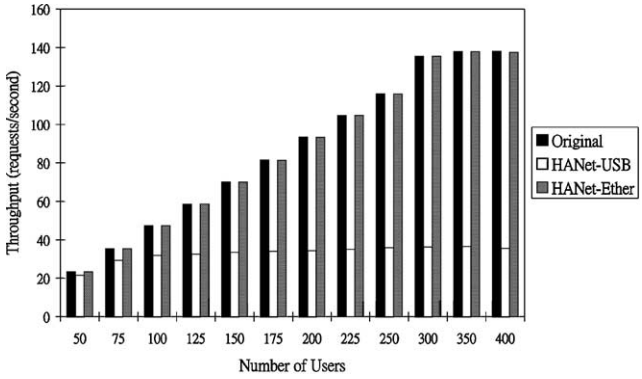

Fig. 10. Throughput of the HANet server through private interface.

USB link is clearly higher than the other cases. And, packet transmission through the private Ethernet link is only about 1% slower than the original case when the load of the server is heavy (e.g., 400 users). This indicates that there is very little overhead introduced by the packet transmission agent.

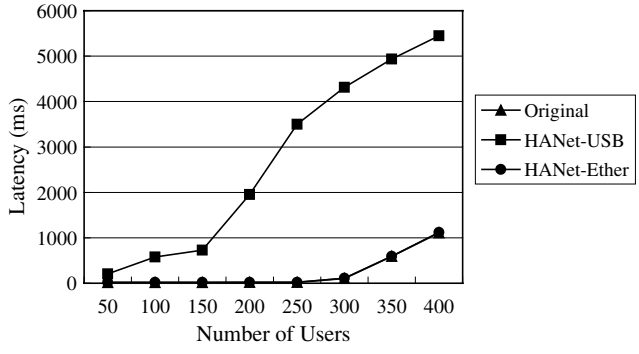

Fig. 11. User-perceived latency of the HANet server through private interface.

5. Discussion

In this section, we address on the issues of applying HANet on a multi-LAN system. We also provide discussions on the bandwidth of the inter-server private channels. Finally, we discuss the support of multiple active interfaces in the HANet framework.

5.1. Multiple LAN support

Currently, HANet is based on the assumption that the servers (including the siblings) are all located on the same LAN. However, it is possible to extend the current implementation and then apply it on a system that across multiple LANs. The goal of putting servers on different LANs is that if the LAN in which the server is located fails to work, the packet transmission job can still be achieved by the sibling computer located on another LAN. This is useful in an organization or a building, where servers may be located on different LANs but are in close proximity.

Taking over the packet transmission job from another LAN raises a problem. That is, packets from the client can not reach the sibling because they will follow the original routing path, and hence be routed to the failed LAN. To solve this problem, HANet must be extended to include the help of the edge router that corresponds to the sibling computer. Fig. 12 illustrates this idea. After determining that LAN A has failed, the server notifies the sibling, which in turn notifies router B. Then, router B sends routing messages to its neighboring routers to update their routing tables. For example, if Routing Information Protocol (Malkin, 1994) is used, router B can send a routing update message (server-subnet, router B, 1) to each of its neighboring routers. The message specifies that the destination subnet address is the subnet of the server, the next hop is router B, and the distance between the server subnet and router B is 1 (i.e., the minimum distance). If Border Gateway Protocol (Rekhter and Li, 1995) is used, router B can send link state update messages with the similar settings in

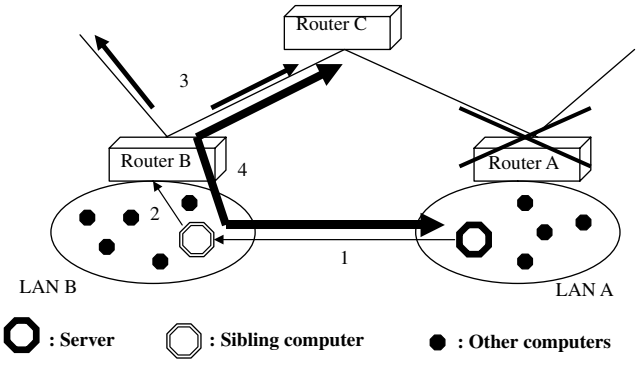

Fig. 12. HANet for a Multi-LAN system.

the AS_PATH attribute and the NEXT_HOP attribute. As a result, packets to the server will be routed to LAN B first, and then be forwarded to the server by the sibling computer.

In our future research, we will investigate more implementation issues for the multiple LAN support, and integrate the support into HANet.

5.2. Private channel bandwidth

In the current prototype implementation, we use RS232, USB, and Ethernet links as the inter-server private channels. However, other kinds of channels can also be used. For example, high-speed inter-server links such as Myrinet (Myricom, 2003) and high-throughput wireless technologies such as 802.11a and UWB are good candidates for private links.

In general, high-speed links are preferred because they provide better performance. With the rapid development of network technology, various types of high speed links have emerged. For example, a Myrinet link already has a 2 Gbps bandwidth in both directions. Both the specifications of USB 2.0 and UWB (i.e., IEEE 802.15.3a) have defined a 480 Mbps bandwidth, and the data rate of an IEEE 1394 device can reach 400 Mbps. By implementing the corresponding drivers that provide the net_device interface, these links can be integrated into our framework easily.

Even when such links are not available on the server system of the service provider, low bandwidth links such as RS232 can still be used. However, under this condition, a server process may decide not to accept any new requests once all of its public interfaces have failed. It can perform a *clean shutdown* (i.e., process all of the pending requests, sending all the results back, and then shutdown). New requests can be handled by other server processes on the other servers. The clean shutdown prevents the loss of the on-line requests, which is beneficial for transaction-based requests (e.g., requests involving database transactions) or dynamic-object requests (e.g., CGI requests). Recovering these requests correctly requires a large effort (Luo and Yang, 2001).

5.3. Multiple active interfaces

In our current implementation, at most one interface is active at any given time. Therefore, the server will not try to send packets simultaneously via multiple interfaces. However, a server does have the ability to utilize multiple interfaces simultaneously for packet transmission and reception. And, the Linux bonding driver also has a round-robin mode that allows the packets to be transmitted through multiple interfaces in a round-robin manner. We did not add this capability into our current implementation due to the following two reasons. First, transmitting packets simultaneously on multiple interfaces requires special support (i.e., link aggregation) from the layer-2 switches (Mehaffey, 2002), and not all of the switches support this functionality. Second, supporting various types of I/O links and multiple active interfaces at the same time requires much more implementation effort. The original bonding driver can easily support multiple active interfaces since its slaves are all of the same type (i.e., Ethernet). As a result, there is no problem for a bonding device to reflect the attributes of its slaves since the attributes of all the slaves are the same. However, different types of I/O links have different attributes. For ease of implementation, we have the bonding device reflect the attributes of its current active slave. Once the active interface changes, the attributes of the bonding device are reconfigured to reflect those of the new active slave. This leads to the support of only one active interface at a time in our current implementation.

To support multiple active interfaces in our framework, a bonding device may need to reflect the attributes of all the available slaves. For example, the hard_header_len field of the net_device structure should be set to the maximum length of all the hardware headers. And, there should be a universal hard_header( ) function that can build the hardware header of a given packet according to the type of the link on which the packet will be transmitted. Furthermore, some additional issues must be addressed. For example, we should develop a policy to determine whether or not to reroute part of the traffic to the sibling if some public interfaces of the server have failed. In the future, we will research the issues of extending our framework to support multiple active interfaces.

6. Related work

Previous work on improving server availability can be divided into several categories: round-robin DNS, Autonomic Computing, software state redundancy, and device redundancy. In the following, we describe them in detail.

Round-robin DNS and DNS aliasing (Brisco, 1995; Garland et al., 1995; McGrath et al., 1995) are used to dispatch user requests to one of multiple redundant servers. Although these approaches increase service availability, the requests being processed will get lost if the corresponding server fails. The lost requests must be re-issued by the user.

Autonomic Computing (Kephart and Chess, 2003) was proposed by IBM, which enables systems to manage themselves according to the administrator's goals. The self-managing means self-configuring, self-healing, self-protecting, and self-optimizing. Especially, the self-healing techniques automatically detect, diagnose, and repair software and hardware problems. Some efforts

related to the self-healing are SRIRAM (Verma et al., 2003) which is a method that facilitates instantiating mirroring and replication of services in a network of servers, K42 (Appavoo et al., 2003) which allows software codes including system monitoring and diagnosis functions to be inserted and removed dynamically without shutting down the running system, and Dynamic CPU Sparing (Jann et al., 2003) that detects failures of a CPU and replaces it with a spare one.

Recovery-Oriented Computing (Patterson et al., 2002) proposed by U.C. Berkeley and Stanford University is an effort related to autonomic computing. It proposes new techniques to deal with hardware faults, software bugs, and operator errors. These techniques include Pinpoint (Chen et al., 2002), which finds the root cause of a system failure in an efficient way; System Undo (Brown and Patterson, 2003), which can perform system recovery from operator errors; and Recursive Restart (Candea et al., 2002), which reduces the service downtime. In addition, they also proposed on-line fault injection and system diagnosis to improve the robustness of the system. The proposed techniques can be integrated with HANet to improve the system availability further.

Software state redundancy is another technique to increase service availability. Process pair (Gray and Siewiorek, 1991) duplicates and synchronizes software states between two servers. If the active server process fails, another backup process takes over its service. However, synchronizing states between two servers has large overhead. Some researchers (Aghdaie and Tamir, 2001; Luo and Yang, 2001; Yang and Luo, 2000) addressed this problem and proposed methods to reduce the synchronization overhead. Although these techniques are transparent to clients, modification to server applications is required. FT-TCP (Alvisi et al., 2001; Zagorodnov et al., 2003) places codes in the Linux kernel to record the I/O of the server application, including packets and some system call return values. If the server fails, it re-produces the server state by running a new copy of the server application from the beginning and feeding it with the logged I/O requests. That is, it replays the process before the server crashes. The advantage of this approach is that it does not need to modify the server applications. However, the replaying process may take a long time.

Device redundancy uses extra copies of hardware devices to increase the system availability. The devices include CPUs, memories, disks, network adapters, etc. If a device fails, the system software can operate on another copy of the device. This approach is a technique fundamental to many other ones. For example, it is the basis of the network bonding driver (Davis, 2003), which provides a fault-tolerant Ethernet network by grouping multiple Ethernet cards into a single Ethernet interface. The limitation of the bonding driver is that it only supports Ethernet cards, but does not support point-to-point communication links such as RS232, parallel, and USB links.

Our work is unique in that it allows network packets to be transmitted and received through all types of communication channels, and it enables a sibling computer to take over the packet transmission job when the public network of the server fails. It requires no modifications to server applications and client systems. Moreover, it is efficient since it eliminates the synchronization overheads and the replaying process.

7. Conclusion

Network failure is one of the major causes of system faults. In this paper, we propose HANet, a framework that masks network failures and hence improves the reliability of network services. It provides a uniform communication interface so that network packets can be transmitted on different kinds of links such as RS232, USB, Ethernet, etc. In addition, it allows a sibling computer to take over the packet transmission job from a server while the public network of the latter is broken. The techniques are transparent to both client-side systems and server-side applications.

In contrast to existing approaches, HANet does not lose any requests. And, the time-consuming replaying process is not needed. In addition, HANet is extensible in the following ways. First, it does not put any constraints on the type of the private interface. The private interface can be an USB link, a Myrinet interface, an IEEE 1394 link, a parallel interface, an Ethernet card, or even a wireless interface such as 802.11 or UWB. Second, new fault detection methods can easily be integrated into the framework.

According to the experimental results, there is no visible throughput degradation and the user-perceived latency can also be neglected as long as the speed of the private interface is not slower than the public one. This indicates that HANet is very efficient and suitable for commercial network services.

References

Aghdaie, N., Tamir, Y., 2001. Client-transparent fault-tolerant web service. In: Proceedings of the IEEE International Conference on Performance, Computing, and Communications, pp. 209–216.

Alvisi, L., Bressoud, T.C., El-Khashab, A., Marzullo, K., Zagorodnov, D., 2001. Wrapping server-side TCP to mask connection failures. In: Proceedings of INFOCOM 2001, pp. 329–337.

Appavoo, J., Hui, K., Soules, C.A.N., Wisniewski, R.W., Silva, D.M.D., Krieger, O., Auslander, M.A., Edelsohn, D.J., Gamsa, B., Ganger, G.R., McKenney, P., Ostrowski, M., Rosenburg, B., Stumm, M., Xenidis, J., 2003. Enabling autonomic behavior in systems software with hot swapping. IBM Systems Journal 42 (1), 60–76.

Barford, P., Crovella, M.E., 1998. Generating representative web workloads for network and server performance evaluation. In: Proceedings of the ACM SIGMETRICS '98, pp. 151–160.

Brisco, T., 1995. DNS support for load balancing. IETF RFC 1794.

Brown, A., Patterson, D.A., 2003. Undo for operators: building an undoable e-mail store. In: Proceedings of the 2003 USENIX Annual Technical Conference, pp. 1–14.

Brownell, D., 2002. The USB Host-to-Host Link (USBnet) Driver. Available at <http://lxr.linux.no/source/drivers/usb/usbnet.c>.

Candea, G., Cutler, J., Fox, A., Doshi, R., Garg, P., Gowda, R., 2002. Reducing recovery time in a small recursively restartable system. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN-2002), pp. 605–614.

Chen, M., Kiciman, E., Fratkin, E., Brewer, E., Fox, A., 2002. Pinpoint: Problem determination in large, dynamic, internet services. In: Proceedings of the International Conference on Dependable Systems and Networks (IPDS Track), pp. 595–604.

Cristian, L., 1991. Understanding fault-tolerant distributed systems. Communications of the ACM 34 (2), 57–78.

Davis, T., 2003. Linux channel bonding. Available at <http://www.sourceforge.net/projects/bonding/usr/src/linux/Documentation/networking/bonding.txt>.

Engler, D., Chelf, B., Chou, A., Hallem, S., 2000. Checking system rules using system-specific programmer-written compiler extensions. In: Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-2000).

Garland, M., Grassia, S., Monroe, R., Puri, S., 1995. Implementing distributed server groups for the world wide web. Technical Report CMU-CS-95-144, School of Computer Science, Garnegie Mellon University.

Gray, J., Siewiorek, D.P., 1991. High-availability computer systems. IEEE computer 24 (9), 39–48.

Horman, S., 2000. Creating linux web farms—linux high availability and scalability. Available at <http://www.vergenet.net/linux/has/html/has.html>.

Intel Corporation, 2003. Intel Networking Technology—Load Balancing. Available at <http://www.intel.com/network/connectivity/resources/technologies/load_ balancing.htm>.

Jann, J., Browning, L.M., Burugula, R.S., 2003. Dynamic reconfiguration: basic building blocks for autonomic computing on IBM pSeries servers. IBM Systems Journal 42 (1), 29–37.

Kephart, J.O., Chess, D.M., 2003. The vision of autonomic computing. Computer Journal 36 (1), 41–50.

Luo, M.Y., Yang, C.S., 2001. Constructing zero-loss web services. In: Proceedings of the IEEE INFOCOM 2001, vol. 3, pp. 1781–1790.

Mackerras, P., 2004. The Paul's PPP Package (PPP) Version 2.4.1. Available at <http://www.samba.org/ppp/index.html>.

Malkin, G., 1994. RIP Version 2—Carrying Additional Information. IETF RFC 1723. Available at <http://www.ietf.org/rfc/rfc1723.txt>.

McGrath, R., Kwan, T., Reed, D., 1995. NCSA's world wide web server: design and performance. IEEE Computer 28 (11), 68–74.

Mehaffey, J., 2002. Highly available networking. embedded linux journal, issue 7. Available at <http://www.linuxjournal.com/article.php?sid=5524>.

Milz, H., 1998. Linux high availability HOWTO. Available at <http://www.ibiblio.org/pub/Linux/ALPHA/linux-ha/High-Availability-HOWTO.html>.

Myricom Inc., 2003. Myricom—Creators of Myrinet. Available at <http://www.myri.com>.

Oppenheimer, D., Ganapathi, A., Patterson, D.A., 2003. Why do internet services fail, and what can be done about it? In: Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03). Available at <http://roc.cs.berkeley.edu/papers/usits03.pdf>.

Patterson, D.A., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J., Enriquez, P., Fox, A., Kiciman, E., Merzbacher, M., Oppenheimer, D., Sastry, N., Tetzlaff, W., Traupman, J., Treuhaft, N., 2002. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. UC Berkeley Computer Science Technical Report UCB//CSD-02-1175.

Patterson, D.A., Chen, P., Gibson, G., Katz, R.H., 1989. Introduction to redundant arrays of inexpensive disks (RAID). In: Digest of Papers for 34th IEEE Computer Society International Conference (COMPCON Spring '89), pp. 112–117.

Performance Technologies Inc., 2001. The Effects of Network Downtime on Profits and Productivity—A White Paper Analysis on the Importance of Non-stop Networking. White Paper. Available at <http://whitepapers.informationweek.com/detail/RES/991044232_762.html>.

Rekhter, Y., Li, T., 1995. A Border Gateway Protocol 4 (BGP-4). IETF RFC 1771. Available at <http://www.ietf.org/rfc/rfc1771.txt>.

Rubini, A., 2000. Making System Calls from Kernel Space. Linux Magazine, Nov. 2000. Available at <http://www.linux-mag.com/2000-11/gear_01.html>.

Scyld Computing Corporation, 2003. Understanding MII Transceiver Status Info. Available at <http://www.scyld.com/diag/mii-status.html>.

Snoeren, A.C., Andersen, D.G., Balakrishnan, H., 2001. Fine-grained failover using connection migration. In: Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems, pp. 221–232.

Stevens, W.R., 1994. TCP/IP Illustrated, Volume 1: The Protocols. Addison Wesley Professional, ISBN: 0-201-63346-9.

Verma, D.C., Sahu, S., Calo, S., Shaikh, A., Chang, I., Acharya, A., 2003. SRIRAM: a scalable resilient autonomic mesh. IBM Systems Journal 42 (1), 19–28.

Yang, C.S., Luo, M.Y., 2000. Realizing fault resilience in web-server cluster. In: Proceedings of the IEEE/ACM Supercomputing Conference 2000 (SC2000).

Zagorodnov, D., Marzullo, K., Alvisi, L., Bressoud, T., 2003. Engineering fault-tolerant TCP/IP services using FT-TCP. In: Proceedings of the IEEE Dependable Computing and Communications Symposium, pp. 393–402.