Journal of KIIT. Vol. 19, No. 2, pp. 53-63, Feb. 28, 2021. pISSN 1598-8619, eISSN 2093-7571
http://dx.doi.org/10.14801/jkiit.2021.19.2.53

RDMA 기반 HDFS를 위한 PCIe NTB Interconnect Network의 JNI 인터페이스 설계 및 구현

지민재*[1], 고병현*[2], 신동렬*[3], 김성현*[4], 임승호**

Design and Implementation of JNI Interface of PCIe NTB Interconnect Network for RDMA-based HDFS

Min-Jae Ji*[1], Byeong-Hyun Ko*[2], Dong-Ryeol Sin*[3], Seong-Hyun Kim*[4], and Seung-Ho Lim**

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (NRF-2019R1F1A1057503). 이 연구는 한국외국어대학교 교내학술연구비의 지원에 의하여 이루어진 것임.

요 약

다중의 노드를 연결하여 Cluster로 구성하는 HPC 시스템은 인터커넥트 네트워크 성능이 전체 시스템의 성능에 영향을 많이 미친다. 이러한 Infiniband, PCIe NTB와 같은 PCIe 인터페이스로 연결된 인터커넥트 네트워크에서는 RDMA(Remote Direct Memory Access) 기반 데이터 전송이 널리 활용된다. 한편, 최근에는 VM 기반 언어인 Java가 HPC에서 주로 사용되었던 컴파일 기반 언어와의 성능 격차가 점차 감소 되면서 HDFS(Hadoop Distributed File System)와 같이 Java로 구현된 HPC 시스템이 널리 활용되고 있다. PCIe 표준에 기반한 인터커넥트 네트워크 인터페이스인 PCIe NTB 인터커넥트 장치가 개발되었지만, Java에서 RDMA를 직접 사용할 수 있는 인터페이스가 제공되지 않는다. 본 논문에서는 Dolphin 사의 PCIe NTB 기반의 인터커넥트 네트워크에서 네이티브 API로 제공되는 RDMA 통신을 JNI(Java Native Interface)를 통해 자바에서 구현하고, 이를 이더넷 소켓과 비교함으로써 실제 시스템에서 활용 가능성과 성능을 고찰해 본다.

Abstract

In the HPC system, which consists of a cluster by connecting multiple computing nodes, the performance of the interconnection network greatly affects the performance of the entire system, and for high performance, RDMA (Remote Network) in the interconnection network with PCIe interfaces such as Infiniband and PCIe NTB. DMA) data transmission is widely used. Meanwhile, recently, as the performance gap between the VM-based language Java and the compilation-based language that was mainly used in HPC has been gradually resolved, HPC systems implemented with Java such as HDFS (Hadoop Distributed File System) are increasing. The PCIe NTB interconnect device, which is an interconnect network interface based on the PCIe standard, was developed, but Java does not provide an interface to directly use RDMA. In this paper, the RDMA communication provided as Native C API in the PCIe NTB-based interconnect network of Dolphin is implemented in Java through JNI and compared with Ethernet Interface to consider the performance in practical use.

Keywords

HPC, HDFS, interconnect network, PCIe NTB, RDMA, JNI

* 한국외국어대학교 컴퓨터공학부 학부생
- ORCID[1]: http://orcid.org/0000-0003-4634-723X
- ORCID[2]: http://orcid.org/0000-0002-1749-9683
- ORCID[3]: http://orcid.org/0000-0003-3839-2049
- ORCID[4]: http://orcid.org/0000-0003-1022-9231
** 한국외국어대학교 컴퓨터공학부 교수(교신저자)
- ORCID: http://orcid.org/0000-0003-3096-0785

· Received: Nov. 23, 2020, Revised: Dec. 22, 2020, Accepted: Dec. 25, 2020
· Corresponding Author: Seung-Ho Lim
Division of Computer Engineering, Hankuk University of Foreign Studies, Korea
Tel.: +82-31-330-4704, Email: lim.seungho@gmail.com

Ⅰ. 서 론

AI, Big Data 기반의 4차 산업혁명 시대에 대규모 콘텐츠 서비스 제공을 위해서는 일반적인 PC 기반의 컴퓨팅 시스템이 아닌 HPC(High Performance Computer) 기반의 컴퓨팅 시스템 활용이 점점 증대된다[1]. 이러한 HPC 시스템은 다수의 노드로 구성된 클러스터를 이루며 구성되는 것이 일반적이며 각 노드를 연결하는 Interconnection이 전체적인 시스템의 성능에 큰 영향을 준다.

HPC에서 각 노드를 구성하는 호스트 시스템의 내부 버스는 프로세서가 외부 장치와 동작하는 데 있어서 중요한 인터페이스이다. 예를 들어, AMD의 CPU 코어 인터커넥트 버스[2]는 DRAM의 클록과 동기 되어서 400GB/s 이상의 대역폭을 보여준다. 하지만 HPC에서 클러스터 노드 간의 자원을 공유하고 데이터를 주고받기 위해서는 프로세서 코어 버스보다 속도가 느린 인터커넥트 네트워크를 거쳐야 한다. 그러므로 클러스터 내 각 노드를 연결하는 인터커넥트 네트워크가 전체 시스템의 병목현상이 될 수가 있다.

인터커넥트 네트워크에서 데이터 전송의 효율을 높이기 위해서 RDMA(Remote Direct Memory Access)[3][4] 기반의 네트워크 기술이 등장하게 되었다. RDMA 프로토콜은 고속 네트워크를 통한 메모리 간 데이터 전송용 기술이다. 구체적으로, RDMA는 CPU를 사용하지 않고 메모리에서/메모리로 직접 원격 데이터를 전송하는 기능을 제공한다. 또한, RDMA는 직접 데이터 배치 기능도 제공하므로 데이터 복사본이 없어져 CPU 작업이 더욱 줄어든다. 따라서 RDMA는 호스트 CPU의 부담을 줄일 뿐 아니라 호스트 메모리 및 I/O 버스에 대한 경합도 줄여 주어 HPC에 유용하다. 병렬 작업이 동기화된 메시지 전송과 함께 많은 계산을 수반할 때 RDMA를 사용하면 성능 향상을 기대할 수 있다. 이러한 RDMA 전송이 가능한 인터커넥트 네트워크로는 Infiniband, Omni-path 등이 있으나, 최근 PCIe Interface Specification의 발전과 더불어 PCIe 자체 표준으로 인터커넥트 네트워크가 가능한 PCIe NTB (Non-Transparent Bridge)[5][6] 기술에 기반한 인터커넥트 장치의 개발에도 불구하고 PCIe NTB에 기반한 RDMA 활용에 대한 연구결과가 없는 편이다.

한편, 최근에는 HDFS(Hadoop Distributed File System) 등과 같이 VM 기반 언어인 Java를 HPC 시스템에서 활용하는 방향으로 가고 있다. 이는 Java가 내장 네트워킹 및 멀티 쓰레딩 환경에서의 폭넓은 지원과 지속적인 JVM 개선을 통해 전통적으로 HPC에서 주로 사용되었던 컴파일 기반 언어와의 성능 격차를 점차 해결하고 있기 때문이다. 그러나 RDMA 구현은 주로 네이티브 디바이스 드라이버 인터페이스를 그대로 사용하는 경우가 많으므로 Java와 같은 시스템에서 PCIe NTB 인터커넥트 네트워크 기반 RDMA를 위한 자바 인터페이스는 잘 제공되지 않는다.

본 연구에서는, PCI-Express NTB 기반의 인터커넥트 네트워크로 구성된 클러스터 시스템의 HDFS에 Native C API로 제공되는 RDMA를 제공하기 위한 JNI(Java Native Interface) 인터페이스를 설계하고 구현하여 이를 직접 JAVA에 적용해 봄으로써 JNI를 통한 RDMA의 사용 가능성을 검증해보았다. 또한, 이를 Dolphin PCIe NTB 인터커넥트 네트워크 시스템[7]-[9]에 직접 적용해 보고, 이더넷 소켓에 기반한 인터커넥트 네트워크와 성능 비교를 해보았다. 이를 통해서 PCIe NTB 인터커넥트 시스템의 RDMA와 JNI[10]를 통한 JAVA 기반 분산 시스템에 대한 활용 가능성을 검증해볼 수 있다.

Ⅱ. 배 경

배경에서는 인터커넥트 네트워크로 사용이 가능한 PCIe NTB의 개념에 관해서 설명하고, Java에서 Native Library를 사용하기 위한 JNI 인터페이스에 대한 구현 방식에 관해서 설명한다.

2.1 PCIe-Express NTB

PCIe NTB[5]는 PCIe 시스템에서 프로세서간 연결을 위한 어댑터를 지원하기 위한 것으로서 2대 이상 시스템의 서로 분리된 메모리 시스템을 같은 PCIe 패브릭으로 연결하게 하는 기술이다. NTB는

이더넷 및 인피니밴드 등의 고속 인터커넥트 네트워크 기술과 비교할 때, PCIe Specification에서 제공하는 최대 대역폭을 보장하는 PCIe 표준 인터커넥트 네트워크 인터페이스라고 볼 수 있다.

NTB는 PCIe TB와 마찬가지로 독립적인 PCIe bus에 대해서 데이터 전송 경로(Path)를 제공한다는 점에서 유사하나, 가장 큰 차이점은 NTB가 사용될 경우 그림 1과 같이 bridge의 양쪽에 각각 독립적인 주소 도메인을 갖는 프로세서 코어를 서로 연결할 수 있다는 점이다. 프로세서는 PCIe NTB에 의해서 서로의 어드레스 공간을 침범할 수 없다. 즉, Bridge의 한편의 호스트는 Bridge 다른 쪽의 호스트의 메모리 공간을 볼 수 없다. Bridge 반대편의 각 프로세서 코어는 반대쪽 영역을 디바이스 영역으로 간주하고 End Point 자체에 메모리 공간을 대응시킨다.

이러한 PCIe NTB 환경에서 PCIe Express 시스템은 서로 데이터를 주고받기 위한 메모리 공간을 서로 확보해야 한다. 즉, 한 호스트의 메모리 공간에서 다른 호스트의 메모리 공간으로 데이터를 주고받기 위한 주소 영역을 서로 교환하여 확보한다. 이렇게 확보된 메모리 공간은 각자의 메모리 영역에 할당되며, 해당 메모리 공간에 데이터를 쓰고 읽음으로써 서로 데이터를 송수신할 수 있다. 한 호스트에서 다른 호스트를 위해서 할당된 메모리 공간에 데이터를 쓰면, 해당 공간의 데이터는 PCIe NTB의 주소 변환을 통해서 다른 호스트에 할당된 메모리 공간으로 데이터 전송이 이루어지게 된다.

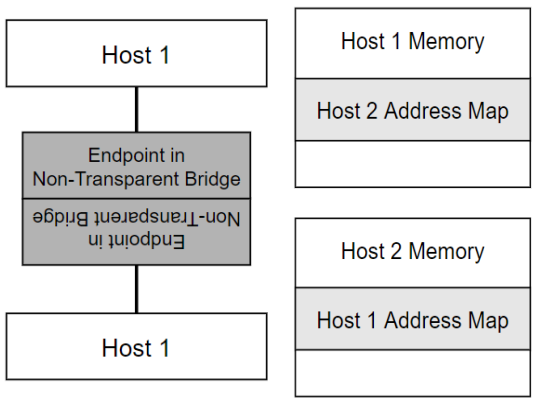


그림 1. PCIe NTB 인터페이스로 연결된 두 프로세스의 데이터 송수신 방식

Fig. 1. Data transmission and reception method of two processes connected by PCIe NTB interface

2.2 JNI

JNI는 JVM 상에서, Native 언어(C언어)로 작성된 코드를 자바에서 사용하거나 반대로 네이티브에서 자바의 클래스와 메소드를 사용할 수 있게 해주는 프로그래밍 인터페이스로 많은 장점이 있다.

첫 번째는 성능적인 목적으로, Java는 원칙적으로 Java 바이트 코드가 JVM 위에서 수행되기 때문에 일반적인 Native 환경보다 처리속도가 느리다. 따라서 빠른 처리를 요구하는 부분에는 JNI를 통해 Native 코드를 호출하여 속도 향상을 기대할 수 있다. 두 번째로 하드웨어 제어 속도 또한 Native 코드를 사용하는 것이 우수함으로 JNI를 통해 자바와 연결하여 자바에서도 하드웨어 제어가 가능하도록 한다. 마지막은 코드의 재사용성으로 기존의 Native로 작성된 코드에 대해서 자바로 재작성할 필요 없이 JNI를 통해 기존 코드를 활용할 수 있다.

JNI 구현의 핵심은 Java Method와 Native Function을 매핑 하는 데 있다. 첫 번째는 자동 매핑으로 프로그래머가 직접 명시하지 않아도 자동으로 Java method와 Native function 간의 매핑을 JNI가 해주는 방식이다. 두 번째는 수동 매핑으로 Java method와 CPP function의 매핑을 직접 기재해주는 방식이다. 본 논문에서는 자동 매핑보다 수동 매핑이 우선순위가 높고, 프로그래머가 직접 연결 작업을 처리하여 로딩 속도 측면에서 더 효율적이므로 수동 매핑 방식을 사용하였다.

III. PCIe NTB 기반 JNI 설계 및 구현

3.1 PCIe NTB 함수

PCIe NTB는 연결된 PCI Bridge에 독립적인 address space를 가지는 두 호스트 간의 Address Translation을 통해서 데이터 송수신을 한다. 이러한 Address Translation을 하기 위해서 기본적으로 PCIe Specification 상 주어진 BAR(Base Address Register), Doorbell Register, ScratchPad Register에 대한 설정과 활용이 필요하며, 각 노드는 PCIe Host Adapter를 통해서 PCIe NTB에 연결된다. 노드 간 연결은 PCIe

NTB Switch를 통해서 인터커넥트 네트워크를 구성할 수 있다. 본 논문은 Dolphin 사의 PXH810 NTB Host Adapter 카드와 IXS600 스위치를 이용하여 PCIe NTB 통신 시스템을 구성하였다[3]. PCIe NTB 관련 함수는 이렇게 구성된 Host Adapter와 스위치 기반으로 된 RDMA 동작에 대한 함수이다. RDMA 동작과 관련된 주요 함수로는 두 호스트 간 데이터 전송을 위해 할당되는 세그먼트 관련 함수, 데이터 전송에 직접 사용되는 DMA 관련 함수, 인터럽트 처리를 위한 함수 등이 있다.

두 호스트 간의 데이터 전송을 위해서 할당되는 세그먼트 관련 함수는 다음과 같이 동작한다. Dolphin PCIe NTB 시스템에서 Buffer의 개념으로 사용되는 것이 세그먼트이다. 호스트 간 데이터를 송수신하기 위해서 먼저 세그먼트를 생성해야 하는데, 세그먼트 생성은 그림 2와 같은 방식으로 진행된다. Initialize를 통해 API를 사용 전에 초기화 작업을 진행한다. 이후 Open을 통해 virtual device를 하나 할당한다. Virtual device는 디바이스 드라이버와 통신하기 위한 채널이다. 이후 Virtual device 위에 세그먼트를 생성하게 되고 PrepareSegment를 통해 세그먼트에 접근할 수 있게 된다. MapLocalSegment를 통해 세그먼트와 프로그램 address space에 매핑 된 address를 받아온다. 마지막으로 생성해준 세그먼트가 다른 노드 들로부터 접근을 가능하게 하고 싶으면 SetSegmentAvailable을 호출해 접근 가능한 상태로 변경할 수 있다. 이렇게 생성된 세그먼트의 상태는 그림 3과 같은 상태를 가지며 상태에 따라서 세그먼트의 사용 가능 여부가 결정된다. 먼저 세그먼트를 생성하면 해당 세그먼트는 NOT PREPARED 상태가 되며, SCIPrepareSegement를 통해서 세그먼트를 사용 가능 상태로 설정하여 사용할 수 있게 해준다.

다음으로 생성된 세그먼트를 통해서 데이터 전송을 하는 RDMA 동작 방식은 다음과 같다. 그림 4는 DMA를 위한 API 동작 흐름에 대해서 순서대로 나타낸 것이다.

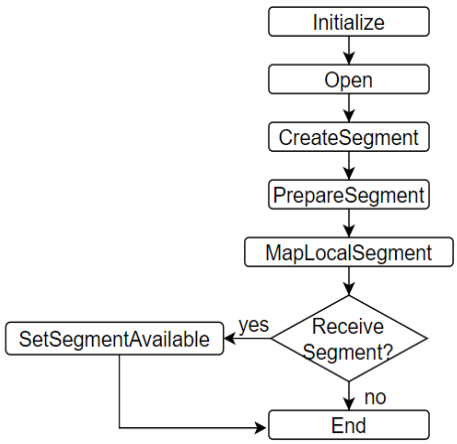


그림 2. 세그먼트 생성 흐름도
Fig. 2. Segment creation flow

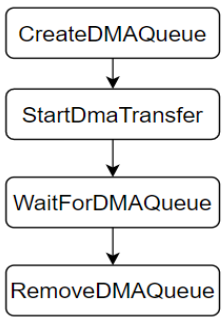


그림 4. DMA 동작 흐름도
Fig. 4. DMA operation flow

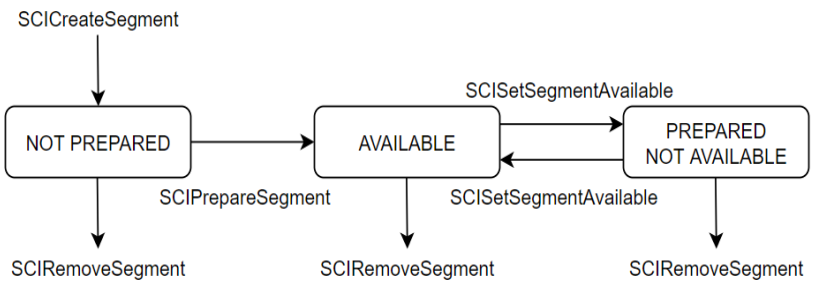


그림 3. 세그먼트 상태 변화
Fig. 3. Segment state transition

PCIe NTB 통신에서 Buffer로 세그먼트를 할당한 후, RDMA 방식으로 데이터를 전송하기 위해서 DMA Queue를 사용하며, CreateDMAQueue 에서 virtual device 위에 DMA Queue를 생성한다. 이후 StartDmaTransfer를 통해 localSegment에서 remote 세그먼트로 전송을 시행한다. WaitForDMAQueue 에선 작업이 완료될 때까지 Block 상태로 대기후 완료되면 RemoveDMAQueue를 이용해 DMA Queue에 할당된 자원들을 제거한다.

마지막으로, 인터럽트를 처리하는 함수를 설명한다. Dolphin PCIe NTB Host Adapter 및 시스템에서는 그림 5와 같은 방식으로 인터럽트를 사용한다. 먼저 CreateInterrupt를 통해 Local에 Interrupt를 생성한다.

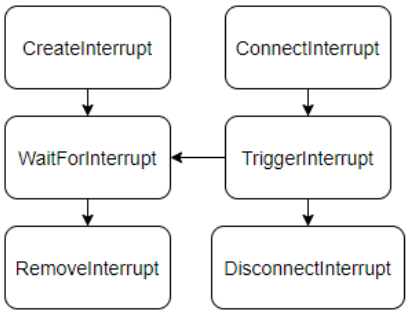
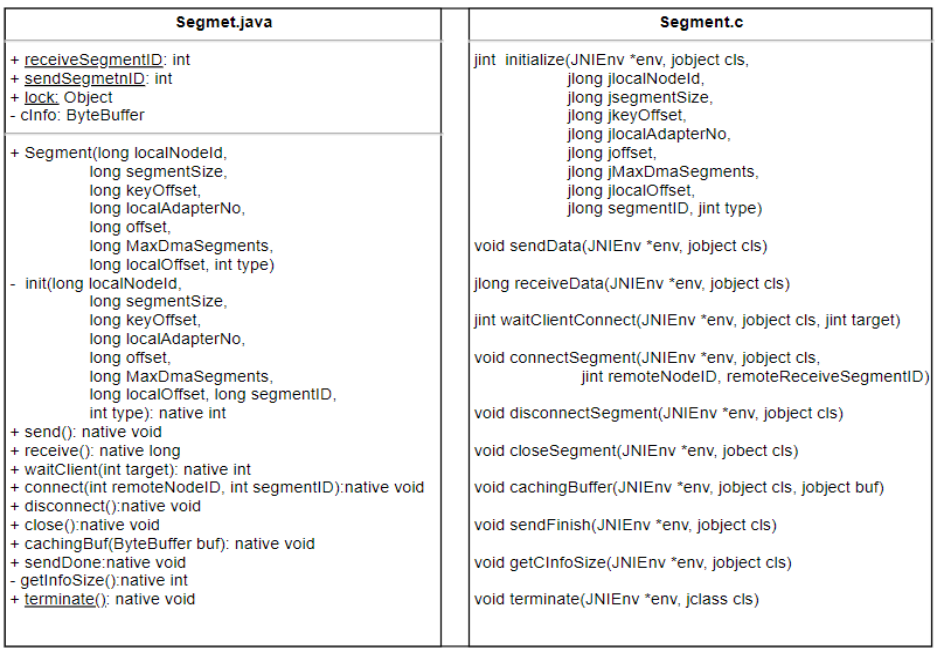


그림 5. 인터럽트 관련 함수 흐름도
Fig. 5. Interrupt-related function flow

이후 Interrupt가 필요한 부분에 WaitForInterrupt 함수를 통해 block 상태로 Interrupt가 Trigger 되기를 기다린다. Interrupt가 Trigger 되면 blocking 상태가 풀리며 Trigger 된 인터럽트를 처리한다. 이후 Interrupt의 사용을 완료했으면 RemoveInterrupt를 이용해 할당된 자원들을 해제하고 삭제한다. Interrupt를 Trigger 하는 쪽에서는 connect-Interrupt를 통해 Remote에 있는 Interrupt를 연결해 온다. 이후 Trigger가 필요한 부분에서 TriggerInterupt를 통해 Interrupt를 Trigger 시키고 완료되면 DisconnectInterrupt를 호출해 RemoteInterrupt와 연결을 끊고 자원들을 해제한다.

3.2 RDMA JNI 인터페이스

본 논문에서 구현한 RDMA JNI 인터페이스의 정의는 그림 6과 같으며 매핑정보는 그림 7과 같이 정리할 수 있다. 그림 6에서, Segment.java의 receiveSegmentID와 sendSegmentID는 Static 변수로 새로 생성되는 Receive 세그먼트의 ID와 Send 세그먼트 ID를 나타내며, Lock은 동기화를 위한 변수이다. cInfo는 Segment.c에서 현재 사용되는 세그먼트의 정보를 담은 구조체인 Info를 저장한다.



Segmet.java

```
+ receiveSegmentID: int
+ sendSegmetnID: int
+ lock: Object
- cInfo: ByteBuffer

+ Segment(long localNodeId,
          long segmentSize,
          long keyOffset,
          long localAdapterNo,
          long offset,
          long MaxDmaSegments,
          long localOffset, int type)
-  init(long localNodeId,
          long segmentSize,
          long keyOffset,
          long localAdapterNo,
          long offset,
          long MaxDmaSegments,
          long localOffset, long segmentID,
          int type): native int
+ send(): native void
+ receive(): native long
+ waitClient(int target): native int
+ connect(int remoteNodeID, int segmentID):native void
+ disconnect():native void
+ close():native void
+ cachingBuf(ByteBuffer buf): native void
+ sendDone:native void
- getInfoSize():native int
+ terminate(): native void
```

Segment.c

```
jint  initialize(JNIEnv *env, jobject cls,
                 jlong jlocalNodeId,
                 jlong jsegmentSize,
                 jlong jkeyOffset,
                 jlong jlocalAdapterNo,
                 jlong joffset,
                 jlong jMaxDmaSegments,
                 jlong jlocalOffset,
                 jlong segmentID, jint type)

void sendData(JNIEnv *env, jobject cls)

jlong receiveData(JNIEnv *env, jobject cls)

jint waitClientConnect(JNIEnv *env, jobject cls, jint target)

void connectSegment(JNIEnv *env, jobject cls,
                    jint remoteNodeID, remoteReceiveSegmentID)

void disconnectSegment(JNIEnv *env, jobject cls)

void closeSegment(JNIEnv *env, jobect cls)

void cachingBuffer(JNIEnv *env, jobject cls, jobject buf)

void sendFinish(JNIEnv *env, jobject cls)

void getCInfoSize(JNIEnv *env, jobject cls)

void terminate(JNIEnv *env, jclass cls)
```

그림 6. Segment.java와 Segment.c의 Java API 정의
Fig. 6. Java API definition in Segment.java and Segment.c

```
{"init","(JJJJJJJJI)I",(void*) initialize},
{"send","()V",(void*) sendData},
{"receive","()J",(jlong*) receiveData},
{"waitClient","(I)I",(jint*) waitClientConnect},
{"connect","(II)V",(jint*) connectSegment},
{"disconnect","()V",(void*) disconnectSegment},
{"close","()V",(void*) closeSegment},
{"cachingBuf","(Ljava/nio/ByteBuffer;)V",(void*) cachingBuffer},
{"getInfoSize","()I",(int*) getInfoSizee},
{"sendDone", "()V", (void*) sendFinish},
{"terminate", "()V", (void*) terminatee}
```

그림 7. Native Method와 C Function 간 매핑 테이블
Fig. 7. JNI Mapping table between native method and C function

```
Info

sci_desc_t sd;
volatile char*  segmentAddr;
volatile char*  data;

sci_local_segment_t  localSegment;
sci_remote_segment_t remoteSegment;

sci_map_t segmentMap;
sci_map_t remoteMap;

int type;
unsigned int  segmentId;
int remoteNodeID;
unsigned int  remoteSegmentID;

unsigned int    DMA_READY;
unsigned int    RECEIVE_DONE;
unsigned int    CLIENT_CONNECT;
unsigned int    CLIENT_CHECK;
```

그림 8. Segment.c의 Info 구조체
Fig. 8. Info structure in Segment.c

Info 구조체는 그림 8과 같으며 Segment.c의 모든 함수는 실행마다 사용되는 세그먼트를 Segment.java의 cInfo 변수로부터 받아와서 사용한다.

Segment.java의 Init()은 세그먼트를 실제로 생성하는 부분이다. 생성된 세그먼트는 어떠한 Connection도 없으며 Receive 용도인지 Send 용도인지만 정해져 있다. watiClient(int target)은 target 세그먼트로부터 Connection을 기다린다. 이때 target이 0이라면 세그먼트 ID와 상관없이 Connection이 이루어진다. connect(int remoteNodeID, int segmentID)는 remoteNodeID 노드에 있는 ID가 segmentID인 세그먼트에 connection을 시도한다.

CachingBuf(ByteBuffer buf)는 Java의 ByteBuffer의 시작 주소를 C의 Pointer가 가리키게 한다. 이때 포인터는 C의 Info 구조체의 volatiole char* data가 된다. Send()는 Info 구조체의 이 데이터에 있는 값을 세그먼트와 매핑 되어있는 segmentAddr에 복사한 다음 DMAQueue를 생성해 Remote 세그먼트로 데이터를 전송한다. 전송이 완료되면 Remote 세그먼트에 데이터 전송이 완료되었다는 Interrupt를 발생시킨다. Receive()는 데이터가 수신되었다는 Interrupt를 수신할 때까지 블록 되며 데이터를 수신할 때 segmentAddr에 있는 값을 데이터 영역으로 복사한다. data는 Java의 ByteBuffer를 가리키고 있으므로 수신받은 데이터는 Java에서 바로 읽을 수 있다. SendDone()은 모든 데이터 전송 시 Interrupt를 발생시켜 Receive()를 종료시킨다. 위와 같이 구현된 세그먼트는 PCIe 객체가 2개씩 가지고 있으며 PCIe 객체는 이를 이용해서 데이터 통신을 수행한다.

3.3 RDMA JNI 인터페이스

3.2에서 설명한 PCIe NTB 함수들을 이용한 JNI API는 Java Socket API와 유사하게 동작한다. 그림 9는 PCIe NTB RDMA에 대한 JNI 인터페이스의 동작 방식에 대해서 도식화한 그림이다. 그림 9에 나타난 바와 같이 기존 Socket 통신 API 경우 Client는 Socket을 생성해 Server의 Server-Socket에 연결을 요청한다.

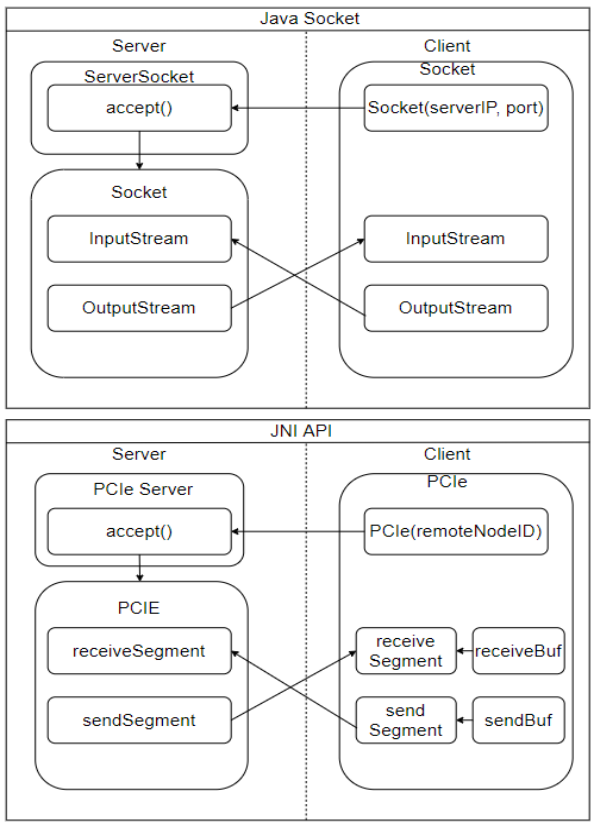


그림 9. Java Socket과 JNI 구현 동작 방식 비교
Fig. 9. Operation comparison between Java socket and JNI interface

연결이 완료되면 ServerSocket은 Client와 연결된 새로운 Socket의 인스턴스를 생성하고 이 인스턴스를 이용해 Client와 통신을 하는 방식이다. 이때, 통신에 이용되는 버퍼가 InputStream OutputStream이며, 사용자는 Socket 객체 내부에 있는 이 Stream들의 reference를 얻어와 데이터 통신을 한다.

이와 마찬가지로 JNI API에서 Client는 PCIe 객체를 생성해 Server의 PCIeServer에게 연결을 요청한다. 연결이 완료되면 PCIeServer객체는 Client와 연결된 새로운 PCIe의 인스턴스를 생성하고 이 인스턴스를 이용해 Client와 통신한다. 이때, sendBuf, receiveBuf 가 사용자가 데이터 전송을 하기 위해 이용하는 버퍼이다. 이를 세그먼트와 통한 매핑 된 address space에 복사해 세그먼트를 통한 DMA 전송을 시행한다.

JNI 인터페이스를 이용한 Java 응용 Programming 방법은 Socket과 유사하다. Client 경우, PCIe 객체를 생성하여 Server와 Connection을 맺는다. 이후, PCIe 객체로부터 receiveSegment, sendSegment, receivBuf, sendBuf의 reference를 가져온다. 다음으로, 데이터를 전송 시 sendBuf에 전송할 데이터를 넣고 sendSegment.send()를 하게 되면 연결된 Server로 데이터가 전송된다. 데이터 수신의 경우, receiveSegment.receive()를 하게되면 연결된 Server로부터 데이터가 수신된다. 수신된 데이터는 receiveBuf에 담겨 있다.

Server의 경우, PCIeServer객체를 생성하고 accept()를 호출한다. 이때 Server는 accept()에서 Block 되어 Client로부터 연결 시도가 오는 것을 기다린다. Client로부터 연결이 완료되면 accept()는 PCIe 객체를 반환하고 이 객체를 이용해 통신한다. 이처럼 Java에서 Socket과 유사한 인터페이스 콜을 이용하여 JNI 인터페이스를 통한 PCIe NTB RDMA를 사용할 수 있으며, RDMA 방식을 통해서 Socket 방식의 네트워크 전송보다 효율적인 데이터 전송이 가능하다.

Ⅳ. 성능평가

4.1 실험환경 구성

본 논문에서 구현한 PCIe NTB 기반 RDMA의 JNI 인터페이스에 대한 성능을 평가하기 위해서 10Gbit Ethernet Socket과 PCIe NTB의 두 가지 인터커넥트 네트워크 실험환경을 구성하였다. Ethernet Socket은 전형적인 HPC 시스템을 구성하는 네트워크 인터페이스로 활용되고 있고, 자바 기반의 HDFS가 이더넷 기반의 Socket에 구현되어 있으므로 이더넷과 PCIe NTB 두 가지 인터커넥트 네트워크 시스템으로 실험환경을 구성하였다. 실험은 6대의 Host 노드를 이용해서 네트워크 시스템을 구성하였으며, PCIe NTB의 경우 Dolphin 사의 PXH810 NTB RDMA NIC 카드[3]와 IXS600 스위치를 사용하여 진행하였다. 각각의 노드는 인텔 i7-9700k와 16GB DRAM을 가지며, 내부적으로 PCIe Express 3.0 x8의 속도로 연결되며 또한, 스토리지 장치로는 인텔의 Optane PCIe Express 900P를 사용하여 네트워크 대역폭에 따르는 스토리지 속도를 확보하였다. 전체적인 소프트웨어 Stack과 H/W Stack은 그림 10과 같이 나타낼 수 있다.

Ethernet과 JNI 구현체 Interface 간의 성능 측정은 micro benchmark program을 사용하였으며 프로그램의 동작은 그림 11과 같다. 전체적인 동작은 데이터를 받는 서버와 데이터를 전송하는 클라이언트를 지정된 Thread 단위만큼 생성하여 interface에 맞게 서로 연결 후 파일을 전송한다. Ethernet 측정 그림 10에 나타난 바대로 측정하였으며, RDMA 측정은 자체적으로 마지막 세그먼트에 대한 interrupt 시간을 측정할 수 있으므로 Packet 캡처 Thread만 제외하면 같은 측정 비교가 된다.

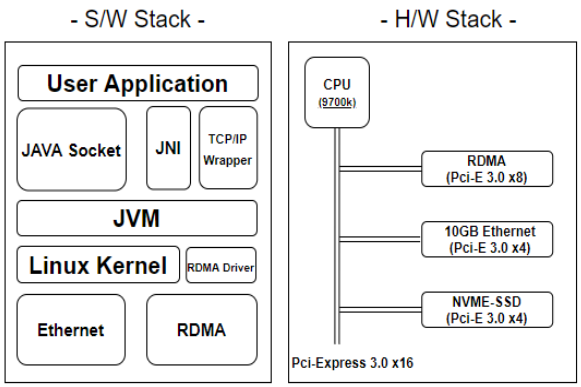


그림 10. 시스템 SW Stack과 H/W 구성
Fig. 10. Configuration of system SW stack and HW stack

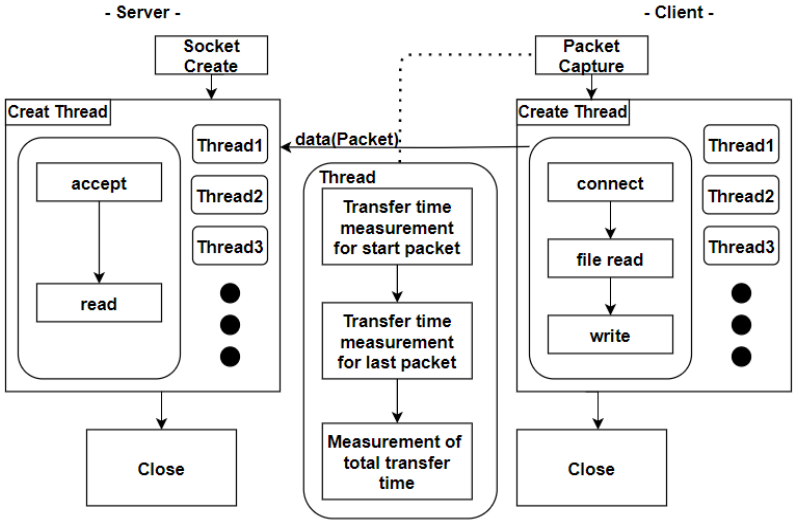


그림 11. Benchmark program 동작 구성도
Fig. 11. Configuration of benchmark program operation

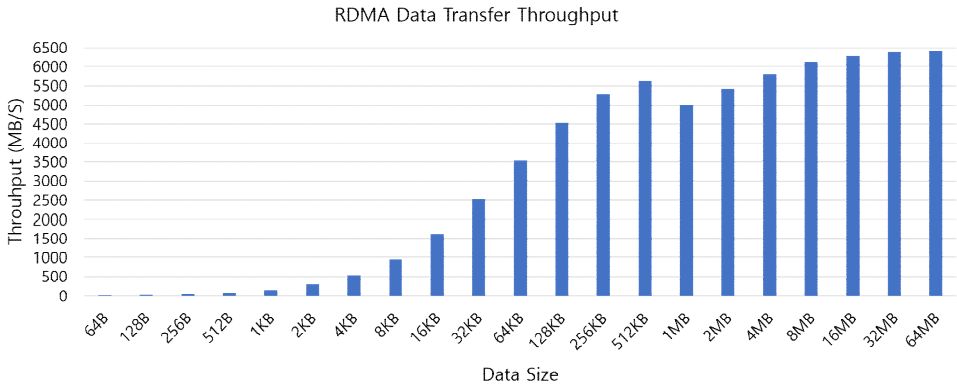


그림 12. 데이터 크기에 따른 Native PCIe NTB RDMA 의 성능 측정 결과
Fig. 12. Measurement results of PCIe NTB RDMA native performance according to data size

실험 평가 Metric은 Throughput이며, 평균 Throughput은 전송 파일 크기를 전체소요시간으로 나눈 값이다. 전체소요시간은 JNI 구현체는 시작 세그먼트부터 마지막 세그먼트의 전송 시간을, Ethernet은 해당 NIC(Network Interface Card)로 전송되는 Packet을 캡처하여 시작 패킷과 마지막 패킷의 전송 시간을 측정하였다.

4.2 실험 결과

먼저, PCIe NTB의 RDMA 성능평가를 시행하였다. 성능평가 방법은 데이터 크기를 64Bytes부터 64MB까지 늘려가면서 1000번씩 데이터 전송을 하였다. 그림 12는 데이터 크기에 따른 RDMA Throughput을 도식화한 것이다. 그림에서 보는 바와 같이 PCIe RDMA의 성능은 데이터 크기가 증가할수록 대역폭이 증가해서 최대 6.4GB/s까지 증가함을 확인할 수 있다. PCIe NTB 3.0의 표준에 근접할 정도로 RDMA 성능이 나옴을 확인할 수 있다.

다음으로 RDMA JNI에 대한 성능평가를 시행하였다. 서버와 클라이언트에서 다중 전송 시 RDMA JNI와 기존 Ethernet Socket의 성능 비교를 위해서 동시에 전송하는 스레드 개수를 2개 및 4개로, 데이터 전송 시 버퍼의 크기에 따른 성능을 평가하기 위해서 버퍼의 크기를 1MB에서 8MB까지 변경하면서 10Gbit Ethernet Socket과 RDMA JNI의 Throughput을 측정하였다.

그림 13은 Thread 개수 2개에 대해서 파일 크기

1GB~8GB를 전송할 때 버퍼 크기를 1, 2, 4, 8MB로 변화시키면서 변화에 따른 Ethernet Socket과 JNI Throughput을 측정해서 도식한 결과이다. 4개의 그래프에서 Ethernet Socket의 Throughput은 810.9567MB/s~949.15MB/s이나, JNI 인터페이스의 Throughput은 979.47MB/s~1337.5MB/s로 높으며 성능이 점점 증가함을 알 수 있다. 버퍼의 크기 증가에 따른 Throughput의 결과에서 볼 수 있듯이 버퍼의 크기가 증가함에 따라 RDMA JNI 인터페이스의 경우 Throughput이 비례해서 점점 증가함을 볼 수 있다. 이는 버퍼의 크기가 증가할수록 Native Method가 call 되는 횟수가 줄어들어 JNI Overhead가 감소하며 RDMA의 성능을 높일 수 있기 때문으로 볼 수 있다. 파일 크기가 1GB에서 8GB로 커질수록 성능 차이가 점점 더 벌어짐을 알 수 있다. 이는 JNI의 버퍼 활용이 Ethernet Socket의 버퍼 활용보다 더 효율적임을 알 수 있다.

쓰레드 개수를 4개로 증가시켜서 실험한 결과를 그림 14에 도식화하였다. 그림에서 보는 바와 같이 쓰레드의 개수가 4개일 때도 Ethernet Socket보다 JNI 인터페이스의 Throughput이 좋아짐을 알 수 있다. 주목할 것은 JNI 인터페이스의 실험 결과에서 알 수 있듯이, 버퍼의 크기가 증가함에 따라서 Throughput이 점점 증가하는 것이다.

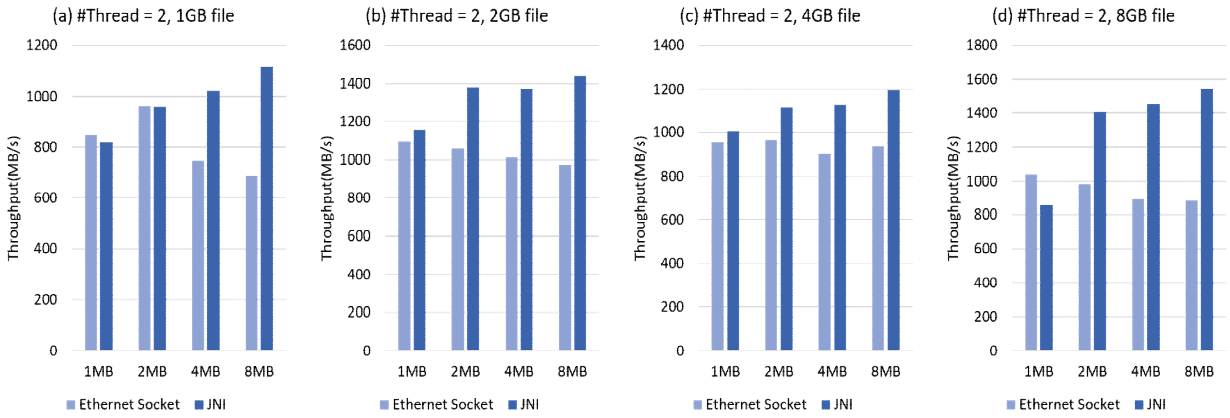


그림 13. Thread 개수가 2개일 때 버퍼 크기 1, 2, 4, 8MB 변화에 따른 Ethernet Socket과 JNI 성능 비교
Fig. 13. Ethernet socket and JNI performance comparison according to the change in buffer size 1, 2, 4, 8MB when the number of threads is 2

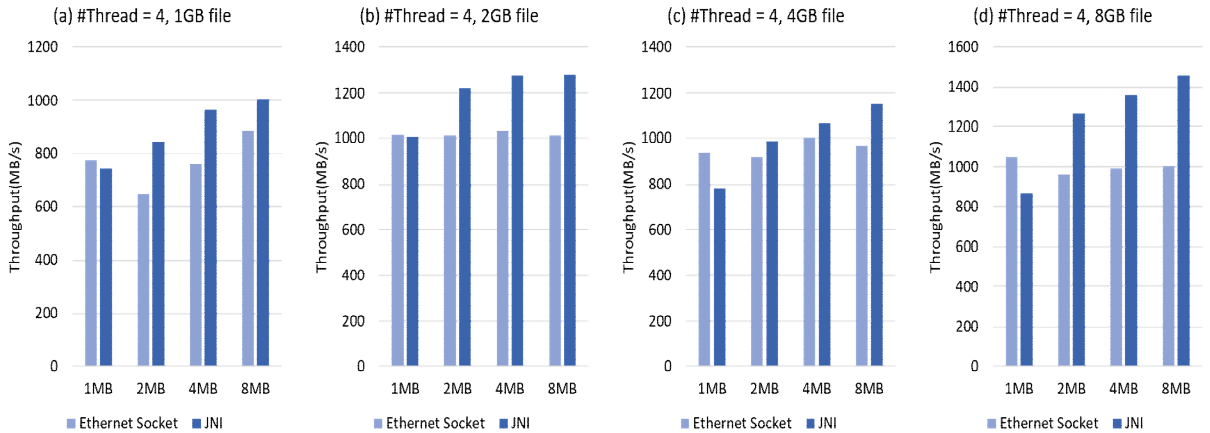


그림 14. Thread 개수가 4개일 때 버퍼 크기 1, 2, 4, 8MB 변화에 따른 Ethernet Socket과 JNI 성능 비교
Fig 14. Ethernet socket and JNI performance comparison according to the change in buffer size 1, 2, 4, 8MB when the number of threads is 4

또한, 동시 다중 전송을 위한 스레드 개수에 따른 성능을 보면, 결과에서 알 수 있듯이 멀티 스레드 환경에서 JNI로 구현한 RDMA의 성능이 단일 쓰레드에 비해서 잘 나옴을 알 수 있다. 그러나 쓰레드의 개수가 많이 증가한 만큼 Throughput이 비례해서 증가하지는 않은데, 이는 RDMA의 처리 방식에서 인터럽트가 쓰레드 별로 독립적으로 처리되지 않는 동기화 문제에서 비롯됨을 파악할 수 있었다. 마찬가지로, RDMA native의 성능에 비해서는 Throughput이 많이 저하되는 것을 알 수 있는데, 이러한 차이는 Java에서 Native Method를 호출하는 인터페이스에서 오버헤드가 있다는 것을 확인할 수 있는 부분이다. 현재 PCIe API에서 Interrupt가 thread 별로 독립적으로 동작하지 못하는 불안정한 동작 방식이 존재하는데, 만약 PCIe NTB의 인터럽트 방식을 thread-safe 한 방식으로 동작하게 한다면 JNI 인터페이스 오버헤드를 줄일 수 있을 것으로 기대하며, 쓰레드의 개수에 비례해서 JNI 인터페이스의 성능이 많이 향상될 것으로 기대하며, 결과적으로 RDMA Native의 Throughput을 잘 활용할 수 있을 것으로 기대한다.

Ⅴ. 결론 및 향후 과제

빅데이터의 발전으로 인해서 HDFS와 같은 대용량 클러스터 시스템의 사용은 점점 증가하고 있다. HDFS, SPARK등 다중의 노드를 연결하여 클러스터로 구성하는 HPC 시스템은 인터커넥트 네트워크 성능이 전체 시스템의 성능에 영향을 많이 미치며, 인터커넥트 네트워크에서 RDMA는 네트워크의 데이터 전송 속도를 높일 수 있는 전송방식이다. 이때까지 인터커넥트 네트워크로 Infiniband와 같은 네트워크 전송방식을 주로 많이 사용해왔으나, PCIe 표준 자체가 발전하고 있는 상황에서 PCIe 네이티브 표준인 PCIe NTB에 기반한 인터커넥트 Network 시스템에 관한 연구는 많이 없다. 한편, 최근에는 VM 기반 언어인 Java가 HDFS(Hadoop Distributed File System) 등과 같이 시스템의 구현 방식으로 활용되고 있으나, JAVA에서 RDMA를 직접 사용할 수 있는 인터페이스가 잘 제공되지 않는다.

본 논문에서는 Dolphin 사의 PCIe NTB 기반의 인터커넥트 네트워크에서 Native C API로 제공되는 RDMA 통신을 JNI를 통해 자바에서 구현하고, 이를 Ethernet Socket과 비교함으로써 실 활용에서의 성능을 고찰해 보았다. 결과적으로 RDMA JNI 방식이 기존 Ethernet Socket보다 10~20%의 Throughput이 증가하는 성능향상을 도출할 수 있었다. 그러나 JNI Overhead와 Interrupt 처리 부분에 있는 오버헤드로 인해서 RDMA 네이티브 인터페이스 자체의 성능에는 못 미치는 수준이다. 이러한 Overhead를 줄일 수 있다면 RDMA JNI를 통한 성능향상을 많이 높일 수 있을 것으로 기대한다. 우리는 이러한 개선을 위한 작업을 향후 진행해 나갈 예정이다.

References

[3] M. Choi and J. H. Park, "Feasibility and performance analysis of RDMA transfer through PCI Express", J. Information Processing System, Vol. 13, No. 1, pp. 95-103, Feb. 2017.

[4] S. Lim, K. Park, and K. Cha, "Developing an OpenSHMEM Model Over a Switchless PCIe Non-Transparent Bridge Interface", 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Rio de Janeiro, Brazil, pp. 593-602, May 2019.

[5] PCI-SIG Peripheral Component Interconnect Special Interest Group, [online] Available: https://pcisig.com/. [accessed: Jan. 01. 2019]

[6] Youngwoong Ju and Min Choi, "Design and Implementation of OpenSHMEM-Light using PCIe NTB", Proceedings of Computer Science Engineering Research Information Center, Vol. 23 No. 02, pp. 0058-0061, Nov. 2016.

[7] Dolphins, SISCI API Functional Specification, https://www.dolphinics.com/download/SISCI/OPEN_DOC/SISCI_API_2_functional_specification.pdf [accessed: Oct. 01. 2019]

[8] Dolphin, Dolphin PCI Express, https://www.dolphinics.com/download/PX/OPEN_DOC/PXH810_

users_guide.pdf [accessed: Oct. 01, 2019]
[9] PLX Technology, "ExpressLane PEX8749 PCI ExpressGen 3 Multi-Root Switch with DMA Data Book", Technical Report, 2013.
[10] Oracle, Java Native Interface Specification, https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html [accessed: March. 01, 2020]
[11] Sang-Kyeum Kim, Yangwoo Lee, and Seung-Ho Lim, "Implementation of Ring Topology Interconnection Network with PCIe Non-Transparent Bridge Interface", KIPS Transactions on Computer and Communication Systems, Vol. 8, No. 3, pp. 65-72, Mar. 2019.

저자소개

지 민 재 (Min-Jae Ji)

2015년 3월 ~ 현재 : 한국외국어대학교 컴퓨터공학부
관심분야 : 빅데이터, 분산처리시스템, HDFS, Java Native Interface

고 병 현 (Byeong-Hyun Ko)

2015년 3월 ~ 2021년 2월 : 한국외국어대학교 컴퓨터공학부
관심분야 : AI, 빅데이터, 분산처리시스템

신 동 렬 (Dong-Ryeol Sin)

2015년 3월 ~ 현재 : 한국외국어대학교 컴퓨터공학부
관심분야 : 빅데이터, 분산처리시스템, HDFS

김 성 현 (Seong-Hyun Kim)

2015년 3월 ~ 현재 : 한국외국어대학교 컴퓨터공학부
관심분야 : 빅데이터, 분산처리시스템, Java Native Interface

임 승 호 (Seung-Ho Lim)

2001년 2월 : 한국과학기술원 전기 및 전자공학과(공학사)
2003년 2월 : 한국과학기술원 전기 및 전자공학과(공학석사)
2008년 2월 : 한국과학기술원 전기 및 전자공학과(공학박사)
2008년 3월 ~ 2010년 2월 : 삼성전자 메모리 사업부 책임 연구원
2010년 3월 ~ 현재 : 한국외국어대학교 컴퓨터공학부 교수
관심분야 : 운영체제, 파일 시스템, 임베디드 시스템, DLA for AI, 비휘발성 메모리 시스템, 빅데이터 처리, Hadoop, HDFS