# MiTS 통신을 위한 메시지 처리 모듈의 설계 및 구현

김태종* · 황훈규* · 서정민* · 윤진식* · 이장세** · 장길웅** · 박휴찬***

## A Design and Implementation of Message Processing Modules for MiTS Communication

Tae-Jong Kim* · Hun-Gyu Hwang* · Jeong-Min Seo* · Jin-Sik Yoon* · Jang-Se Lee** · Kil-Woong Jang** · Hyu-Chan Park***

이 논문은 2009년도 지식경제부 및 정보통신산업진흥원 IT핵심기술개발사업의 연구비를 지원받았음

요 약

MiTS 통신은 선박 내에서 발생하는 다양한 형태의 정보를 시스템 상호 간에 교환하기 위한 프로토콜이다. 또한, 이러한 정보 교환을 위한 약 30여 가지의 특수한 메시지 포맷도 정의되어 있다. 따라서, MiTS 통신 프로토콜을 따르는 시스템의 구축을 위해서는 이러한 메시지를 일관성 있게 처리해야 한다. 본 논문에서는 이를 위한 메시지 처리 모듈의 설계와 구현을 제시한다. 또한, 구현된 모듈의 테스트 결과를 제시한다.

ABSTRACT

MiTS communication is a protocol for the exchange of various information among shipboard systems. It also specifies more than 30 specialized message formats for the information exchange. To build MiTS systems, therefore, the messages should be processed consistently. This paper proposes the design and implementation of such message processing modules. It also describes the test results of the implemented modules.

키워드

MiTS, 정보 교환, 메시지 처리, IEC61162-4

Key word

MiTS, Information exchange, Message processing, IEC61162-4

* 한국해양대학교 대학원
** 한국해양대학교
*** 한국해양대학교 (교신저자)

접수일자 : 2009. 12. 05
심사완료일자 : 2010. 01. 14

## Ⅰ. 서 론

MiTS(Maritime information Technology Standard)는 선박에서 발생하는 다양한 형태의 정보를 통합하여 관리하고 교환하기 위한 표준으로 제안되었다. 이후, 세계적인 표준화 기구인 IEC(International Electrotechnical Commission)에 의해 IEC61162-4 표준으로 발전 채택되었다[1].

이러한 MiTS는 선박 내에서 네트워크로 연결되어 있는 시스템 상호간에 정보를 교환할 수 있는 통신 프로토콜을 명세하고 있다. 특히, 정보 교환을 위한 약 30여 가지의 특수한 메시지 포맷도 정의하고 있다. 따라서 MiTS 통신 프로토콜을 따르는 시스템은 이 메시지 포맷에 따라 정보를 송수신해야 한다. 즉, 송신하는 측에서는 데이터를 메시지 형태로 변환한 후 전송해야 하며, 수신하는 측에서는 수신된 메시지를 데이터 형태로 역변환한 후 사용해야 한다. 따라서 선박에서 MiTS 시스템을 구축하기 위해서는 이러한 메시지 처리 모듈의 개발이 필수적이다.

본 논문에서는 이러한 메시지 처리 모듈의 설계와 구현 결과를 제시한다. 즉, 데이터를 메시지 형태로 패킹(packing)하는 기능과 메시지를 데이터 형태로 언패킹(unpacking)하는 기능에 대하여 설명한다. 또한 구현된 모듈에 대한 기능 테스트 결과를 제시한다. 이러한 메시지 처리 모듈을 실제로 활용한 선박용 통합 정보시스템이 개발되고 있다[2,3].

이 논문의 구성은 2장에서 MiTS 통신에 대하여 개략적으로 소개하고, 3장에서 메시지 처리 모듈의 설계에 관하여 다루며, 4장에서는 실제 구현 및 테스트 결과에 대하여 기술한 후, 5장에서 결론 및 향후연구로 끝을 맺는다.

## Ⅱ. MiTS 통신 개요

### 2.1 통신 프로토콜

MiTS 통신 프로토콜을 따르는 시스템은 다양한 형태의 구조가 가능하다. 그림 1은 그 중 하나의 가능한 시스템 구조를 보여주고 있으며, 크게 게이트웨이(G/W), 미들웨어 서버(M/W Server), 어플리케이션(APP)의 세 시스템으로 구성되어 있다. 각 시스템은 정보의 교환을 위하여 LNA(Local Network Administrator)와 MAU(MiTS Application Unit)를 가진다[2,3].

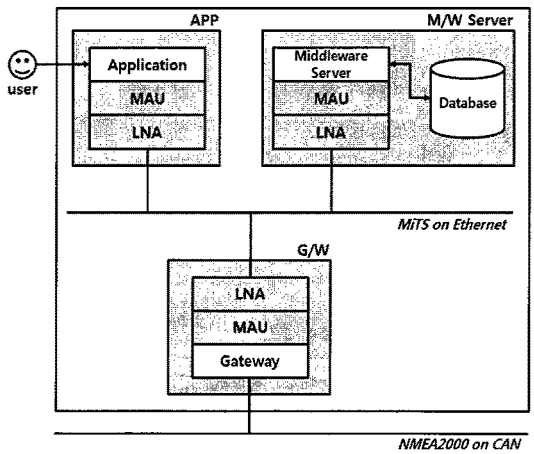


그림 1. MiTS 시스템 구조
Fig. 1 System architecture of MiTS

정보의 교환을 위한 통신 모듈인 LNA와 MAU 간의 관계는 그림 2에 나타나 있다[1]. 그림에서 알 수 있듯이, 각 호스트 컴퓨터는 LNA와 MAU를 가질 수 있으며, LNA 간에는 상호 통신할 수 있고 MAU는 LNA의 통신 서비스 사용할 수 있다. LNA는 하나 이상의 MAU를 서비스할 수 있으며, MAU는 상위 어플리케이션과 통신한다.

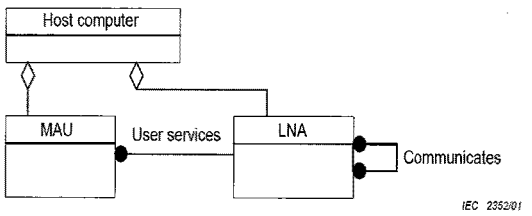


그림 2. LNA와 MAU의 관계[1]
Fig. 2 Relationship of LNA and MAU

LNA는 네트워크를 관리하고 MAU 간의 메시지 통신을 담당하는 모듈이다. 또한, 세션 관리(session management), 트랜잭션 관리(transaction management), MAU 이름 검색(MAU name look-up)기능을 담당한다.

특히 세션 관리를 통하여 MAU의 연결, 상태변경, 해지 등을 할 수 있으며, 원격 LNA와의 연결 유지 및 해제도 가능하다. 각 LNA들은 원격 LNA와 연결 시, 서로의 정보를 교환하기 위하여 핸드쉐이킹 프로토콜(handshaking protocol)을 사용한다.

MAU의 기능은 크게 상태를 관리하는 기능(state management), MAU 내부의 접속점들 간의 세션을 관리하는 기능(session control), 보안을 위해 인증을 담당하는 인증 기능(authentication), 통신 문제를 해결하는 기능(congestion control), 내부 접속점들의 집합인 인터페이스를 관리하는 기능(interface management) 등으로 나눌 수 있다. 그 외에 대용량 전송을 위한 기능(bulk data transfer)도 가진다.

MAU와 LNA간, 혹은 LNA와 LNA간의 상호 통신은 약속된 메시지를 주고받음으로써 데이터 및 제어의 신뢰성을 높게 만든다. 메시지 처리 모듈은 이런 약속된 메시지를 만들고, 받은 메시지를 해체하여 형태를 분석하며, 그에 따른 처리를 하는 모듈이라 할 수 있다. 메시지를 만들기 위하여 메시지 포맷 분석과 메시지 처리를 위한 모듈은 다음 장에서 자세히 설명한다.

### 2.2 메시지 포맷

MiTS 통신에서는 여러 형태의 메시지가 존재 할 수 있다. 그림 3에서 옥텟으로 구분된 여러 형태의 메시지 중 몇 가지의 예를 볼 수 있다[1].

그림의 (a)는 세션 메시지의 형태이며, (b)는 목적지가 메시지를 수신할 수 있는 상태인지 검사하는 메시지 형태이다. 또한 (c)는 데이터를 요청하는 메시지 형태이며, (d)는 데이터를 전송하는 메시지 형태이다.

특히 그림 3의 (c)와 (d) 메시지 내의 옥텟 순서에 따른 각 필드를 자세히 보면, 처음 두 옥텟은 MAGIC_CODE 필드이다. MAGIC_CODE라는 것은 메시지의 처음을 나타내는 필드로써 여러 메시지가 동시에 전송되는 경우 각 메시지를 구분할 수 있는 필드이다. 그 다음의 두 옥텟(LL_DATA, MAPI_xREQ)은 메시지의 타입을 나타내는 필드이다. 메시지 타입은 크게 2가지의 메시지로 나눌 수 있는데, MAU와 LNA간의 메시지, LNA와 LNA간의 메시지로 크게 나눌 수 있고 각각 14개, 15개의 타입을 가진다.

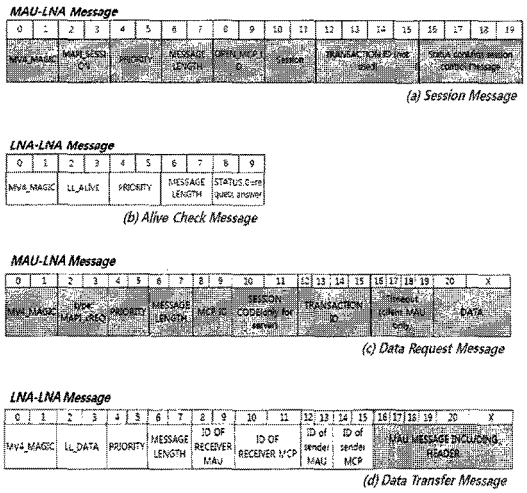


그림 3. 메시지 포맷의 예
Fig. 3 Examples of message format

그 뒤의 두 옥텟(PRIORITY)은 메시지의 우선순위를 나타내는 필드로 URGENT, NORMAL 그리고 LOW의 세 우선순위가 있다. 다음 두 옥텟은 메시지의 길이를 나타내는 MESSAGE LENGTH 필드이며 메시지의 길이는 MAGIC_CODE의 시작인 0번째 옥텟으로부터 데이터 마지막까지의 총 길이이다. LNA-LNA 메시지의 8번, 9번 옥텟은 수신자 MAU의 ID를 가지며 MAU-LNA 메시지의 8번, 9번 옥텟은 MCP의 ID를 가진다. 앞서 설명한 내용은 일반적인 메시지의 옥텟 순서이고 이후의 옥텟은 LNA-LNA 메시지, MAU-LNA 메시지에 따라 각각 다르게 나타난다. 또한 메시지도 여러 종류가 있으므로 메시지는 다양한 형태와 길이를 가질 수 있다.

## III. 메시지 처리 모듈 설계

### 3.1 메시지 처리 과정

메시지 처리 모듈은 메시지의 타입 필드를 분석하여 어떤 메시지인지 파악한 후 처리를 하도록 하는 기능을 가진 모듈이며 패킹 및 언패킹 과정으로 나누어진다. 이러한 시스템 상호간의 메시지 처리 과정은 그림 4에서 보여주고 있다.

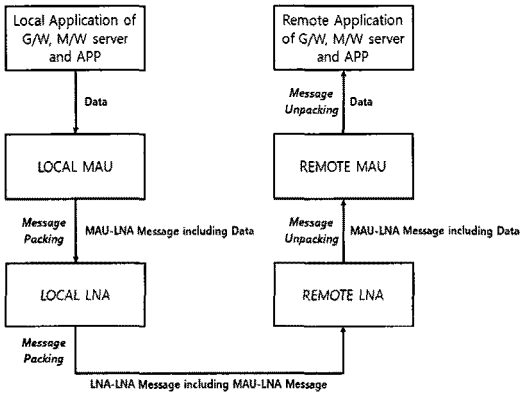


그림 4. 메시지 처리 과정
*Fig. 4 Procedure of message processing*

그림에서 알 수 있듯이, 게이트웨이, 미들웨어 서버, 혹은 어플리케이션이 상호 데이터를 전송하거나 데이터를 요청하고자 하면, 로컬 MAU는 목적에 맞는 메시지를 만들어 로컬 LNA에 보낸다. 또한 로컬 LNA는 로컬 MAU로부터 메시지를 받아 이 메시지가 어떤 목적의 메시지인지 판별하고 원격 LNA로 전송하기 위한 메시지를 만드는데, 이러한 과정이 메시지 패킹이다.

로컬 LNA에서 원격 LNA로 메시지를 전송한 후, 원격 LNA가 메시지를 수신하면 그 메시지가 어떤 메시지인지 확인하고 처리를 한 후에 원격 MAU로 전송하기 위하여 데이터 부분만 남기고 헤더 부분은 제거하게 된다. 또한 그 메시지를 수신한 MAU는 메시지의 형태를 보고 어떤 행동을 하게 될 것이다. 데이터를 요청하는 데이터일 경우 어떤 데이터를 요구하는지를 알기 위하여 메시지를 해체하게 된다. 이런 각각의 메시지를 해체하는 일련의 과정들이 메시지 언패킹이다.

메시지 패킹과 언패킹 모듈은 표준에 정의되어 있는 트랜잭션 타입 중에서 데이터를 요청하고 요청된 데이터를 응답하는 함수(function) 타입의 트랜잭션 메시지를 기반으로 설계하였다.

### 3.2 메시지 처리 모듈 설계

그림 5, 6은 MAU-LNA 메시지 및 LNA-LNA 메시지의 패킹 및 언패킹 모듈 구현을 위해 설계된 클래스 다이

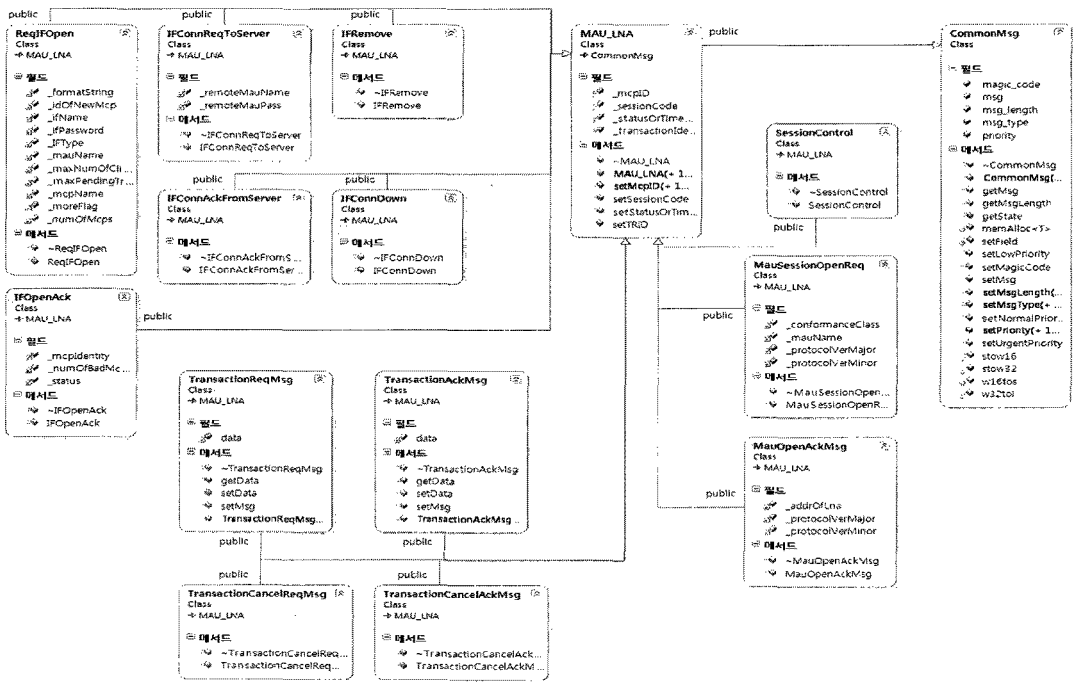


그림 5. MAU 메시지 클래스 다이어그램
Fig. 5 Class diagram for MAU messages

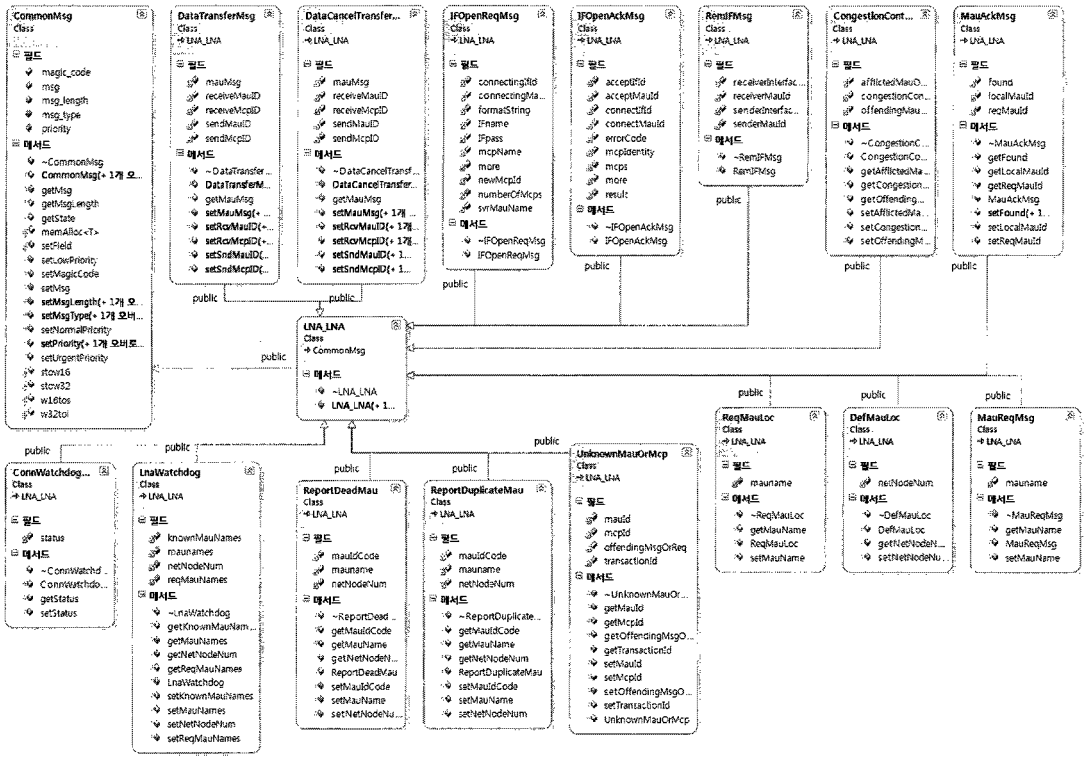


그림 6. LNA 메시지 클래스 다이어그램
Fig. 6 Class diagram for LNA messages

어그램이다. 모든 클래스들의 최상위 부모 클래스인 CommonMsg 클래스는 3.1절에서 설명한 메시지 옥텟 순서 중 8번째 옥텟까지를 나타낸다. 각각 필드들을 멤버 변수로 가지고, get~과 set~의 멤버 함수들을 이용하여 접근할 수 있도록 하였다. 또한 전체 메시지를 나타내는 멤버 변수를 두어 전체 메시지에 접근하기 용이하도록 설계하였다. 각각 필드에 맞는 데이터로 변환하고자 데이터 변환하는 멤버 함수들을 설계하고 구현하였다. CommonMsg 클래스로부터 상속받은 MAU_LNA 클래스는 14개의 MAU-LNA메시지의 공통부분 옥텟을, LNA_LNA 클래스는 15개의 LNA-LNA 메시지의 공통부분 옥텟을 멤버 변수화 하였고 CommonMsg와 같이 get~, set~ 함수를 통하여 멤버 변수에 접근할 수 있도록 하였다. 각각의 클래스에는 생성자를 여러 개를 두어 메시지를 패킹 및 언패킹 작업을 쉽게 할 수 있도록 설계하였다.

그림 5에서는 CommonMsg 클래스로부터 상속받은 MAU_LNA 클래스, MAU_LNA에서 상속받은 14개의 메시지를 각각 클래스로 설계하였다. MAU_LNA 클래스는 하위 14개의 클래스에 공통적으로 사용될 부분을 추출하였다.

그림 6에서는 CommonMsg 클래스로부터 상속받은 LNA_LNA 클래스, 그리고 그것으로부터 상속받는 15개의 메시지를 각각 클래스로 설계하였다. LNA_LNA 클래스는 하위 15개의 클래스에 공통적으로 사용될 부분을 추출하였다. 각각의 메시지를 처리하는데 편의를 두고자 이런 계층구조로 설계하였다.

## Ⅳ. 메시지 처리 모듈 구현

### 4.1 구현

그림 7은 LNA에서 소속된 MAU로부터 메시지를 받아 원격 LNA로 전송할 메시지로 만들어주는 패킹 부분이다. 각각의 함수는 인자로 입력부와 출력부로 나누어져 있다. 입력부는 메시지 타입, 데이터로서 각각의 메시지를 어떠한 메시지로 만들지에 대한 것과, 그 메시지에 담을 데이터로 구성된다. 출력부는 만들어진 메시지의 문자열과 만들어진 메시지의 길이로 구성된다.

메시지 타입과 MAU-LNA 메시지, 출력될 LNA-LNA메시지를 담을 공간, 출력될 메시지의 길이를 담을 공간을 입력으로 하여 함수를 호출하게 되면 패킹되어 LNA-LNA메시지를 반환한다. 그림에서는 데이터 전송 메시지 타입인 LL_DATA에 대한 코드를 보여주고 있다.

```
void pack(short msgType, char* inMsg, char* target, int* msgLen){
    int tmpMsgLen = 0;
    LNA_LNA* rtnMsg;
    MAU_LNA* tempMsg = new MAU_LNA(inMsg);
    switch(msgType){
        case LL_DATA:
            rtnMsg = new DataTransferMsg(tempMsg);
            rtnMsg->setNormalPriority();
            ((DataTransferMsg*)rtnMsg)->setRcvMauID(11);
            ((DataTransferMsg*)rtnMsg)->setRcvMcpID(22);
            ((DataTransferMsg*)rtnMsg)->setSndMauID(33);
            ((DataTransferMsg*)rtnMsg)->setSndMcpID(44);
            ((DataTransferMsg*)rtnMsg)->setMauMsg(tempMsg);
            break;
        case LL_ALIVE:
        case LL_DATAC:
        case LL_IFACK:
        case LL_IFREM:
        case LL_IFREQ:
        case LL_MAUACK:
        case LL_MAUREQ:
        case LL_NOMCP:
        case LL_SESSION:
//LNA_LNA multi-cast messages
        case CC_DEADMAU:
        case CC_DEFNAU:
        case CC_METOO:
        case CC_REQMAU:
        case CC_WATCHDOG:
        case LLBC_AACK:
        case LLBC_SACK:
            break;
    }
    tmpMsgLen = rtnMsg->getMsgLength();
    for(int i=0;i<tmpMsgLen;i++){
        target[i] = rtnMsg->getMsg()[i];
    }
    target[tmpMsgLen] = '₩0';
    *msgLen = tmpMsgLen;

    free(tempMsg);
    tempMsg = NULL;
    free(rtnMsg);
    rtnMsg = NULL;
}
```

그림 7. LNA에서의 메시지 패킹
Fig. 7 Message packing in LNA

그림 8은 원격 LNA로부터 전송받은 메시지를 MAU 메시지로 만들어주는 언패킹 부분이다. 메시지 타입과 LNA-LNA 메시지를 입력으로 하여 함수를 호출하게 되면 언패킹되어 MAU-LNA메시지를 반환한다. 그림에서는 데이터 전송 및 함수 로딩 요청 메시지 타입인 MAPI_FREQ와 데이터 전송 및 함수 로딩 요청 메시지의 응답 타입인 MAPI_FACK에 대한 코드를 보여주고 있다.

```
void unpack(short msgType, char* source, char* target, int* msgLen){
    int tmpMsgLen = 0;
    LNA_LNA* tempMsg = new LNA_LNA(source);
    if(tempMsg->getState() == LL_DATA){
        tempMsg = new DataTransferMsg(source);
        MAU_LNA* rtnMsg;
        switch(msgType){
            case MAPI_FREQ:
                rtnMsg =  new TransactionReqMsg(
                    ((DataTransferMsg*)tempMsg)->getMauMsg());
                break;
            case MAPI_FACK:
                rtnMsg = new TransactionAckMsg(
                    ((DataTransferMsg*)tempMsg)->getMauMsg());
                break;
            case MAPI_AACK:
            case MAPI_AREQ:
            case MAPI_BACK:
            case MAPI_BREQ:
            case MAPI_CACK:
            case MAPI_CREQ:
            case MAPI_DACK:
            case MAPI_DREQ:
            case MAPI_FBACK:
            case MAPI_FDACK:
            case MAPI_FSACK:
            case MAPI_IACK:
            case MAPI_IDOWN:
            case MAPI_IOACK:
            case MAPI_IOPEN:
            case MAPI_IREM:
            case MAPI_IREQ:
            case MAPI_NREQ:
            case MAPI_OREQ:
            case MAPI_OACK:
            case MAPI_RACK:
            case MAPI_RREQ:
            case MAPI_SACK:
            case MAPI_SESSION:
            case MAPI_SREQ:
            case MAPI_WACK:
            case MAPI_WREQ:
                break;
        }
        tmpMsgLen = rtnMsg->getMsgLength();
        for(int i=0;i<tmpMsgLen;i++){
            target[i] = rtnMsg->getMsg()[i];
        }
        target[tmpMsgLen] = '₩0';
        *msgLen = tmpMsgLen;
        free(rtnMsg);
        rtnMsg = NULL;
    }
    free(tempMsg);
    tempMsg = NULL;
}
```

그림 8. LNA에서의 메시지 언패킹
Fig. 8 Message unpacking in LNA

그림 9는 MAU에서 데이터로부터 MAU와 LNA간의 메시지를 만들어주는 패킹 부분이다. 각각의 함수는 인자로 입력부와 출력부로 나누어져 있다. 입력부는 메시지 타입, 데이터로서 각각의 메시지를 어떠한 메시지로 만들지에 대한 것과, 그 메시지에 담을 데이터로 구성된다. 출력부는 만들어진 메시지의 문자열과 만들어진 메시지의 길이로 구성되거나 추출된 데이터의 문자열과 데이터의 길이로 구성된다.

메시지 타입과 데이터, 데이터의 길이를 입력으로 하

여 함수를 호출하게 되면 패킹되어 MAU-LNA메시지를 반환한다. 그림에서는 데이터 전송 및 함수 로딩 요청 메시지 타입인 MAPI_FREQ와 데이터 전송 및 함수 로딩 요청 메시지의 응답 타입인 MAPI_FACK에 대한 코드를 보여주고 있다.

```
void pack(short msgType, char* source, char* target, int* msgLen){
    MAU_LNA* ml_msg;
    int strLen = 0;
    switch(msgType){
    case MAPI_FREQ:
        strLen = (*msgLen) + 20;
        ml_msg = new TransactionReqMsg();
        ml_msg->setNormalPriority(); // or ml_msg->setPriority(PT_NORMAL);
        ml_msg->setMsgType(MAPI_FREQ);
        ml_msg->setMcpID("aaaa");
        ml_msg->setSessionCode(SESSION_CO);
        ml_msg->setTRID(1);
        ml_msg->setStatusOrTimeout(LM_OK);
        ml_msg->setMsgLength(strLen);
        ((TransactionReqMsg*)ml_msg)->setData(source);
        break;
    case MAPI_FACK:
        strLen = (*msgLen) + 20;
        ml_msg = new TransactionAckMsg();
        ml_msg->setNormalPriority(); // or ml_msg->setPriority(PT_NORMAL);
        ml_msg->setMsgType(MAPI_FACK);
        ml_msg->setMcpID("aaaa");
        ml_msg->setSessionCode(SESSION_CO);
        ml_msg->setTRID(1);
        ml_msg->setStatusOrTimeout(LM_OK);
        ml_msg->setMsgLength(strLen);
        ((TransactionAckMsg*)ml_msg)->setData(source);
        break;
    case MAPI_AACK:
    case MAPI_AREQ:
    case MAPI_BACK:
    case MAPI_BREQ:
    case MAPI_CACK:
    case MAPI_CREQ:
    case MAPI_DACK:
    case MAPI_DREQ:
    case MAPI_FBACK:
    case MAPI_FDACK:
    case MAPI_FSACK:
    case MAPI_IACK:
    case MAPI_IDOWN:
    case MAPI_IOACK:
    case MAPI_IOPEN:
    case MAPI_IREM:
    case MAPI_IREQ:
    case MAPI_NREQ:
    case MAPI_OREQ:
    case MAPI_OACK:
    case MAPI_RACK:
    case MAPI_RREQ:
    case MAPI_SACK:
    case MAPI_SESSION:
    case MAPI_SREQ:
    case MAPI_WACK:
    case MAPI_WREQ:break;
    }
```

그림 9. MAU에서의 메시지 패킹
Fig. 9 Message packing in MAU

그림 10은 MAU와 LNA간의 메시지에서 데이터부분을 추출하는 언패킹 부분이다. MAU-LNA 메시지를 입력으로 하여 함수를 호출하게 되면 언패킹되어 데이터를 반환한다. 그림에서는 데이터 전송 및 함수 로딩 요청 메시지 타입인 MAPI_FREQ와 데이터 전송 및 함수 로딩 요청 메시지의 응답 타입인 MAPI_FACK에 대한 코드를 보여주고 있다.

```
void unpack(short msgType, char* source, char* target, int* msgLen){
    MAU_LNA* inMsg;
    int _msgLen = 0;
    switch(msgType){
    case MAPI_FREQ:
        inMsg = new TransactionReqMsg(source);
        _msgLen = inMsg->getMsgLength()-20;
        for(int i=0; i<_msgLen; i++){
            target[i] = ((TransactionReqMsg*)inMsg)->getData()[i];
        }
        break;
    case MAPI_FACK:
        inMsg = new TransactionAckMsg(source);
        _msgLen = inMsg->getMsgLength()-20;
        for(int i=0; i<_msgLen; i++){
            target[i] = ((TransactionAckMsg*)inMsg)->getData()[i];
        }
        break;
    case LL_ALIVE:
    case LL_DATAC:
    case LL_IFACK:
    case LL_IFREM:
    case LL_IFREQ:
    case LL_NAUACK:
    case LL_NAUREQ:
    case LL_NOMCP:
    case LL_SESSION:
        //LNA_LNA multi-cast messages
    case CC_DEADMAU:
    case CC_DEFMAU:
    case CC_METOO:
    case CC_REQMAU:
    case CC_WATCHDOG:
    case LLBC_AACK:
    case LLBC_SACK:
        break;
    }
    target[_msgLen] = '₩0';

    free(inMsg);
}
```

그림 10. MAU에서의 메시지 언패킹
Fig. 10 Message unpacking in MAU

### 4.2 테스트

구현한 메시지 처리 모듈을 테스트하기 위하여 델(Dell)사의 서버를 사용하였다. 프로세서는 2개의 Quad-Core Intel Xeon Pro E5405 2GHz로 구성되어 있고 메모리는 2GB이다. 운영체제는 Windows Server 2008을 사용하였고 개발 툴은 Visual Studio 2008에서 C++를 사용하였다. 그림 11은 실제로 구성한 시스템의 사진을 보여주고 있다.

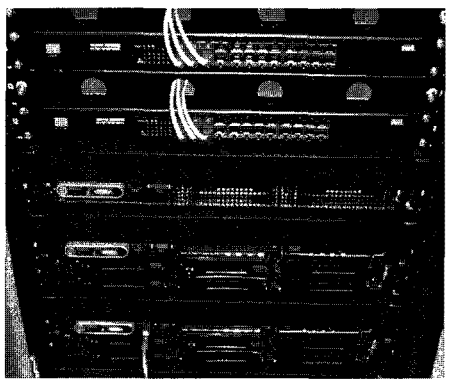

그림 11. 시스템 사진
Fig. 11 System photograph

구현한 메시지 처리 모듈의 기능이 정상적으로 동작하는지 확인하기 위하여 다음과 같은 시나리오로 테스트하였다. 먼저, MAU는 getTagOneValue와 36이라는 데이터를 MAU 메시지로 패킹하여 LNA로 전달한다. LNA는 전달 받은 메시지를 LNA 메시지로 패킹하여 원격 LNA로 전송한다. 원격 LNA는 수신한 메시지를 MAU 메시지로 언패킹하여 자신에 속한 원격 MAU에게 전달한다. 원격 MAU는 전달 받은 메시지를 언패킹하여 데이터를 얻을 수 있게 된다.

위의 테스트에 대한 각 MAU 및 LNA 모듈에서의 출력 결과를 그림 12에 제시하였고, 시나리오 대로 정상적으로 동작함을 보여주고 있다.


그림 12. 테스트 결과
Fig. 12 Test results

## V. 결 론

본 논문에서는 MiTS 시스템에서 구성 요소들 간의 통신에 필요한 메시지 처리 모듈을 설계하고 구현하였다. 또한, 구현한 메시지 처리 모듈이 MiTS 통신에서 패킹과 언패킹의 과정을 통해 메시지가 정상적으로 처리됨을 확인하였다. 따라서, 본 논문에서 설계하고 구현한 메시지 처리 모듈을 이용하면 선박에서의 정보시스템이 MiTS 표준을 기반으로 통신함을 보장할 수 있다

향후 연구로는 표준에 제시되어 있는 다양한 타입의 트랜잭션에 대하여 메시지를 처리할 수 있도록 확장 하는 것이다. 또한, 모든 플랫폼에서 동작이 가능하도록 범용성을 높이는 것이 필요하다.

## 감사의 글

본 연구는 지식경제부 및 정보통신산업진흥원의 IT핵심 기술개발사업의 일환으로 수행하였음. [2008-F-046-01, E-Navigation 대응 IT-선박 융합 핵심기술 개발]

## 참고문헌

[1] IEC 61162-4: Maritime Navigation and Radiocommunication Equipment and Systems - Digital Interfaces - Multiple Talkers and Multiple Listeners - Ship Systems Interconnection, 2001.

[2] 박휴찬, 이장세, 장길웅, 이정우, 정희섭, 박중현, 강순열, "선박에서의 통합 정보처리를 위한 시스템 아키텍쳐", 2009년도 전기학술대회논문집, 한국마린엔지니어링학회, 2009.

[3] 이장세, 박휴찬, 장길웅, 이주형, 장남주, 이주영, 이부형, "선박 내 정보의 통합관리를 위한 정보 아키텍쳐", 2009년도 전기학술대회논문집, 한국마린엔지니어링학회, 2009.

## 저자소개

김태종(Tae-Jong Kim)

2008년 한국해양대학교
IT공학부(공학사)
2010년 한국해양대학교
컴퓨터공학과(공학석사)

※관심분야: 데이터베이스, 해양정보시스템

황훈규(Hun-Kyu Hwang)

2009년 한국해양대학교 IT공학부(공학사)
2009년~현재 한국해양대학교 대학원 컴퓨터공학과 석사과정
※관심분야: 정보보안, 시뮬레이션, 네트워크

서정민(Jung-Min S대)

2009년 한국해양대학교 IT공학부(공학사)
2009년~현재 한국해양대학교 대학원 컴퓨터공학과 석사과정
※관심분야: 데이터베이스, 해양정보시스템

윤진식(Jin-Sik Yoon)

2009년 한국해양대학교 IT공학부(공학사)
2009년~현재 한국해양대학교 대학원 컴퓨터공학과 석사과정
※관심분야: 정보보안, 네트워크, 포렌식

이장세(Jang-Se Lee)

1997년 한국항공대학교 컴퓨터공학과(공학사)
1999년 한국항공대학교 컴퓨터공학과(공학석사)
2003년 한국항공대학교 컴퓨터공학과(공학박사)
2004년~현재 한국해양대학교 조교수
※관심분야: 컴퓨터보안, 지능시스템, 모델링 및 시뮬레이션

장길웅(Kil-Woong Jang)

1997년 경북대학교 컴퓨터공학과(공학사)
1999년 경북대학교 컴퓨터공학과(공학석사)
2002년 경북대학교 컴퓨터공학과(공학박사)
2003년~현재 한국해양대학교 부교수
※관심분야: 네트워크 프로토콜, 유비쿼터스 네트워킹

박휴찬(Hyu-Chan Park)

1985년 서울대학교 전자공학과(공학사)
1987년 한국과학기술원 전기및 전자공학과(공학석사)
1995년 한국과학기술원 전기및전자공학과(공학박사)
1987년~1990 금성반도체
1997년~현재 한국해양대학교 교수
※관심분야: 데이터베이스, 데이터마이닝, 해양정보 시스템