제 18 회 한국정보처리학회 추계학술발표대회 논문집 제 9 권 제 2 호 (2002.11)

JNI를 이용한 소켓 구현

조희남, 정명조, 차태성, 이철훈
충남대학교 컴퓨터공학
e-mail : hncho@ce.cnu.ac.kr

Implementation of Socket Using JNI

Hee-Nam Jo, Myung-Jo Jung, Tae-Sung Cha , and Cheol-Hoon Lee
Dept. of Computer engineering, Chung-Nam National University

요 약

JNI 는 자바언어로 이루어진 응용프로그램과 C 나 C++같은 다른언어로 구현된 응용프로그램 사이에 양방향 인터페이스를 제공하는 자바 플랫폼의 강력한 특징중 하나이다. 소켓의 경우 자바언어만으로는 호스트에 의존적인 특징을 지원하지 못하기에 필연적으로 Native 메소드를 사용해야만 한다. 본 논문은 호스트 의존적인 부분에 대해 JNI 을 사용하여 연결지향 프로토콜에서 이용되는 소켓을 구현한다.

1. 서론

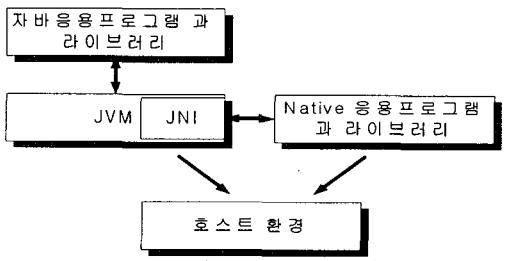
자바 플랫폼이 프로그래머들에게 자바언어만으로 프로그램밍하게 만들었다면 Legacy 코드에 대한 그들의 노력은 쓸모없는 것이 되었을 것이다. 그러나 자바 플랫폼은 Java™ Native Interface(JNI)라는 Native 코드와의 양방향 인터페이스를 제공함으로써 Legacy 코드들을 효율적으로 사용할 수 있도록 하고 있다.

본 논문은 이러한 JNI 을 사용하여 플랫폼에 의존적인 소켓을 자바언어와 함께 C 언어로 구현한다. 본 논문의 구성은 다음과 같다. 2 장에서 JNI , Native 라이브러리의 로딩, 그리고 구현하고자하는 소켓에 대해 설명하고 3 장에서는 JNI 을 이용한 소켓의 설계 및 구현에 대하여 기술하며 4 장에서는 실험결과에 대해서 살펴보고 마지막으로 5 장에서는 결론 및 향후과제에대해 기술한다.

2. 관련연구

2.1 자바 플랫폼과 호스트 환경에서의 JNI

자바 플랫폼은 자바가상머쉰(VM)과 Java Application Programming Interface(API)으로 이루어진 프로그래밍 환경이다. 자바 응용프로그램들은 이러한 API 을 이용하여 구현되며, 기계독립적인 이진 클래스 형식으로 컴파일된 후 어떠한 VM 에서도 작동되도록 되어 있다. 호스트 환경은 호스트의 운영시스템을 뜻하며 운영시스템은 Native 라이브러리 집합들과 CPU 명령어 집합으로 이루어져 있다. Native 응용프로그램들은 C 나 C++같은 Native 언어들로 프로그래밍되며 Native 라이브러리와 링킹되어 호스트에 의존적인 이진 코드로 컴파일 된후 실행된다. 자바 플랫폼은 호스트 환경의 상위 계층에 배치되는데 JNI 는 이러한 호스트 환경과 자바 플랫폼 사이에서 양방향 인터페이스를 제공함으로써, 서로 다른 언어를 사용할 수 있도록 해준다. [그림 1]은 JNI 의 역할에 대해 보여주고 있다.



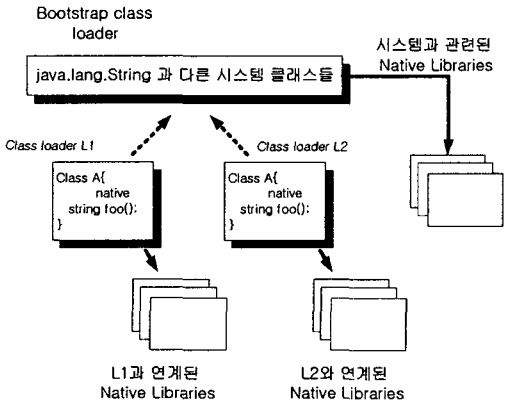
[그림 1] JNI 의역할

그러나 이러한 JNI 를 사용하게 되면 자바의 플랫폼 독립성이 깨지게 되는데 이는 자바의 언어는 여전히 플랫폼 독립적이지만 Native 언어는 호스트 환경에 맞게 다시 컴파일 해주어야 함을 의미한다. 또한 자바언어의 형 안정성(type-safe)이 보장되지 못하는데 이것은 C 나 C++ 같은 Native 언어들이 형 안정성을 지원하지 못하기 때문이다. 형안정성이 보장되지 않는 C

나 C++로 잘못 구현된 Native 메소드의 사용은 자칫 전체 응용프로그램의 붕괴를 초래할 수 있으므로 Native 메소드를 사용하는데 있어서는 신중을 가해야 한다.

2.2 Native 라이브러리의 로딩

JDK1.2 이후로 자바는 정형화된 Parent-delegation 모델을 사용하여 클래스들을 로딩하게 된다. Parent-delegation 모델은 가장 하위 단계의 클래스로더가 부모 클래스 로더에게 필요로하는 타입을 로딩해줄 것을 요청하면 부모 클래스 로더역시 그 자신의 부모 클래스로더에게 같은 요청을 하면서 일치하는 타입을 찾는 모델을 말한다. [그림 2]는 클래스로더와 Native 라이브러리와의 관계를 도식화하는 것으로서, 그림에서 보는 것과 같이 각각의 클래스 로더는 자기자신의 Native 라이브러리 집합을 유지하며 각각의 클래스 로더는 독립적인 성격을 띄게 된다. 즉 L1 에서 L2 의 Native 라이브러리를 접근할 수는 없다.



[그림 2] Native 라이브러리의 로딩

2.2 소켓

클라이언트가 서버측 주소와 포트번호를 갖고 소켓을 생성하여 서버측에 요청 메시지를 보내게되면 서버는 클라이언트와의 통신을 위해서 도착한 요청 메시지에 대해 새로운 소켓을 생성하게 된다. 이러한 소켓은 Time-Out, Linger-on-Close, 그리고 Nagle's 알고리즘을 사용할 것인지에 대한 옵션을 지원해야 한다.

(1) Socket Time-Out Period
Time-out 은 데이터가 소켓에 도착할 때까지 읽기 연산이 차단되는 시간 간격을 말한다.

(2) Linger-on-Close
Linger-on-Close 은 소켓을 닫을 때 소켓이 어떤 행동을 취해야 하는가에 대해 영향을 미치는 옵션이다. Linger-on-Close 옵션이 설정되지 않으면 소켓은 close() 메소드를 바로 호출하게 되며 전송되지 않고 큐잉되어 있는 데이터를 모두 전송하게된다. Linger-on-close 가 설정되면 Time-Out 값이 매개변수값으로 입력되는데, Time-Out 기간동안 큐에 있는 데이타가 전송될 때가지 close()메소드는 블로킹되게된다. Time-Out 값이 0 이면 close()메소드가 바로 호출되며 큐잉된 모든 데이터는 버려진다.

(3) TCP Data Coalescing
많은 작은 패킷의 전송은 혼잡과 오버헤드의 원인이다. 대부분의 TCP 프로토콜의 경우 Nagle's 알고리즘 기법을 사용하는데, 이는 전송되는 많은 작은 패킷의 오버헤드를 줄이기 위해서 일단 처음 패킷이 ACQ 를 받으면 연속적으로 데이터를 전송하여 패킷의 수를 줄이는 기법으로, 오버헤드 및 혼잡을 방지한다. 본 논문에서 구현하고자 하는 소켓도 Nagle's 알고리즘을 지원할 것인지 아닌지에 대한 옵션을 제공한다.

3. 설계 및 구현

3.1 자바 언어부

클라이언트는 처음 비연결 소켓을 생성한 후, 소켓을 사용하고자 할 때 서버와의 연결을 시도한다. 서버는 일단 자기자신의 주소를 갖고 소켓을 생성한 후 요청 메시지가 도착하면 그에 해당하는 새로운 소켓을 생성하고 클라이언트와 통신을 시작한다. 더 이상 소켓을 사용하지 않을 때에는 반드시 소켓을 닫아주어 시스템 자원이 효율적으로 재사용될 수 있도록 해주어야 한다. 자바 언어부에서는 위의 기술된 내용을 지원키 위해서 9 개의 native 메소드가 필요하다.

(1) localSocketAccept()
소켓에 대한 연결 요청을 승인하는 메서드
(2) localSocketBind()
소켓과 로컬 포트번호와의 바인딩
(3) localSocketClose()
소켓 닫기
(4) localSocketConnect()
소켓과 도착 주소와의 연결
(5) localSocketCreate()
비연결 소켓의 생성
(6) localSocketListen()
stream 소켓에 대한 연결 요청을 기달리는 메소드
(7) localSocketSetOption()
소켓이 지원해야 하는 옵션들을 설정하는 메소드
(8) localSocketGetOption()
설정된 옵션값이 무언인지를 돌려주는 메소드
(9) localNeedInit()
메소드 코드내에서 글로벌하게 사용하게될 변수들의 초기화에 필요한 Native 메소드

[그림 3]은 자바 클래스에서 Native 메소드가 선언된 것을 보여준다.

```
private native void localSocketSetOption(int option, Object data) throws SocketException;
private native int localSocketGetOption(int option) throws SocketException;
private native void localSocketAccept(SocketImpl sock);
```

```
private native int   localSocketAvailable();
private native void localSocketBind(InetAddress addr, int
port);
private native void localSocketClose();
private native void localSocketConnect(InetAddress
addr, int port);
private native void localSocketCreate(boolean stream);
private native void localSocketListen(int count);
private static native void localNeedInit();
```

[그림 3] native 메소드 목록

이 밖에 자바클래스에서는 소켓 연결 후 데이터 전송과 관련된 IO stream 에 대한 메소드를 제공한다. 관련연구에서 설명된것처럼 클래스로더가 Native 라이브러리를 로딩하기 위해서는 System.loadLibrary 라는 메서드를 반드시 호출해주어야 하며, 솔라리스의 경우 로딩하고자 하는 Native 라이브러리명 앞에 lib 이라는 접두어와 .so 라는 확장자가 붙게된다. [그림 4]는 구현된 자바클래서에서 Native 라이브러리의 호출 부분으로 라이브러리 명은 libnetsocket.so 이라는 이름으로 로딩되게 된다.

```
static
{
        System.loadLibrary("netsocket");
        localNeedInit();
}
```

[그림 4]Native 라이브러리의 로딩

3.2 Native 언어부

본 논문에서 구현된 Native 언어는 C 언어이며 다음과 같은 환경에서 프로그래밍 되었다.

Machine hardware:	sun4u
OS version:	5.7
Processor type:	sparc
Hardware:	SUNW,Ultra-Enterprise

[그림 5] 호스트 환경

C 로 구현된 native 는 플랫폼에 의존적인 부분에 대한 것으로 자바 언어부에서 명시한 native 메서드의 실제적인 구현부분이다. [그림 6]은 native 메서드의 입력 형식을 보여준다.

```
/*
 * Class:      클래스이름
 * Method:     메소드 이름
 * Signature:  signature
 */
JNIEXPORT void JNICALL Java_<클래스이름>_<메서드
이름> (JNIEnv *, jobject, jint);
```

[그림 6] native 메소드 입력형식

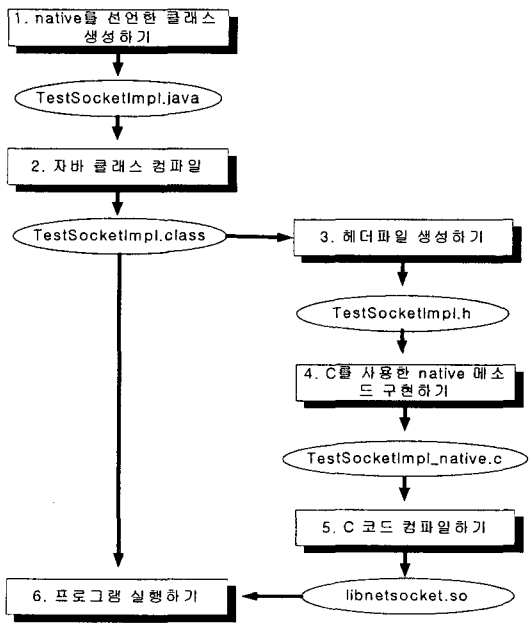
위의 형식은 자바에서 지원하는 javah 의 -jni 옵션을 주면 자동적으로 생성이 되면 [그림 7]은 [그림 6]의 형식을 이용하여 실제 구현한 Native 메소드의 구현을 보여준다.

```
// 헤드부분 일부생략
#include "TestSocketImpl.h"
#include "net_util.h"
// 중략
/*
  * Class:      TempSocketImpl
  * Method :   localNeedInit native method
  * Signature: ()V
  * 설명    : 자바 클래스 필드내의 초기값들을 할당받아
다른 native 함수 구현시 필요한 전역변수를 선언코자 하
여 만들어진 initial 메소드
*/
JNIEXPORT void JNICALL
Java_TempSocketImpl_localNeedInit(JNIEnv *env, jclass cls){
  jclass class= (*env)->FindClass(env, "java/io/FileDescriptor");
  need_fdID = (*env)->GetFieldID(env, cls , "fd",
    "Ljava/io/FileDescriptor;");
  need_addID = (*env)->GetFieldID(env, cls, "address",
    "Ljava/net/InetAddress;");
  need_pID = (*env)->GetFieldID(env, cls, "port", "I");
  need_lpID = (*env)->GetFieldID(env, cls, "localport", "I");
  need_toID = (*env)->GetFieldID(env, cls, "timeout", "I");
  CLS_fd_fdID=(*env)->GetFieldID(env,class,"fd","I");
  if(socketEx == NULL){
     socketEx = (*env)->FindClass(env,
       "java/net/SocketException");
     socketEx = (jclass)(*env)->NewGlobalRef(env,socketEx);
     if(socketEx==NULL)
       return;
   }
}
```

[그림 7] Native 메소드

3.3 통합과정

자바언어와 Native 언어의 통합은 [그림 8]과 같이 6 단계를 거쳐 이루어진다.



[그림 8] Native 코드와 자바 코드의통합 단계

(1) 클래스 파일 생성하는 단계이다. 여기서는 TestSocketImpl.java 라는 이름의 Native 메소드를 선언한 클래스를 사용한다.
(2) TestSocketImpl.java 를 컴파일하는 단계이다.
(3) javah 의 -jni 옵션을 이용하여 TestSocketImpl.h 이라는 헤더파일을 만든다.
(4) native 메소드 구현하는 단계이다.
(5) 호스트기반의 Native 파일을 컴파일하는 단계이며 아래 그림은 Native 파일을 컴파일하기위한 makefile 이다.

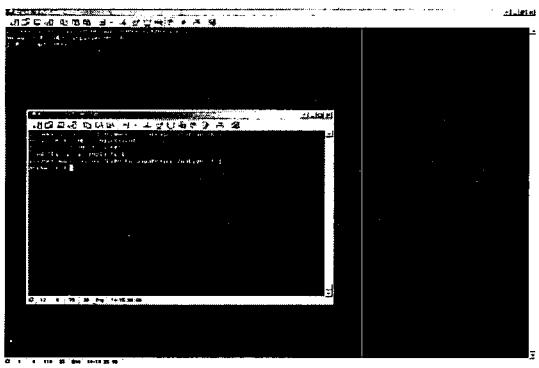
```
.SUFFIXES=.o
CFLAGS = -G
OBJS = TestSocketImpl_native.c
SRCS = $(OBJS:.o=.c)
CC = gcc
TARGET = libnetsocket.so
$(TARGET) : $(OBJS)
        $(CC) -G -I ./include -I ./include/solaris -I .
$(OBJS) -o $(TARGET)
```

[그림 9] makefile 작성

(6) 실행단계로서, 실행을 위해서는 Native 라이브러리 패스를 지정해 주어야 하는데 솔라리스에서 C shell 를 사용하는 경우는 환경변수 LD_LIBRARY_PATH 를 다음과 같이 지정해 주어야 한다. 'setenv LD_LIBRARY_PATH .'

4. 실험결과

실험환경은 Native 메소드가 구현된 환경과 같다. 테스팅은 블랙박스 테스팅으로 이루어졌으며 모듈 테스트와 기능 테스트까지 이루어 졌다. [그림 10]은 하나의 결과화면이다.



[그림 10] 실험결과 화면

5. 결론 및 향후과제

본 논문는 소켓 구현시 플랫폼에 의존적인 부분을 JNI 를 사용하여 구현해 보았다. Native 메서드를 사용함으로써, 기존 Legacy 코드를 재활용할 수 있다는 큰 이점을 누릴 수 있고, 자바언어로는 구현할 수 없는 부분을 Native 언어로 구현함으로서 자바 프로그래밍의 깊이를 한차원 높일 수 있었다. 그러나 본 논문에서 구현된 소켓은 연결지향 프로토콜들에서만 사용되는 것으로써 앞으로는 Datagram 패킷을 서로 주고받는 비연결지향 프로토콜들에서 사용할수 있는 소켓도 JNI 사용하여 구현되어야 할 것이다.

참고문헌

[1] Sheng Liang, *The Java™ Native Interface*,June 1999
[2] Sun Microsystems,Inc., *Java™ Native Interface Specification, 1997*
[3] W.Richard Stevens, *Unix Network Programming*,1998
[4] Elliotte Rusty Harold, *Java™ secrets*, 1997
[5] Sun Microsystems, Inc., *Java™ Object Serialization Specification*, Revision 1.4.3, JDK™ 1.2 Beta4, September 30,1998
[6] Bill Venners, *Inside the Java Virtual Machine*,1999