

# Linux as Virtual Interface Architecture for Network Block Device

Prof. Vinit A. Sinha

Prof. S. V. Joshi

**Abstract-** VIA is a user-level memory mapped communication model which provides zero-copy communication by removing the operating system from the critical communication path. Typically, an NBD layer offers the abstraction of a storage media across the network. GNBD/VIA attempts to improve the performance of the NBD layer by employing the lightweight VIA communication mechanisms between NBD servers and clients. This paper describes a design and implementation of GNBD/VIA, a Network Block Device (NBD) over Virtual Interface Architecture (VIA), and evaluates its performance on Linux-based cluster of PCs. To our best knowledge, GNBD/VIA is the first implementation of NBD on VIA. GNBD/VIA outperforms the normal NBD placed on top of TCP/IP protocol stacks, and achieves the performance comparable to local disk devices.

## I. INTRODUCTION

The System Area Network (SAN) is a key element in building scalable cluster systems by providing low latency and high bandwidth communication. However, the traditional communication models were unable to fully exploit the raw performance of the recent SANs operating at gigabit speeds, due to the high overhead added by software layers [8]. The Virtual Interface Architecture (VIA) [1] is an industry standard on user-level memory-mapped communication model, whose main objective is to reduce the communication overhead further for high-speed SANs. The basic idea in user-level communication is to factor out protection from the critical path of communication operations. To provide protected communication, two conditions must be satisfied. First, the kernel must grant the permission for a process to communicate with another process by providing a communication channel. Second, the network interface must multiplex user-level DMA performed through these channels. This support eliminates the need to trap into the kernel each time a send is executed, and makes the send operation lightweight. At the same time, no copy is necessary by sending data from the user space to a remote receive buffer and the end-to-end communication bandwidth approaches to the raw bandwidth provided by the network hardware. There are several hardware and software implementations of VIA today. Emulex (Giganet before) has a hardware VIA implementation called cLAN with drivers for Linux and

Windows NT. VIA implementations at the firmware level are available for ServerNet (Tandem) and Myrinet (Myricom) interconnects. M-VIA [6] provides Linux software VIA drivers for various fast ethernet and gigabit Ethernet adapters. In this paper, we describe a design and implementation of a network block device (NBD) over VIA for a Linux-based cluster of PCs. The NBD is a software layer which offers the abstraction of a storage media across the network, where a remote server provides the real physical storage. NBD clients can access the server's disk device as if it were a local one through a virtual device created at the client side. The virtual device acts exactly like a traditional block device to client applications and it is even possible to make a file system on it using the UNIX mkfs command. As each disk read/write request to the virtual device is delivered to the NBD server over the communication network, the performance of NBD heavily depends on the underlying communication performance. Our goal is to design and implement a highly efficient NBD layer which takes advantage of the low latency and high bandwidth characteristics of VIA, in order to minimize the performance gap between local disks and NBDs. The existing NBD layers utilize TCP/IP protocols to communicate between NBD servers and clients. However, if NBD servers and clients are interconnected through VIA-enabled SAN within a cluster, we can accelerate the performance of NBD by replacing the TCP/IP protocols with the lightweight VIA communication mechanisms. Among many NBD implementations, we have modified GNBD (GFS NBD), which is used to build GFS (Global File System) on IP-based networks [9].

## II. BACKGROUND

### 2.1. Virtual Interface Architecture (VIA)

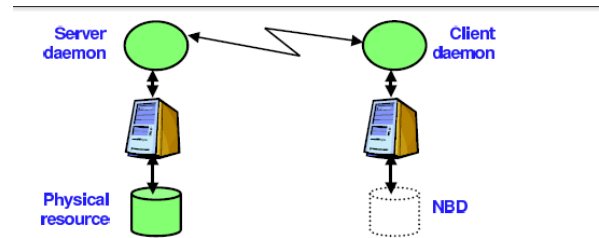
In the traditional network architecture, the operating system (OS) virtualizes the network hardware into a set of logical communication endpoints available to network consumers. The OS multiplexes access to the hardware among these endpoints. In most cases, the OS also implements protocols that make communications between connected endpoints reliable. This model permits the interface between the network hardware and the OS to be very simple. The drawback of this organization is, however, that all

communication operations require a call or trap into the OS kernel, which can be quite expensive to execute. The demultiplexing process and reliability protocols also tend to be computationally expensive. The Virtual Interface Architecture (VIA) eliminates the system processing overhead of the traditional model by providing each consumer process with a protected, directly accessible interface to the network hardware - a Virtual Interface (VI). Each VI represents a communication endpoint. A pair of VI endpoints can be logically connected to support bi-directional, point-to-point data transfers. A process may own multiple VIs exported by one or more network adapters. A network adapter performs the endpoint virtualization directly and subsumes the tasks of multiplexing, de-multiplexing, and data transfer scheduling normally performed by an OS kernel and device driver. An adapter may completely ensure the reliability of communication between connected VIs. Alternately, this task may be shared with transport protocol software loaded into the application process, at the discretion of the hardware vendor [1].

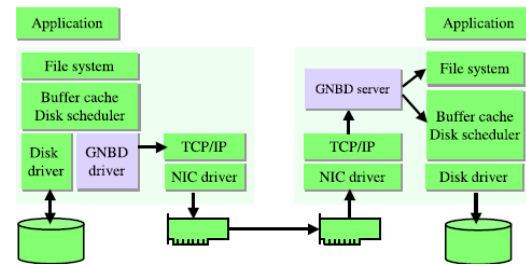
## 2.2. Network Block Device (NBD)

Figure 1 illustrates the basic concept of Network Block Device (NBD). The NBD offers an access model that simulates a block device, such as a hard disk or a hard-disk partition, on the local client, but connects across the network to a remote server that provides the real physical storage. For clients, the device looks like a local disk partition, but it is only an entrance for the remote. Even though the actual access Requests and data blocks are communicated on the network, the NBD layer hides all the details and the client simply uses the virtual device as if it were a local disk device. This is a little lower level and more basic than network file systems such as NFS or Samba, which require more kernel interaction to properly handle the file-level access requests from the remote host.

There are several implementations of the generic NBD: Linux/NBD, DRBD, ENBD, ODR, and GNBD. Linux/ NBD is the basic NBD driver that is included in the Linux kernel.



**Figure 1. The concept of Network Block Device**



**Figure 2. The structure of GNBD (GFS Network Block Device)**

## 2.3. Motivation

Generally, the NBD layer is built on top of the TCP/IP Layer, which means the location of NBD server and client can be separated across the LAN or WAN. There are, however, many application domains of the NBD layer where the NBD server and client are closely linked within a cluster. For example, combined with software RAID driver, GNBD can provide a shared storage between two servers configured with a fail-over support. Another example is the GFS: it aggressively uses a set of GNBD servers and clients to implement a cluster file system on IP-based networks, even in the absence of Fiber Channel switch and storages. In these environments, communication within a cluster is a major factor which limits the overall performance. As the performance of NBD heavily depends on the underlying communication performance, it is important to extract the raw performance of network hardware as much as possible in order to minimize the performance gap between local disks and NBDs.

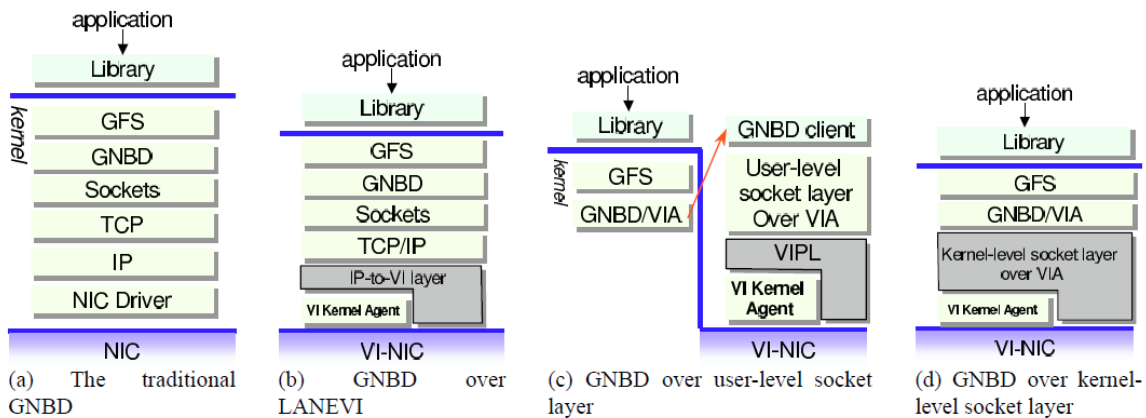


Figure 3. Design alternatives for GNBD client

### III. DESIGN ISSUES

#### 3.1. Design Alternatives

The original GNBD works on Sockets interface on top of the TCP/IP stacks as can be seen in figure 3(a). The GNBD can be implemented on the VIA-enabled system area networks in several ways: (1) by emulating IP layer over VIA, (2) by using a user-level Sockets layer over VIA, and (3) by using a kernel-level Sockets layer over VIA. We briefly examine the characteristics of each alternative and discuss which is the desirable choice for our goal.

**Using a kernel-level Sockets layer over VIA.** The two approaches explained earlier do not effectively exploit the VIA's advantages even though they are built on the NIC that supports VIA mechanisms in hardware. Instead, we implement GNBD/VIA based on the kernel-level Sockets layer over VIA, as depicted in figure 3(d). We design and implement a slim layer which provides a subset of Sockets-like interfaces over VIA inside the kernel. We believe this approach is the simplest way to port GNBD over VIA with minimal efforts while producing the maximum performance. Although VIA enables user-level communication, the previous observation in figure 3(c) suggests it is not necessary to move the disk I/O requests and data to the user-level because such requests and data are already present in the kernel.

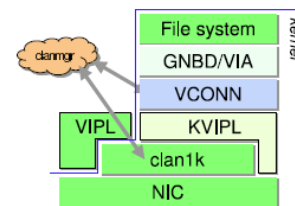


Figure 4. The layering of GNBD/VIA implementation

#### 3.2. KVIPL (Kernel level VIPL) layer

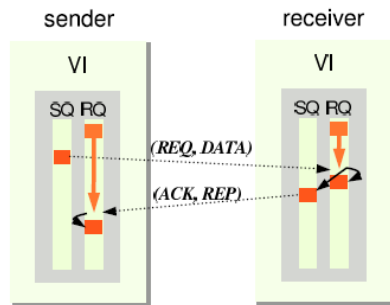
For user-level applications, the VIA specification defines a set of standardized API called VIPL (VI Provider Library). However, as the VIPL is provided in the form of a user-level library, it can not be used inside the kernel. Fortunately, the cLAN driver from Emulex has a set of kernel level VI Provider Library or KVIPL, as part of its LANEVI layer.

Even though KVIPL is included in the cLAN driver, KVIPL does not provide a complete set of APIs for VI programming, because it is an unofficial submodule for implementing the LANEVI layer. For example, since it relies on clanmgr when a node connects or disconnects with a remote node, it does not provide APIs related to the connection

management such as `VipConnectRequest()`, `VipConnectAccept()`, `VipConnectWait()`, `VipConnectReject()`, and `VipDisconnectVi()`. It also lacks blocking send / receive APIs such as `VipSendWait()` and `VipRecvWait()`, and error handling ones.

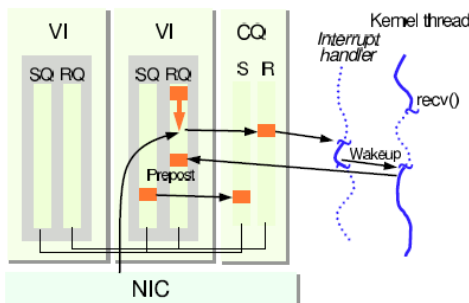
#### 3.3. VCONN layer

Modifying GNBD directly over the KVIPL layer requires a significant change in GNBD internals, because the



**Figure 5. Sender/receiver synchronization**

interface provided by the KVIPL layer is very primitive compared to the Sockets used by GNBD. Instead, we introduce an intermediate layer called VCONN (VIA Connection), which emulates Sockets interface over the KVIPL layer. As a normal kernel-level Sockets-like interface is provided by the VCONN layer, we can minimize the code modification in GNBD. We discuss several issues in implementing the VCONN layer in this subsection. Our goal is to make the VCONN layer as efficient as possible so that the performance of native VIA can be delivered to the upper kernel-level applications such as GNBD/VIA.



**Figure 6. Message handling**

**Message handling.** When a message arrives at a node, the corresponding descriptor should be extracted from a queue and an appropriate action needs to be taken. Normally, the arrival of an asynchronous message is not au- **Figure 6.**

### Message handling

tomatically notified to the application in VIA. As cLAN's implementation of VIA does not support asynchronous notification either, we should invent a scheme to handle the asynchronous messages efficiently. In the kernel-level VCONN layer, it is possible to run a specific code upon the

completion of a descriptor by registering an interrupt handler. When a message arrives, the cLAN hardware issues an interrupt and the corresponding interrupt handler is activated. The handler checks the Completion Queue (CQ), dequeues the entry in CQ and Work Queue, and then wakes up a kernel thread which is waiting for a new message. The kernel thread preposts a descriptor for receiving another message, as depicted in figure 6.

Therefore, we design and develop a very simple but decent flow control method tuned to GNBD communications. VCONN supports a flow control mechanism similar to the TCP's sliding window protocol [10] by extending the twoway handshaking shown in figure 5. Our implementation of VCONN also has the notion of window size  $w$ , which denotes the maximum number of messages the sender is allowed to transmit without waiting for an acknowledgment. Initially, the receiver preposts  $w$  descriptors to RQ. Whenever the sender transmits a (REQ, DATA), it decreases  $w$ , which means that one of the preposted descriptors on the receiving end has been consumed.

## IV. EVALUATION

### 4.1. Evaluation Methodology

The hardware platform used for performance evaluation is two Linux servers running Linux kernel 2.2.18 patched with UDMA-enabling and GFS-enabling code. Each server consists of Pentium III-1GHz microprocessor with 256KB of L2 cache, 512MB of main memory, UDMA100 hard disks and an on-board Intel EtherExpress 10/100 FastEthernet adapter. Additionally, a cLAN1000 adapter has been installed to the 32-bit 33MHz PCI slot of each server. The cLAN1000 adapters are connected in a back-to-back topology without any intermediate switch. To evaluate GNBD performance, we have measured the file read and write bandwidth after mounting EXT2 and GFS file systems on GNBD, varying the transport layers:

LANEVI and VCONN on cLAN, and TCP/IP on 100MbpsFastEthernet (FE for short). We have used `bonnie++` [7] to measure the file system performance.

### 4.2. Performance

**Basic Performance.** First, we report the latency of LANEVI and VCONN in table 1. The message sizes, 22 bytes and 4096 bytes, are the most frequently used sizes in GNBD if a block size is 4096 bytes. The 22 byte-long message is used to send a request or receive a response, and 4096 byte-long message to send or receive a block itself. As can be seen in

table 1, the latency of VCONN is far better than that of LANEVI, meaning VCONN is able to work faster and more efficiently than LANEVI. Note that although the latency is very important for single block read/write, it can be hidden by pipelining the requests and responses.

**EXT2/GNBD performance.** Figure 7 shows the performance of EXT2 file system over GNBD. Looking at the performance results, we can see that GNBD/VCONN results in higher read/write bandwidth than GNBD/LANEVI and GNBD/FE. The read bandwidth of EXT2/GNBD/VCONN (30.6MB/s) reaches nearly the local EXT2 read bandwidth (32.5MB/s) in figure 7(a), while the write bandwidth (25.9MB/s) is far lower than the local bandwidth (38.1MB/s). We observe that the followings are responsible for the degradation in the write bandwidth of EXT2 file system over GNBD:

**Synchronous I/O:** the GNBD server intentionally opens a file for synchronous I/O and then exports it to the clients. Turning on the synchronous I/O mode reduces the write bandwidth significantly because the buffering system is not utilized. For example, a block device with synchronous I/O mode shows the bandwidth around 30MB/s, while the normal bandwidth is about 41MB/s.

**No block clustering:** the kernel combines multiple small I/O operations into a larger single I/O operation in order to decrease the number of read/write requests to disk. However, his block clustering is not enabled for the virtual device created

by the GNBD layer on the client side. **Dual buffering:** a block should be written in the buffer memory of the client and also in the memory of the server before being flushed into a physical disk.

The graphs shown in figure 7(a) are obtained when the server exports a single, large file to the clients. Alternatively, the server can also export a raw partition to clients and figure 7(b) plots the performance of EXT2 file system in this case. First, we see that the write bandwidth is slightly improved when compared to the results of the file-exported case in figure 7(a). The improvement in write bandwidth is caused by the simplicity in writing mechanism of the block device: when a block is written by the client, the server stores nothing but the requested block. When a file is exported, however, the server also needs to touch other meta-data (for example, an i-node block for the exported file) in the server's own file system.

**GFS/GNBD performance.** Figure 8 shows the performances of GFS over GNBD. Generally, the write performance of GFS/GNBD is slightly lower than that of EXT2/ GNBD due to the overhead of GFS itself such as journaling.

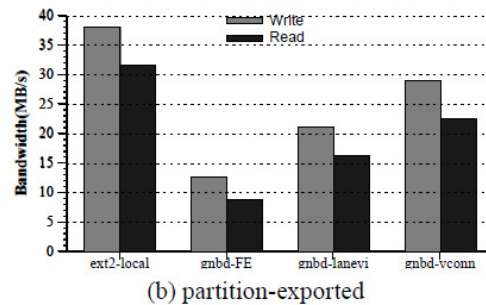


Figure 7. The performance of EXT2 file system over GNBD

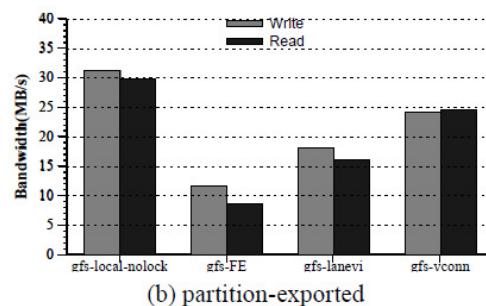
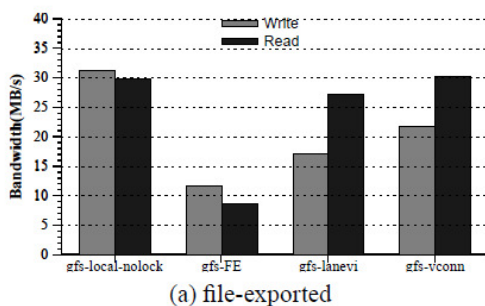


Figure 8. The performance of GFS file system over GNBD

## V. CONCLUSION

In this paper, we present a design and implementation of GNBD/VIA, a Network Block Device (NBD) over Virtual Interface Architecture (VIA). First, we have extended the KVIPL layer included in the VIA driver of Emulex cLAN adapters so that it supports the same set of APIs as VIPL in the kernel. And then we have developed an intermediate layer called VCONN which provides a set of kernel-level Sockets-like interfaces over KVIPL. Using the VCONN layer, we can minimize the code modification in GNBD, while maximizing its performance.

Our measurement results show that GNBD/VIA outperforms the normal NBD placed on top of TCP/IP protocol stacks, and realizes the performance comparable to local disk devices, showing the read and write bandwidth of 30.6MB/s and 25.9MB/s on cLAN, respectively.

We plan to extend the VCONN layer to a general kernellevel Sockets layer over VIA in the near future.

## REFERENCES

- [1] Compaq Computer Corp., Intel Corp., and Microsoft Corp. Virtual Interface Architecture Specification Draft Revision 1.0. <http://www.viarch.org/>, Dec. 1997.
- [2] K. Duncan. Fibre Channel and Gigabit Ethernet: A Look atTechnology for Storage Networking Solutions. Fibre channel group, University of Minnesota, 2001.
- [3] Gigaset Inc. *cLAN for Linux: Software user's Guide*, 2001.
- [4] J.-S. Kim, K. Kim, and S.-I. Jung. SOVIA: A User-level Sockets Layer over Virtual Interface Architecture. In *Proceedings of the 3rd IEEE Int'l Conf. on Cluster Computing*, pages 399–408, Oct. 2001.
- [5] National Energy Research Scientific Computing Center. MVICH: MPI for Virtual Interface Architecture. <http://www.nersc.gov/research/FTG/mvich/>.
- [6] National Energy Research Scientific Computing Center. M-VIA: A High Performance Modular VIA for Linux. <http://www.nersc.gov/research/FTG/via>, 1999.
- [7] R. Coker. Bonnie++ Project. <http://www.coker.com.au/bonnie++/>.
- [8] M. Rangarajan and L. Iftode. Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance. Technical report, Department of Computer Science, Rutgers University, 2000.
- [9] Sistina Software, Inc. Global File System. <http://www.sistina.com>.
- [10] W. R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.