리눅스 클러스터를 위한 커널 수준 통신 모듈

(A Kernel-Level Communication Module for Linux Clusters)

박 동 식 † 박 성 용 †† 양 지 훈 †††
(Dong-Sik Park) (Sung-Yong Park) (Ji-Hoon Yang)

요 약 클러스터를 위한 기존의 커널 수준 통신 시스템들은 플랫폼에 종속적이고, 사용이 어렵거나 클러스터를 위한 다양한 기능을 제공하고 있지 못하다. 본 논문에서는 리눅스 클러스터 상에서 커널 수준의 어플리케이션 개발을 위한 통신 모듈인 KCCM(Kernel level Cluster Communication Module)을 설계하고 개발하였다. KCCM은 송수신(send/receive) 기반의 동기 통신과 원격 함수 호출(remote procedure call) 기반의 비동기 통신을 모두 지원하며 포팅 가능성을 고려하여 커널 소켓을 이용하여 구현되었다. 또한 TCP와 같은 연결 기반(connection oriented)의 시스템에서 발생할 수 있는 연결 상태의 장애를 복구할 수 있는 기능을 갖춘 동시에 사용하기 쉬운 인터페이스를 가지도록 설계되었다. 본 논문에서는 실험을 통하여 KCCM의 성능을 RPC(Remote Procedure Call)와 비교해 보았으며 특히 비동기 통신을 요구하는 통신 구조에서 적합함을 보였다.

키워드 : 통신 시스템, 리눅스 클러스터

Abstract Traditional kernel-level communication systems for clusters are dependent upon computing platforms. Futhermore, they are not easy to use and do not provide various functions for clusters. This paper presents an architecture and various implementation issues of a kernel-level communication system, KCCM(Kernel level Cluster Communication Module), for linux cluster. The KCCM provides asynchronous communication services as well as standard synchronous communication services using send and receive. The KCCM also automatically detects and recovers connection failures at runtime. This allows programmers to use KCCM when they build mission critical applications over TCP-based connection-oriented communication environments. Having developed using standard socket interfaces, it can be easily ported to various platforms. The experimental results show that the KCCM provides good performance for asynchronous communication patterns.

Key words : Communication System, Linux Cluster

1. 서 론

1980년대의 수퍼컴퓨팅(supercomputing)적 문제 해결 방식은, 가장 빠르고 가장 효율적인 프로세서(processor)가 가장 높은 성능을 발휘할 수 있다는 논리에서 출발하였다. 둘 이상의 컴퓨터를 동시에 이용해 주어진 문제를 해결하는 병렬처리(parallel processing) 개념의 등장은 새로운 문제 해결 환경의 등장을 예고하였고, 1990년대 초부터 저렴한 가격과 높은 성능을 동시에 갖춘 범용 통신 장치들이 개발됨으로서 MPP (Massively Parallel Processor)등 주류를 이루던 수퍼컴퓨팅 장치들에 비해 상대적으로 저가인 워크스테이션이나 개인용 컴퓨터를 이용하여 제작된 클러스터가 비중 있게 수퍼컴퓨터 시장에 등장하게 되었다[1,2]. 그러나 클러스터가 그 자체로 MPP 등이 제공하는 수퍼컴퓨팅적 문제 해결

· 본 연구는 2002년도 산업자원부 차세대신기술개발사업 및 서강대학교 산업기술연구소 지원으로 이루어졌음.
† 비 회 원 : (주) 엔버전스
bilbo@dcclab.sogang.ac.kr
†† 종신회원 : 서강대학교 컴퓨터학과 교수
parksy@ccs.sogang.ac.kr
††† 비 회 원 : 서강대학교 컴퓨터학과 교수
jhyang@ccs.sogang.ac.kr
논문접수 : 2002년 11월 6일
심사완료 : 2002년 3월 11일

환경을 대신할 수 있는 것은 아니다. 클러스터는 두 개 이상의 워크스테이션이나 개인용 컴퓨터를 이용해 제작되며, 따라서 이러한 두 개 이상의 노드를 유기적으로 결합시켜줄 소프트웨어적 계층이 그 위에 놓여져야 한다. 그렇게 함으로서 클러스터를 구성하고 있는 노드들이, MPP를 이루고 있는 두 개 이상의 프로세서가 서로 연동되듯이 유기적으로 움직여 사용자로 하여금 하나의 수퍼컴퓨터 앞에 앉아 있는 것과 마찬가지인 환경을 제공하게 된다.

클러스터를 이용한 문제 해결 환경을 구축하기 위해서는 구성 노드간의 통신 메커니즘이 필요하기 때문에 낮은 가격의 워크스테이션과 개인용 컴퓨터로 만들어진 클러스터에서 효율적인 통신 방식에 관한 연구는 항상 중요한 연구 쟁점이 되어 왔다[3,4,5,6,7,8,9,10]. 하지만 이러한 연구들은 대부분 이식성을 위해서 사용자 수준의 미들웨어를(예: MPI[4], PVM[5], CORBA[9], RMI[10], 소켓 등) 통하여 구현되어 왔거나 또는 고성능만을 고려하여 특정 플랫폼에만 동작하도록 구현되어 있어서[3,6,7,8], 범용의 커널 수준 응용 프로그램 개발에는 사용될 수 없다는 단점이 있다. 커널에도 범용으로 사용될 수 있는 커널 소켓이나 RPC(Remote Procedure Call)와 같은 통신 방법이 제공되지만, 인터페이스 자체가 커널 프로그램을 위해서 공식적으로 제공되는 것이 아니기 때문에 사용이 어렵다.

본 논문에서는 리눅스 클러스터 상에서 커널 수준의 어플리케이션 개발을 위한 통신 모듈로서 쉬운 인터페이스를 제공하여 사용이 용이하고, 다양한 통신 패러다임과 연결 복구 기능을 갖는 통신 시스템인 KCCM (Kernel-level Cluster Communication Module)의 설계 및 구현에 대하여 기술한다. KCCM은 송수신(send/receive) 기반의 동기 통신과 원격 함수 호출(remote procedure call) 기반의 비동기 통신을 모두 지원하여 다양한 통신 패러다임을 요구하는 클러스터에 적합하도록 설계되었다. 포팅 가능성을 고려하여 커널 소켓을 이용하여 구현되었고, 사용하기 쉬운 인터페이스를 제공한다. 또한 TCP와 같은 연결 기반(connection oriented)의 시스템에서 발생할 수 있는 연결 상태의 장애를 복구할 수 있는 기능을 갖추고 있기 때문에 고 가용성의 클러스터를 구현하는데 유용하게 사용될 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 기존의 커널 수준 통신 시스템들의 특징들을 살펴보고 그 문제점을 알아보며, 3장에서는 KCCM의 특징 및 상세한 구조와 구현 내용을 기술한다. 4장에서는 KCCM과 RPC와의 비교를 통하여 KCCM의 성능을 평가하고, 마지막으로 5장에서는 결론 및 향후 과제에 대하여 기술한다.

2. 관련 연구

커널 수준의 통신을 위해 사용되는 방법은 크게 두 가지로 나누어진다. 디바이스 드라이버의 직접적인 수정을 통해 높은 성능을 가지고 있는 고성능 통신 시스템을 채택하여 그 시스템의 공개된 인터페이스로 통신에 접근하는 방법(예: AM(Active Message)[3], FM(Fast Message)[6], U-Net[7], VIA[8] 등)과, 커널 소켓이나 RPC 등 기존 커널 시스템에서 기본적으로 제공하는 커널 수준 통신 인터페이스를 사용하는 방법이다.

2.1 고성능 통신 시스템

AM(Active Message)[3]은 병렬 분산 시스템에서 널리 쓰이는 메시지 기반의 통신 시스템이다. 각각의 메시지는 처리할 핸들러가 표시되어 있어서 메시지가 도착하면 핸들러가 메시지를 처리한다. 메시지가 도착하면 처리 할 때까지 따로 버퍼링을 하지 않고 도착과 동시에 핸들러가 처리하기 때문에 버퍼링으로 인한 오버헤드가 줄어드는 장점이 있으며 비동기적으로 핸들러가 수행되기 때문에 병렬화로 인한 장점을 얻을 수 있다. 그러나 AM은 디바이스 드라이버의 수정을 통하여 구현되어 있기 때문에 드라이버에서 지원하지 않으면 사용할 수 없는 문제가 있다. 디바이스 드라이버는 하드웨어마다 다르게 구현되어야 하며 하드웨어마다 특성도 다르고 조작 방법도 다르기 때문에 통신 시스템에서 제공해야 하는 요구사항을 하드웨어에서 제공하지 않을 경우 소프트웨어적으로 디바이스 드라이버에서 구현해야 한다. 리눅스용으로 개발된 액티브 메시지인 GAMMA [11]는 현재 이더넷 드라이버로 3Com509b, DEC, EtherExpress만을 지원하고 있다.

FM(Fast Message)[6]은 기가비트 인터페이스인 미리넷(Myrinet)[12]을 위한 저 수준 통신 인터페이스로, 프로그래밍 인터페이스는 AM과 유사하지만 미리넷이라는 특별한 하드웨어를 고려하고 설계되었기 때문에 다른 시스템에서는 사용할 수 없다는 한계가 있으며, 오류가 거의 발생하지 않는 미리넷의 특성을 이용하여 오류에 대한 처리나 장애 상황에 대한 고려가 되어있지 않다는 문제점을 안고 있다.

또한 U-Net[7]이나 VIA[8]와 같이 사용자 수준에서 네트워크 하드웨어를 직접 접근하여, 전통적인 통신 시스템에서 발생되는 프로토콜의 오버헤드나 버퍼링의 오버헤드를 제거하기 위한 연구도 활발히 진행되고 있지만 이 또한 하드웨어에 종속적이기 때문에 범용적으로 사용되기는 어렵다.

2.2 커널 소켓

커널 소켓은 사용자 수준의 인터페이스인 소켓에서 제공하는 시스템 콜을 커널 중간에서 가로챈 인터페이스로 커널 내부에서 사용할 수 있다는 점을 제외하면 사용자 수준의 소켓의 사용과 같다. 사용자 수준에서는 소켓을 이용하려면 시스템 콜을 사용해서 접근해야 하기 때문에 소프트웨어 인터럽트 오버헤드가 있으며 소프트웨어의 흐름을 바꾸게 되지만 커널에서는 소켓 내부에서 블록킹(blocking)을 하지 않아서 커널의 제어를 잃어버리지 않는다는 점이 다르며, 커널 내부에서는 디스크립터로 *struct socket*을 직접 사용한다는 것이 다르다.

리눅스의 경우 커널에서의 소켓 인터페이스가 커널 프로그램을 위해서 공식적으로 제공되는 것은 아니지만 커널 기반 웹 서버인 khttpd[13]에서 이 인터페이스를 이용하고 있다. 유저 수준에서 사용하던 방법과 거의 같은 방법으로 사용할 수 있기 때문에 유저 수준에서 소켓으로 구현된 알고리즘을 커널 수준으로 옮길 때 쉽고 빠르다는 장점이 있다. 그러나 이 방법은 최소한의 접속 기능과 송수신 기능만을 제공하고 있으며 동기식 통신만을 위해 설계되어 있어서 비동기 통신을 필요로 할 경우 별도의 구현이 필요하다. 또한 일대일 통신만을 제공하기 때문에 클러스터에서 범용으로 사용하기 위한 통신 시스템으로는 적합하지 않다.

2.3 원격 프로시저 호출(RPC)

원격 프로시저 호출은 소켓이 기본적인 데이타를 주고받는 방법만을 지원하기 때문에 보다 높은 수준의 프로그램 작업을 위해 함수 호출 형태로 네트워크 자원을 이용할 수 있는 인터페이스로 개발되었다. 커널에서 NFS (Network File System)를 제공하는 시스템들은 NFS가 내부적으로 SUNRPC(SUN's Remote Procedure Call)를 사용하기 때문에 파일 공유를 목적으로 SUNRPC 코드가 포함되어 있다. 리눅스의 경우 소스가 공개되어 있기 때문에 커널 내부에 구현된 RPC를 접근할 수 있지만 유저 수준처럼 RPC 프로토콜 컴파일러가 없고 저수준의 RPC 인터페이스를 사용해야 하기 때문에 사용이 어렵다는 단점이 있다.

원격 프로시저 호출은 단순히 메시지를 전달하는 것이 아닌 원격지의 함수를 호출하는 높은 수준의 추상화를 제공하기 때문에 응용 프로그램의 제작이 쉽다. 그러나 동기적으로 동작하여 수행이 끝난 후에 리턴되기 때문에 병렬화의 잇점을 살리지 못한다는 단점도 지적되고 있다. 또한 원격 함수 호출은 관리하기 위한 인터페이스가 없기 때문에, 장애 발생 시 이에 대처하는 시스템을 구성할 수 없다는 한계가 있다.

3. KCCM의 설계 및 구현

3.1 KCCM의 개요

리눅스 클러스터를 위한 커널 수준 통신 시스템인 KCCM(Kernel-level Cluster Communication Module)은 인터넷 프로토콜(IP)을 기반으로 하는 퍼스널 컴퓨터 또는 워크 스테이션으로 구성된 클러스터에서 커널 수준의 응용 프로그램 작성과 서비스 제공을 위한 통신 시스템이다. KCCM은 커널 내부의 하나의 모듈로 존재하며 클러스터 시스템 커널에서 네트워크 환경을 이용하여 통신 서비스를 제공한다 (그림 1 참조). 프로그래밍 인터페이스는 함수의 이름을 커널에 공개해서 커널의 다른 부분이나 모듈들이 부를 수 있도록 하여 함수 호출로 서비스를 제공한다. 이것을 이용해서 커널에서는 공유 파일시스템 서비스(Shared File-system Service)나 광범위 프로세스 아이디 서비스(*Global Process Id* Service), 프로세스 이주 서비스(Process Migration *Service*) 등의 통신을 사용하는 커널 수준의 서비스를 개발할 수 있다. 다음은 KCCM의 기능을 요약한 것이다.

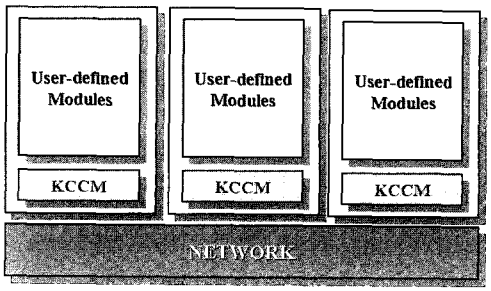

그림 1 클러스터 환경에서의 KCCM

3.1.1 동기 통신

클라이언트-서버 구조가 주류를 이루었던 기존의 통신 시스템은 대개 동기 통신(synchronous communication) 패러다임을 사용하고 있다. 동기 통신 모델에서는 두 프로세스가 통신을 한다고 할 때, 하나의 프로세스의 송신(send) 요청과 다른 하나의 프로세스의 수신(receive) 요청이 대응되어, 송신을 요청한 프로세스는 수신을 하는 프로세스가 그와 일치하는 수신 요청을 하였을 때까지 정지(block)하게 된다. 데이타의 수신측 프로세스가 송신측 프로세스의 데이타가 도착하는 것을 기다리는 것이다.

KCCM에서 동기 통신은 단순한 송신과 수신으로 이루어진다. 수신 측에서 수신을 시작하면 메시지가 도착

할 때까지 기다려 받아온다. 모듈번호와 시퀀스 번호를 키(key)로 하여 수신을 요청하면 이미 도착한 메시지가 존재하면 흐름을 멈추지 않고 받아오며 존재하지 않은 경우 수신을 요청한 커널의 흐름은 정지 상태로 도착을 기다린다. KCCM에서 동기 통신의 동작은 IPC(Inter Process Communication)의 메시지 큐와 거의 같은 방식으로 동작하도록 설계되었다.

3.1.2 비동기 통신

고성능 네트워크 인터페이스(high performance network interface)의 등장에도 불구하고 하드웨어적으로 엄격히 결합된(tightly coupled) 데이타 통신 채널을 갖춘 기존 SMP(Symmetric Multiprocessor) 시스템이나 MPP 시스템에 비해 LAN등의 네트워크에 통신을 의존하는 클러스터 시스템은 여전히 송수신 종단(sending/receiving ends) 사이의 지연시간이 클 수밖에 없다. 이러한 하드웨어적 한계 자체가 극복되어지기는 어렵기 때문에, 이를 극복하기 위한 연구는 소프트웨어 계층 차원에서 이루어져 왔다. 그 중 대표적으로 꼽을 수 있는 것이 통신과 연산(computation)을 중복(overlap)시키는 비동기 통신 수법이다.

KCCM에서 비동기 통신은 원격 함수 호출로 이루어진다. 비동기 통신은 원격지에서 미리 등록한 메시지 핸들러를 이용하여 메시지의 도착과 동시에 핸들러가 메시지를 처리하는 방식이다. KCCM에서는 원격 함수를 호출할 때 파라미터로 데이타를 보내는 것으로 비동기 통신을 구현하였다. 호출하는 측에서 데이타를 보내면 받는 측에서는 헤더를 받은 후 데이타의 길이만큼 받아 분석해서 핸들러의 파라미터로 넘긴다. 헤더에는 기본적인 송수신 노드 번호와 핸들러 번호를 포함하고 있으며 따라올 데이타의 타입과 길이정보 등이 포함된다. 핸들러는 사용의 편의를 위해서 가변길이의 아규먼트(variable argument)를 사용하고 있다.

3.1.3 이식성

이식성은 통신 시스템을 리눅스 환경에서 다른 여러 가지 하드웨어를 사용할 때 소스 수정 없이 사용하기 위한 측면과 다른 하드웨어 환경이나 운영체계를 사용하기 위해 소스수준에서 수정을 쉽게 하기 위한 측면으로 나눠볼 수 있다. KCCM은 다른 하드웨어 환경에서 문제없이 사용할 수 있도록 하기 위하여 커널에서 제공하는 기본 기능만을 이용하도록 하였으며 특히 통신을 위하여 TCP 소켓을 이용하였다. TCP 소켓을 이용할 경우 어떤 하드웨어든 드라이버만 제공되면 소프트웨어를 바꿀 필요 없이 통신 시스템을 그대로 사용할 수 있는 장점이 있는 반면 소켓 자체의 하드웨어의 성능을 그대로 사용하지 못하고 소켓 버퍼를 한번 거치는 오버헤드를 가진다는 단점이 있다.

한편 커널 수준의 응용 프로그램을 제작할 때 커널을 직접 수정하는 경우 커널의 모든 기능을 이용할 수 있기 때문에 프로그램이 자유스럽다는 장점이 있다. 그러나 커널을 수정하면 배포와 수정이 어려워지며 소스의 관리가 힘들어지는 단점이 있다. 본 시스템에서는 커널을 직접 수정하지 않고 모듈을 사용하며, 커널에서 외부에 공개한 인터페이스만을 사용하여 관리를 편하게 하였으며 배포를 쉽게 하였다.

3.1.4 접속 관리 및 복구

TCP를 이용하는 프로그램은 모든 노드와 통신을 하기 위해서 통신을 시작하기 전 초기화 과정에서 접속을 형성해두는 것이 필요하다. 또한 장애 발생이나 관리 목적으로 접속이 끊어지는 상황이 생길 수 있으므로 이런 상황에서 다시 통신을 시작하기 위해 접속을 복구하는 것이 필요하다. KCCM에서는 TCP 기반의 시스템에서 발생할 수 있는 연결 상태의 장애를 자동으로 검출하고 회복시킬 수 있는 기능을 포함하고 있다.

처음 통신을 위한 노드를 설정하는 과정이나 접속을 끊거나 접속이 끊긴 노드의 접속을 복구하는 과정은 클러스터 시스템의 관리와 밀접한 관계가 있다. 그래서 본 통신 시스템에서는 접속 관리와 복구를 위한 관리자를 통신 시스템 외부에 두어 통신 시스템의 관리를 클러스터 시스템의 관리와 함께 연동하여 할 수 있도록 하였다. 또, 관리를 위한 인터페이스를 공개하여 응용 프로그램이 통신 시스템의 상태에 따라 유기적으로 동작할 수 있도록 응용 프로그램 제작을 위한 관리자를 직접 제작할 수 있게 하였다.

3.2 응용 프로그램 인터페이스

KCCM은 접속 관리를 위한 응용 프로그래밍 인터페이스들과 일반적인 네트워크 응용 프로그램들 간의 동기 통신 방법인 송수신(Send/ Receive)을 위한 응용 프로그래밍 인터페이스들, 그리고 원격 함수 호출을 위한 응용 프로그램 인터페이스 등 크게 세 종류를 제공한다(표 1 참조).

KCCM에서는 기본적인 통신을 위해 TCP를 사용하기 때문에 처음 통신을 시작하기 위해 접속을 형성하는 것이 필요하다. 또, 사고나 관리 점검을 위해 접속을 끊고 다시 연결하는 상황이 있거나 새로운 노드의 추가로 새로운 접속이 생길 경우 접속을 맺어줄 필요가 있다. 이런 과정은 유저 수준의 클러스터 관리 시스템이나 장애 복구 시스템과 연동하면 보다 신뢰성 있는 시스템을 구성할 수 있기 때문에 외부에 접속을 위한 인터페이스

표 1 KCCM의 프로그래밍 인터페이스

종 류	API	설 명
접속관리를 위한 API	CCM_Close(); CCM_Listen_to(); CCM_Connect_to();	연결을 끊는다. 연결을 기다린다. 연결한다.
동기 통신을 위한 API	CCM_bsend(); CCM_brecv(); CCM_arecv();	메시지를 보낸다. 메시지를 받아온다. 기다리지 않고 도착한 메시지를 받아온다.
비동기 통신을 위한 API	CCM_rcall_l(); CCM_rcall_s(); CCM_regist(); CCM_unregist();	원격지의 함수를 쓰레드 형태로 호출한다. 원격지의 함수를 호출한다. 함수를 등록한다. 등록된 함수를 제거한다.

를 공개하는 것이 필요하다. 통신시스템이 적재된 상태에서 사용을 시작하거나 또는 사용을 중지하거나 관리 점검을 위해 노드의 상황이 변하는 상황을 클러스터 시스템의 관리 인터페이스를 통해 다른 클러스터 응용 프로그램과 함께 통신시스템을 관리 할 수 있다. 응용 프로그램 인터페이스는 접속을 기다리는 인터페이스인 *CCM_Listen_to()*와 접속을 하는 인터페이스인 *CCM_Connect_to()*, 그리고 접속을 끊어줄 때 사용하는 *CCM_Close()*로 이루어져 있다. 모두 노드의 논리적 번호를 입력으로 받아 접속을 관리하게 되어있다.

KCCM에서는 기본적인 동기식 통신을 위해 송신과 수신을 제공하고 있다. 각 메시지들은 노드 번호, 타입, 모듈 번호로 구분되며 각각의 메시지는 필요할 경우 시퀀스 번호로 구분할 수 있다. 메시지를 보내는 응용 프로그램 인터페이스는 *CCM_bsend()*이며 상대 노드 번호, 상대 모듈 번호로 메시지를 전달할 대상을 결정하게 되어 있으며 시퀀스 번호 필드를 필요에 따라 선택적으로 사용할 수 있다. 시퀀스 번호 필드는 메시지들 간의 구분이나 순서를 보장할 필요가 있을 때 사용할 수 있다. 메시지를 받는 인터페이스는 *CCM_brecv()*와 *CCM_arecv()*가 있다. *CCM_brecv()*는 도착한 메시지가 있는지 살펴본 후, 도착한 메시지가 있다면 그 메시지를 가지고 리턴되지만, 그렇지 않다면 대기큐(Wait Queue)에 들어가 메시지가 도착할 때까지 대기(sleep) 상태가 된다. *CCM_arecv()*는 도착하지 않은 메시지를 기다리지 않고 진행하기 위해 만들어진 인터페이스이다.

비동기 통신은 동기식 통신이 연산의 흐름을 멈추고 통신을 하기 위해 프로세스가 멈추는 단점을 피하기 위해 메시지에 대해 핸들러를 연결시켜 메시지가 도착하면 별도의 핸들러가 수행되는 방식이다. KCCM에서는 비동기 통신을 원격 함수 호출을 통해 제공하며 크게 2종류의 인터페이스를 제공한다. *CCM_rcall_s()*는 메시지 도착으로부터 메시지를 처리하는 핸들러가 수행되기까지의 시간을 최소로 하기 위해 메시지 수신시 직접 핸들러 함수를 호출한다. 하지만, 원격 함수 호출은 비동기 통신을 위해서도 쓰이지만 일반적인 RPC 호출처럼 단순한 프로그램의 편리를 위해서 사용할 수도 있다. 이런 경우 핸들러는 단순히 메시지를 받아서 처리하는 역할 뿐 아니라 입출력을 수행하거나 또는 핸들러 안에서 별도의 통신을 수행할 수도 있다. 따라서 메시지 수신시 함수로 핸들러를 호출하여 오랜 시간을 기다리도록 할 수 없기 때문에 원격 함수 호출을 위한 쓰레드를 생성시키고 이 쓰레드에서 핸들러를 수행시키는 *CCM_rcall_l()* 인터페이스를 제공한다.

3.3 KCCM의 구현

3.3.1 KCCM의 구조

그림 2는 KCCM의 구조를 보여주고 있다. KCCM은 리스너(listener) 쓰레드를 중심으로 동작하게 되어 있다. 리스너 쓰레드는 항상 메시지를 기다리는 상태를 유지하고 있으며 메시지 패킷이 도착하면 패킷을 분석하여 패킷의 종류에 따라 처리하고 다음 처리를 위해 다시 기다리는 구조로 되어 있다. 메시지의 종류는 크게 동기 통신 메시지와 비동기 통신 메시지로 구분될 수 있다. 동기 통신 메세지는 송수신을 위한 메시지로 모듈의 상황에 따라 큐잉(queuing)하거나, 해당하는 프로세스에 전달해준다. 비동기 통신 메시지는 원격 함수 호출을 위한 메시지로 메시지의 종류에 따라 짧은 호출(short)과 긴 호출(long)로 나누어 처리한다. 어떤 프로세스 또는 커널 흐름(flow)에서 데이타를 받기 위해 수신을 요청하면 이미 도착했는지를 확인하기 위해 수신큐(Recv_Queue)를 살펴본 후 도착하지 않았다면 대기상태에서 기다리게 되며 도착한 경우에는 수신 큐에서 가져간다. 대기 상태에서 기다리는 프로세스가 있다는 것을 리스너 쓰레드에 알리기 위해 별도의 큐(Recv_Wait_Queue)를 구성하여 메시지가 도착하면 곧바로 깨워서 전달할 수 있게 한다.

3.3.2 데이타 구조

(1) 메시지 구조

KCCM이 주고받는 메시지는 그림 3에서 보여지는 것처럼 크게 헤더 부분과 데이타 부분으로 나눠진다. 헤더에는 송수신 데이타에는 송신할 노드의 번호(src_id)와 수신할 노드의 번호(dst_id) 그리고 처리할 모듈번호와 타입(type), 시퀀스 번호(seqn)를 포함하고 있으며 보내는 데이타의 크기 정보(size)를 가지고 있다. 보내는 쪽에서는 항상 고정된 길이의 메시지 헤더를 보낸 후 가

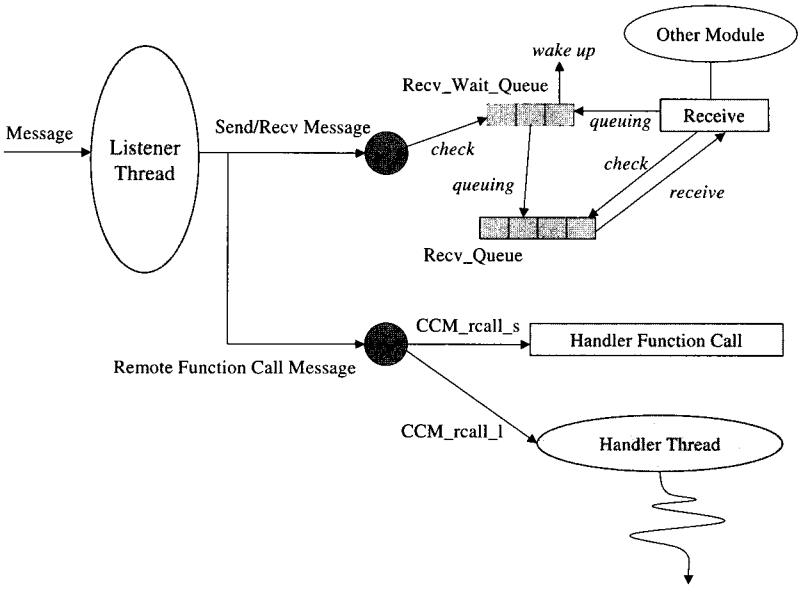

그림 2 KCCM의 동작 구조

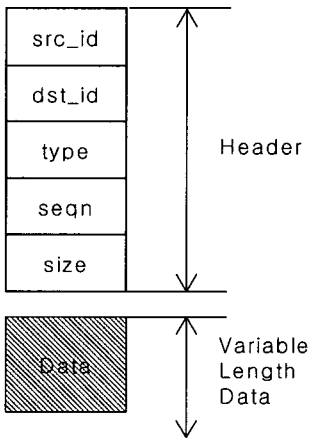

그림 3 메시지 구조

변 길이의 데이타를 보내며 받는 쪽에서는 헤더를 받은 후 데이타의 크기 정보필드에 따라 데이타 공간을 할당하여 받거나 이미 할당된 공간에 받는다.

(2) 수신 대기 큐

수신 대기 큐는 사용자가 수신 명령을 내렸을 때 아직 도착하지 않은 메시지의 도착을 기다리는 큐이다. 동기식 통신에서 사용자가 수신 명령을 내리면 메시지가 도착해 있으면 흐름을 멈추지 않고 받아오지만 메시지가 아직 도착하지 않은 경우에는 그 흐름을 멈추고 메시지가 도착하는 것을 기다려야 한다. 이 때에 지연시간이 짧은 효율적인 통신 시스템이 되기 위해서는 메시지의 도착으로부터 해당 프로세스에 전달되어 다시 진행하기까지의 시간이 짧고, 프로세스 점유가 작아야 한다. KCCM에서는 메시지가 도착하면 그 흐름이 바로 깨어날 수 있도록 커널 상에서 기다리는 프로세스를 (수신 명령을 내렸으나 메시지가 아직 도착하지 않은 경우) 위한 웨이트 큐(wait queue)를 생성하고, KCCM의 수신 대기 큐에 연결시켜 메시지가 도착하면 그 흐름에 해당하는 프로세스가 대기 상태에서 곧바로 깨어날 수 있도록 하였다. 또한, 도착한 메시지를 다른 임시 공간에 복사했다 다시 사용자가 넘겨준 공간에 복사하는 오버헤드를 피하기 위해 수신 대기 큐에 수신자의 버퍼 포인터를 가지고 있어 직접 그 공간에 메시지를 받을 수 있도록 하였다. 이렇게 되면 수신 대기 큐에 들어간 수신에 대해서는 불필요한 복사 오버헤드를 모두 피할 수 있다. 그림 4는 수신 대기 큐의 구조와 동작 방법을 보여주고 있다.

먼저 프로세스는 특정 타입의 메시지를 기다린다는 것을 적어 놓은 헤더를 가진 채(Header1, 2, 3 등) 대기 큐에 들어가 대기 상태가 된다. 그 후 메시지가 도착하면(그림 4 ①) KCCM은 메시지가 가지고 있는 헤더와 일치되는 헤더가 있는가 대기 큐를 검색한다(그림 4의 ②). 대기 큐에 그러한 프로세스가 없다면 메시지는 수신 데이타 큐로 들어가지만, 만약 일치되는 헤더가 있다

면 해당 큐 요소(element)에 있는 데이타 포인터에 도착한 메시지의 데이타를 매달고(그림 4의 ③) 신호를 주어 시스템 대기 큐에 있는 프로세스를 깨우게 된다(그림 4의 ④). 깨어난 프로세스는 도착된 데이타를 가지고 멈추어 있던 일을 계속한다.

(3) 수신 데이타 큐

수신 데이타 큐는 메시지가 도착했을 때 수신 대기중인 프로세스가 없을 경우 임시로 버퍼링을 하는 큐이다. 메시지를 받으면 헤더 정보와 함께 받은 메시지를 리스트 형태로 저장해두게 된다. 저장된 메시지는 사용자의 수신 요청이 있을 경우 적합한 송신 노드의 번호, 타입, 시퀀스 번호에 따라 사용자에게 데이타를 넘겨준다. 임시로 버퍼링을 하기 위해 받은 데이타를 사용자에게 전달하기 위해서는 사용자 공간으로 복사를 하는 오버헤드를 지닌다. KCCM에서는 임시로 저장하기 위해 만든 버퍼의 포인터를 직접 사용자에게 넘겨서 복사하는 오버헤드를 줄일 수 있게 하였다. 그러나 상황에 따라 사용자가 메시지를 위해 할당한 공간에 메시지를 다시 복사해야 하는 경우, 불필요한 프로그램 오버헤드만 증가시키기 때문에 사용자 자신이 지정한 수신 버퍼를 사용할 수 있는 옵션을 추가하였다. 그림 5는 수신 데이타 큐의 구조 및 동작 방법을 보여주고 있다.

먼저 그림 5의 ①처럼 메시지가 도착한다. 도착한 메시지는 그림 4에서처럼 그 메시지를 기다리는 프로세스가 있는가를 검사한 다음, 없다면 수신 데이타 큐에 들어간다(그림 5의 ②). 이때 저장되는 데이타는 카피를 줄이기 위해 도착한 메시지의 데이타를 포인터로 가리키게 된다(그림 5의 ③). 이때 만약 이 메시지를 받기를 원하는 프로세스가 있어서 수신 동작을 시작했다면(그림 5의 ④) 원하는 메시지의 타입을 이용하여 수신 데이타 큐를 검색한다(그림 5의 ⑤). 일치되는 메시지를 발견하면 이 그 메시지의 데이타가 수신 프로세스에게 전달되며, 프로세스는 주어진 일을 계속한다(그림 5의 ⑥). 만약 이 경우에 일치하는 타입 메시지를 발견하지 못했다면 프로세스는 원하는 메시지가 도착하지 않았다고 판단하고 자신을 수신 대기 큐에 매달고 대기 상태에 들어가게 된다.

3.3.3 커널 인터페이스

KCCM은 커널에서 구현되었기 때문에 커널에서 제공하는 함수들을 사용한다. 즉, 통신을 위해 소켓을 이용하기 때문에 커널에서 사용할 수 있는 소켓이 필요하고, 커널 상의 메모리 및 프로세스 관리를 위한 몇 가지 함수가 KCCM의 구현에 필요하다. 특히, 커널 소켓은 유저 수준에서의 소켓의 이용과 비슷하지만 사용하는 데이타 구조가 조금 다르고 사용할 수 있는 함수들이 다르다. 표 2는 KCCM의 구현에 사용된 커널 수준의 함수들을 정리해 놓았다. 이해를 돕기 위해 사용자 수준에서 사용되는 동일 함수(C 라이브러리)들을 함께 정리하였다.

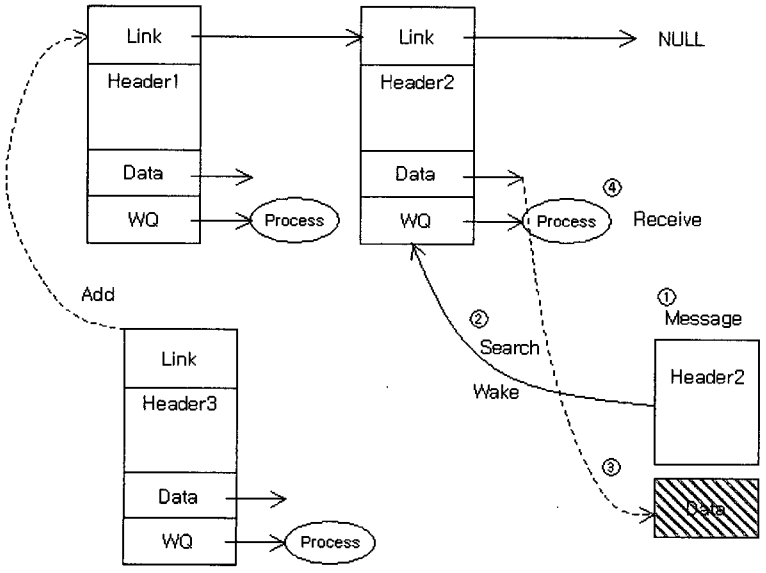

그림 4 수신 대기 큐의 구조와 동작방법

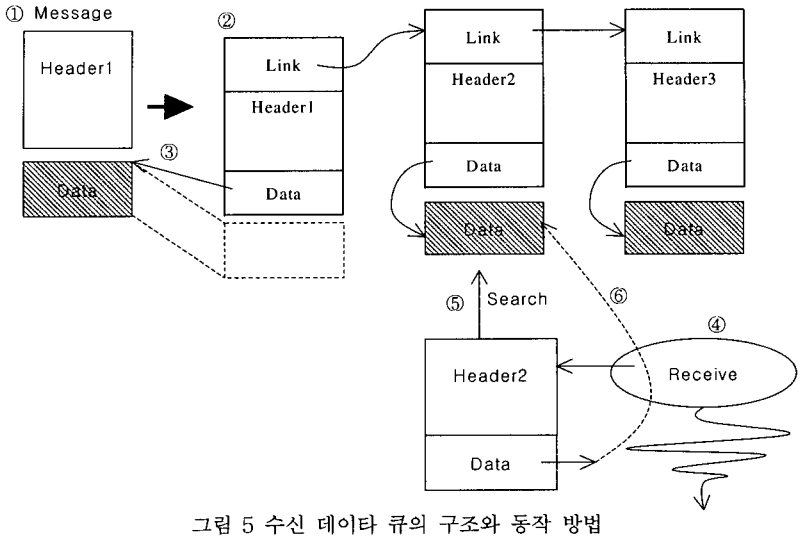

그림 5 수신 데이타 큐의 구조와 동작 방법

3.3.4 동기 통신 과정 구현

메시지를 수신하는 리스너 쓰레드의 흐름과 데이타의 수신을 요청하는 사용자 프로세스 사이에서, 두 프로세스의 시간적 선행관계에 따라 두 가지 경우의 수를 예상할 수 있다(그림 6 참조).

첫째는 보낸 쪽에서 미리 보내놓고 상대편 노드의 큐에 저장된 상태에서 상대편 노드에서 수신을 부르면 커널의 흐름은 멈추지 않고 바로 다음으로 넘어가게 된다. 큐에 메시지가 2개 이상일 경우에는 먼저 도착한 메시지를 전달해준다.

두 번째는 한 개의 흐름이 수신을 부른 상태로 큐를 살펴서 도착한 메시지가 없기 때문에 대기 상태로 들어가는 상황이 있을 수 있다. 이 경우 수신을 부른 쪽에서는 수신 대기 큐에 자기가 대기 상태에 있으며 어떤 메시지를 기다리는지를 표시해둔다. 리스너 쓰레드는 상대편 노드에서 메시지가 도착하면 수신 대기 큐를 보고 기다리던 흐름을 깨워서 그 메시지를 전달한다.

3.3.5 비동기 통신 과정 구현

KCCM에서는 원격 함수 호출을 할 경우 파라미터와 함께 상대편 노드로 *Rcall* 메시지가 전달되게 되어 있다. 상대편 노드에 메시지가 도착하면 리스너 쓰레드는 메시지 헤더를 보고 타입을 확인한다. 원격 함수 호출은 긴 호출(long) 타입과 짧은 호출(short) 타입 두 가지가 있다(그림 7 참조). 긴 호출 타입은 *CCM_rcall_l()*으로 호출하였을 때 핸들러가 쓰레드 형태로 수행되며, 리스너 쓰레드는 원격 함수 호출 메시지를 받으면 곧바로 핸들러 쓰레드를 생성시키고 다음 메시지를 기다리며

표 2 KCCM의 구현에 사용된 커널 인터페이스

유저 수준 인터페이스	커널 수준 인터페이스
malloc()	kmalloc();
free()	kfree();

커널 수준 인터페이스	설 명
init_waitqueue_head(&(x));	
interruptible_sleep_on_timeout (&(x), (y*HZ));	ych만큼 sleep
wake_up_interruptible(&(x));	sleep하고 있는 프로세스를 깨운다.

유저 수준 인터페이스	커널 수준 인터페이스
socket();	sock_create();
bind();	sock->ops->bind();
listen();	sock->ops->listen();
accept();	NewSock=sock_allow(); NewSock->type=소켓->type; NewSock->ops=소켓->ops; sock->ops->accept();

send();	struct msghdr msg;
	struct iovec iov;
	msg.msg_iov=&iov;
	msg.msg_iovlen=1;
	msg.msg_iov->iov_len=Length;
	msg.msg_iov->iov_base=Buffer;
	len=sock_sendmsg(sock, &msg,
	(size_t) (Length));

recv();	struck msghdr msg;
	struck iovec ivo;
	msg.msg_iov=&iov;
	msg.msg_iovlen=1;
	msg.msg_iov->iov_len=Length
	msg.msg_iov->iov_base=&Buffer[0];
	len=sock_recvmsg(sock, &mgs,
	Length, &msg, MSG_WAITALL);

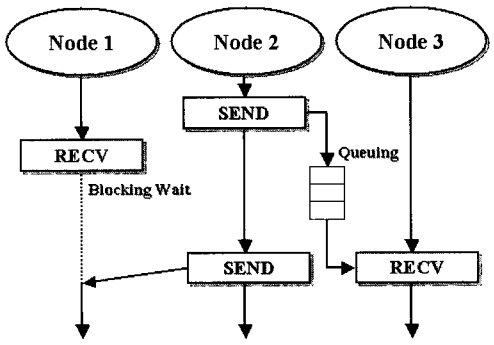

그림 6 송수신 과정

생성된 핸들러 쓰레드는 등록된 함수를 수행한다. 짧은 호출 타입은 *CCM_rcall_s()* 함수를 호출하였을 때 리스너 쓰레드 안에서 핸들러 함수를 수행하도록 구현하였으며 핸들러 수행이 끝나면 다시 다음 메시지를 처리한다. 짧은 호출 타입의 경우 불필요한 쓰레드의 생성에서 생기는 오버헤드를 감소시킬 수 있기 때문에 원격 함수 호출 시에 성능 향상을 기대할 수 있다.

원격 함수 호출에서 긴 호출 타입으로 깨어난 1개 이상의 쓰레드가 송수신 메시지를 주고받게 되면 같은 타입의 메시지들이 어떤 쓰레드로 전달되어야 하는지 알 수 없는 상황이 생긴다. 이것을 피하기 위해 인터페이스의 파라미터 중 하나인 *seqn*을 사용할 수 있다. *Rcall*을 부를 때 프로그래머에 의해 모듈 내부에서 결정된 *seqn* 숫자는 상대편의 쓰레드에 식별자로 사용되어 쓰레드를 구분할 수 있게 해준다. 핸들러 함수의 두 번째 아규먼트로 *seqn*이 전달되며 이것을 이용하여 원하는 쓰레드로 전달되도록 할 수 있다. 수신 옵션에서 *NOSEQ*를 이용할 경우 seqn을 무시한 상태에서 타입과 모듈 번호만을 이용해서 메시지를 전달받는다.

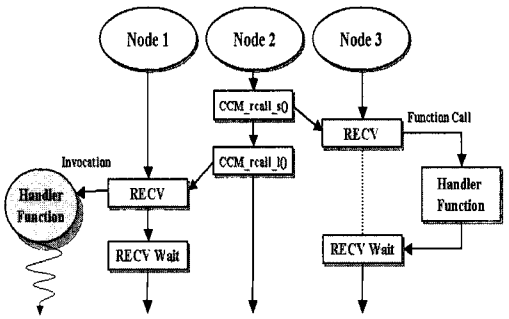

그림 7 CCM_rcall_s()와 CCM_rcall_l()

3.3.6 접속 및 접속 복구 과정 구현

KCCM은 TCP위에서 만들어졌기 때문에 접속을 관리하는 것이 필요하다. TCP는 접속을 하는 곳(client side)과 접속을 받는 곳(server side)이 나뉘어 있기 때문에 두 노드가 접속을 형성할 때 항상 한쪽이 다른 한쪽을 기다리도록 만들어져야 한다. KCCM이 접속을 관리하는 과정은 크게 처음 부팅하여 모든 노드로의 접속들을 형성하는 과정과 동작 상태에서 노드에 이상이 생겨 접속을 다시 형성하는 복구 과정으로 구성된다.

모든 노드로의 접속은 항상 노드 번호가 낮은 노드에서 접속을 시도하고 노드 번호가 높은 노드에서 접속을 기다리는 원칙으로 모든 노드에서 동시에 노드 순서대로 접속 형성을 시도하는 방법을 통해 클러스터에 참여하는 전체의 노드들이 상호 접속 상태를 형성한다.

한편 네트워크 이상이나 컴퓨터 이상 또는 관리상의 이유로 한 노드 이상이 통신을 할 수 없게 된 경우 문제가 되는 노드를 복구하는 과정에서 통신 시스템의 접속도 다시 형성해야 할 필요가 있다. 접속 관리자는 그림 8처럼 노드의 번호에 따라 끊어진 접속에 대해 새로운 접속을 시도하여 전체 접속 상태를 회복한다. 접속을 복구하는 과정은 크게 3단계로 이루어진다. 첫 단계는 끊어진 접속이나 문제가 생긴 노드를 확인하는 과정으로 접속 관리자가 연결된 노드들에 대하여 주기적인 메시지(heartbeat message)를 보냄으로서 끊어진 접속을 확인한다. 두 번째 단계는 끊어진 접속의 소켓을 제거하

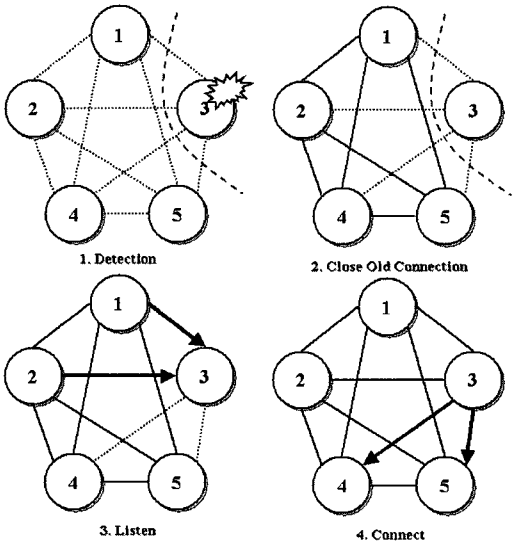

그림 8 접속 복구 과정

는 과정으로 문제가 생긴 노드에 대한 통신을 더 이상 진행하지 못하도록 막는 역할을 한다. 세 번째 단계는 다시 접속을 복구하는 과정으로 각 노드는 끊어진 접속에 대해 노드 번호에 따라 새로운 접속을 시도한다. 모든 노드들의 접속 관리자는 접속을 하는 부분과 접속을 받는 부분을 모두 포함하고 있기 때문에 복구의 주체가 되는 노드가 필요하지 않으며 개별적으로 접속을 복구한다.

4. 성능 평가

3장에서 설계하고 구현된 KCCM의 성능을 최적으로 이용하는 커널수준 서비스나 응용프로그램이 아직 없기 때문에 성능평가의 수행은 구현된 커널 수준의 통신 시스템을 사용하는 가상의 서비스 모듈을 커널에 설치하고 이 서비스를 이용하는 유저 수준의 응용 프로그램을 이용하여 통신 패턴과 메시지의 크기에 따른 대역폭을 실험을 통해 알아보았다. 본 실험에서는 일반적인 통신 패턴을 단 방향 원격 함수 호출 요구, 원격 함수 호출을 통한 원격 데이타 요청 그리고 단순한 메시지 송수신으로 분류하고 소켓을 이용할 경우와 RPC를 이용할 경우의 성능을 비교해본다. 실험에 사용된 하드웨어 환경으로는 256KB캐쉬(cache)를 가진 펜티움 800MHz 대칭형 멀티프로세서(SMP) 시스템으로 구성된 클러스터를 이용하였다. 네트워크 인터페이스로 3c905B 100Mbps 이더넷을 이용하였으며, 소프트웨어 환경으로 리눅스 커널 2.4.14에서 egcs2.91.66으로 최적화 옵션 없이 실험하였다.

4.1 메시지 송수신

단순하게 메시지를 주고받는 형태의 통신이다. 두 노드가 송수신 인터페이스(*CCM_bsend* 및 *CCM_brecv*)를 이용하여 서로 메시지를 주고받는 핑퐁(ping-pong) 형태의 통신 패턴을 10000번 왕복하는데 걸리는 시간을 측정하였고 측정된 시간은 커널에서 직접 소켓을 사용했을 경우 걸리는 시간과 비교해 보았다. 송수신을 위하여 단지 하나씩의 쓰레드를 사용하였기 때문에 통신과 연산(computation)을 중복(overlap)시킴으로서 얻을 수 있는 잇점을 배제하였다. 본 실험의 목적은 커널 소켓 위에서 구현된 KCCM이 어느 정도의 오버헤드를 지니고 있는지를 측정하여 KCCM의 통신 구조를 평가해 보기 위함이다.

그림 9는 KCCM과 소켓의 성능을 비교한 것이다. 이 그래프에 의하면 KCCM을 이용한 통신이 소켓을 이용한 통신 방법에 비해 약 10%정도의 오버헤드를 가진다는 것을 알 수 있다. 크기가 적은 메시지의 경우에는 생성된 오버헤드가 미비하고 메시지의 크기가 커질수록 오버헤드가 커짐을 알 수 있다. 이 오버헤드는 리스너 쓰레드의 오버헤드로 소켓의 내용을 임시로 저장하기 위한 버퍼링 및 복사에 의한 오버헤드와 메시지 도착을 전달해주기 위해 리스너 쓰레드를 한번 더 거친 후 수신을 요구하는 측에 전달하는 오버헤드로 생긴다. KCCM에서는 이식성과 쉬운 배포를 위해 커널을 수정하지 않고 구현하였기 때문에 소켓 위에서 리스너 쓰레드를 이용하였으며 이 때문에 피할 수 없는 오버헤드이다.

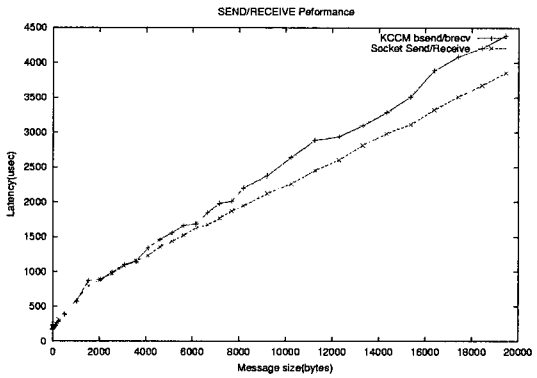

그림 9 송수신 메시지의 성능 측정 결과

4.2 원격 함수 호출

KCCM의 원격 함수 호출 성능을 알아보기 위해서 KCCM과 RPC(동기식)의 성능을 비교해 보았다. 동일한 실험 환경에서 동작하는 비 동기식 RPC를 구할 수 없었기 때문에 비 동기식 동작의 경우는 KCCM의 성능만을 측정하였다. KCCM의 원격 함수 호출은 대상 노드에서 등록된 함수를 수행시킨다. RPC가 등록된 함수의 리턴 값을 다시 돌려주는 동기식 동작을 하는데 비해서 KCCM은 대상 노드에서 일방적으로 수행시키는 비동기식 동작을 한다. 그러므로 본 실험에서는 호출될 핸들러 내부에서 *CCM_bsend*를 통해서 송신 쓰레드에 리턴 값을 보내주는 것으로 비동기 동작을 대신하여 비교하였다.

그림 10은 원격 함수 호출의 성능 측정 결과를 보여준다. 그림 10에서 보면 알 수 있듯이 동기 원격 함수 호출에 비해 비동기 원격 함수 호출이 월등히 빠른 것을 볼 수 있으며 이것은 매번 호출한 노드에 메시지를 보내지 않기 때문에 일어나는 당연한 결과이다. 리턴 값을 받지 않아도 되거나 일정 주기로 확인만 하면 되는 응용 프로그램을 작성할 때 비동기 원격 함수 호출을 이용하면 성능향상에 도움이 될 수 있다. RPC와

KCCM을 비교할 때 3000바이트 이상의 메시지를 보낼 때 KCCM이 더 좋은 성능을 보이고 그 보다 작은 메시지를 보낼 때 RPC가 더 좋은 성능을 보이는 것을 알 수 있다. 일반적으로 원격 호출을 하는 통신 시스템들은 작은 메시지에 대하여 최적화가 잘 되어 있어서 좋은 성능을 보이며 메시지가 커질 때 데이타를 보내기 위한 인코딩 오버헤드(예: XDR) 등으로 성능이 떨어진다. 이 결과를 볼 때 KCCM에 최적화의 여지가 많다고 보여진다.

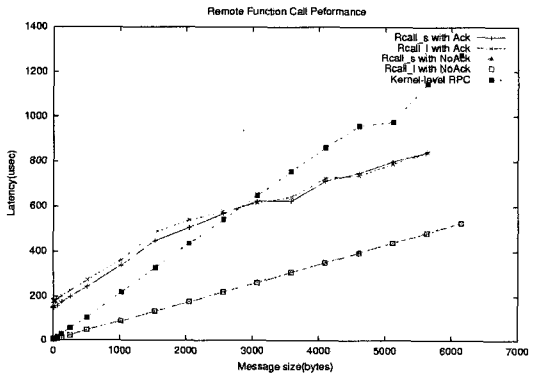

그림 10 원격 함수 호출의 성능 측정 결과

5. 결론 및 향후 과제

본 논문에서는 클러스터 환경에서 커널 수준의 응용 프로그램이나 서비스 개발을 위한 사용이 간편하고 효율적인 통신 시스템인 KCCM을 설계하고 구현하였으며 그 성능에 대해 논의하였다. 기존에 존재하던 통신 시스템들은 대부분 사용자 수준에서 구현되어 있었기 때문에 커널 수준의 응용프로그램이나 서비스를 개발하는 경우 사용할 수 없었으며 커널 수준으로 작성된 경우에도 기본적인 기능만 지원하거나 사용 방법이 어려워서 문제가 있었고 성능을 위해서 특정 하드웨어를 필요로 하는 경우에는 범용성이 떨어지기 때문에 일반적인 응용에는 적합하지 않았다. KCCM은 일반적인 동기식 통신 방법이외에 원격 함수 호출을 통해 비동기 패러다임을 사용할 수 있도록 설계하였으며 사용자의 응용 프로그램을 쉽게 하기 위한 단순한 RPC의 용도로도 사용할 수 있도록 하였다. 또한, KCCM은 여러 다른 하드웨어 환경에서 소스 수정을 최소화하여 사용할 수 있도록 하기 위해 커널 소켓을 이용해 구현하였다. 시스템의 장애 상황에 효과적으로 대처하기 위해 접속을 관리하는 인터페이스를 외부로 공개하여 응용 프로그램과 연동하여 관리할 수 있도록 하였다.

구현된 통신 시스템의 성능을 측정해본 결과 동기식 통신은 소켓에 비해 약 10% 정도의 오버헤드를 가지며 비동기 통신을 사용할 수 있는 경우 동기식으로 동작하는 RPC에 비해 유리하다는 것을 확인하였다. KCCM의 경우 작은 메시지에 대해 소켓을 이용함으로 인해 생기는 오버헤드가 크고 통신 시스템을 운영하기 위해 소모하는 프로세스나 메모리 자원이 많다는 단점이 있으며 이를 해결하는 것이 추후 과제가 될 수 있을 것이다.

참 고 문 헌

[1] T. Anderson, D. Culler, D. Patterson, and the NOW Team, "A Case for NOW(Network of Workstations)," IEEE Micro, pp.1-6, 1995.

[2] G. Ciaccio, "A Communication System for Efficient Parallel Processing on Clusters of Personal Computers," PhD Thesis DISI-TH-1999-02, pp.20-66, 1999.

[3] A. Mainwaring, D. Culler, S. Goldstein and K. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation," Proc. of the 19th ISCA, 1992.

[4] E. Lusk, B. Saphir, M. Snir, "MPI-2: Extensions to the Message-Passing Interface," MPI Standard 2.0, 1997.

[5] "Parallel Virtual Machine," http://www.epm.ornl.gov/pvm/

[6] S. Pakin, V. Karamcheti, and A. Chien, et. al., "Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors," IEEE Concurrency, vol. 5, no. 2, 1997.

[7] T. von Eicken, et al., "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," Proc. of the 15th ACM SOSP, 1995.

[8] T. von Eicken and W. Vogels, "Evolution of the Virtual Interface Architecture," IEEE Computer, 1998.

[9] Object Management Group Inc., "The Common Object Request Broker: Architecture and Specification, Revision 2.0," OMG TC Document, 1994.

[10] http://java.sun.com, "Java Remote Method Invocation," Sun Microsystems.

[11] G. Ciaccio, "Messaging on Gigabit Ethernet: Some experiments with GAMMA and other systems," IPDPS '01, April 2001.

[12] N. J. Boden, D. Cohen, R. Felderman, et al., "Myrinet-A Gigabit-per-Second Local-Area Network," IEEE Micro, Feb. 1995.

[13] http://www.fenrus.demon.nl, "kHTTPd - Kernel httpd accelerator," 2001.

박 동 식

1999년 6월~1999년 12월 ㈜ 타이니미디어 근무 2000년 서강대학교 컴퓨터학과 졸업 2002년 서강대학교 컴퓨터학과 석사 2002년 3월~현재 ㈜ 엔버전스 근무. 관심분야는 클러스터 통신시스템, 운영체제 시스템

박 성 용

1987년 서강대학교 전자계산학과 졸업 1994년 Syracuse 대학 Computer Science 석사. 1998년 Syracuse 대학 Computer Science 박사. 1998년~1999년 Bellcore 연구소 연구원. 1999년~현재 서강대학교 컴퓨터학과 조교수. 관심분야는 분산시스템, 클러스터 시스템, 고성능 네트워크

양 지 훈

1987년 서강대학교 전자계산학과 졸업 1989년 Iowa State University, Computer Science 석사. 1999년 Iowa State University, Computer Science 박사 1999년 1월~2000년 10월 HRL Laboratories 연구원. 2000년 10월~2002년 2월 SRA International, Inc. 연구원. 2002년~현재 서강대학교 컴퓨터학과 조교수. 관심분야는 데이타 마이닝, 생물정보학