H-0246 (H0611-007) November 13, 2006
Computer Science

IBM Research Report

The Design and Implementation of an IP Only Server

Muli Ben-Yehuda, Oleg Goldshmidt, Elliot K. Kolodner, Zorik Machulsky, Vadim Makhervaks, Julian Satran, Marc Segal, Leah Shalev, Ilan Shimony
IBM Research Division
Haifa Research Laboratory
Mt. Carmel 31905
Haifa, Israel

The Design and Implementation of an IP Only Server

Muli Ben-Yehuda[1] Oleg Goldshmidt[1] Elliot K. Kolodner[1] Zorik Machulsky[1]
Vadim Makhervaks[2] Julian Satran[1] Marc Segal[3] Leah Shalev[1]
Ilan Shimony[1]

[1]{muli,olegg,kolodner,machulsk,satran,leah,ishimony}@il.ibm.com
[2]vadim.makhervaks@gmail.com
[3]marcs@cs.technion.ac.il

IBM Haifa Research Laboratory
Haifa University Campus, Mount Carmel
Haifa, 31905, Israel

Abstract

Present day servers must support a variety of legacy I/O devices and protocols that are rarely used in the day to day server operation, at significant costs in board layout complexity, reliability, power consumption, heat dissipation, and ease of management. We present a design of an IP Only Server, which has a single, unified I/O interface: IP network. All of the server's I/O is emulated and redirected over IP/Ethernet to a remote management station, except for the hard disks which are accessed via iSCSI. The emulation is done in hardware, and is available from power-on to shutdown, including the pre-OS and post-OS (crash) stages, unlike alternative solutions such as VNC which can only function when the OS is operational. The server's software stack — the BIOS, the OS, and applications — will run without any modifications.

We have developed a prototype IP Only Server, based on a COTS FPGA running our embedded I/O emulation firmware. The remote station is a commodity PC running a VNC client for video, keyboard and mouse. Initial performance evaluations with unmodified BIOS and Windows and Linux operating systems indicate negligible network overhead and acceptable user experience. This prototype is the first attempt to create a diskless and headless x86 server that runs unmodified industry standard software (BIOS, OS, and applications).

This is is the revised and extended version of a short paper that was originally published in Proceedings of USENIX '06 Systems Practice & Experience Track.

1 Introduction

Present-day server systems support the same set of I/O devices, controllers, and protocols as desktop computers, including keyboard, mouse, video, IDE and/or SCSI hard disks, floppy, CD-ROM, USB, serial and parallel ports, and quite a few others. Most of these devices are not utilized during the normal server operation. The data accessed by a server frequently exists on remote disks, and the only reason to have a local hard drive is for the operating system boot. However, modern networked storage — both Fibre Channel [1] and iSCSI [2] — now supports booting off remote disk devices, so directly attached hard disks are also becoming unnecessary for this purpose. Various media devices, such as floppies and CD-ROMs, are only used for installation of operating systems and applications, and that can also be avoided with modern remote storage management systems.

Moreover, there are no users who work directly on the server, using keyboard, mouse, and display — normal administrative tasks are usually performed over remote connections, at least while the server is operational. Remote management is done via protocols such as Secure Shell (SSH) and X [11] for Linux/UNIX systems, Microsoft's Windows Terminal Services [8], and cross-platform protocols such as the Remote Framebuffer (RFB, [6]), used by the popular Virtual Network Computing (VNC, [7]) remote display scheme — see also Section 2 below. However, local console access is still required for some operations, including low-level BIOS configuration (pre-OS environment) and dealing with failures, such as the Windows "blue screen of death" and Linux kernel panics (post-OS environments). Local console is usually provided either via the regular KVM (keyboard-video-mouse) interface or a serial line connection.

The legacy protocols and the associated hardware have non-negligible costs. The board must contain and support the multitude of controllers and the associated auxiliary electrical components to deliver the right voltage and current to each of them. This occupies a significant portion of the board real estate that could otherwise contain, say, an additional CPU or memory. The multitude and complexity of the legacy components reduce the mean time between failures (MTBF), and increase the complexity of the associated system firmware and software.

We propose that future servers will only need one or more CPUs, memory, a northbridge, and one or more network interface cards (NICs). All the legacy I/O that is done today over PCI or or other legacy protocols will be done over a single, universal I/O link — the ubiquitous IP network. All communication with storage devices will be done over iSCSI [2], including boot (iBOOT — Remote Boot over iSCSI[1]), so no local disks will be necessary. All system management tasks that require local console access will also be performed over an IP network. Protocols such as USB can also be emulated over IP [16], providing a variety of remote peripherals such as CD-ROM, printers, or floppy if they are needed.

To illustrate this point, Figure 1 identifies the various components of an x86 server (an IBM HS20 blade server in this case, chosen here for illustration because the layout of the board is not obstructed). The CPUs, the DRAM, the northbridge, the BIOS flash, and the network hardware are necessary, while the southbridge, the SCSI and IDE controllers, the graphics adapter, and the integrated legacy I/O chip (implementing the keyboard and mouse controllers, various timers, etc.) can be removed and their functionality can be emulated over the network.

This "remoting" of I/O can be achieved without any modifications at all to the firmware and software stack running on a server: neither the applications, nor the operating system, nor the BIOS need to be modified.

This can be achieved if the protocol emulation is done in hardware. Substituting a single hardware component for all the legacy controllers, capturing all the bus transactions involving the legacy devices, remoting the transactions over the IP network (possibly including protocol translation), and performing the actual I/O at remote stations will be completely transparent to the BIOS and the operating system, and thus to the applications.

[1] `http://www.haifa.il.ibm.com/projects/storage/iboot/`

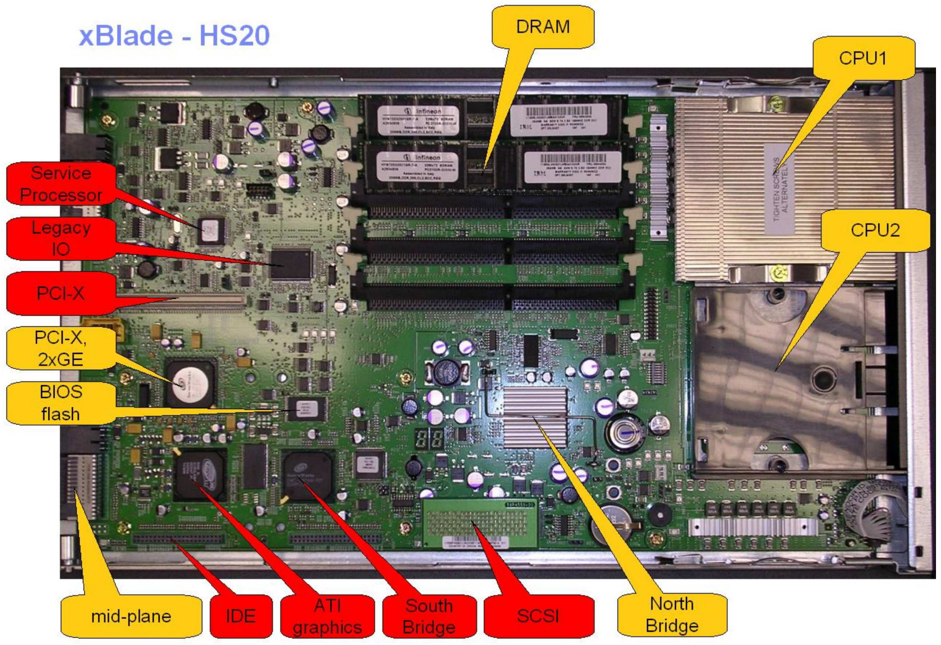

Figure 1: Components of an IBM HS20 blade server.

While many software based alternatives exist for remoting IO when the operating system is up and running (see Section 2 for a survey), doing the protocol emulation in hardware is essential for supporting the pre-OS (e.g., BIOS or bootloader) and post-OS environments.

We designed and implemented a research prototype of such an "IP Only Server", a server that uses Ethernet/IP for all of its I/O needs. We designed and implemented a legacy I/O controller/emulator based on a COTS FPGA. The FPGA, connected to the host via the PCI bus, serves as the local keyboard, mouse, and VGA controller. All keyboard, mouse, and VGA traffic reaches the FPGA and is sent to a remote station over IP. The user is able to perform all the management operations — throughout the lifetime of the server, i.e., during boot, BIOS, operating system initialization, normal operation, and post-OS (e.g., "blue screen of death") stages — from the remote station. No changes were needed for the software running on the host. In particular, neither the BIOS nor the operating systems were modified.

We developed two methods for remoting PCI over IP: iPRP, a straightforward mapping of PCI to IP, and local emulation of PCI devices with remote access over RFB, the stock VNC protocol.

The performance of the prototype is acceptable for the usual server management tasks. The only significant network load may come from the remote iSCSI storage, the network utilization due to remote management is small, and the user experience is without change.

In this paper, we present our design for an IP Only Server and a prototype implementation. The rest of the work is organized as follows. In Section 2 we discuss some related work. In Section 3, we give an overview of the server's design, pertaining to hardware, firmware, and remote station software. In Section 4, we describe our prototype implementation. In Section 5, we evaluate the server's performance when using IP for legacy I/O vs. the performance of an unmodified server using legacy I/O controllers and devices.

Our results show that the IP Only Server provides acceptable user experience when no accelerated graphics are needed. In Section 6, we discuss possible enhancements to the IP Only Server, as well as various novel uses.

2 Related Work

The concept of allowing the user to interact with a remote computer over a network has had a long history [17, 18, 19, 20]. Today there exists a wide variety of thin clients, from the X11 remote display concept [11], often used in conjunction with Secure Shell, to Virtual Network Computing [7] to SunRay [10] to Citrix Metaframe [9] to Windows Terminal Server [8]. Our approach differs from all these solutions in two major ways: we allow remoting of legacy I/O over the network without any host software modifications, and we allow remoting of legacy I/O over the network from the moment the computer has powered on until it has powered off, including when no operating system is present (BIOS stage as well as OS crash).

We did not do any special work on iSCSI [2] or iBOOT (boot over iSCSI), reusing existing software. Support for iSCSI exists in various operating systems, notably in Linux and Windows — the operating systems we experimented with. We had iBOOT-enabled BIOS installed on our prototype servers. The BIOS is generic, and was not created to fit our needs. It is expected that vendors will provide iBOOT support in BIOSes and in hardware (HBAs) in the near future. If iBOOT support is not available, other ways could be used to to boot a server from a remote disk, e.g., the Preboot Execution Environment (PXE).

The Super Dense Server research prototype [3] presented servers with no legacy I/O support (keyboard, VGA or serial). It used Console Over Ethernet [4], which requires operating system modifications and is therefore operating system dependent, and supported text mode only. The Super Dense Server ran Linux, and also used LinuxBIOS [15] rather than a conventional BIOS. In contrast, the IP Only Server runs unmodified operating systems and BIOSes, and supports graphical VGA modes as well as text based modes.

"USB over IP" [16] is used as a peripheral bus extension over an IP network. Using a virtual peripheral bus driver allows the system to remote I/O devices over an IP network with good performance if the network is fast enough. USB/IP is similar to our iPRP protocol (cf. Section 4.4.1): in both cases a hardware bus is extended using a network protocol. The difference is that USB/IP requires a special OS-specific driver, which grabs URBs (USB Request Blocks) at the PDD (USB Per-Device Driver) layer, encapsulates them, and send them over the network, while iPRP does not require any main CPU intervention — it listens on the PCI bus for transactions using a hardware component. The more advanced RFB protocol we use (cf. Section 4.4.2) also listens on the PCI bus, emulates the I/O device, and sends the compressed data over the network. USB/IP is only available while the OS is operational, unlike our solution.

Baratto et al. presented THINC [21], a remote display architecture that intercepts application display commands at the OS's device driver interface. This approach looks promising for remoting the display while the OS is running, but does not handle either pre-OS or exception (post-OS) conditions. THINC could be used together with modifications to the system's BIOS to build a pure software IP Only Server. The main advantage of such an approach is that no hardware modifications are required. Its main disadvantage is the necessity to combine at least two separate software solutions — one for the BIOS, and another for the OS. For instance, BIOS updates are complex and costly, and affect only the pre-OS environment. Remoting post-OS exception states requires the BIOS to remain available and functional even after the OS has come up. This is not the case, e.g., for Linux on x86. It should be noted, however, that Linux does include post-OS remoting facilities (e.g., netconsole and netdump).

The main reason we prefer the hardware solution is the enhanced compatibility and robustness. By using specialized hardware for remoting I/O peripherals the system can be remoted at all times — from power up to power down, or even after a system crash. The OS and the applications are not aware in any way that the I/O is remoted, and since the I/O remoting hardware includes its own processor and network interface, controlling the system requires neither the main CPU nor the main NIC to be operational. Additionally, the possibility of emulating all I/O over the network opens interesting opportunities for system management scenarios, simplifies hardware virtualization (cf. also Section 6 below), and facilitates automatic deployment.

Another advantage of the hardware solution is that some I/O operations are emulated locally, unlike USB/IP[16] that requires dispatching of all I/O operations to the remote machine. Handling events such as a VGA framebuffer read without requiring a network round trip alleviates the need for a high speed/low latency network connection and improves the interactive user experience. Even a loaded Fast Ethernet link does not affect the system's performance.

Hardware vendors are investing significant efforts in the integration of various I/O controllers into single hardware components, sometimes into the southbridge. While these developments address some of the issues mentioned above, e.g., board real estate savings, they do not integrate external I/O channels or protocols, nor do they provide opportunities for multiplexing I/O as the IP Only Server does.

IBM's JS20 PowerPC-based blades do not contain any video/mouse/keyboard components. The main I/O channel into those machines in the pre-OS and post-OS states is Serial-over-LAN that tunnels text-only serial console messages over UDP datagrams. Once the OS is up, the usual methods of accessing it apply (e.g., SSH and VNC).

KVM over IP products (e.g., Cyclades AlterPath KVM/net) provide an easy way to remote all operating environments. However, such products carry a non-negligible price tag. Servers using KVM over IP still require a full complement of hardware components — a graphics card, keyboard and mouse controllers — that remain unused, thus wasting the board real estate and power.

Several months after we conducted this research a news report was published [2] about Sun developing an "integrated lights-out" management processor and a full keyboard-video-mouse-storage (KVMS) emulation for their x86 servers. We do not have any further information on the architecture, although superficially it appears to be similar to our IP Only Server.

3 Design

The design of the IP Only Server was developed according to the following guidelines:

1. The IP Only Server must preserve software backward compatibility. It must be able to run unmodified host software, including OS and BIOS.

2. The IP Only Server must provide remote access to the host from the moment it has been powered on until the moment it has been powered off. This includes the BIOS stage, the operating system's lifetime, and even post-operating system environments such as the "blue screen of death" or a Linux kernel oops. However, during normal operation the OS may provide a more effective remote console or terminal support through X-Windows, Windows Terminal Server, or another similar solution.

3. The server should have the minimal amount of local state required for disconnected operation. The hard drives should be remoted over IP (including boot), and the RTC (Real Time Clock) should be initialized by a remote device.

4. The IP Only Server must be able to work even when no remote management station is connected, or when one has been connected and then disconnected. The IP Only Server must also support arbitrarily many re-connections by the remote management station. Obviously, the remote storage that is necessary for boot and normal operation of the server must be available

[2]`http://news.com.com/Bechtolsheims+machine+dreams/2008--1010_3--5857470.html`

even if the remote station is disconnected at any given moment.

5. The IP Only Server must provide text (console) support and graphical mode support. There is no requirement to provide more than plain VGA mode support — the IP Only Server is not aimed at users who need accelerated graphics.

6. The remote management station should not require a custom or proprietary client, e.g., the KVM-over-IP (Keyboard/Video/Mouse-over-IP) protocol should be based on open standards.

7. A single remote station should be able to control multiple IP Only Servers concurrently.

8. Support for remoting desktop peripherals such as USB connected devices, CD-ROM etc. should also be provided over IP.

The IP Only Server can be based on any standard architecture, for instance on Intel (and compatible) x86-based servers. The CPU, memory, the northbridge, and the BIOS flash are not modified. The server will include at least one network interface. The following components, normally present in any modern server, will not be needed:

- video controller and related video memory
- hard drives or hard drive controller
- keyboard and mouse controllers
- real time clock backup battery
- southbridge
- serial port
- parallel port
- USB ports

The functionality of the above controllers can be emulated on an ASIC (or FPGA) that presents the legacy I/O interfaces to the host (via a PCI bus), and remotes them over an IP based protocol to the remote station. Such an ASIC (or FPGA) is substantially smaller, cheaper and simpler than the sum of these components. It leads to significant savings in board layout complexity, power consumption, heat dissipation and ease of management, and allows the server to run unmodified firmware and software. The choice between an ASIC or an FPGA is a trade-off between price and programmability: research prototypes are likely to be based on FPGA, while subsequent mass production may switch to ASICs. This scheme will satisfy design guidelines 1 and 2 above.

The IP Only Server will not include any local disks. Instead, it boots from a remote boot device, such as an iSCSI disk via iBOOT or PXE. Alternatively, disk access can be remoted like the other legacy I/O protocols. A mixture of the two approaches is possible in principle: the emulation hardware can include an implementation of a boot-capable iSCSI initiator. This will satisfy design guideline 3 above.

For the prototype described below we designed an FPGA that presented itself as a VGA/keyboard/mouse device. The server's BIOS and OS accessed the FPGA using their standard drivers. The FPGA received all hosts accesses as PCI transactions, and handled them appropriately.

We experimented with two different approaches to remoting these PCI transactions to a remote station. The first approach, the Internet PCI Remote Protocol (iPRP) was essentially "PCI over IP": PCI transactions were wrapped by IP packets, sent to the remote station, and processed there. Responses were sent back as IP packets as well, and the emulation FPGA passed them to the PCI bus of the host. Clearly iPRP does not satisfy design guideline 4 above, and was used mainly as an intermediate debugging tool, in particular to get over the hurdle of finding all the places in which BIOS and OS touch local devices. It is described in Section 4.4.1.

The second approach, using the RFB protocol, is described in Section 4.4.2. In this scheme the emulation FPGA translates keyboard, mouse, and video PCI transactions into the high level RFB protocol that allows using any VNC client (and any operating system) on the remote station. Communication with a remote station is only needed to present the video display to the user, and to accept the user's keyboard and mouse input. Otherwise the PCI transactions are processed locally by the FPGA.

The difference between the two schemes is highlighted by the diagram in Figure 2: with iPRP the FPGA is essentially transparent, and the emulation is done in the remote station; with RFB the remote station exists for the user interface only, the emulation is done in the FPGA, and some PCI transactions (reads) are handled locally inside the FPGA.

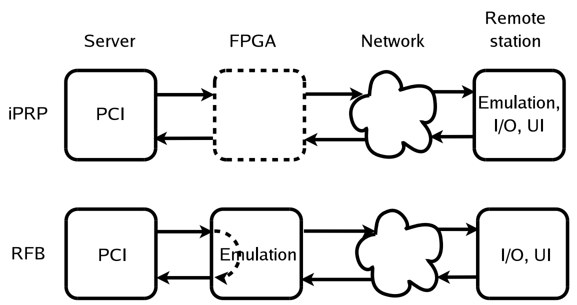

Figure 2: Comparison of iPRP and RFB implementations of an IP Only Server.

Thus, in the RFB design, if the user is not interested in interacting with the server the latter can operate without a remote station. On the other hand, a user can open VNC sessions against multiple IP Only Servers simultaneously. Clearly, this approach supports design guidelines 4 through 7.

Design guideline 8 — support for USB over IP and other remote peripheral devices (CD-ROM, floppy, etc., most of which can be provided over USB) from power-on to power-off — was not implemented in our prototype, and is left for future work (cf. Section 6).

4 IP Only Server Implementation

For the prototype we used an FPGA evaluation board, Dini-group DN3000k10s with a Metanetworks Inc. MP1000TX Ethernet prototyping plug-in board. The FPGA is a Xilinx Virtex-II x2v8000 FPGA. The design ran at 50 MHz, used 378 KB of local memory, and 8400 logic slices (equivalent to about 1M gates).

The FPGA firmware was divided into four main modules: a BIOS expansion ROM, a PCI Interface Module, a Network Interface Module, and a Transaction Processing Module. We cover these components in more detail below.

4.1 Expansion ROM

We implemented a BIOS expansion ROM on the evaluation board. The expansion ROM contained the VGA BIOS initialization routines that are invoked by the main system BIOS during boot. The expansion ROM was based on the GPL VGA BIOS included in the Bochs x86 emulator (`http://bochs.sourceforge.net`) with insignificant modifications.

4.2 PCI Interface

This module is responsible for the correct behavior and presentation of the device on the PCI bus. The module answers to the following addresses on the bus:

- I/O addresses 0x60, and 0x64 — keyboard controller
- I/O address ranges 0x3B0–0x3BB, 0x3C0–0x3DF — VGA controller
- Memory address range 0xA0000–0xBFFFF — VGA memory

The PCI configuration space of the device is set up in such a way that the device identifies itself as a PCI-based VGA adapter. Upon discovery of the VGA add-in card, standard BIOSes consider it as the default VGA adapter, and configure the chipset in such a way that all VGA I/O and memory transactions are routed to the device. Having keyboard controller I/O addresses routed to the device is trickier, and requires additional northbridge and I/O bridge configuration done by the BIOS. The transactions are captured by the device and placed into a Transaction FIFO queue which serves as the interface to Transaction Processing module (cf. Section 4.4 below).

4.3 Network Interface

The network interface consists of a low-level interface to the board's Ethernet adapter and a DMA engine. The DMA engine transfers the packets built by the Transaction Processing Module and the TCP/IP stack from the FPGA memory to the Ethernet device.

In general, an IP Only Server does not need a separate management Ethernet port. The overhead of the management traffic is small, and it is unlikely to interfere with the rest of the server's traffic. It may be beneficial to use a separate management interface for other reasons — for instance, if the main interface is down or saturated (e.g., due to heavy load or to a denial of service attack). The iSCSI storage interface may also be separate, for performance and/or security reasons.

In our prototype implementation we used separate interfaces for iSCSI and for KVM emulation. The former was handled by the server's regular ethernet interface, and the latter was handled by the FPGA's Ethernet interface.

4.4 Transaction Processing

The Transaction Processing firmware consists of a single loop that gets the PCI transactions out of the Transaction FIFO queue (cf. Section 4.2 above), and either handles them locally or wraps them into the appropriate network protocol (iPRP or RFB — see Sections 4.4.1 and 4.4.2 below). The protocol messages are passed to the Network Interface Module that sends them over Ethernet to the remote station.

To improve network utilization subsequent write transactions are coalesced into bigger network packets, while read transactions are either handled locally (in the RFB version, cf. Section 4.4.2) or sent to the network immediately (in the iPRP implementation, cf. Section 4.4.1), to avoid bus parking and host CPU stalling for a long time. Traffic received from the remote machine may include read response transactions and interrupt requests. Read response transactions are forwarded by the Transaction Processing Module to the PCI Interface Module that releases the pending bus read request transactions.

Interrupt requests are generated by the remote machine in reaction to keyboard events. Upon receipt of an interrupt request our prototype adapter is expected to cause IRQ1 interrupt which is associated with the keyboard controller. An add-in card, being a PCI device, cannot generate IRQ1 interrupt. To overcome this obstacle we used the chipset's Open HCI USB legacy keyboard emulation feature to emulate IRQ1 interrupt. Upon arrival of an interrupt request the Transaction Processing Module issues a PCI write transaction to the PCI mapped chipset Open HCI USB register that is responsible for legacy keyboard emulation. The chipset raises an IRQ1 interrupt as a result. When interrupted, the host CPU accesses the 0x60 I/O port of the keyboard controller to figure out the reason for the interrupt. The access is captured by the FPGA and is delivered to the emulated keyboard controller, which responds with the 0x60 content. The FPGA resets the interrupt by writing to the appropriate Open HCI USB legacy keyboard emulation register of the chipset.

4.4.1 iPRP — Internet PCI Remote Protocol

The iPRP protocol is a very simple UDP protocol used to emulate memory and I/O space PCI transactions over an IP network. The PCI configuration space accesses are also defined for completeness. The protocol was designed for ease of implementation: it is not very efficient in terms of coding, and contains redundant information. It is in no way a PCI implementation, and in its current state can not be used for a PCI bridge: the iPRP client side is to be implemented on a PCI target (and not a PCI bridge), although it could potentially be used as a bridge, with some modifications.

The protocol uses UDP for transport, and includes only a simple recovery mechanism. This allows for a quick and dirty implementation, with no need for operating systems or complicated protocol stacks. The FPGA firmware is particularly simple: the FPGA does essentially nothing but ships the PCI transaction data between the PCI and network interfaces and back. The protocol has been designed and implemented primarily to ease the initial debugging and bring-up of the IP Only server hardware and firmware, allowing to concentrate on the idiosyncrasies of the PC low level architecture (boot, BIOS, VGA interface, PCI usage).

The protocol relies on UDP checksum for data correctness, and uses a go-back-N scheme [14] with sequence numbers for delivery correctness.

In the following description a *command* is defined as a single I/O command such as memory read, I/O space write, or acknowledgment. Multiple commands may be packed into a single Ethernet frame. Accordingly, a *message* is one or more commands, mapped into a single Ethernet PDU. In a multi-command message only the last command may be a read type command, since the PCI-based system can not proceed until the read data arrives back. Each commands has an attached *sequence number* (SN).

Each message is a single UDP datagram, which may include one or more commands. For simplicity the protocol acts in a dual bi-directional fashion, i.e., an ACK can not be attached to a message in the other direction — it must be sent as a separate message. A message is acknowledged by a single ACK, even if there are multiple commands in the message — the ACK message acknowledges all commands with sequence numbers less the the ACK's SN. The protocol uses 'go-back-N' acknowledgment, i.e., there may be up to N unacknowledged commands in flight. ACKs for read commands contain the returned data.

A read request will trigger sending a message — like with most I/O devices strict ordering must be kept, and the CPU must wait for the read data before continuing. In order to make sure that the display screen is updated at all times a timeout may also trigger a message send. In case the maximum message size is reached the message is also sent.

Since this protocol is an extension to a local PCI bus it is expected that the host will crash (or just lock up) due of a broken connection. No provisions were made to keep the host alive in case of a network problem or a remote side failure. This was enough for our purposes: iPRP was essentially a debugging tool. In practice the protocol exhibited rather remarkable robustness given its simplicity: we successfully ran (not heavily loaded) servers for hours and even days on end.

The remote station software was based on the Bochs open source x86 emulator. We extracted the relevant PCI device emulation code, for instance the code for a VGA and keyboard/mouse controllers, and fed it the PCI transactions received as iPRP payload as input. For host PCI write transactions the VGA and keyboard/mouse state machines are updated, and the Bochs VGA screen is displayed to the user. For host PCI reads, a return packet with the response is sent back to the FPGA.

As a particular example of using iPRP for debugging it is instructive to mention that it is trivial to extract and log the actual PCI transactions performed by the server on the remote station. The transaction log can then be used to drive a standalone or emulation version of the RFB-based FPGA firmware (see Section 4.4.2) to verify its correctness and debug the possible problems. We used this technique on a number of occasions.

4.4.2 RFB — Remote Frame Buffer

To overcome the shortcomings of the iPRP version, especially the fact that the server cannot operate without the remote station, we have to emulate the device controllers in the FPGA firmware rather than in the remote station software.

The Bochs code again proved immensely useful: we took Bochs implementations of VGA, keyboard, and mouse controllers — the same we used in the iPRP remote client — and based our firmware on them. In particular, we ported the relevant Bochs C++ code to C, eliminated the use of any libraries and system facilities unavailable on our FPGA, and invested significant effort into reducing the footprint and improve the efficiency of the resulting code.

To transfer the keyboard, mouse, and video events between the FPGA and the remote station we chose the Remote Framebuffer (RFB) protocol [6]. To this end, we implemented a VNC server [7] in the FPGA firmware. The choice of RFB was motivated by two important considerations. First, RFB is a well known and widely used open protocol, and this eliminates the need to implement remote station software: any VNC client on any platform will do the job. Secondly, the Bochs emulator includes an implementation of a VNC server as one of the choices for interface, and we based our implementation on that, albeit with significant modifications.

In particular, our FPGA platform was limited in both space (350 KB of memory total) and speed (50 MHz CPU, no memory caches). Accordingly, we started with a straightforward hardware VGA controller emulation in software, based on the Bochs VGA controller, and optimized it both in space and time to fit our FPGA environment. For instance, by removing support for VGA modes that were not used by either Linux or Windows, we managed to reduce the VGA framebuffer to 128 KB.

Since the RFB protocol is TCP/IP based, we also added a TCP/IP stack to the FPGA firmware. Due to the limited memory and processing resources of the FPGA we had to implement a custom embedded TCP/IP stack. The stack is minimalistic and is specifically designed to fit our firmware environment, but it implements a complete TCP state machine and is streamlined: it has a very low memory footprint and avoids copying of data as much as possible.

Like the iPRP version, the RFB-based FPGA firmware is based on a single looping execution thread: there is no operating environment, no context switching, no memory management. The firmware receives the host's PCI transactions from the PCI interface. Unlike the iPRP version, it processes some transactions locally. Host PCI reads are answered immediately. Host PCI writes update the local device state machines. Every so often, the frame buffer updates are sent to the remote station to be displayed. The decision of when to update the remote station is crucial for establishing reasonable performance with our constrained FPGA; we developed heuristics that performed fairly well (cf. Section 5).

5 Performance Evaluation and Analysis

The most important performance metric for evaluating the IP Only Server is user experience, which is notoriously hard to quantify. In order to approximate the user's experience, we performed several measurements.

All tests were performed on two identical IBM x235 servers including the PCI-based FPGA evaluation board with a 100 Mb/s Ethernet network interface. The remote station software ran on two R40 Thinkpad laptops with a 1.4 GHz Pentium M CPU and 512 MB RAM each. The servers were booting either Windows 2003 Server or Red Hat Enterprise Server Linux 3.0. The iSCSI connection was provided through a separate network interface, and the FPGA was not involved in communication with the iSCSI target at all. The performance of iSCSI storage has been studied independently, and we were primarily interested in the performance of our FPGA-based KVM emulation. Therefore, for the purpose of these measurements we used local disks, to achieve a clean experimental environment.

First, we measured the wall time of a server boot for each of the three scenarios: a server with native legacy I/O peripherals, an IP Only Server with an FPGA using iPRP, and an IP Only Server with an FPGA using RFB. In each scenario, we measured the time from power on until a Linux console login prompt appeared, and from power on until a Windows 2003 Server "Welcome to Windows" dialog showed up.

As depicted in Figure 3, an unmodified server booted to a Linux login prompt in 238 seconds, while a server using the iPRP version of the FPGA booted in 343 seconds, and a server using the RFB version of the FPGA booted in 330 seconds. This is a 38% slowdown (for the RFB version), which is acceptable for the very first, unoptimized prototype.

As depicted in Figure 4, an unmodified server booted to the Windows 2003 Server "Welcome to Windows" dialog in 186 seconds, while the server with the iPRP FPGA booted in 299 seconds, and the server with the RFB FPGA booted in 289 seconds. This is slightly worse, a 55% slowdown, but again, it is acceptable.

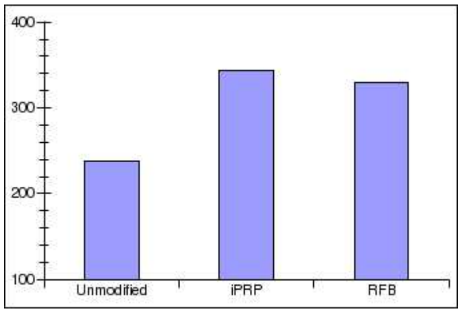

Figure 3: Linux boot time in seconds.

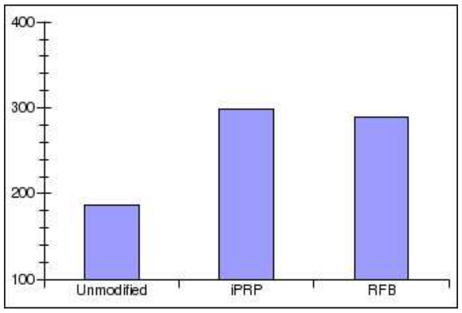

Figure 4: Windows boot time in seconds.

Server	Native	iPRP	RFB
Linux	238	343 (144%)	330 (138%)
Windows	186	299 (160%)	289 (155%)

Table 1: Linux and Windows boot time in seconds.

It should be noted that the RFB and iPRP versions performed nearly the same. While iPRP sends much more traffic over the network than RFB the bulk of the emulation in the iPRP version is performed by the 1.4 GHz Pentium M CPU of the remote station, whereas in the RFB version it is done by the severely constrained 50 MHz FPGA CPU. This leads us to conclude that there exists a trade off between performance and cost here: by throwing a stronger (more expensive) FPGA or ASIC at the emulation, the performance can be easily improved.

While the user is mostly concerned with how quickly the IP Only Server responds, the system administrator would be rightly concerned about the network utilization of such a server. Unlike legacy I/O devices, which are a local resource, the IP Only Server uses the shared network infrastructure.

We measured the network utilization using the RFB version of the FPGA in the the same Linux and Windows boot scenarios described above. We used the Ethereal (`http://www.ethereal.com`) network sniffer to capture all network traffic and the tcptrace (`http://www.tcptrace.com`) network analyzer to process the captured data. Table 2 shows the number of actual data bytes exchanged, the number of TCP packets and the throughput for both Linux and Windows boot sequences. The throughput is of the order of 1 Mb/s — not a significant load for a modern local area network, and even acceptable for a wide area connection.

Booted OS	Unique Bytes	Packets	Throughput
Linux	52324982	189920	131488 Bps
Windows	47457592	155542	164384 Bps

Table 2: Network Utilization.

Figure 5 shows the throughput in bytes/sec of the VGA traffic from the FPGA to the remote station, as distributed throughout the boot sequence. Most of the traffic occurs when the BIOS draws the initial eServer logos on the screen and when grub (the Linux bootloader) draws its graphical menus. Note that grub can be trivially modified to use a text based menu, which will significantly cut down on the boot time network utilization.

An additional observation is related to an important class of systems that may be built on the basis of IP Only Servers— the so-called "hosted clients". Hosted clients are interesting for a number of reasons: diskless user machines may be required for se-

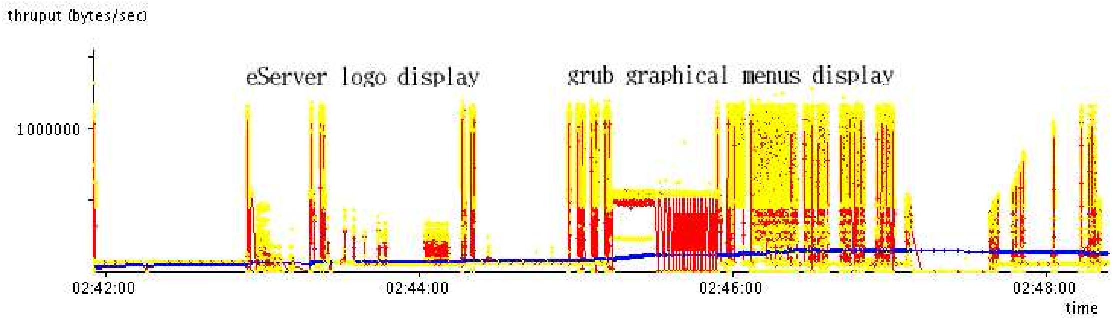

Figure 5: Linux boot network throughput.

curity, particular applications may require a machine with more computing power or memory on a temporary basis, several virtual clients can be consolidated on one powerful server, etc. There are vendors that provide hosted client solutions by capturing the low level video access stream, processing it (mainly applying different compression schemes), and shipping to a remote display.

We ran some experiments to assess whether our low level display remoting scheme could be used to support a hosted client solution that deals with graphically intensive applications, and found, similarly to Baratto et al. [21], that remoting the higher level graphics access APIs was more effective than remoting the low level hardware graphic access. The reason is that important information accessible to graphics device drivers is not normally available to the graphics hardware. This indicates that there is even less reason to keep local graphics adapters on servers supporting hosted clients. For such servers our emulation scheme can be relegated to supporting the pre-OS environment and exception handling (possibly supporting more than one virtual console), i.e., functions that are not graphically intensive.

6 Future Work

Future IP Only Server work includes supporting more protocols, such as USB, serial, and parallel. USB support during pre-OS and post-OS stages is particularly desirable, as it would help provide support a diverse set of devices, including floppy and CD-ROM drives.

The IP Only Server prototype runs on x86 only at the moment, although there's nothing inherently x86-centric in its design. A port to other architectures such as PowerPC [5] will validate the design.

The IP Only Server provides interesting opportunities when combined with para-virtualization technologies such as Xen [12] or full virtualization technologies such as VMWare [13]. It can be used to give unmodified guest operating systems physical device access, while providing isolation between different guests and offering each guest its own view of the I/O hardware.

The IP Only Server could also be used to provide a transparent virtualization layer on top of physical devices, by providing several sets of device control registers, associating each register set with a given partition. Additionally, it could provide several servers with access to the same remote physical device. The host OS on each server would access what appears to it as a physical device, but is actually the IP Only hardware. The IP Only hardware would remote the device accesses to a remote station, which would multiplex the device usage between several servers.

Remoting all I/O— including privileged and confidential I/O— over the untrusted medium of an IP network, opens up all sorts of interesting security questions. Which host is allowed access to which remote stations, and vice versa? How can we protect against man in the middle attacks, spoofing and other network attacks? Disk data that may contain sensitive business information must be protected. The screen contents must be protected, since confidential information may be displayed. Keystrokes must be protected, since they may convey passwords and other security-sensitive data. It should be noted that today's remote access protocols must deal with these issues as well. However, today one can choose whether to use VNC or the secure local console; with IP Only, there is no alternative. Widespread IP Only Server use will require solving these problems.

7 Conclusion

We present a novel approach to legacy I/O support in servers. We designed an IP Only Server, utilizing the IP network as the single I/O bus. All user interaction throughout the lifetime of the server, i.e., during boot, BIOS, operating system initialization, normal operation, and post-OS (e.g., "blue screen of death") stages are done via the remote station. No changes were needed for the software running on the host — in particular, neither the BIOS nor the operating systems were modified. While diskless servers have been attempted before, and headless servers exist as well (e.g., IBM's JS20 blades), as far as we know this is the first attempt to create a diskless, headless server that runs industry standard software (BIOS, Windows or Linux OS) without any modifications.

We assumed that in an industrial setting BIOS is tightly coupled with the Reliability and Availability Services, and that attempting sweeping changes in BIOS would affect both testing and management. At least in the IBM production setting this has proved to be true. The emulation approach enables a gradual introduction of servers that do not carry legacy devices yet enable box testing and management to stay unchanged and or to change gradually. We assumed also that there will be a wider than expected dependencies on legacy devices (keyboard, mouse, timer), especially in operating environment with a strong desktop legacy. This also proved to be true, e.g., Windows boot touches keyboard and video hardware more than one would expect. The emulation approach that we took turned out a good way to support Windows without requiring difficult and expensive (and not necessarily feasible) changes in the OS. One of the major reasons for the popularity of the x86 architecture is that it's affordable, and one reason for the low cost of x86 systems is ready availability of software built for large numbers of desktops. The emulation approach is instrumental in keeping this argument valid.

Our research prototype implementation of the IP Only Server provides a user experience that is comparable to that of a regular server, with reasonable latency and low network utilization.

The IP Only Server can provide significant savings in hardware and software costs, power consumption, heat dissipation and ease of management. It eliminates some legacy aspects of the PC architecture, replacing them with a single, simple, and modern counterpart.

Acknowledgments

The authors would like to thank Kevin Lawton and other Bochs developers for their excellent emulator which made our work far less frustrating than it could have been. The authors would also like to thank Michael Rodeh and Alain Azagury for their interest and support of this project.

References

[1] X3T9.3 Task Group of ANSI: Fibre Channel Physical and Signaling Interface (FC-PH), Rev. 4.2, 1993

[2] K. Z. Meth, J. Satran, "Design of the iSCSI Protocol", Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies, pp. 116–122, 2003.

[3] W. M. Felter, T. W. Keller, M. D. Kistler, C. Lefurgy, K. Rajamani, R. Rajamony, F. L. Rawson, B. A. Smith, and E. van Hensbergen, "On the Performance and Use of Dense Servers", IBM

Journal of R & D, vol. 47, no. 5/6, pp. 671–688, 2003.

[4] M. D. Kistler, E. van Hensbergen, and F. Rawson, "Console Over Ethernet", Proceedings of FREENIX Track, USENIX Annual Technical Conference, pp. 125–136, 2003.

[5] C. May, E. Silha, R. Simpson, H. Warren, "The PowerPC Architecture: A Specification for a New Family of RISC Processors", Morgan Kaufmann, San Francisco, CA, 1994.

[6] T. Richardson, "The RFB Protocol", RealVNC Ltd., Version 3.8, 2004.

[7] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper, "Virtual Network Computing", IEEE Internet Computing, 2(1), 1998.

[8] B. C. Cumberland, G. Carius, and A. Muir, "Microsoft Windows NT Server 4.0 Terminal Server Edition: Technical Reference", Microsoft Press, Redmond, WA, 1999.

[9] T. W. Mathers, S. P. Genoway, "Windows NT Thin Client Solutions: Implementing Terminal Server and Citrix Metaframe", Macmillan Technical Publishing, Indianapolis, IN, 1998.

[10] B. K. Schmidt, M. S. Lam, and J. D. Northcutt, "The Interactive Performance of SLIM: A Stateless, Thin-Client Architecture", Proceedings of the 17th ASM Symposium on Operating Systems Principles (SOSP), v. 34, pp. 32–47, Kiawah Island Resort, SC, 1999.

[11] R. W. Scheifler, J. Gettys, "The X Window System", ACM Transactions on Graphics, 592:79–106, 1986.

[12] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, "Xen and the Art of Virtualization", Proceedings of the 19th ASM Symposium on Operating Systems Principles (SOSP), pp. 164–177, Bolton Landing, NY, 2003.

[13] J. Sugerman, G. Venkitachalam, B.-H. Lim, "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor", Proceedings of General Track, USENIX Annual Technical Conference, pp. 1–14, 2002.

[14] D. Bertsekas, R. Gallager, "Data Networks", 2nd ed., Simon & Schuster, Saddle River, NJ, 1991.

[15] R. Minnich, J. Hendricks, and D. Webster, "The Linux BIOS", Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, GA, 2000.

[16] T. Hirofuchi, E. Kawai, K. Fujikawa, and H. Sunahara. "USB/IP — A Peripheral Bus Extension for Device Sharing over IP Network", USENIX Annual Technical Conference, FREENIX Track, Anaheim, CA, 2005.

[17] M. Wilkes, R. Needham, "The Cambridge Model Distributed System", ACM SIGOPS Operating Systems Review, v. 24/1, pp. 21–29, 1980.

[18] R. van Renesse, A. Tanenbaum, and G. Sharp, "Functional Specialization in Distributed Operating Systems", Proceedings of the 3rd ACM SIGOPS European Workshop: Autonomy or interdependence in distributed systems? Cambridge, United Kingdom, pp. 1–4, 1988.

[19] M. Hayter, D. McAuley, "The Desk-Area Network", ACM Operating System Review v. 25, pp. 14–21, 1991.

[20] H. Eberle, E. Oertli. "Switcherland: A QoS Communication Architecture for Workstation Clusters", Proceedings of ACM ISCA '98, Barcelona, Spain, 1998.

[21] R. Baratto, L. Kim, J. Nieh, "THINC: a Virtual Display Architecture for Thin-Client Computing", Proceedings of the 20th ACM Symposium on Operating systems, Brighton, United Kingdom, pp. 277–290, 2005.