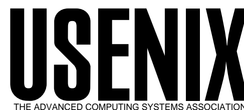
USENIX Association

Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference

San Antonio, Texas, USA
June 9-14, 2003



© 2003 by The USENIX Association All Rights Reserved For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

Console over Ethernet

Michael Kistler, Eric Van Hensbergen, and Freeman Rawson
IBM Austin Research Laboratory

Abstract

While console support is one of the most mundane portions of an operating system, console access is critical, especially during system debugging and for some low-level system-administration operations. As a result, densely packed servers and server appliances have either keyboard/video/mouse (KVM) cabling and KVM switches or serial ports, cabling, and concentrators. Further increases in the density of server appliances and blades require eliminating these items. We did so in the design of a prototype dense-server-blade system that includes none of the standard external ports for console devices. Since the standard means for console accesss, KVM or serial, were not possible, we developed low-level software to redirect console activity to the Ethernet interface. This paper describes the console support over the network interface that we developed for both our LinuxBIOS-based boot-time firmware and the Linux operating system. We refer to this code as *console over Ethernet* or *etherconsole*. We found it invaluable in debugging and evaluating our prototype server blades. We also describe ways of extending our work to make it more transparent to existing firmware and operating systems.

1 Introduction

Console support has been a feature of operating systems since their inception, and it is perhaps one of their most basic and mundane components. Modern computer systems make relatively little use of the console during normal operation and, for reasons of cost, space and simplicity are often run without dedicated terminal equipment connected to the console interface. This is particularly the case for servers packaged as server appliances, where a *server appliance* is defined to be a computer system specialized to run a single server application, such as a web server or firewall. However, during system debug and for certain types of low-level system-administration tasks, console support is essential.

In this paper we describe several variants of an implementation of Linux console support that use the Ethernet connection as the input/output device. We incorporated versions of this console support into both LinuxBIOS [18] and Linux to provide the console support for a prototype dense-server-blade system that includes none of the standard external ports for console devices. We call our console support *console over Ethernet* or *etherconsole*. We found this code to be invaluable for both the hardware and software bring-up and ongoing systems management of our prototype. As we used our implementation, we refined and extended its usefulness, but we also identified some areas where further enhancements are clearly desirable.

Currently, there are some who believe that consoles are an anachronism, a hold-over from earlier days, and, clearly, there are embedded systems environments that have no console support at all. However, we believe that for servers, especially during certain portions of their life-cycles, console function is critical. As noted above, we needed console support during our debugging efforts, and we would note that even embedded systems developers generally use some type of temporary console support, usually some form of console over a serial line, during the debugging of their code. For servers, given the complexity and flexibility of their configurations and their tendency to have a repair-or-replace rather than a pure replacement strategy for dealing with failures, there is a need for console support beyond the development phase to set low-level configuration values and capture system message output. In the future, servers may become more like embedded systems and no longer require console support except during development, but that does not seem to be the current situation.

The following section provides some additional motivational background for our work as well as describing the constraints of our environment. Section 3 describes other console technologies and previous, related work. In Section 4 we give a detailed description of the design and implementation of etherconsole. Section 5 provides the results of our experimental evaluation of etherconsole, and Section 6 indicates a number of ways in which our work can be enhanced. Section 7 concludes the paper.

2 Motivation and Intended Environment

Our work on etherconsole is in support of two larger projects at the IBM Austin Research Laboratory, the MetaServer and Super-Dense Server projects [10]. The goal of the MetaServer project is to develop new approaches to systems management for clusters of server appliances. A MetaServer cluster consists of a collection of compute nodes, each of which contains a pro-

cessor and memory but no local disk, network-attached-storage nodes, which are used to store application data, and a MetaServer system that provides a single point of management and administration for the cluster. These components are connected by a high speed, switched network infrastructure, and the cluster connects to an external network through one or more gateways that hide its internal organization. The compute nodes share a single operating system kernel image and root file system supplied to them over NFS from the MetaServer system. The operating system environment of the compute nodes is based on the Red Hat 7.1 Linux distribution and the Linux 2.4.17 kernel, with key modifications to support diskless operation. We call this environment the *Linux Diskless Server Architecture* (Linux-DSA).

The Super-Dense Server (SDS) project is exploring ways of building servers to reduce their power consumption and heat production as well as increase their density. It uses a number of boards, or *server blades*, plugged into a chassis or backplane where each server blade is a single-board computer with a processor, memory, an Ethernet connection, an interface to the I2C system management bus, and supporting logic. The server blades were intentionally designed with a very limited amount of I/O to reduce their power consumption and heat production and to increase the overall density of the prototype. In particular, our server blades do not have local disks and do not include any of the standard external ports for console devices. Along with the server blades, other specialized blades are also plugged into the chassis including an Ethernet switch and a specialized single-board computer that acts as a system management controller. Each server blade runs its own copy of a common operating system image and server-application set, and the server blades in a chassis are typically configured into a cluster. We developed our own boot-time firmware based on LinuxBIOS [18] and use the Linux-DSA operating system and runtime environment from the MetaServer project. Recently, a number of commercial dense-server and server-blade products have appeared in the marketplace, including ones from Amphus [1], Compaq [6] (prior to its merger with Hewlett-Packard), Hewlett-Packard [12], IBM [14], and Dell [7]. All of these products share concepts similar to those found in the SDS work, but all still include either standard or proprietary ports for console devices and require extra cabling and switches for console access.

The goal of etherconsole is to attach a console or console emulator to a server over a standard Ethernet connection. In so doing, it provides a complete replacement for all console alternatives for both the firmware and the operating system. Since a server must have a network connection (typically Ethernet in commercial data center environments), reusing it for console attachment adds no additional hardware or cabling for console support. The network interfaces, cables (or backplane wiring) and switches are all required and would be present no matter how the console is attached. Given our desire to maximize the packaging density of our server blades, which led to our decision not to include the ports required to support standard console interfaces, we were forced to develop an alternative mechanism for console support. Our only options were the Ethernet connection and the I2C system management bus. We rejected the I2C bus for a number of reasons. First, we wanted to reserve it for hardware control and certain low-level monitoring functions that we needed for our other research work. Second, although the latest versions of the bus have up to 3.4 megabits per second of bandwidth [23] in high-speed mode, our implementation runs it at the bus's original bandwidth of 100 kilobits per second. Third, experimentally, we found the I2C bus to be somewhat unreliable in our implementation. Once we eliminated the I2C bus from consideration, we were forced to implement the console support for our prototype over the Ethernet interface.

Our network environment also influenced our design. We made several important assumptions about the Ethernet and the systems on it, all of which are true in our environment and many other clusters and data centers. First, we assumed the use of high-speed, highly reliable links connecting the individual server systems to an Ethernet switch. Second, we assumed that the switch is capable of Ethernet data-link-layer [22] switching at line speed, which means that it does not experience congestion problems or drop packets. Therefore, the network itself is highly reliable, and only the system endpoints create transmission or reception hazards. Third, since all of the systems, including the management system that runs the console monitor or console emulator program, are on the same physical subnet, we assumed that no routing is needed. These assumptions imply that reasonably reliable, high-performance communication is possible using only link-layer Ethernet protocols and hardware addressing.

Since we were developing a prototype hardware platform, we had to do system debug and bring-up. We initially developed two forms of console over Ethernet, a link-layer, output-only one, and a full console implementation using the TCP/IP stack. Recently, we consolidated them into a single link-layer implementation that can function both as a simple message transmitter and as a full console supporting both input and output. We needed the link-layer, output-only function to allow us to track the progress (or lack of it) of early system initialization and the full-function support as a complete replacement for the standard console function. Full console support is required, for example, when some type

of low-level configuration is needed and the system must run in single-user mode. Although our systems have network access in single-user mode, standard daemon processes such as telnet servers cannot run, so the console is the only way of providing the input that the machine requires.

3 Background and Related Work

Others have recognized the cost and complexity issues with providing console access to large clusters of servers and have developed a variety of solutions to address these problems. In this section, we describe both the alternative approaches to console support used in dense servers and blades and some work that is a pre-cursor to our own solution to the problem.

3.1 Alternative Approaches to Console Support

We have identified five distinct approaches to providing console support that are in common use in commercial data centers. The first approach employs a Keyboard/Video/Mouse (KVM) switch which connects a single keyboard, mouse, and display to multiple servers, typically 4, 8, 16, or 32 machines. To interact with the console of a particular server, the user selects the server using a mechanical or programmable selector on the KVM switch, and the keyboard, mouse, and display are connected by the switch to that server's console ports. A KVM switch that supports N servers requires $3N + 3$ cables – 3 between each server and the KVM switch and 3 from the KVM switch to the shared keyboard, mouse, and display. In addition to standard KVM switches, there are a number of KVM extenders as well as KVM-to-Ethernet concentrators that have appeared on the market. One advantage of KVM switches and extenders is that they are capable of presenting a graphical user interface. However, in comparison to console over Ethernet, all of these products require additional hardware components, cabling, and space.

A common alternative to KVM switches is the use of a serial console which redirects the console traffic over a serial port connected to a serial concentrator. Unlike KVM switches but like etherconsole, serial console does not support a graphical user interface. Serial consoles and concentrators offer the advantage of reducing the number of cables for N systems to $N + 1$. Some newer serial concentrators also offer conversion to an Ethernet uplink. However, they still add to the cost of the installation, take up space, and require additional cabling. Given that the console is a vital but rarely used function in a production installation, this seems like an excessive price to pay for supporting it.

Recent developments in hardware, firmware, and operating system support now make it possible to use a Universal Serial Bus (USB) [27] port for the console. The current support in Linux for a full console over USB is similar to one of our implementations in that it is based on the pre-existing serial console support in Linux and is not graphical in nature. The USB serial console supports communication with a terminal emulator over a USB-connected serial port. Like the serial console, the USB serial console requires at least $N + 1$ cables for N systems as well as the additional serial concentration hardware and a USB-to-DB9-serial adaptor for every system. In addition, given the late initialization of the USB support and infrastructure in Linux, the USB serial console offers less support for system debug than the standard serial console. Moreover, USB cables are limited to 5 meters in length, complicating the cabling problem. Although there are also KVM switches that support the USB mouse and keyboard interfaces, they still use the standard video interface and cable and have properties that are very similar to standard KVM switching schemes.

IBM has recently introduced a new form of KVM switching, C2T Interconnect cable chaining and the NetBAY Advanced Connectivity Technology (ACT) [15]. Rather than using a star topology for KVM switching, ACT daisy chains the server systems together. C2T is one of the two chaining techniques used; it reduces the cabling and conversion hardware at the cost of adding a specialized port to every server in the chain. The daisy chain ends in either a special KVM switch or KVM-to-Ethernet converter. Although ACT reduces the amount of cabling and the number of switches or converters required, it still relies on cabling, connectors, and supporting equipment that reduce density and increase cost.

Finally, there are systems that provide a form of integrated, hardware-based console. A good example of such an implementation is the Embedded Remote Access (ERA) port [9] found on some recent server systems from Dell. The ERA port provides console access to firmware functions as well as hardware monitoring information using an independent microprocessor, memory, and set of firmware, and a dedicated network port, presumably an Ethernet port. Based on the published information, it appears that the ERA collects information over the I2C bus. It serves only as a hardware and firmware console, but not as the console for the operating system. It may be possible to modify the operating system to interact with the ERA over the I2C. This approach is probably adequate for a single system, but the characteristics of the I2C bus may well make it unsuitable for clusters.

In summary, all of these approaches require additional cabling or hardware, which decreases reliability and adds complexity to the installation and management of a cluster of server blades or appliances. In ad-

dition, they increase costs, in terms of equipment and floor space requirements, because they require additional hardware devices such as KVM switches or serial concentrators. Finally, all of these console implementations decrease the density of the server systems themselves since they require parts on the server boards that can be eliminated if console access is provided through the network interface.

3.2 Other Software Technologies

Portions of our implementation of console over Ethernet were based on the recently introduced Linux netconsole feature [17, 25]. After we describe our design and implementation, we will compare our work with netconsole in Section 5.4. Here we merely review the key features of netconsole. As implemented in the openly released patch to Linux 2.4, netconsole is a module that is dynamically loaded into a running kernel image. It is designed to support the transmission of console messages, especially in emergency situations, but it is not intended as a full console replacement. The netconsole implementation uses UDP and either broadcasts messages or transmits them to a particular receiving host, whose IP address is specified as a parameter to the netconsole kernel module when it is loaded. Netconsole registers as a console to allow it to receive the messages being written by the kernel. To avoid problems when the system is low on memory, netconsole maintains a dedicated pool of Linux `sk_buffs` [3], the network buffer structures used by the kernel. It also relies on special routines added to the network device drivers that send out frames waiting in the transmission queue without enabling interrupts, polling for the device to become available if necessary. Netconsole uses a syslog server running on another system to capture the messages that it sends.

In addition to netconsole, there have been other console-over-a-network implementations in recent years. The Fairisle network console [2] is similar to etherconsole in that it provides remote console support over a network, in this case, an ATM network. Based on the readily available documentation, it is difficult to make a detailed comparison of etherconsole with the Fairisle network console. In particular, it is not clear whether the Fairisle network console is a complete replacement for all local console function.

One traditional method of accessing systems is through terminal-oriented protocols such as telnet [5] and SSH [30]. Although these protocols provide tty access to the system and most of the needed function once the operating system and all of the required infrastructure are running, unlike etherconsole, they do not offer true console access and are not available until after kernel and mult-user system initialization has been completed.

Virtual Network Computing (VNC) [26] is a very commonly used way of gaining access to a graphical user interface (GUI) over the network. It is often used, for example, to provide an X Windows session with a remote Linux host on a Microsoft Windows-based system. Like telnet and SSH, VNC depends on a broad range of underlying system services and thus cannot be used during system initialization. VNC also requires more network bandwidth since it must transfer graphical screen contents to the remote system. Also unlike etherconsole, which is intended to work with server programs that catch messages from and interact with large numbers of server blades, VNC is intended to allow remote, one-to-one access to the graphical user interfaces of a relatively small number of host systems. Another commercially available remote GUI-access protocol is Citrix's Independent Computing Architecture (ICA) [4], which is used with Microsoft Windows-based hosts. Although the details of ICA are very different from those of VNC, they are conceptually quite similar [21].

Finally, although there is a SourceForge project entitled "Network Console Project" [20], as of this writing (early 2003), it appears to be dormant.

4 Design and Implementation

We developed three different console over Ethernet implementations, and each implementation contributed features that we found very useful in our work. The first implementation, which we call *etherconsole-T*, is a full replacement for the standard serial console. Etherconsole-T uses a kernel-level interface to the socket library to send and receive console messages on a TCP connection over the Ethernet instead of doing low-level device operations on the serial port as the serial console does. The second implementation, referred to in this paper as *coe*, is actually more primitive in terms of function and reuses some technology from the Linux netconsole implementation described in Section 3. Despite its simplicity, we found coe extremely useful in our work. The final implementation, called *etherconsole*, merges the best features of the two previous implementations. This section describes, in turn, each design and implementation along with the console emulator code that we developed to work with it.

4.1 TCP/IP-Based Implementation

Our initial implementation of console over Ethernet is a straightforward implementation of all of the console and tty function of the serial console using a TCP connection rather than a serial line as the transport. Structurally, as well as functionally, etherconsole-T is very similar to the serial tty and console code in the standard Linux distribution with the major difference being the way that the driver code communicates with the console emulation

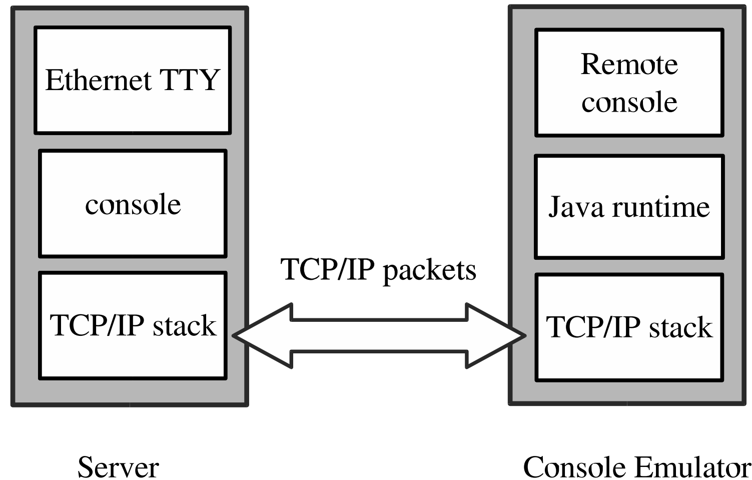


Figure 1: etherconsole-T: a TCP/IP-based console over Ethernet

program running on another system. Since etherconsole-T predated our other work with link-layer communication protocols, we chose to implement it using transport-layer protocols rather than link-layer networking. Figure 1 shows the overall structure of etherconsole-T and its console emulator program.

Since etherconsole-T is an Ethernet tty, it must fit into Linux's overall tty structure including the kernel line discipline code. It is structured as a character device with a major and minor number, relying on either `/dev` entries or the devfs [11] file system to provide the required special files. The driver supports multiple minor numbers and, therefore, multiple Ethernet ttys. The `console=` boot parameter is used to specify the name of the etherconsole-T device and the IP address of the system running the console emulator program. Since etherconsole-T depends on a fully configured TCP/IP stack, console initialization must be delayed until after TCP/IP is initialized and configured. If etherconsole-T is to be used for diagnosing early boot problems, TCP/IP configuration must be done as part of kernel initialization, using either DHCP or parameters passed to the kernel. Our server blades use DHCP to obtain the kernel boot image and configuration parameters and always perform TCP/IP configuration as part of kernel initialization. When etherconsole-T is initialized, it establishes two socket connections with the console emulator program on the remote host. One of the connections is used to accept input from the console emulator program while the other is used to write data to it. The etherconsole-T code does all of its communication with the console emulator using a kernel-level interface to the standard socket library; the programming technique it uses is very similar to the one used by the in-kernel Tux [24] web server. Since input from the console emulator arrives asynchronously, etherconsole-T uses a dedicated kernel thread to read it and pass it up through the tty code.

In addition to the code that works with the tty line discipline function in the Linux kernel, there is also a set of direct I/O routines for use by the kernel console function. These operate in much the same manner as the functions that interface with the tty code except that they are invoked through the Linux console structure.

We modified a terminal emulator and telnet program written in Java [16] to act as a console emulator for etherconsole-T. The result uses standard telnet protocol and terminal emulation techniques once communication is initiated, but unlike a standard telnet client, it listens for connections from a system running etherconsole-T, and when it accepts a pair of connections (one input and one output) from a server, it opens a separate window for the console session with the remote system.

Etherconsole-T adds two new files to the Linux kernel sources with a combined size of 1243 lines of code. In addition, there are 12 lines of code added to the tty and system initialization. Our changes to the Java telnet program to support console emulation are about 220 lines of Java.

4.2 Link–Layer Console Message Transmitter

Although etherconsole-T works well in the context of our Linux-DSA operating environment, it requires a full Linux TCP/IP stack and, thus, cannot be used in very low-level code such as boot-time firmware. Our need for console support to help debug our server blades led to our second implementation, coe. Figure 2 depicts the

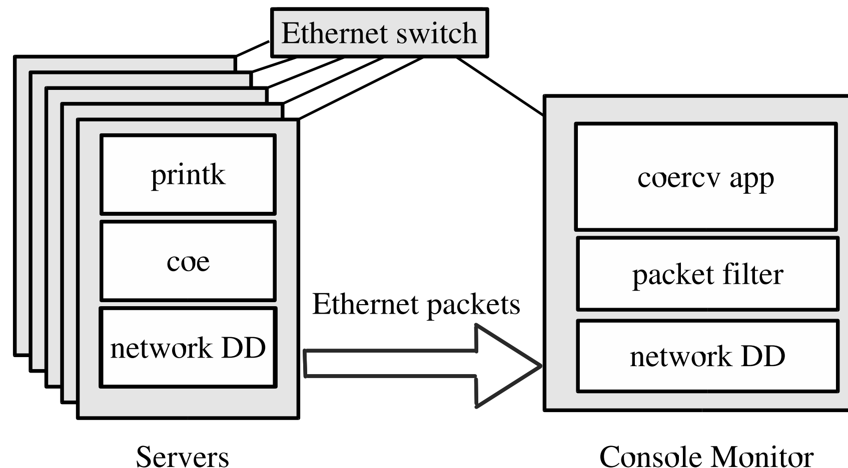


Figure 2: coe as deployed on prototype server blades

implementation of coe on our prototype server blades.

We began our work on coe by implementing some simple extensions to Etherboot [29] and LinuxBIOS which allowed redirection of print messages over the Ethernet device as opposed to a serial port. We then combined this code with portions of Red Hat's netconsole patch in order to provide a unified, link-layer, message transmitter for both our firmware and the Linux kernel.

Our implementation of coe uses link-layer networking only, transmitting raw Ethernet frames with a special frame type. This allows our console to begin operation much earlier in system initialization since it does not depend on IP configuration. Even when done at the kernel level, IP configuration occurs relatively late in initialization and relies on the proper exchange of messages between a kernel-level DHCP client and a DHCP server or the proper processing of kernel parameters. The special Ethernet frame type allows other systems to identify and process coe packets properly and ensures that these packets do not conflict with the existing traffic on the network. The use of link-layer networking also reduces the number of locks that must be obtained while formatting the message for transmission since the network stack's sequence number lock is no longer needed. To minimize the amount of configuration required and to eliminate the dependency on module parameters, we changed the code to broadcast Ethernet frames to the hardware broadcast address of `ff:ff:ff:ff:ff:ff`.

We retained from netconsole the use of a pool of reserved, pre-allocated network buffers that allows for the transmission of up to 32 console messages in out-of-memory situations in which the standard network buffer allocation fails. We also used the same device polling technique for message transmission over the network. Network hardware generally has some limitation on the size of its transmission queue. The coe implementation checks to determine if the transmission queue of the network device is stopped. If so, it calls a routine in the network device driver that enters the interrupt handler to force the processing of transmission interrupts, thus clearing the queue and allowing coe to transmit a packet. This scheme improves the reliability and timeliness of the transmissions.

To receive and display console messages sent by coe, we wrote a console monitor program, coercv, that runs on a separate system in the same physical subnetwork. It uses the packet socket support provided in the standard Linux kernel to send and receive raw network packets. Coercv creates a packet socket using the standard `socket` system call, specifying `PF_PACKET` as the protocol family and the coe Ethernet frame type as the protocol. This causes the Linux kernel to set up a packet filter [19] for the coe Ethernet frame type; the packet filter directs incoming frames with the specified frame type to coercv. The coercv program uses the `recvfrom` system call to receive the incoming messages. For sockets in the `PF_PACKET` protocol family, the kernel returns the hardware address of the source of the packet in the `sock_addr` structure passed on the `recvfrom` call. This allows coercv to either tag the messages that it prints with the Ethernet hardware address or filter the incoming messages and print only those whose source hardware address matches a specified value. We also added support to the standard Linux syslogd program, which logs messages, to capture and log ones being sent by coe. It uses a map of network

hardware addresses to tag the messages that it receives with the hostname of the server to allow the reader of the log file to determine what system originated each message.

Neither coe nor coercv contains much code or is very complex. Building on the then-current version of netconsole, coe adds 35 lines of code to the source file, of which 26 lines are preprocessor conditional compilation statements. The coercv program with optional filtration based on the sending system is slightly more than 100 lines of code. The changes to syslogd are approximately 75 lines of code.

4.3 Merged Implementation

Finally, our etherconsole implementation combines the features of the coe and etherconsole-T implementations. It offers both the low-level, output-only broadcast of messages from the system and a complete console and tty similar in function to etherconsole-T. Although it incorporates the kernel module packaging features, it is designed to be built statically into a kernel image, as we must do with LinuxBIOS, and this also allows it to be initialized earlier than would be possible if it were a kernel module. Once the network device driver and basic sockets support have been initialized, etherconsole registers as a console and can start broadcasting messages. The Linux kernel maintains a buffer of the most recently issued console messages and passes these to the console during the registration process. This generally allows etherconsole to process all the messages generated by the system during the boot process, even those issued before the network is available. The size of the kernel message buffer can be expanded to prevent the loss of early boot messages because of buffer wrapping, but this has not been necessary in our environment. Although etherconsole initialization is later than standard console initialization, it still precedes most system initialization processing and, therefore, supports error diagnosis for a large fraction of system initialization.

Etherconsole uses a link-layer protocol and, thus, it can communicate only with other machines on the same physical subnet. In its default configuration, it performs all of its output by setting the Ethernet MAC address to `ff:ff:ff:ff:ff:ff` to force the hardware to do a broadcast. However, it can be configured to send output to a particular target MAC address with the `console=` kernel boot parameter. The syntax of this boot parameter is

```
console=ttyNn,mac_address
```

where `ttyNn` is a character-device node in `/dev` or created by devfs [11] with a major and minor number indicating it is an etherconsole device and *mac_address* is an Ethernet MAC address in 6-byte, hexadecimal notation.

Packets are transmitted by queuing them directly on the transmission queue of the Ethernet driver and invoking a special routine in the device driver to send out available packets without enabling interrupts, polling for the device to become available if necessary. To handle input using link-layer protocols, etherconsole registers as the packet handler for its Ethernet frame type, so that it receives all incoming packets with that frame type. Etherconsole discriminates between packets that are console input and those that are broadcasts from other etherconsoles on the same subnet by checking the destination address in the Ethernet frame header. If it matches the hardware address of the local Ethernet interface, the packet is assumed to be console input. Otherwise, it is treated as a broadcast from another etherconsole and discarded.

Like etherconsole-T, etherconsole contains tty support as well, which allows us to use it as a full console replacement. To provide tty support, etherconsole requires the character-device node described above. Early in boot, the system uses the console entry points directly, but once the init thread opens `/dev/console`, subsequent console interactions use the tty entry points. Etherconsole provides all of the entry points needed by the tty code. However, many of the entry points do nothing more than return since they are not applicable to the underlying Ethernet device.

Our final etherconsole implementation supports coercv and the modified syslogd we developed for coe as well as a Java console emulator similar to that used with etherconsole-T.

Our link-layer etherconsole including both the code inherited from the original netconsole implementation and all of the function that we added is 470 lines of code.

5 Experience and Evaluation

This section describes our experience with our console-over-Ethernet implementations.

5.1 Usage

Since our server blades are prototypes, we wrote our own boot firmware based on LinuxBIOS [18] and etherboot [29], which allowed us to integrate etherconsole into both our firmware and our Linux kernels. Using etherconsole allowed us to debug and perform low-level systems management for server blades that do not have any standard external ports for console devices. We were able to track the progress of boot from power-on or reset through power-on self-test, firmware initialization, hardware set-up, and operating system load to the completion of kernel initialization. This proved invaluable

for fault isolation and problem determination. It also allowed us to monitor the concurrent initialization of multiple blades on a single display, making it easy to determine when the blades were all up following a cluster reboot. Moreover, it externalized kernel messages regarding problems encountered during normal operation. Our full console function allowed us to manage systems running in single-user mode, start and stop subsystems outside the context of normal init processing, and perform other types of management and debugging operations when normal telnet or SSH services were unavailable.

One usage difficulty that is not an issue in our environment, but might be in other situations, is the fact that there are a few commonly used commands such as `ifconfig` that manipulate the Ethernet interface and the network configuration. Since our systems are diskless and totally dependent on network communication, shutting down the Ethernet interface with an `ifconfig down` command effectively shuts down an entire node of our cluster. However, for disk-based environments, care must be taken to restrict the usage of commands that may unintentionally terminate console operation.

5.2 Performance and Reliability

Since console and tty support is not a performance path, we did not perform any formal performance studies. However, in our very extensive experience with our implementations, we did not observe any performance problems, in terms of either the speed of the console or the overhead imposed on the system. Also we found that we did not lose messages even when using the implementations that operate at the link layer. Even for a configuration where 8 blades are broadcasting boot-time messages simultaneously, the period of heaviest activity, the load imposed on a switched 100 Mbit Ethernet is insignificant. If one or more specific MAC addresses are used instead, the network load is even less. As mentioned in Section 2, our network environment is such that the network itself is reliable at the link layer, and the implementation techniques used in our link-layer code avoid the problems that can occur with network transmission on busy systems. We have not found any need for additional reliable delivery or flow control mechanisms. In passing, it is worth noting that we also use link-layer networking very successfully in our environment for remote disk access [28].

Having a second network interface and dedicating it to console support offers the advantages of clean traffic separation and the possibility of having a different network topology for console messages, but our experience indicates that these benefits are outweighed by the cost, complexity and additional space required for this design. Our implementation of console over Ethernet achieves more than adequate performance and reliability with very little impact on other network traffic using the single Ethernet interface on our blades. The paper by Felter, *et al*, [10] contains detailed performance studies of our server blades. All of these studies were conducted with one or more of our console-over-Ethernet implementations active on all the blades.

5.3 Security Considerations

One possible concern with console over Ethernet is security. Putting the system console on the network creates the possibility of break-ins and denial-of-service attacks. By using link-layer protocols, etherconsole hides the console from systems outside the physical subnet. Link-layer protocols do not use IP addresses and are not routable beyond the subnet. In addition, the hardware addresses required for interaction with the console are not exposed and have no meaning outside the subnet. Within the subnet, access to the console can be further restricted using VLANs [13]. A private VLAN can be created consisting of only those machines that require console access; machines in the physical subnet but not part of the VLAN will be unable to access consoles within the VLAN. A more sophisticated console emulation program can act as a form of firewall and bridge, receiving connections from the outside on another VLAN, doing the necessary protocol conversions, and forwarding legitimate messages to the etherconsoles over the private VLAN. This idea is explored further in Section 6.

Denial-of-service attacks are a special form of security exposure that have recently received considerable attention. One possible form of this type of attack is to send the console a very large number of messages or faulty command strings to handle. Processing them can consume a substantial amount of resource, slowing the system down significantly. A related form of attack is to send a command that somehow causes a very large amount of console output. As most software developers recognize, one way to slow down a program dramatically is to have it do a very large number of `printf` calls, and one way to slow down a kernel module in Linux is to fill it with `printk`s. In many cases, Linux protects against this form of attack by limiting message generation rates. For example, the rate limiting support in the TCP/IP protocol stack of the Linux 2.4 kernel detects and suppresses duplicate messages issued faster than some fixed rate. This decreases the load imposed by any console implementation, including etherconsole, and reduces the possibility of lost messages.

5.4 Comparison with Netconsole

We made three very significant extensions to netconsole. First, we designed etherconsole to be built into the ker-

nel and enabled at boot time, rather than being packaged as a kernel module that can be used only after the system is fully initialized. This allows etherconsole to start operation much sooner than netconsole, so that messages generated during system boot can be observed remotely. Second, etherconsole uses a link-layer protocol that requires only basic network hardware support and MAC-level addressing, whereas netconsole uses UDP sockets, requiring full IP configuration and addressing. Third, we added support for input by incorporating the tty support described above. In addition, etherconsole also derives from our very first implementation of the full console over Ethernet, which was a completely independent development that started before netconsole appeared.

6 Future Work

There are a number of areas in which our work can be extended, something that we hope to encourage by making our code available to the open source community.

6.1 Functional Extensions

Our implementation has a few functional deficiencies. In particular, it does not handle the `control-alt-delete` key sequence for rebooting nor does it process the `sys req` key used by some Linux kernel debugging features. Rather than relying on command line parameters, the DHCP [8] support in the firmware and kernel can be used to receive additional console configuration and control information such as the hardware address of a particular workstation running the console emulation program for this system.

An interesting and more challenging functional extension is to support a remote gdb debugger over the Ethernet. In principle, many of the same techniques used in the private gdb serial code should also work with an Ethernet interface. In particular, our code already uses polling to clear the transmission queue and operates with minimal overhead at the link layer. The major open question is how to handle the network buffers. The debugging environment is even more restrictive than the console, and we need to recycle a fixed pool of network buffers that is completely private to the gdb stub code to avoid unwanted interactions with the system being debugged.

6.2 Subnet Console Server and Reflector

A useful extension to our console emulator software would be to incorporate a telnet or SSH server that would allow console access outside the physical subnet. Such a program can accept telnet or SSH connections over TCP/IP from workstations outside the subnet, perform appropriate authorization and authentication checking, and then link these connections with the console of a particular system, doing the conversion between the TCP/IP and link-layer protocols.

6.3 Serial Port Emulation

Most server firmware and operating systems allow console interactions to be directed to one of the computer's serial ports. As discussed above, this is commonly used as the means of accessing the console of a server appliance or blade. For systems using x86-compatible processors, the serial port is an I/O port with an address defined in the system configuration. The firmware or operating system uses the OUTB instruction to send data and commands to the serial port and the INB instruction to receive data or read the port status. An alternative console over Ethernet design can intercept data and commands issued to the serial port and redirect these over the network to a console emulator running on another system. In this solution, no change is necessary to the operating system; it is simply configured to use the serial console. Figure 3 illustrates this concept. As indicated in the figure, communication can be at either the link layer using Ethernet frames or the transport layer using TCP or UDP.

A variety of mechanisms can be used to intercept read and write requests to a serial port. One approach is to add this feature to the firmware of the machine. In this case, the firmware is configured to trap INB and OUTB instructions written to a specific port and process them on its own. Since machines capable of DHCP and remote boot already have the basic mechanisms for IP connectivity built into their firmware, we can leverage this existing support to provide the transport for console communications. Another approach is to develop special serial port hardware that, instead of transmitting data to a physical port on the machine, transmits it to the network interface. Finally, a programmable network interface can pretend to be the serial device by owning the bus resources for it. It would then translate operations on the serial I/O port into the corresponding network interactions.

6.4 Frame Buffer Emulation

Some operating systems do not support character-based consoles and require the use of a graphical user interface instead. They are better supported with a frame buffer emulation approach. In this case, an area of memory is used to hold a dummy frame buffer, and the network interface sends characters from the frame buffer. Input occurs by transmitting characters and mouse movements to the system through the network interface and then passing them to the operating system by emulating normal keyboard and mouse input.

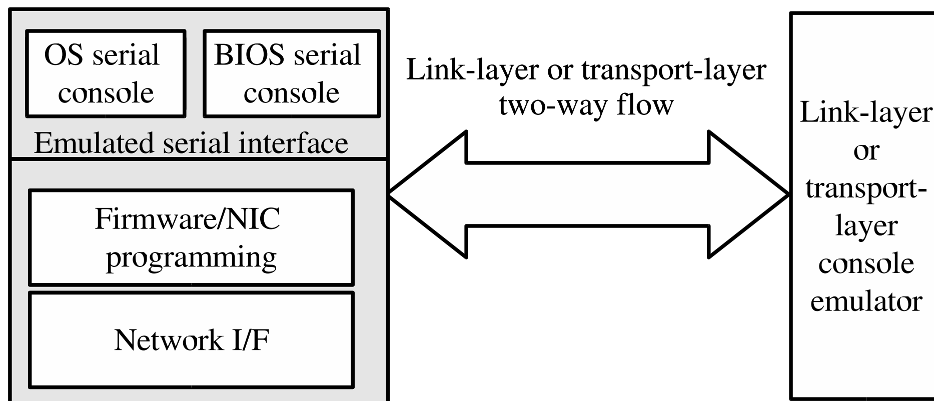


Figure 3: Serial device emulation

7 Conclusions

By redirecting console interactions over the network interface, console over Ethernet allows us to construct servers without standard external ports or cables and the associated infrastructure for console devices. This eliminates the cost, complexity, and fragility of console cabling and switches, while allowing for higher density packaging of server systems. As with the chassis that we used for our prototype server blades [10], even the Ethernet cables become unnecessary if the server is a blade that plugs into an appropriately wired backplane. Eliminating the separate network for console traffic simplifies the design and wiring of the backplane, reducing the cost of the chassis and the total cost of a blade server deployment.

Source Code Availability

Source code for our implementation is available from the IBM Linux Technology Center web site at `http://oss.software.ibm.com-/developerworks/opensource/linux`.

Acknowledgments

Wes Felter, Tom Keller, Elmootazbellah Elnozahy, Bruce Smith, Ram Rajamony, Karthick Rajamani, and Charles Lefurgy contributed very heavily to the SDS project. We would also like to thank the management of the IBM Austin Research Laboratory for their support of our work. Our shepherd, Chuck Cranor, and the anonymous referees provided many helpful comments that improved the quality of this paper. All trademarks are the property of their respective owners.

References

[1] Amphus, Inc. Virgo: A ManageSite-enabled, fully manufacturable, ultra-dense server design, 2001.

[2] R. Black. The Fairisle network console. http://www.cl.cam.ac.uk/Research/SRG-/bluebook/18/netcon/netcon.html.

[3] D. Bovet and M. Cesati. *Understanding the Linux Kernel, Second Edition*. O'Reilly and Associates, Inc., 2002.

[4] Citrix Corporation. Citrix independent computing architecture. http://www.citrix.com/press-/corpinfo/ica.asp, 2002.

[5] D. Comer and D. Stevens. *Internetworking with TCP/IP, Volume 3: Client–Server Programming and Applications, BSD Socket Edition*. Prentice Hall, 1993.

[6] Compaq, Inc. Proliant BL10e Server, January 2002.

[7] J. Delaney. Dell blade: Maximum power, minimum space. *PC Magazine*, February 2003.

[8] R. Droms and T. Lemon. *The DHCP Handbook: Understanding, Deploying, and Managing Automated Configuration Services*. Macmillan Technical Publishing, 1999.

[9] Y.-C. Fang, J. Jancic, A. Saify, and S. Zaiback. Using Dell embedded remote access to facilitate remote management of HPC clusters. *Dell PowerSolutions*, November 2002.

[10] W. Felter, T. Keller, M. Kistler, C. Lefurgy, K. Rajamani, R. Rajamony, F. Rawson, B. Smith, and E. VanHensbergen. On the performance and use of dense servers. To appear in the *IBM Journal of Research and Development*, 2003.

[11] R. Gooch. Linux devfs (device file system) FAQ. http://www.atnf.csiro.au/people/rgooch-/linux/docs/devfs.html, August 2002.

[12] Hewlett-Packard Company. HP bc1100, December 2001.

[13] IEEE. IEEE standards for local and metropolitan area networks: Virtual bridge local area networks, IEEE Standard 802.1Q-1998, 1998.

[14] International Business Machines Corporation. BladeCenter. http://www.ibm.com, September 2002.

[15] International Business Machines Corporation. The Decision Maker's Guide to Server Connectivity and Console Switching. http://www.ibm.com, July 2002.

[16] The Java telnet/ssh applet. http://javassh.org, 2002.

[17] M. Johnson. Red Hat, Inc.'s network console and crash dump facility. http://www.redhat.com-/support/wpapers/redhat/netdump, 2002.

[18] R. Minnich, J. Hendricks, and D. Webster. The Linux BIOS. In *The Fourth Annual Linux Showcase and Conference*, October 2000.

[19] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, pages 39–51, 1987.

[20] Network Console Project. http://sourceforge.net-/projects/netconsole, 2003.

[21] J. Nieh, J. Yang, and N. Novik. A comparison of thin–client computing architectures. Technical Report CUCS-022-00, Columbia University, November 2000.

[22] L. Peterson and B. Davie. *Computer Networks, Second Edition*. Morgan Kaufman, 2000.

[23] Philips Corporation. I2C-Bus. http://www.-semiconductors.philips.com/buses/i2c, 2003.

[24] Red Hat, Inc. TUX 2.1, 2001.

[25] Red Hat, Inc. Netconsole, 2002.

[26] T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1), January/February 1998.

[27] USB. http://www.usb.org, 2003.

[28] E. Van Hensbergen and F. Rawson. Revisiting link-layer storage networking. Technical Report RC22609, IBM Research, October 2002.

[29] K. Yap and M. Gutschke. Etherboot user manual, July 2002.

[30] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. SSH protocol architecture, September 2002.