VIA를 이용한 네트웍 블록 디바이스

김강호° 김진수 정성인
한국전자통신연구원 리눅스연구팀
{khk, jinsookim, sijung}@etri.re.kr

The Network Block Device Using the VIA

Kangho Kim° Jin-Soo Kim Sungin Jung
Linux Research Team, ETRI

요 약

VIA는 클러스터 또는 시스템 영역 네트워크를 위한 표준화된 사용자수준 통신 아키텍쳐이고, GFS는 LINUX 클러스터에서 사용할 수 있는 공유 파일 시스템이다. 클러스터 환경에서 GFS를 사용할 때 특별한 스토리지 네트워크가 설치되어 있지 않으면 GNBD를 사용한다. GNBD는 TCP/IP 상의 소켓을 기반으로 구현되어 있기 때문에, VIA를 사용하는 클러스터이더라도 VIA 하드웨어 상에서 TCP/IP 소켓을 통하여 GNBD를 작동시킨다. VIA와 같이 물리적 연결이 신뢰성이 높고 높은 수준의 기능을 제공하는 경우는 같은 클러스터 안에서 TCP/IP 프로토콜 스택을 사용할 필요가 없다. 본 논문은 VIA상에서 GNBD를 위한 고속 통신 계층(VCONN)을 제안하여, 동일한 VIA 하드웨어에서 지원되는 TCP/IP 모듈을 사용했을 때보다 읽기, 쓰기 성능을 각각 약 22%, 30% 향상시키는 방법을 소개한다.

1. 서론

표준화된 하드웨어 부품으로 이루어진 클러스터 시스템이 점차 고성능 계산과 확장성 가능한 인터넷 서버의 기준으로 받아들여지고 있다[9, 5]. 통신 성능을 향상시키기 위하여 많은 클러스터 시스템이 미리넷(Myrinet), 기가비트 이더넷(Gigabit Ethernet), SCI와 같이 기가비트 이상의 속도를 지원하는 시스템 영역 네트워크(System Area Network, SAN)을 도입한다. 그러나 네트워크 하드웨어의 처리속도가 빨라질수록 메시지를 보내기 위한 소프트웨어의 오버헤드가 전체시간에서 많은 부분을 차지하게 된다. 특히 TCP/IP 프로토콜을 기반으로 한 전통적인 통신 구조는 프로토콜의 오버헤드, 문맥교환 오버헤드, 자료복사 오버헤드 때문에 하드웨어의 성능을 제대로 이용할 수 없다고 알려져 있다[6]. 그 결과 운영체제가 메시지 전송에 관여하지 않는 여러가지 사용자 수준의 통신 아키텍쳐가 제안되었다. VIA는 Compaq, Intel, Microsoft가 사용자수준 통신 아키텍쳐를 표준화하기 위해서 제안한 것이다.

새로운 네트워킹 기술이 등장함에 따라 저장 매체를 여러 노드가 공유할 수 있게 되었다. 공유된 저장 매체에 기록된 파일을 여러 노드가 동시에 접근할 수 있도록 만들어 주는 것이 공유 디스크 파일 시스템이다. GFS는 LINUX에서 사용할 수 있는 공유 클러스터 파일 시스템이다. GFS 클러스터 내의 노드들은 동일한 저장매체를 Fiber Channel, 공유 SCSI, 또는 네트워크 블록 디바이스(Network Block Device)를 사용하여 물리적으로 공유한다[1].

FC 또는 공유 SCSI와 같은 별도의 스토리지 영역 네트워크(Storage Area Network)을 갖추지 않은 클러스터 환경에서 GFS를 사용하려면 GFS가 제공하는 NBD(GNBD)를 사용해야 한다. GNBD를 포함한, 일반적인 NBD는 TCP/IP 계층 위의 소켓을 사용하여 구현되어 있기 때문에 전통적인 LAN 또는 WAN환경에 적합하지만 고성능 네트워크를 구비한 클러스트 환경에 그대로 적용하는 것은 적합하지 않다. 특히 VIA와 같이 물리적 연결이 신뢰성이 있고 높은 수준의 기능을 제공한다면 같은 클러스터 안에서 TCP/IP 프로토콜 스택을 사용하여 통신할 필요가 없다.

본 논문은 VIA[8]상에서 GNBD를 사용할 때, TCP/IP 스택을 사용하지 않는 간단한 커널 소켓 API를 구현하여 GNBD 인터페이스는 유지하면서 VIA 하드웨어의 성능을 최대한 이용할 수 있는 기법을 소개한다.

본 논문의 구조는 다음과 같다. 2절에서 VIA와 GNBD를 간단히 설명하고, 3장에서 VIA상에서 GNBD를 위한 통신 모듈을 설계할 때 고려한 요소들을 설명한다. 4장은 구현한 통신 모듈을 GNBD에 적용하여 얻은 성능측정 결과를 보이고, 5장에서 결론을 맺는다.

2. 배경

2.1 Virtual Interface Architecture (VIA)

VIA는 네 개의 기본 요소, 즉 가상 인터페이스 (VI: Virtual Interface), 완료 큐, VI-제공자, VI-수요자로 구성된다. VI-제공자는 물리적인 네트웍 어댑터와 커널 에이전트 및 라이브러리를 포함하며, VI-수요자는 VIA를 이용하는 응용 프로그램을 말한다.

일반적으로 노드상의 여러 프로세스는 하나의 네트웍

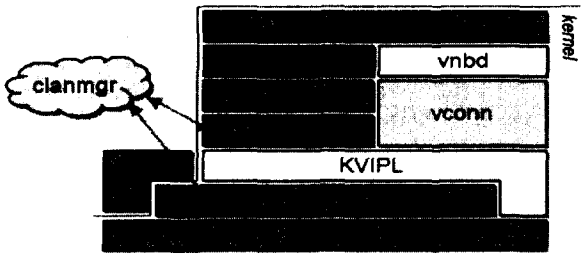
어댑터를 공유하지만, VIA에서는 각각의 VI-수요자에게 가상으로 하나의 전용 어댑터가 할당되어 있는 것과 같이 보이도록 지원한다. VI는 송신 큐와 수신 큐의 쌍으로 이루어지며, VI-수요자는 전달할 또는 받을 메시지에 대한 정보를 담고 있는 디스크립터(descriptor)를 큐에 추가함으로써 데이터를 교환한다. 한편, 여러 개의 송수신 큐를 하나의 완료 큐(CQ: Completion Queue)와 연결시킴으로써, 매번 모든 큐를 조사할 필요가 없이 완료 큐를 통해 완료된 디스크립터를 처리할 수 있다.

현재 사용할 LINUX에서 사용할 수 있는 VIA구현은 MVIA[8], Berkeley VIA[7], cLAN[3]이 있다.

2.2 GNBD

NBD는 리눅스 커널에 포함된, 기본적인 네트워크 블록 디바이스로서 클라이언트가 다른 노드의 자원을 블록 디바이스처럼 사용하고자 할 때 필요하다. NBD는 클라이언트에게 마치 로칼 디스크인 것처럼 보이도록 만들어준다. 이것은 NFS와 같은 분산 파일 시스템 서비스보다 좀 더 낮은 수준이고 기본적인 것으로 NFS보다 적은 커널 작업이 필요하다.

GNBD는 GFS에 포함된 네트워크 블록 디바이스로서 위에서 설명한 NBD를 변형한 것이다. NBD와 GNBD의 가장 큰 차이점은 NBD가 한번에 오직 하나의 클라이언트에게 블록 디바이스 접근을 허용하는 반면에 GNBD는 동시에 여러 클라이언트에게 접근을 허용한다는 것이다 [4].



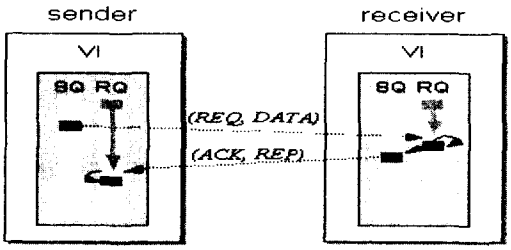
[그림 1] VIA를 이용한 GFS와 GNBD의 서비스 구조

3. GNBD/VIA 설계

3.1 KVIPL

GNBD를 VNBD로 수정할 때, VIA 커널 에이전트와 VIA 커널 프로그래밍 라이브러리(KVIPL)를 사용하기로 결정했다. 그 이유는 GNBD 드라이버가 커널에 구현되어 있고, 가능하다면 GNBD의 구조를 손상시키지 않고 VIA의 성능을 사용할 수 있는 가장 간단한 방법이라고 생각했기 때문이다. GNBD의 요구(request)를 처리하는 사용자 수준 프로세스를 사용하는 방법이 있으나 버퍼의 추가 복사와 다른 부담으로 성능을 발휘하지 못 할 수 있다. 서버 측을 고려할 때에도 커널 수준에서 구현되는 것이 더욱 유리하다. 파일 입출력을 수행할 때마다 커널 상태로 이전하지 않아도 되고, LINUX에서 2G바이트보다 큰 크기의 디스크를 서비스할 수 있다.

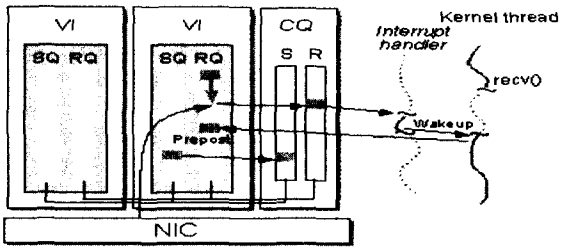
CLAN1000 드라이버가 커널 수준의 VIPL을 제공하고 있으나, 하나의 독립 API로 지원하는 것이 아니라 LANEVI를 구현하기 위한 서브 모듈이어서 완전한 API을 갖추지 않았다. 이 문제를 해결하기 위하여 KVIPL을 확장하여 부족한 기능을 구현하였다.



[그림 2] 송수신 동기화

3.2 흐름제어

GNBD의 통신 패턴을 분석한 결과, 복잡한 흐름제어를 만들지 않아도 될 만큼 통신 패턴이 간단하고, 통신 함수 호출 순서가 일정하다는 것을 알아냈다. 그래서 우리는 간단하지만, GNBD통신에 적용하는데 적합한 흐름제어를 설계하였다. Send()와 recv()를 동기화 시키기 위해서 기본적으로 그림2처럼 2-way handshaking 기법을 사용한다. 수신측은 수신큐에 미리 한 개의 디스크립터를 두어서 recv()함수 호출과는 비동기적으로 도착하는 메시지를 수신한다. 메시지 수신 후 ACK을 보내기 전에 소비한 디스크립터를 보충한다. 수신된 메시지는 버퍼에 저장했다가 recv()가 호출될 때 전달한다. 송신측은 ACK을 받았을 때 다음 메시지를 전달할 수 있다. 기본 구조에 추가하여, 송신측이 ACK을 받지 않고 N개의 메시지를 연속으로 보낼 수 있게 하였다. N은 수신측의 수신큐에 미리 포스팅된 디스크립터의 개수이다. 이렇게 하면 요청을 겹쳐서 보낼 수 있게 된다.



[그림 3] 메시지 처리

3.3 메시지 핸들링

커널 모드에서 수신 메시지 처리는, 그림3에서 볼 수 있듯이, 하드웨어 인터럽트 핸들러(hardware interrupt handler)가 담당한다. 메시지가 도착했을 때 인터럽트 핸들러가 동작하여 해당 디스크립터를 뽑아내고, 커널 쓰레드에게 메시지를 전달하고, 대기큐에서 쉬고 있는 커널 쓰레드를 깨운다. 그 때 커널 쓰레드는 메시지를 수신하고, 다음 메시지 수신을 위해서 한 개의 디스크립터를 수신 큐에 미리 저장한다.

인터럽터 핸들러는 LINUX가 지원하는 bottom half[10]를 이용하여 구현하였다. 기존의 방법을 사용하면 인터럽트 핸들러를 수행하는 시간보다 메시지가 도착하는 시간이 더 작은 경우에 메시지가 큐에 도착했으나 핸들러

면 프로세스는 준비 상태가 되어 프로세스 레디 큐에 들어가게 된다.

3.3 시간성 튜플과 관련된 프리미티브들

시간성 튜플에 접근할 수 있는 프리미티브에는 in_timed()와 out_timed()라는 두 가지가 있다. in_timed() 프리미티브는 기존의 시스템 콜 sleep(), wait()과 같은 역할을 한다. 일정한 시간을 정하여 프로세스를 슬립 상태로 놓을 수 있으며, 정해진 시간이 지나면 스스로 깨어나 대기 상태에 들어가고 오류 메시지를 발생시킨다. 이 프리미티브의 형식은 in_timed(int key, int time) 이다. 이 프리미티브를 호출한 프로세스는 key에 매핑되는 시간성 튜플을 찾아 counter 필드에 time 값을 복사하고 수면 상태로 들어간다.

out_timed(int key) 프리미티브는 기존의 시스템 콜 signal()과 같이 남은 수면시간에 상관없이 지정된 프로세스를 깨워서 대기 상태에 집어넣는다. 이때 rtn_value 필드에는 OK 값이 들어가고 counter 필드는 0 으로 리셋되며 이 튜플의 키와 관련된 모든 수면 상태의 프로세스는 대기 상태에 들어간다. 그리고 counter 필드의 값이 이미 0 이면 rtn_value 필드에 NOK(=0) 값이 들어가고 오류 메시지를 발생시킨다. 결국 슬립 상태의 프로세스는 스스로 깨어나거나 out_timed() 프리미티브에 의해 깨어나므로 프로그램이 데드락에 빠질 가능성을 일부 배제할 수 있다. 그리고 in_timed()와 out_timed()가 잘 동기화되지 못하였을 경우에는 두 프리미티브를 호출한 프로세스 각각에게 오류 메시지를 보내게 된다.

4. 시간성 튜플의 장점

4.1 구조의 단순성

앞에서도 언급하였듯이 기존의 튜플을 이용하여 데이터의 교환과 프로세스 동기화를 모두 구현하기 위해서는 튜플에 데이터 필드와 카운터 필드를 같이 갖고 있어야 한다. 따라서 많은 공간의 낭비가 있을 수 있고, 또한 튜플에 접근할 수 있는 프리미티브들이 데이터의 교환을 위한 것인지, 동기화를 위한 것인지 구분될 수 있어야 하는데 이것은 사용자에게는 단순하지 못하여 사용상 실수를 유발시킬 수도 있다. 또한 접근 알고리즘도 각각의 경우를 모두 포함하고 있어야 하므로 복잡하게 된다.

4.2 처리 속도

하나의 튜플에 두 가지 목적을 위한 필드가 같이 존재하면 하나의 목적을 위한 오퍼레이션을 수행할 때 다른 오퍼레이션을 위한 필드도 읽히거나 어떤 특정한 값이 모든 튜플에 쓰일 수 있기 때문에 시간성 튜플을 따로 사용할 때 보다 많은 처리 시간이 요구된다. 앞의 3.1에서도 언급하였듯이 카운터 필드는 시스템 tic이 발생할 때 마다 1씩 줄여가야 하는데 이러한 오퍼레이션은 데이터 교환을 위한 튜플에서는 전혀 사용되지 않기 때문에 컴퓨팅 파워를 낭비하는 결과를 초래한다. 시간성 튜플을 도입함으로써 결국 필요 없는 처리시간을 줄일 수 있다.

5. 결론

본 논문에서는 LINDA개념을 커널에 도입하여 프로세스간 동기화를 구현하기 위해서 필요한 것들을 고찰하였다. 기존의 튜플을 그대로 사용했을 경우 커널 공간을 많이 사용하게되고, 튜플의 구조나 접근 프리미티브 알고리즘이 복잡해 질 수 있고, 필요 없는 오퍼레이션으로 컴퓨팅 파워를 낭비하는 문제가 있었다. 이러한 문제점을 극복하기 위하여 시간성 튜플을 도입하였고 이러한 도입을 통하여 커널 공간 사용을 줄일 수 있었으며, 간단한 구조의 튜플을 구성하여 처리 알고리즘을 단순화 할 수 있었다. 또한 필요 없는 오퍼레이션이 없어지게 됨으로써 컴퓨팅 파워를 낭비하지 않게 할 수 있었다.

Reference

[1] S.Ahuja, N.Carriero, and D.Gelernter, "Linda and Friends", Computer, Vol. 19, No. 8, Aug. 1986.

[2] S.Ahuja, N.Carriero, D.Gelernter, and V.Krishnaswamy "Matching Language and hardware for Parallel Computation in the Linda Machine", IEEE Trans. on Comp., Vol.37, No.8, Aug. 1988.

[3] 박 영환 "실시간 커널 : RTLINDA", 추계학술발표회 논문집, 정보과학회, 제 25권 2호, 59-61 pp, 1998.

[4] E.Yao, B.Jardin, Y.H.Park, and K.M.Hou, "Real-Time Multiprocessor Kernel : Hierarchical LINDA", 4th Int. Conf. on Signal Processing Application and Technology, Santa-Clara, California, USA, Oct, 1993.

[5] 박 영환 "병렬 프로그래밍 개념 LINDA의 간편화 ", 추계학술발표논문집, 한국정보과학회, 제 27권 2호, ,642-644 pp., 2000.