한국마린엔지니어링학회지 제34권 제1호, pp. 147~153, 2010. 1 (ISSN 1226-9549)
Journal of the Korean Society of Marine Engineering

상태 천이로 명세된 MiTS 통신 프로토콜의 객체지향 설계 및 C++ 구현

박휴찬† · 이장세[1] · 장길웅[2]

(원고접수일 : 2009년 12월 4일, 원고수정일 : 2010년 1월 7일, 심사완료일 : 2010년 1월 12일)

An Objected-Oriented Design and C++ Implementation of MiTS Communication Protocol Specified in State Transitions

Hyu-Chan Park† · Jang-Se Lee[1] · Kil-Woong Jang[2]

요 약 : MiTS 통신 프로토콜은 선박에서 발생하는 다양한 정보의 통합 처리 및 교환을 위하여 제안된 표준이다. 일반적인 통신 프로토콜과 유사하게 MiTS 통신 프로토콜도 상태 천이로 명세되어 있다. 이러한 통신 프로토콜의 구현에는 많은 노력과 시간이 필요하므로, 이를 경감할 수 있는 체계적인 설계 및 구현 방법이 요구된다. 이러한 요구에 적합한 방법론으로 디자인 패턴을 들 수 있다. 본 논문에서는 이러한 디자인 패턴을 적용하여 MiTS 통신 프로토콜을 객체지향적으로 설계하고 C++로 구현한 결과에 대하여 기술한다.

주제어 : MiTS, 통신 프로토콜, 상태 천이, 객체지향, C++

Abstract: MiTS Communication Protocol is a standard for the integrated processing and exchange of information on shipboard. It is specified in the form of state transitions as normal communication protocols. The design and implementation of such communication protocol require huge amount effort and time. To alleviate such burden, some systematic methodologies need to be devised. The design pattern may be the most adoptable one. This paper describes an object-oriented design and C++ implementation of MiTS Communication Protocol by adopting such methodology.

Key words: MiTS, Communication protocol, State transition, Object-oriented, C++

1. 서 론

MiTS(Marine Information Technology Standard) 통신 프로토콜은 선박에서 발생하는 다양한 정보를 통합하여 관리하고 상호 교환할 수 있도록 제안된 것이다. 이 프로토콜은 세계적인 표준화 기구인 IEC(International Electrotechnical Commission)에서 IEC61162-4이라는 표준으로 발전시켰다[1]. 많은 경우의 통신 프로토콜과 유사하게 MiTS 통신 프로토콜도 상태 천이(state transitions)로 명세되어 있다. 즉, 이벤트(event)가 발생하면 적절한 액션(action)을 취한 후 다음 상태(state)로 천이(transition)하게 된다.

많은 경우에 이러한 통신 프로토콜은 C언어와 같은 절차적 언어로 개발되어 왔다[2]. 하지만, 통신 프로토콜의 복잡도와 규모가 증대됨에 따라 과도한 노력과 시간을 투입해야하는 문제점이 발생하

† 교신저자(한국해양대학교 IT공학부, E-mail:hcpark@hhu.ac.kr, Tel: 051-410-4573)
1 한국해양대학교 IT공학부
2 한국해양대학교 데이터정보학과

고 있다.

이러한 문제점을 극복하고자 하는 연구로 디자인 패턴(design pattern)[3]과 객체지향적(object oriented) 개발 방법이 제안되어 왔다. 즉, 통신 프로토콜을 구현함에 있어서 사안별로 구현하는 것보다는 디자인 패턴 등 체계적인 구현 방법론에 따라 구현하면 구현 시간을 단축할 수 있을 뿐만 아니라 구현 결과물의 신뢰성과 효율성을 달성할 수 있다. 또한, 디자인 패턴은 설계의 유연성, 모듈화, 재사용성, 이해도를 증대시킬 수 있다[3].

본 논문에서는, 위와 같은 디자인 패턴을 적용하여 상태 천이로 명세된 MiTS 통신 프로토콜을 설계하고 구현한 결과를 제시한다. 우선, 설계에서는 프로토콜의 각 상태를 개별 클래스로 정의하고, 상태 천이는 클래스 오브젝트의 변경으로 처리한다. 각 상태에서의 이벤트와 액션은 클래스의 멤버 함수로 정의한다. 이러한 객체지향적 설계의 구현을 위하여 객체지향 언어인 C++를 사용한다. 최종적으로, C++로 구현된 MiTS 프로토콜의 검증 결과를 제시한다.

2. 관련 연구

복잡한 시스템을 개발함에 있어서 패턴의 중요성은 많은 분야에서 인식되어 왔고, 다양한 형태의 디자인 패턴이 제시되어 왔다. 특히, 상태 천이로 명세된 시스템의 설계 및 구현에 적용할 수 있는 디자인 패턴이 제시되어 있다[3].

통신 프로토콜도 상태 천이로 명세할 수 있으므로 이러한 디자인 패턴을 적용할 수 있다. 상태 천이로 명세된 통신 프로토콜의 객체 지향적 설계 및 구현을 위한 디자인 패턴은 Figure 1과 같다[3,4].

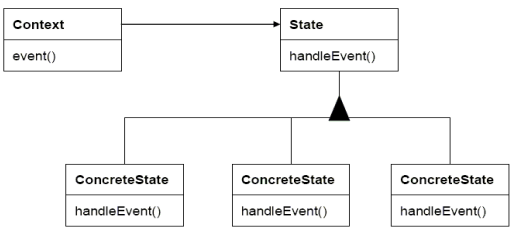

Figure 1: Design pattern for state transitions

State 클래스는 프로토콜의 개별 상태를 표현하는 모든 concreteState 클래스들에 공통적인 인터페이스를 정의한다. 각 concreteState 클래스는 State 클래스의 서브 클래스로서 프로토콜의 특정 상태에서의 이벤트와 액션을 구현한다. 프로토콜에서 정의된 상태의 개수와 동일한 개수의 concreteState 클래스가 정의된다. Context 클래스는 프로토콜의 현재 상태를 나타내기 위하여 concreteState 클래스를 관리한다. 이를 통하여 현재 상태에서의 모든 이벤트를 concreteState 클래스로 전달하고, 전달 받은 concreteState 클래스는 적절한 처리를 하게 된다. Context 클래스는 프로토콜의 상태가 바뀔 때마다 concreteState 클래스를 변경한다[3].

이러한 디자인 패턴은 다음과 같은 장점을 가진다. 첫째, 프로토콜의 동작을 각 상태별로 분할 할 수 있으며, 특정 상태에서의 동작을 지역화할 수 있다. 즉, 특정 상태에서의 모든 동작은 하나의 concreteState 클래스 내에 둘 수 있다. 또한, 특정 상태와 관련된 모든 프로그램 코드를 대응되는 concreteState 클래스 내에 둘 수 있다. 따라서 프로토콜에 새로운 상태가 추가되면 새로운 concreteState 클래스만 추가하는 것으로 쉽게 대응할 수 있다. 둘째, 상태 천이가 명시적이다. 즉, 상태에 대한 정보를 단순한 값으로 관리하는 것보다, 각각의 상태에 대하여 서로 다른 concreteState 클래스를 둠으로써 상태 천이가 보다 명시적으로 구현될 수 있다[3].

3. MiTS 통신 프로토콜의 설계 및 구현

3.1 MiTS 통신 프로토콜

MiTS 통신 프로토콜은 몇 개의 상태 천이도(state transition diagram)로 명세되어 있는데, 그 중에서 MAU(MiTS Application Unit) 상태 천이도가 Figure 2에 제시되어 있다[1].

MAU 상태 천이도는 다섯 개의 상태(MAU_DEFINED, MAU_OPENING, MAU_OPEN, MAU_CLOSED, MAU_ERROR)로 구성되어 있고, 각 상태에는 이벤트와 그에 대응되는 액션이 정의되어 있다.

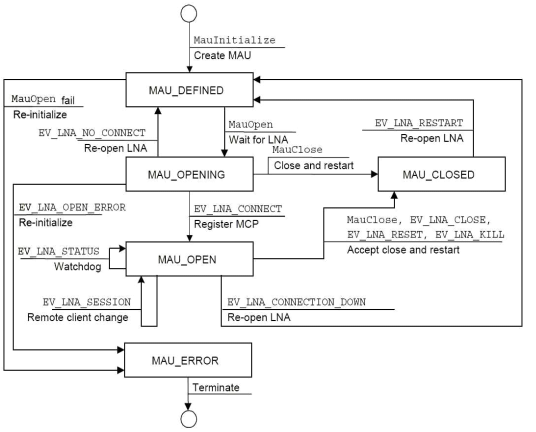

Figure 2: A state transition diagram

3.2 MiTS 통신 프로토콜의 설계

디자인 패턴을 따르는 MiTS 통신 프로토콜의 객체지향적 설계는 Figure 3과 같다. 그림에서 알 수 있듯이, 핵심적인 클래스 구성은 하나의 MauContext 클래스, 하나의 MauState 클래스, 그리고 프로토콜의 다섯 가지 상태 각각에 대응되는 다섯 개의 상태 클래스로 구성되어 있다. 이러한 상태 클래스는 MauState 클래스의 서브 클래스로서 MauDefinedMauState, MauOpening MauState, MauOpenMauState, MauClose MauState, MauErrorMauState 클래스로 구성되어 있다.

MauContext 클래스는 모든 이벤트 인터페이스와 액션 함수를 정의하고 있다. 각 상태 클래스 내에 액션을 정의할 수도 있지만 여러 상태 클래스에 동일한 액션이 존재할 수 있으므로 MauContext 클래스에 정의하여 공동으로 호출하여 사용할 수 있도록 하였다. 또한, 프로토콜의 현재 상태를 관리하기 위하여 상태 클래스에 대한 포인터와 프로토콜의 다음 상태로 천이하기 위한 상태 천이 함수를 가지고 있다.

MauState 클래스는 각 상태 클래스 내에 정의되지 않은 이벤트 핸들러(event handler)가 호출되었을 때 그 상태 클래스를 대신하여 적절한 행위를 수행한다. 즉, 모든 상태 클래스에 공통으로 적용할 수 있는 디폴트 이벤트 핸들러를 정의한다.

각각의 상태 클래스는 자신이 처리해야 할 이벤트에 대한 핸들러를 정의한다. 이벤트 핸들러는 자신에게 부과된 액션을 수행하기 위하여 MauContext에 정의되어 있는 액션 함수를 호출한다. 또한, 다음 상태로 천이하기 위하여 MauContext에 정의되어 있는 상태 천이 함수를 호출한다.

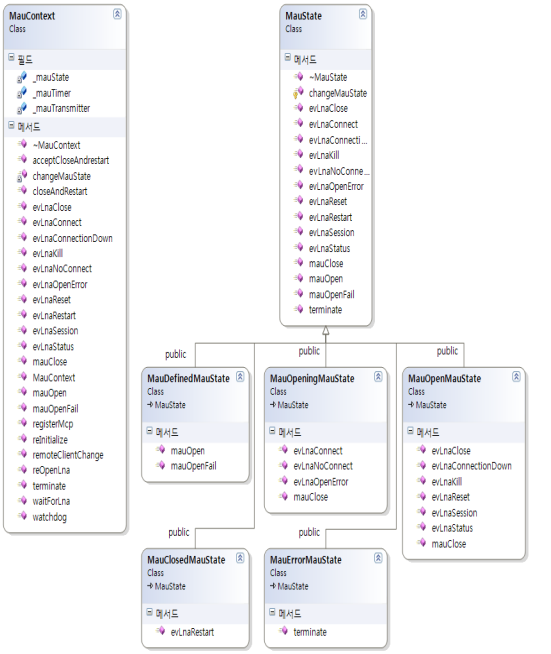

Figure 3: Designed class diagram

위의 클래스들은 다음과 같은 상호동작을 하게 된다. 먼저, MiTS 통신 프로토콜의 클라이언트가 발생시킨 이벤트를 MauContext가 받게 된다. MauContext는 프로토콜의 현재 상태에 대응하는 상태 클래스에게 이 이벤트의 처리를 위임하게 된다. 위임 받은 상태 클래스는 자신의 이벤트 핸들러를 통하여 적절한 처리를 완료하게 된다. 이후, MauContext는 다음 상태로 천이하게 된다. 이상의 과정을 반복적으로 수행하게 된다.

3.3 MiTS 통신 프로토콜의 구현

MiTS 프로토콜의 객체 지향적 설계를 C++로 구현하였다. 각 클래스의 실제 정의 및 구현에 대하여 설명한다.

3.3.1 MauContext 클래스의 정의 및 구현

프로토콜의 모든 이벤트(예, mauOpen)와 액션(예, waitForLna)에 대한 함수를 정의하고, 다음 상태로 천이하기 위한 함수(changeMauState)도 정의한다. 그리고 현재 상태를 관리하기 위하여 상태 클래스에 대한 포인터(_mauState)를 가지고 있다.

실행 예를 들면, MauContext의 mauOpen() 이벤트가 호출되면 _mauState 변수가 포인팅하고 있는 MauDefinedMauState 상태 클래스의 이벤트 핸들러인 mauOpen()을 호출한다. 이후, 호출된 MauDefinedMauState로부터 다음 상태에 대응하는 상태 클래스를 넘겨받아 _mauState를 변경한다.

```
class MauContext
{
public:
        MauContext();

          //--- Events -----
          // for MauDefined state
        void mauOpen();
        void mauOpenFail();

          // for MauOpening state
        void mauClose();
        void evLnaConnect();
        void evLnaOpenError();
        void evLnaNoConnect();

          // for MauOpen state
        void evLnaClose();
        void evLnaReset();
        void evLnaKill();
        void evLnaConnectionDown();
        void evLnaSession();
        void evLnaStatus();

          // for MauCLosed state
        void evLnaRestart();

          // for MauError state
        void terminate();

          //--- Actions -----
          // for MauDefined state
        void waitForLna()
        void reInitialize()

          // for MauOpening state
        void closeAndRestart()
        void registerMcp()
        void reOpenLna()

          // for MauOpen state;
        void acceptCloseAndrestart()

          //void reOpenLna()
        void remoteClientChange()
        void watchdog()

          // for MauClosed state

          // for MauError state

private:
        void changeMauState(MauState*);
        MauState* _mauState;
}


//--------------------
MauContext* mauInitialize()
{
        return new MauContext;

}
```

```
MauContext::MauContext()
{
        _mauState = GetInstance::mauDefinedMauState();
}

void MauContext::mauOpen()
{
        _mauState->mauOpen(this);
}

void MauContext::mauOpenFail()
{
        _mauState->mauOpenFail(this);
}

    /*
     * similar for other member functions
     */

void MauContext::terminate()
{
        _mauState->terminate(this);
}

void MauContext::changeMauState(MauState* mauState)
{
        _mauState = mauState;
}
```

3.3.2 MauState 클래스의 정의 및 구현

프로토콜의 모든 이벤트에 대하여 디폴트 핸들러를 정의한다. 만일, MauContext가 상태 클래스에 없는 이벤트 핸들러를 호출하게 되면 MauState의 디폴트 핸들러가 자동으로 실행된다. 실행 결과로 "유효하지 않는 이벤트가 수신되었다"는 로그 메시지를 출력한다. 또한, 상태 클래스들이 공동으로 사용할 수 있는 상태 천이 함수(changeMauState)도 정의하고 있다.

```
class MauState
{
public:
          // for MauDefined state
        virtual void mauOpen(MauContext*);
        virtual void mauOpenFail(MauContext*);

          // for MauOpening state
        virtual void mauClose(MauContext*);
        virtual void evLnaConnect(MauContext*);
        virtual void evLnaOpenError(MauContext*);
        virtual void evLnaNoConnect(MauContext*);

          // for MauOpen state
        virtual void evLnaClose(MauContext*);
        virtual void evLnaReset(MauContext*);
        virtual void evLnaKill(MauContext*);
        virtual void evLnaConnectionDown(MauContext*);
        virtual void evLnaSession(MauContext*);
        virtual void evLnaStatus(MauContext*);

          // for MauCLosed state
        virtual void evLnaRestart(MauContext*);

          // for MauError state
        virtual void terminate(MauContext*);

protected:
        void changeMauState(MauContext*, MauState*);
};


//-------------------------------------
void MauState::mauOpen(MauContext* mauContext)
{
        Logger::log("Invalid MAU Event 'mauOpen' Received");
}

void MauState::mauOpenFail(MauContext* mauContext)
```

```
{
        Logger::log("Invalid MAU Event 'mauOpenFail'
         Received");
}

    /*
     * similar for other member functions
     */

void MauState::terminate(MauContext* mauContext)
{
        Logger::log("Invalid MAU Event 'terminate' Received");
}

void MauState::changeMauState(MauContext* mauContext,
         MauState* mauState)
{
        mauContext->changeMauState(mauState);
}
```

3.3.3 상태 클래스들의 정의 및 구현

상태 클래스는 MauState 클래스로부터 상속받아 정의한다. 프로토콜의 다섯 가지 상태 각각에 대하여 대응되는 상태 클래스(MauDefined MauState, MauOpeningMauState, MauOpen MauState, MauClosedMauState, MauError MauState)를 정의하였다.

각 상태 클래스(예, MauDefinedMauState)는 자신에게 유효한 이벤트(예, mauOpen)에 대하여 핸들러를 가지고 있다. 각 핸들러는 MauContext에 정의되어 있는 액션(예, waitForLna)을 호출하여 적절한 행위를 완료한다. 이후, 다음 상태로 천이하기 위하여 MauState 클래스에서 상속받은 change MauState()를 호출한다.

```
class MauDefinedMauState : public MauState
{
public:
        void mauOpen(MauContext*);
        void mauOpenFail(MauContext*);
};

class MauOpeningMauState : public MauState
{
public:
        void mauClose(MauContext*);
        void evLnaConnect(MauContext*);
        void evLnaOpenError(MauContext*);
        void evLnaNoConnect(MauContext*);
};

class MauOpenMauState : public MauState
{
public:
        void mauClose(MauContext*);
        void evLnaClose(MauContext*);
        void evLnaReset(MauContext*);
        void evLnaKill(MauContext*);
        void evLnaConnectionDown(MauContext*);
        void evLnaSession(MauContext*);
        void evLnaStatus(MauContext*);
};

class MauClosedMauState : public MauState
{
public:
        void evLnaRestart(MauContext*);
};

class MauErrorMauState : public MauState
{
public:
        void terminate(MauContext*);
};


//-----------------------------------------
void MauDefinedMauState::mauOpen(MauContext*
         mauContext)
{
        mauContext->waitForLna();
        changeMauState(mauContext,
                        GetInstance::mauOpeningMauState());
}

void MauDefinedMauState::mauOpenFail(MauContext*
         mauContext)
{
        mauContext->reInitialize();
        changeMauState(mauContext,
                        GetInstance::mauOpeningMauState());
}

    /*
     * similar for other classes and member functions
     */

void MauErrorMauState::terminate(MauContext*
         mauContext)
{
        ;
}
```

3.3.4 기타 정의 및 구현

프로토콜에서 공통적으로 사용하는 함수들을 네임스페이스로 정의한다. 로그 메시지 출력을 위한 Logger를 정의하였다. 또한, 상태 클래스의 오브젝트를 생성하기 위한 GetInstance도 정의하였다. 각 상태 클래스에 대하여 하나의 오브젝트만 생성하여 재사용할 수 있도록 싱글톤(singleton) 기법[3]을 적용하였다.

```
namespace Logger
{
        void log(const char*);
};

namespace GetInstance
{
        MauState* mauDefinedMauState(void);
        MauState* mauOpeningMauState(void);
        MauState* mauOpenMauState(void);
        MauState* mauClosedMauState(void);
        MauState* mauErrorMauState(void);
};


//-----------------------
void Logger::log(const char* s)
{
        cout << s << endl;
}

MauState* GetInstance::mauDefinedMauState()
{
        if (_mauDefinedMauState == NULL)
                _mauDefinedMauState = new MauDefinedMauState;
        return _mauDefinedMauState;
}

       /*
       * similar for other classes
       */

MauState* GetInstance::mauErrorMauState()
{
        if (_mauErrorMauState == NULL)
                _mauErrorMauState = new mauErrorMauState
        return _mauErrorMauState;
}
```

4. MiTS 프로토콜의 검증

구현된 MiTS 프로토콜을 검증하기 위하여 가상의 상태 천이 시나리오를 Figure 2를 기준으로 정의하였다. 먼저, MauInitialize 이벤트를 호출하여 프로토콜의 상태를 MAU_DEFINED로 초기화한다. 이후, MauOpen 이벤트로 MAU_OPENING 상태로 천이하고, EV_LNA_CONNECT 이벤트로 MAU_OPEN 상태로 천이하고, EV_LNA_SESSION 이벤트로 다시 MAU_OPEN 상태로 천이한다. 이 상태에서 EV_LNA_RESTART 이벤트를 가하면 유효한 이벤트가 아니므로 오류 메시지가 출력되어야 한다.

위와 같은 상태 천이 시나리오를 따르는 테스트 프로그램은 다음과 같다.

```
void main()
{
        MauContext* myMau;

        cout << "(test) mauInitialize...\n";
        myMau = mauInitialize();

        cout << "(test) myMau->mauOpen...\n";
        myMau->mauOpen();

        cout << "(test) myMau->evLnaConnect...\n";
        myMau->evLnaConnect();

        cout << "(test) myMau->evLnaSession...\n";
        myMau->evLnaSession();

        cout << "(test) myMau->evLnaRestart(invalid event)...\n";
        myMau->evLnaRestart();
}
```

```
(test) mauInitialize...
[MAU_TRACE] mauInitialize
[MAU_TRACE] MauContext::MauContext
[MAU_TRACE] GetInstance::mauDefinedMauState

(test) myMau->mauOpen...
[MAU_TRACE] MauContext::mauOpen
[MAU_TRACE] MauDefinedMauState::mauOpen
[MAU_TRACE] MauContext::waitForLna
[MAU_TRACE] GetInstance::mauOpeningMauState
[MAU_TRACE] MauState::changeMauState
[MAU_TRACE] MauContext::changeMauState

(test) myMau->evLnaConnect...
[MAU_TRACE] MauContext::evLnaConnect
[MAU_TRACE] MauOpeningMauState::evLnaConnect
[MAU_TRACE] MauContext::registerMcp
[MAU_TRACE] GetInstance::mauOpenMauState
[MAU_TRACE] ...new MauOpenMauState creating
[MAU_TRACE] MauState::changeMauState
[MAU_TRACE] MauContext::changeMauState

(test) myMau->evLnaSession...
[MAU_TRACE] MauContext::evLnaSession
[MAU_TRACE] MauOpenMauState::evLnaSession
[MAU_TRACE] MauContext::remoteClientChange
[MAU_TRACE] GetInstance::mauOpenMauState
[MAU_TRACE] MauState::changeMauState
[MAU_TRACE] MauContext::changeMauState

(test) myMau->evLnaRestart(invalid event)...
[MAU_TRACE] MauContext::evLnaRestart
[MAU_TRACE] MauState::evLnaRestart
Invalid MAU Event 'evLnaRestart' Received
```

Figure 4: Protocol output

테스트 프로그램을 구현된 MiTS 프로토콜에 가했을 때, 그 출력이 Figure 4에 제시되어 있다. 그림에서 알 수 있듯이, 주어진 시나리오 대로 MiTS 프로토콜에서 상태 천이가 일어나고 있음을 확인할 수 있고, 또한 잘못된 이벤트에 대하여 적절한 오류 메시지가 출력됨을 확인할 수 있다.

5. 결 론

본 논문에서는 상태 천이로 명세된 통신 프로토콜의 설계 및 구현 방법에 대하여 고찰하였고, 이를 MiTS 통신 프로트콜의 설계와 구현에 적용하였다. 이를 통하여 객체지향적 설계와 C++ 구현의 효용성과 신뢰성을 검증할 수 있었다.

본 논문에서는 MiTS 통신 프로토콜에 대하여 적용하였지만 다른 통신 프로토콜의 설계 및 구현에도 유사하게 적용할 수 있을 것이다. 또한, C++ 구현을 제시하였지만 다른 객체지향언어를 사용한 구현에도 동일하게 적용할 수 있을 것이다.

후 기

본 연구는 지식경제부 및 정보통신산업진흥원의 IT핵심기술개발사업의 일환으로 수행하였음. [2008-F-046-01, E-Navigation 대응 IT-선박융합 핵심기술 개발]

참고문헌

[1] IEC61162-4: Maritime Navigation and Radiocommunication Equipment and Systems - Digital Interfaces - Multiple Talkers and Multiple Listeners - Ship Systems Interconnection, 2001.

[2] J. Darroch, "Implementing protocol state machines," Proceedings of Embedded System Conference, 2003.

[3] H. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Objected-Oriented Software, Addison-Wesley, 1994.

[4] H. A. Sugar, "Efficient Coding

Communication Protocols in C++," 2006

저 자 소 개

박휴찬(朴烋讚)

1985년 서울대학교 전자공학과(공학사), 1987년 한국과학기술원 전기및전자공학과(공학석사), 1995년 한국과학기술원 전기및전자공학과(공학박사), 1987년 - 1990년 금성반도체, 1997년 - 현재 한국해양대학교 IT공학부(교수). 관심분야: 데이터베이스, 데이터마이닝, 해양정보시스템

이장세(李章世)

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

장길웅(張吉雄)

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