결함허용 중개자 스터브 방식에서 실시간객체를 감시하는 구조

(An Architecture to Monitor Real-Time Objects in FTB Stub Approach)

임 형 택 † 양 승 민 ††

(Hyung-Taek Lim)(Seung-Min Yang)

요 약 RMO(Region Monitor Object)는 결함전파나 객체군에 주어진 요구사항의 위반에 의해 발생하는 오류를 처리하는 실시간객체로써 여러 실시간객체의 상태를 감시 및 분석하여 오류를 감지하고, 증상을 진단한 후 알맞은 복구 및 재구성을 실행한다. 이를 위하여 RMO는 응용 실시간객체를 감시할 수 있는 권한을 갖는다. RMO의 권한을 지원해주는 구조는 결함허용 중개자를 이용한다. 결함허용 중개자(FTB 또는 Fault Tolerance Broker)는 RMO가 응용 실시간객체를 감시할 때에 응용의 설계와 응용의 위치에 투명하게 수행될 수 있게 중개자 역할을 한다. 제안하는 감시 구조에는 결함허용 중개자가 응용 실시간객체마다 스터브로 붙는 스터브 방식과 각 노드의 커널에 모듈로 존재하는 커널 모듈 방식이 있다.

본 논문은 스터브 방식에서 RMO가 응용 실시간객체를 감시하는 구조를 제시하고 구현한다. 결함허용 중개자 스터브는 응용 실시간객체와 같은 주소 공간에 존재하면서 응용 실시간객체에서 발생하는 메시지를 가로채고 소속자료에 접근한다. RMO는 결함허용 중개자 스터브가 제공하는 인터페이스를 통해서 응용 실시간객체에 대한 감시 정보를 얻는다. 제안한 감시 구조는 실시간객체 모델인 dRTO(dependable RTO) 모델에 기반하여 설계하였고 실시간 커널인 dKernel 상에서 구현 및 실험하였으나 다른 모델이나 커널에도 적용될 수 있다

Abstract RMO(Region Monitor Object) is an RTO(Real-Time Object) that handles errors caused by fault propagation and/or violation of requirement imposed on a set of RTOs. It detects error by monitoring and analyzing status of RTOs, diagnoses the symptoms, and then performs appropriate recovery and/or reconfiguration. To perform these functions, RMO has the authority to monitor application RTOs. The architecture that support RMO's authority uses FTB(Fault Tolerance Broker). FTB plays the role of the broker for RMO to monitor application RTOs in application transparent and location transparent way. There are two possible architectures. In stub approach, FTB is attached to each application RTO as a stub. In kernel module approach, it resides in kernel of each node as a module.

This paper proposes and implements an architecture for RMO to monitor application RTOs in stub approach. FTB stub that resides in the same address space as application RTO's, intercepts messages occurred in the application RTO and accesses member data in the RTO. RMO acquires monitoring information of application RTO via interfaces provided by FTB. The monitoring architecture is designed based on an RTO model, dRTO (dependable RTO) model, and is implemented and tested on a real-time kernel, dKernel. However, proposed architecture can be applied to other models and kernels.

· 본 연구는 한국과학재단 특정기초 연구과제 지원(과제번호: 1999-1-303-006-3)으로 수행되었습니다.

† 비 회 원 : 숭실대학교 컴퓨터학과
htlim@realtime.soongsil.ac.kr
†† 종신회원 : 숭실대학교 컴퓨터학과 교수
yang@computing.soongsil.ac.kr

논문접수 : 1999년 12월 28일
심사완료 : 2000년 11월 6일

1. 서 론

모델은 복잡한 시스템을 구축할 때 불필요하게 자세한 부분을 숨기고 중요한 특징만을 나타냄으로써 시스템 개발의 매우 이른 단계에서부터 결점을 찾아낼 수 있고 시스템을 효율적이고 체계적으로 개발할 수 있게 하는 청

사진의 역할을 한다. 실세계를 객체라는 단위로 모델링하는 객체 모델[1]은 가장 널리 사용되고 있는 대표적인 모델이다. 실시간객체(RTO 또는 Real-Time Object) 모델은 객체 모델을 실시간 시스템의 모델링에 적합하도록 확장한 것으로써 대표적인 예로는 TMO(Time-triggered Message-triggered Object) 모델[2]과 dRTO(dependable Real-Time Object) 모델[3]이 있다. 유연하고(flexible) 재사용 가능한 결함허용 시스템 구축을 위하여 메타객체(metaobject)를 이용한 연구[4]도 있다.

실시간 시스템을 위한 모델이나 결함허용 시스템을 위한 모델은 각각 활발히 연구가 되고 있고 이미 제시된 모델도 있으나 실시간 개념과 결함허용 개념을 한 모델에 통합적으로 지원하여 결함허용 실시간 시스템 구축을 돕기 위한 연구는 부족하였다.

본 논문의 저자는 결함허용 실시간 시스템 구축을 위한 모델을 제안하였다[5][6]. 제안한 모델은 실시간객체(Real-Time Object 또는 RTO) 모델에 기반하며 Robust RTO(Robust Real-Time Object)와 RMO(Region Monitor Object)로 구성된다. RobustRTO는 자신 안에서 발생하는 결함을 감지하고 처리하는 실시간객체이다. 복구블럭(Recovery Block)[7], N 버전 프로그래밍(N Version Programming)[8], TMR(Triple Modular Redundancy)[9], standby spare[9], 분산복구블럭[10], active/passive replication 기법[11][12][13]과 같이 단위 객체의 결함을 허용하기 위하여 여분의 객체를 이용하는 여분기반(redundancy based) 결함허용 기법들은 RobustRTO로 모델링된다. RMO는 여러 실시간객체들의 상태를 감시 및 분석함으로써 시스템의 비정상적인 행위를 감지한 후, 그 증상을 진단하여 결함을 찾아 알맞은 복구 및 재배치 방법을 결정하고 수행하는 실시간객체이다. [5]와 [6]에서 RobustRTO와 RMO의 개념과 이들을 모델링하는 방법을 제시하였다.

내장 실시간 시스템은 여러 종류의 하드웨어 및 소프트웨어 객체들로 구성되므로 운행 시 다양한 오류가 발생할 수 있고 그 원인도 다양하다. 하드웨어의 결함은 소프트웨어의 오류를 발생시키고 궁극적으로 시스템이 제 기능을 수행하지 못하게 한다. 시스템 운용시 한 객체의 결함이 다른 객체에 전파되어 오류를 발생시킬 수 있으므로 오류가 발생한 부분을 복구한다고 결함이 처리되는 것은 아니다. 또한, 단위 객체의 요구사항이 모두 만족되더라도 객체군에 대한 요구사항은 만족되지 않을 수 있다. 이와 같은 오류는 RMO가 처리한다.

RMO는 실시간객체에 대한 감시, 감지한 오류에 대한 진단 및 분석, 복구 및 재배치, 비상사태 처리, 기록의 다섯 가지 기능을 갖는다. 그리고, 다섯 가지 기능을 수행할 수 있도록 RMO는 응용 실시간객체들의 상태와 이들이 주고 받는 메시지를 감시하고 이들을 제어할 수 있는 권한을 갖는다.

RobustRTO와 캡슐화된 복제객체들, 그리고 RMO가 소프트웨어로 구현되는 경우 RMO의 권한을 지원해주는 감시 및 제어 구조가 필요하다. RMO가 응용 실시간객체를 감시 및 제어할 수 있도록 지원하는 구조는 실시간 커널과 결함허용 중개자(Fault Tolerance Broker)로 구성된다. 결함허용 중개자는 RMO와 응용 실시간객체(RobustRTO 포함)의 사이에서 RMO의 권한이 응용의 설계와 응용의 위치에 투명하게 수행될 수 있도록 중개자 역할을 한다. 즉, RMO 때문에 응용의 설계가 변경되지 않게 하며 RMO의 권한이 응용 실시간객체의 위치와 무관하게 수행될 수 있게 한다. 결함허용 중개자는 RMO가 응용 실시간객체를 감시하거나 제어하는데 필요한 정보와 메쏘드를 제공한다. RMO는 RobustRTO도 응용 실시간객체와 동일하게 감시하고 제어한다. 감시 및 제어 구조는 결함허용 중개자의 위치에 따라서 스터브 방식과 커널 모듈 방식이 있다. 스터브 방식에서는 응용 실시간객체마다 결함허용 중개자가 스터브로 붙고 커널 모듈 방식에서는 각 노드 당 하나의 결함허용 중개자가 모듈 형태로 커널에 존재한다.

본 논문은 스터브 방식에서 RMO가 실시간객체를 감시하는 구조를 제시하고, 이 구조 상에서 메시지와 실시간객체의 상태인 실시간자료를 감시하는 방법을 제안하고 구현한다. 본 논문의 감시 구조는 실시간객체 모델인 dRTO 모델[4]에 기반하여 설계하였고 dRTO 모델을 위한 실시간 커널인 dKernel 상에서 구현 및 실험을 하였다. 그러나, 감시 구조는 다른 모델이나 커널에도 적용될 수 있다.

2 장에서 dRTO 모델과 dKernel, 그리고 개발환경을 소개한다. 3 장에서 RobustRTO와 RMO를 설명한다. 4 장에서는 스터브 방식과 커널 모듈 방식을 소개하고 두 방식의 장단점을 분석한 후 스터브 방식을 위한 개발환경과 개발과정을 설명한다. 스터브 방식에서 실시간객체들이 주고받는 실시간메시지를 감시하는 구조를 5 장에서, 그리고 실시간객체의 실시간자료를 감시하는 구조를 6 장에서 설명한다. 7 장에서 구현한 예를 보이고 8 장에서 본 논문을 마친다.

2. 실시간 시스템을 위한 모델과 커널

본 장에서는 저자의 연구실에서 개발한 dRTO 모델과 dKernel의 주요 특징을 소개하고 dRTO 모델과 dKernel

상에서 응용을 개발하는 환경과 과정을 기술한다.

2.1 dRTO 모델

dRTO 모델은 신뢰도 높은 내장 실시간 시스템의 효율적인 구축을 위하여 객체지향과 실시간성, 그리고 신뢰성의 세 가지 기본 개념을 단일 모델에 통합한 것이다. dRTO 모델에서는 내장 실시간 시스템의 모든 하드웨어 및 소프트웨어가 실시간객체로 추상화된다. 그림 1처럼 하나의 실시간객체는 실시간자료와 실시간메쏘드로 구성된다.

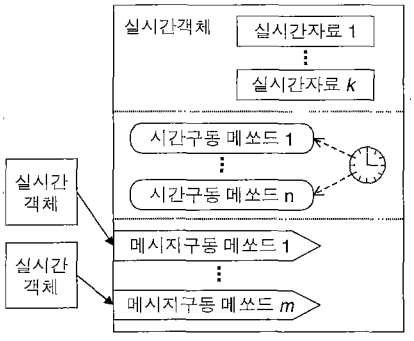


그림 1 dRTO 모델의 실시간객체

실시간메쏘드는 명세된 시간에 따라 구동되는 시간구동 메쏘드(TM 또는 Time-triggered Method)와 메시지에 의하여 구동되는 메시지구동 메쏘드(MM 또는 Message-triggered Method)로 구분된다. 실시간객체들 간의 통신은 메시지구동 메쏘드의 호출을 통해서 이루어진다. 실시간메쏘드는 긴급(immediate)과 경성(hard), 그리고 연성(soft)의 세 스케줄링 클래스로 구분되며 각 스케줄링 클래스의 실시간메쏘드는 중요도(criticalness)를 갖는다. 중요도는 오버로드 상황에서 어떤 실시간메쏘드를 포기하는 것이 피해를 최소화할 수 있는지를 판단하기 위하여 사용된다. 시간구동 메쏘드는 경성이나 연성 스케줄링 클래스에 속하며 설계시 정해진 실행주기, 최악수행시간(WCET 또는 Worst Case Execution Time), 마감시간, 중요도에 따라 수행된다. 메시지구동 메쏘드는 긴급, 경성, 연성 중 하나가 되며 긴급은 시스템에서 가장 우선순위가 높다. 메시지구동 메쏘드는 설계시 최악수행시간만 갖고 있고 스케줄링 클래스와 마감시간, 그리고 중요도는 호출자에 의해 결정된다.

메시지구동 메쏘드를 호출하는 방식에는 동기 호출(blocking call)과 비동기 호출(nonblocking call)의 두 가지가 있다. 동기 호출에서 피호출자의 스케줄링 클래스와 마감시간, 그리고 중요도는 호출자의 것을 물려받는다. 비동기 호출에서는 호출자가 피호출자의 스케줄링 클래스와 마감시간, 그리고 중요도를 지정한다. 즉, 경성인 호출자가 피호출자를 연성 스케줄링 클래스로 호출할 수 있다.

실시간객체들 간에 실시간 정보를 가지고 전달되는 메시지를 실시간메시지라 한다. 실시간메시지는 메시지구동 메쏘드를 호출하였을 때 발생하며 메시지구동 메쏘드의 호출방식과 전달인자, 스케줄링 클래스, 마감시간, 중요도 등을 갖는다. 실시간쓰레드는 실시간메쏘드가 실체화된 실행단위이다.

2.2 dKernel

dKernel은 dRTO 모델의 실행엔진으로써 그림 2와 같이 여러 관리자(또는 모듈)로 구성된다. 실시간객체 관리자는 실행 중인 실시간객체에 대한 정보를 관리하고, 실시간 스케줄러는 dRTO 모델의 실행단위인 실시간쓰레드를 스케줄링한다. 현재 dKernel에는 LLF(Least Laxity First) 스케줄러가 구현되어 있으나 다른 실시간 스케줄러로 재구성하기 용이하게 구현되었다. IOC(Inter-Object Communication) 관리자는 dRTO 모델의 실시간객체들이 통신할 수 있도록 동기 호출과 비동기 호출 기능을 제공하고 네트워크 관리자는 원격 노드 간의 통신을 지원한다. dKernel은 VxWorks처럼 응용과 커널의 주소공간이 구분되어 있지 않고 모두 한 주소공간에서 실행되며 가상 메모리는 지원하지 않는다.

그림 3처럼 dKernel은 dkSim과 dkI386의 두 버전이 존재한다. dkSim은 리눅스 상에서 dKernel을 시뮬레이

그림 2 dKernel의 구조

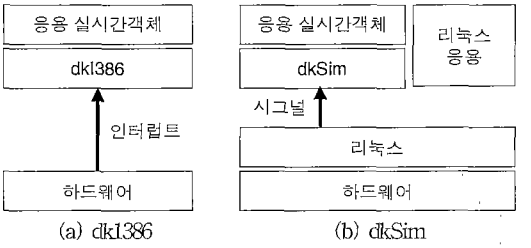


그림 3 dKernel의 두 버전: dkI386과 dkSim

션해주는 커널로써 리눅스의 사용자 프로세스로 동작한다. dkI386은 인텔 80386 이상의 프로세서에서 직접 동작하는 커널이다. dkSim과 dkI386은 HAL 부분을 제외하고 핵심적인 커널의 기능은 동일하다. 즉, dkI386은 하드웨어 인터럽트를 이용하여 동작하고 dkSim은 리눅스의 시그널을 이용하여 동작한다.

2.3 dRTO 모델과 dKernel의 개발환경

dSource는 dRTO 모델에 기반한 프로그래밍을 지원하는 스크립트이다(그림 4).

```
[rto|rmo|robustrto] rto_name {
ac:
  other_rto_name.mm_name;
  . . .
ods:
  . . .
tm:
  (hard|soft) tm_name {
    aac {
      from start_time to end_time
      every period by deadline
    }
    code wcet wcet stack stack_size crit criticality {
      ... user routine in C syntax ...
    }
  }
  . . .
mm:
  mm_name {
    code(argument_list)
      wcet wcet stack stack_size crit criticality {
      ... user routine in C syntax ...
    }
  }
  . . .
};
```

그림 4 dSource

dSource에서 실시간객체에 대한 명세는 **rto, rmo, robustrto** 중 하나의 키워드와 그 이름으로 시작한다. **rto, rmo, robustrto**는 각각 응용 실시간객체, RMO, 그리고 RobustRTO를 나타낸다. 실시간객체는 **ac**(access capability), **ods**(object data store), **tm, mm**의 네 개 구역으로 나뉜다. **ac** 구역에는 이 실시간객체가 참조하는 다른 실시간객체의 메시지구동 메쏘드를 명세한다. **ods** 구역에는 실시간객체의 실시간자료를 C 문법에 따라 선언한다. **tm** 구역에는 시간구동 메쏘드를 명세한다. 시간구동 메쏘드의 스케줄링 클래스(경성 또는 연성)와 이름을 명세하고 **aac**(automatic activation condition) 부분과 **code** 부분을 명세한다. **aac** 부분에는 시간구동 메쏘드가 자동으로 활성화되기 위한 시간으로써 동작기간(**from** *start_time* **to** *end_time*), 주기(**every** *period*), 그리고 마감시간(**by** *deadline*)을 명세한다. 시간구동 메쏘드는 *start_time*에서 *end_time*까지 매 *period*마다 실행된다. **code** 부분에는 시간구동 메쏘드의 행위와 함께 최악수행시간 (**wcet** *wcet*), 스택의 크기(**stack** *stack_size*), 그리고 중요도(**crit** *criticality*)를 명세한다. **mm** 구역에는 메시지구동 메쏘드를 명세한다. **mm** 구역에는 **aac** 부분이 없고 인자가 있다는 점이 **tm** 구역과 다르다. 실시간객체의 생성자(constructor)와 파괴자(destructor)는 **mm** 구역에 명세되며 각각 "init"과 "dest"라는 고정적인 이름을 갖는 메시지구동 메쏘드이다. 이들은 인자가 없다는 점이 일반 메시지구동 메쏘드와 다르다. 생성자와 파괴자는 생략할 수 있다. 실시간메쏘드의 행위는 C와 같은 프로그래밍 언어로 기술한다.

다른 실시간객체의 메시지구동 메쏘드를 호출할 때에는 다음과 같이 피호출자의 스케줄링 클래스와 마감시간, 그리고 중요도를 지정할 수 있다. 동기호출 시에는 호출자의 스케줄링 클래스와 중요도가 피호출자에게 상속되므로 생략한다. 같은 실시간객체의 메시지구동 메쏘드를 호출하는 경우에는 실시간객체 이름을 생략한다.

[immediate|hard|soft][*rto_name.*]*mm_name*(*argument_list*) **[by** *deadline*][**crit** *criticality*]

dSource 전처리기는 dSource로 만든 프로그램이 특정 커널 상에서 수행될 수 있도록 커널이 제공하는 프리미티브를 사용하는 C나 C++과 같은 실행 가능한 프로그래밍 언어로 변환한다. 그림 5는 dkSim에서 dRTO 모델 기반 프로그램을 개발하는 과정을 보여준다. dSource 전처리기는 dSource로 구현한 RTO들을 dkSim이 제공하는 프리미티브를 사용하는 C 언어로 변환한다. 그런 다음 gcc를 이용하여 목적파일을 생성한 후 리눅스에서 "dksim *objectfile*"과 같이 실행하면 dkSim 상에서 실시간객체들이 수행된다.

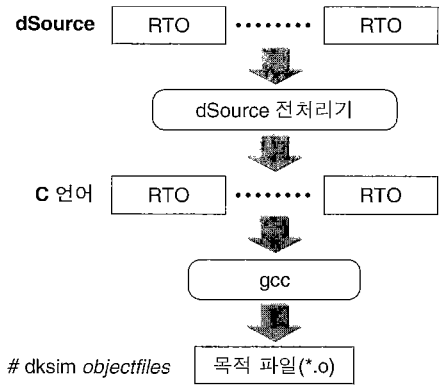


그림 5 dkSim의 개발환경

3. RobustRTO와 RMO

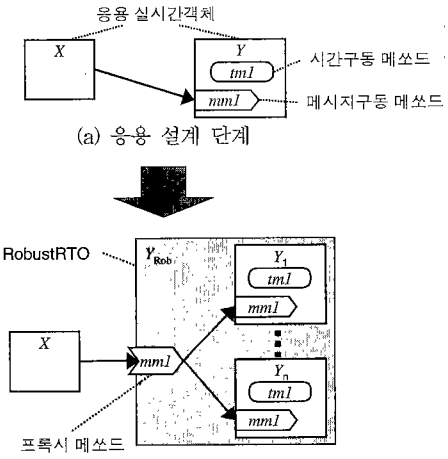
3.1 RobustRTO

RobustRTO는 자신 안에서 발생하는 결함을 처리하는 실시간객체로써 복구블럭, N 버전 프로그래밍, TMR과 같이 여분을 이용하는 하드웨어 및 소프트웨어 결함허용 능력을 갖는다. RobustRTO는 응용 실시간객체를 자신 안에 캡슐화시킨 후 이 응용 실시간객체를 공간 및 시간 복제하고 주어진 결함허용 기법에 따라 이들을 관리함으로써 응용 실시간객체의 신뢰도를 높인다. RobustRTO의 목적은 내부에서 발생한 오류를 다른 실시간객체가 알 수 없도록 하고, 항상 자신의 올바른 상태만 외부에 보일 수 있게 하는 것이다. RobustRTO와 그 안에 캡슐화되는 응용 실시간객체의 특성과 크기에는 제한이 없다. 즉, RobustRTO나 응용 실시간객체는 작은 하드웨어 칩이나 소프트웨어 객체일 수도 있고, 하드웨어 노드일 수도 있다.

RobustRTO는 결함허용과 관련된 모든 정보를 자신 안에 캡슐화함으로써 응용 실시간객체의 설계와 결함허용 설계를 분리한다. 따라서, 결함허용을 위하여 응용 실시간객체가 복제된 경우에도 기존 응용 실시간객체의 인터페이스를 그대로 이용할 수 있도록 해주며 결함허용 구성에 따라 응용 실시간객체의 설계가 변경되지 않게 해준다.

RobustRTO는 캡슐화된 응용 실시간객체의 메시지구동 메쏘드와 이름이 동일한 메시지구동 메쏘드를 포함하는데 이를 프록시 메쏘드(proxy method)라고 한다. 프록시 메쏘드는 결함허용 구성에 상관없이 외부에서 투명하게 RobustRTO 안에 캡슐화된 복제객체들을 접근할 수 있게 해주며 RobustRTO의 결함허용 구성에 따라 복제객체를 제어하고 올바른 결과값을 산출해내는 역할을 하는 메시지구동 메쏘드이다. 프록시 메쏘드는 캡슐화된 응용 실시간객체의 메시지구동 메쏘드마다 존재하며 각 프록시 메쏘드는 이들과 동일한 인터페이스를 갖는다.

그림 6은 응용 실시간객체 Y가 결함허용 능력을 갖게 하기 위하여 RobustRTO로 구성하는 과정을 보여준다. 그림 6 (b)와 같이 응용 실시간객체 Y는 n 개의 복제객체로 복제되고 RobustRTO Y_{Rob}은 이들을 캡슐화한다. 실시간객체 X가 $Y.mm1$을 호출하면 RobustRTO의 프록시 메쏘드 $Y_{Rob}.mm1$은 이 호출을 가로챈 후 RobustRTO의 결함허용 구성 방식에 따라서 복제객체들에 있는 자신과 동일한 이름을 지닌 메쏘드(예를 들면, 그림 6 (b)에서 $Y_1.mm1$이나 $Y_n.mm1$)를 호출한 후 올바른 결과값을 산출하여 X에게 돌려준다. 이와 같이 X는 Y_{Rob}의 결함허용



(a) 응용 설계 단계

(b) 결함허용 설계 단계

그림 6 RobustRTO의 구성 과정

구조에 상관없이 응용 실시간객체 Y와 통신할 수 있으며, 캡슐화된 Y는 복제될 뿐 자체 설계가 변경되지 않는다.

복제객체들은 RobustRTO 안에 물리적으로 포함된 것이 아니고 논리적으로 캡슐화된 것이다. 따라서, RobustRTO와 복제객체간의 물리적인 관계는 구현에 따라 달라지며 세 가지 경우가 있다. 첫째는 복제객체가 RobustRTO 안에 물리적으로 내장되는 경우이다. 하드웨어의 경우, 복제객체는 물리적으로 RobustRTO 안에 패키징될 수 있다. 둘째는 RobustRTO가 하나 이상의 복제객체에 내장되는 경우이다. 복제객체가 노드인 경우 RobustRTO는 그 노드에서 수행되는 소프트웨어 실시간객체일 수 있다. 셋째는 복제객체가 RobustRTO와 동등한 실시간객체인 경우이다. 소프트웨어의 경우 이들은 독립적인 실시간객체로 서로 다른 노드에 분산되어 수행될 수 있다.

RobustRTO의 프록시 메쏘드가 응용 실시간객체에 대한 호출을 가로챌 수 있는 방법으로 정적인 방법과 동적인 방법이 있다. 정적인 방법은 설계나 구현시에 RobustRTO의 이름을 RobustRTO가 캡슐화한 응용 실시간객체의 이름으로 바꾸는 것이다. 즉, 응용 실시간객체 X가 RobustRTO에 캡슐화된 경우 RobustRTO의 이름이 X가 되고 복제객체는 다른 이름을 갖는다. 동적인 방법은 실행시에 호출된 응용 실시간객체가 어느 RobustRTO에 캡슐화되었는지를 알아내어 해당하는 RobustRTO를 호출하게 하는 것이다. 동적인 방법은 실행시 오버헤드를 발생시키므로 본 논문에서는 정적인 방식을 사용한다.

3.2 RMO

RMO는 감시지역(region)에 소속된 실시간객체들의 상태를 감시 및 분석함으로써 실시간객체들의 비정상적인 행위를 감지한 후, 그 증상을 진단하여 결함을 찾아 알맞은 복구 및 재배치 방법을 결정하고 수행하는 실시간객체이다. 감시지역은 결함이나 오류에 연관된 실시간객체들의 집합이다. 여기서 오류란 감시지역 내의 실시간객체들이 준수해야 하는 요구사항을 어긴 것이나 감시지역 내의 특정 실시간객체에서 감지한 것으로써 다른 실시간객체의 결함전파에 의해 발생할 수 있는 것 등을 말한다. 한 오류는 다양한 결함에 의해 발생할 수 있고, 한 결함은 여러 종류의 오류로 나타날 수 있다. 대개 한 오류를 발생시킬 수 있는 실시간객체들을 하나의 감시지역으로 설정한다. 설계자는 한 시스템에 대한 감시지역들을 구성하고 각 감시지역마다 하나의 RMO를 할당한다. 즉, 감시지역은 RMO의 감시, 진단, 복구 행위가 수행되는 범위가 된다. RobustRTO는 동일한 복제객체들을 대상으로 하는 반면 RMO는 서로 다른 실시간객체들을 대상으로 한다는 점이 다르다.

감시지역의 멤버객체로는 응용 실시간객체(그림 7에서 *T*와 *S*)나 RobustRTO(그림 7에서 *X, Y, Z*)가 될 수 있다. RobustRTO 안에 캡슐화된 복제객체 중에서 일부만이 감시지역의 멤버객체가 될 수는 없다. 예를 들면, 그림 7에서 Z_1과 Z_2만 감시지역 *RegB*에 소속되고 Z_3은 감시지역 *RegB*에 소속되지 않은 경우는 잘못된 것이다. 감시지역은 서로 겹칠 수 있다. 따라서, 한 감시지역의 멤버객체는 동시에 다른 감시지역에도 소속될 수 있다(그림 7에서 *S*와 *Y*). 그림 7에서 RMO *RmoA*는 감시지역 *RegA*의 멤버객체인 *X, Y, T, S*를 감시하고 제어한다.

RMO의 기능은 감시, 진단, 복구, 비상사태 처리, 기록의 다섯 가지로 나뉜다. 1) 감시 기능 : 감시지역에 소속된 실시간객체들의 상태와 이들이 주고받는 실시간메시지를 감시한다. 2) 분석 및 진단 기능 : 감시를 통해 얻어진 정보를 분석하여 감시지역이 주어진 요구사항을 만족하고 있는지 판단함으로써 오류나 비정상적인 상태를 감지한다. 오류나 비정상적인 상태가 감지되면 감시지역 내의 실시간객체들을 진단하여 결함을 갖고 있는 실시간객체를 찾고, 알맞은 복구 및 재배치 방법을 결정한다. 3) 복구 및 재배치 기능 : 결정된 복구 및 재배치 방법에 따라 결함을 지닌 실시간객체들을 제어하여 이들을 복구 또는 재배치한다. 예를 들면, 실행 중인 실시간객체를 다른 노드로 이주시키거나 hot standby spare 방식으로 이중화된 경우 부노드와 주노드를 교체한다. 4) 비상사태 처리 기능 : 복구나 재배치가 불가능하여 시스템의 정상적인 운행을 보장할 수 없다고 판단한 경우 시스템을 안전하게 정지시키기 위한 비상사태 처리를 수행한다. 5) 기록 기능 : RMO는 감시정보를 히스토리 형태로 보관하여 분석하는 데 이용한다. 또한, 진단 및 복구 결과 등도 기록하여 시스템에 대한 오프라인 분석을 할 수 있게 한다. 이와 같은 RMO의 다섯 가지 기능을 지원하기 위하여 RMO는 응용 실시간객체들을 감시하고 제어할 수 있는 권한을 갖는다.

4. 기본적인 감시 및 제어 구조

4.1 두 가지 감시 및 제어 구조

그림 8처럼 감시 및 제어 구조는 결함허용 중개자의 위치에 따라 스터브 방식과 커널 모듈 방식으로 구분된다. 스터브 방식에서는 응용 실시간객체마다 결함허용 중개자가 스터브로 붙어서 중개자 역할을 한다. 응용 실시간객체가 RobustRTO에 캡슐화된 경우 결함허용 중개자 스터브는 RobustRTO와 캡슐화된 복제객체에도 붙는다. 커널 모듈 방식에서는 각 노드 당 하나의 결함허용 중개자가 모듈 형태로 커널에 존재한다.

커널 아키텍처가 숨겨져 있거나 커널을 수정하기 어려운 경우, 또는 커널 모드와 유저 모드의 구분이 있어서

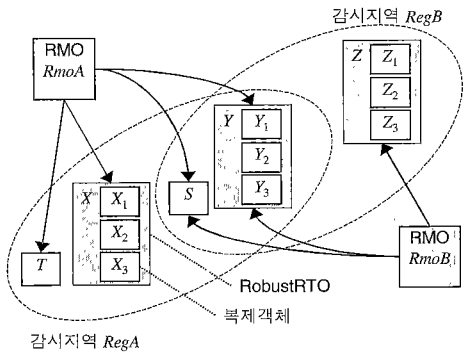


그림 7 감시지역의 예

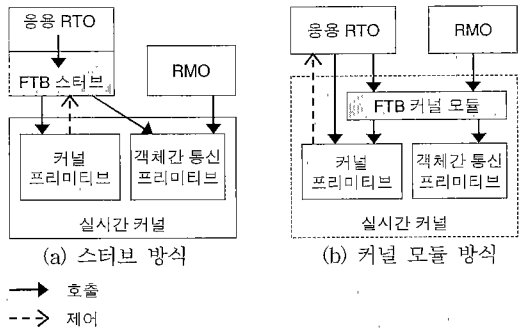


그림 8 기본적인 감시 및 제어 구조

응용 실시간객체마다 주소공간이 분리되어 있는 경우 스터브 방식이 적합하다. 스터브는 응용 실시간객체와 같은 주소공간에 있으므로 결함허용 중개자가 직접 응용 실시간객체를 감시 및 제어하기 쉽다. 또한, 커널을 거의 수정할 필요가 없다. 스터브 방식은 어떤 특성을 갖는 커널에도 적용할 수 있다. 반면 스터브 생성기가 필요하고 스터브 때문에 전체 코드의 크기가 응용 실시간객체의 수에 비례해서 증가할 수 있다. 결함허용 중개자 스터브가 응용 실시간객체의 결함에 영향을 받아서 RMO가 해당하는 응용 실시간객체를 제대로 감시하거나 제어하기 어려울 수 있다.

VxWorks와 같은 대표적인 내장 실시간 운영체제나 dKernel은 커널 모드와 유저 모드의 구분 없이 커널과 응용이 모두 같은 주소 공간에서 수행되며 커널의 깊숙한 부분까지 사용자가 접근하기 용이하도록 풍부한 API를 제공한다. 이 경우 커널 모듈 방식이 적합하다. 결함허용 중개자 커널 모듈이 같은 노드에 있는 모든 응용 실시간객체에 접근하기 용이할 뿐만 아니라 감시 및 제어를 위하여 커널도 쉽게 커스터마이즈할 수 있기 때문이다. 커널 모듈 방식을 유저 모드와 커널 모드가 구분된 커널에 적용하려면 결함허용 중개자가 서로 다른 주소공간을 접근하면서 감시 및 제어해야 하므로 구현이 쉽지 않다. 커널 모듈 방식은 커널 모드와 유저 모드의 구분이 없는 커널에 적합하므로 스터브 방식보다 범용성이 떨어진다. 그러나, 복잡한 스터브 생성기가 필요하지 않고 응용 실시간객체가 많은 경우 전체 코드의 크기가 스터브 방식보다 작다. 응용 실시간객체의 결함이 커널에 영향을 미치지 않는 이상 RMO는 응용 실시간객체를 감시하고 제어할 수 있다.

설계자는 커널의 특징에 따라 스터브 방식과 커널 모듈 방식을 선택하여 사용한다. 스터브 방식은 응용 실시간객체마다 독립된 주소공간이 제공된다고 가정하였으며, 커널 모듈 방식은 커널과 응용이 하나의 주소공간에서 수행된다고 가정하였다. 따라서, 그림 8의 스터브 방식에서 커널은 실선으로 그렸고 커널 모듈 방식에서 커널은 점선으로 그렸다.

본 논문은 스터브 방식에서 실시간객체를 감시하는 구조와 방법을 제시하고 구현한다.

4.2 스터브 방식의 개발환경 및 개발과정

그림 9는 스터브 방식에서 프로그램을 개발하는데 필요한 도구와 개발과정을 보여준다. RobustRTOspec과 RMOspec[5][6]은 각각 RobustRTO와 RMO를 명세하는 틀이다. RobustRTO 생성기(RobustRTOgen)와 RMO 생성기(RMOgen)는 RobustRTOspec과 RMOspec, 그리고 응용

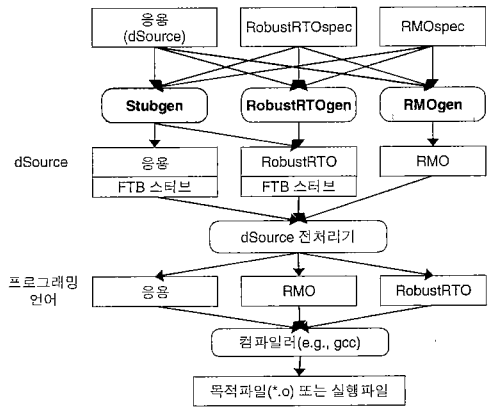


그림 9 스터브 방식의 개발과정

실시간객체를 입력으로 받는다. 그런 다음 두 생성기는 시간구동 메쏘드와 메시지구동 메쏘드, 그리고 실시간자료로 구성된 RobustRTO와 RMO의 기본 골격을 dSource 형태로 출력한다. 이로부터 설계자는 실시간메쏘드의 구체적인 코드를 작성한다.

스터브 생성기는 dSource로 프로그래밍된 응용 실시간객체와 RMOspec, 그리고 RobustRTOspec으로부터 필요한 정보를 입력받아서 각 응용 실시간객체를 위한 결함허용 중개자 스터브를 생성한다. 즉, 결함허용 중개자 스터브는 실시간객체마다 다르다. RMOspec에는 각 RMO들이 어떤 실시간메시지와 실시간자료를 감시하는지가 명세되어 있다. 스터브 생성기는 이 정보를 이용하여 각 응용 실시간객체를 감시하는데 필요한 결함허용 중개자 스터브의 메쏘드와 자료구조를 생성한다. 응용 실시간객체가 RobustRTO에 캡슐화된 경우 스터브 생성기는 RobustRTO와 복제객체를 위한 결함허용 중개자 스터브도 생성한다.

dSource로 된 응용 실시간객체, RobustRTO, RMO가 완성되면 dSource 전처리기에 의해 타겟 프로그래밍 언어로 변환된다. 그런 다음, 컴파일러에 의해 타겟 플랫폼과 운영체제 상에서 수행될 수 있는 목적파일이나 실행파일이 만들어진다. 개발환경에 필요한 도구의 설계 및 구현은 본 논문에서 다루지 않는다.

4.3 실시간객체의 3 가지 상태

실시간객체는 CREATED, INIT, ACTIVE의 세 가지 상태를 갖는다. CREATED 상태는 실시간객체가 생성만 되고 실시간메쏘드는 실행될 수 없는 상태이다. 이 상태에 있는 실시간객체는 "생성되었다"고 한다. INIT 상태는 시간구동 메쏘드는 실행되지 않고 메시지구동 메쏘드만 실행될 수 있는 상태이다. 이 상태에 있는 실시간객체는

"초기화되었다"라고 한다. 실시간객체가 초기화되면 제일 먼저 실시간객체의 생성자 메쏘드가 실행된다. ACTIVE 상태는 시간구동 메쏘드까지 활성화된 상태이다. 이 상태에 있는 실시간객체는 "활성화되었다"고 한다. RMO와 응용 실시간객체들이 활성화되기 전에 결함허용 중개자가 필요로 하는 정보가 초기화되어야 하기 때문에 실시간객체는 3 가지 상태를 차례로 거쳐야 한다. 따라서, 커널은 실시간객체를 생성하고, 초기화하고, 활성화하는 프리미티브를 각각 제공해야 한다.

5. 실시간메시지의 감시

RMO는 응용 실시간객체가 다른 응용 실시간객체의 메시지구동 메쏘드를 호출할 때 발생하는 실시간메시지를 감시한다. 호출되는 응용 실시간객체가 RobustRTO 안에 캡슐화되어 있으면 이 실시간메시지는 원래 응용 실시간객체의 이름을 가진 RobustRTO의 프록시 메쏘드에게 전달된다. 이와 같이 RMO가 실시간메시지를 감시할 수 있도록 결함허용 중개자 스터브는 인터셉터 메쏘드와 실시간메시지 테이블을 제공하고 RMO는 메시지감시 메쏘드를 제공한다. 인터셉터 메쏘드는 실시간메시지를 가로채서 감시 중인 RMO에게 전달하고 이 실시간메시지를 피호출자(RobustRTO 또는 응용 실시간객체)에게 전달한다. 인터셉터 메쏘드가 실시간메시지를 가로챌 수 있도록 스터브 생성기는 호출자의 코드 중에서 피호출자를 호출하는 부분을 인터셉터를 호출하도록 변경한다. 실시간메시지 테이블에는 각 실시간메시지를 감시중인 RMO들이 기록되어 있다. 즉, 인터셉터는 어떤 RMO에게 실시간메시지를 보내야 하는지를 알아내기 위하여 실시간메시지 테이블을 참조한다. 메시지감시 메쏘드는 RMO의 메시지구동 메쏘드로써 인터셉터가 보내준 실시간메시지를 받아서 RMO의 히스토리에 실시간메시지에 대한 정보를 기록한다. 한 실시간메시지를 감시하는 RMO는 여러 개 존재할 수 있으며, 한 RMO는 여러 실시간메시지를 감시한다. 그림 10은 실시간메시지 처리를 위한 기본 구조를 보여준다.

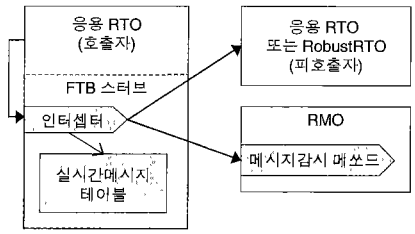


그림 10 실시간메시지 처리를 위한 기본 구조

5.1 인터셉터 메쏘드

인터셉터는 메시지구동 메쏘드이며 실시간객체에서 발생하는 실시간메시지마다 하나씩 존재한다. 인터셉터 메쏘드의 이름은 "FtbIntercept_"로 시작하며 그 뒤에 실시간메시지의 호출자와 피호출자의 이름을 쓴다. 호출자와 피호출자 사이와 실시간객체와 실시간메쏘드 사이는 '_'로 구분한다. 예를 들면, *X.mm1*이 *Y.mm2*를 호출한 경우 이를 가로채는 인터셉터 메쏘드의 이름은 *FtbIntercept_X_mm1_Y_mm2*가 된다. 인터셉터 메쏘드의 인자로는 피호출자의 인자들과 종료시간, 그리고 스케줄링 클래스가 있다. 그림 11처럼 스터브 생성기는 *X.mm1*이 *Y.mm2*를 경성 비동기 호출하는 dSource 코드를 인터셉터 메쏘드를 호출하는 코드로 바꾸고 인터셉터 메쏘드를 만든다.

```
/* X.mm1의 코드 */
  hard Y.mm2(l,m,n) BY 10;
```

스터브 생성기

```
/* X.mm1의 코드 */
  FtbIntercept_X_mm1_Y_mm2(
                l,m,n,10,SCHEDCLASS_HARD);
```

```
/* X를 위한 결함허용 중개자 스터브 코드 */
FtbIntercept_X_mm1_Y_mm2
(int l, float m, char n,
 int deadline, int sched_class)
{
  . . .
}
```

그림 11 인터셉터 메쏘드의 실시간메시지 가로채기

5.2 실시간메시지 테이블

실시간메시지 테이블에는 한 응용 실시간객체에서 발생하는 실시간메시지와 이를 감시 중인 RMO들이 기록된다. 실시간메시지는 호출자와 피호출자의 쌍으로 식별하며, 호출자와 피호출자는 각각 실시간객체와 실시간메쏘드의 쌍으로 나타낸다. 실시간메시지 테이블의 자료구조는 그림 12와 같다. 한 실시간메시지를 감시하는 RMO는 여러 개일 수 있으므로 한 엔트리에 RMO의 RID (RTO ID)는 여러 개 기록될 수 있어야 한다. RID와 MID (Method ID)는 구현에 따라서 정수형태의 ID나 문자열형태의 이름이 된다.

그림 12 실시간메시지 테이블의 자료구조

RMO가 실시간메시지 테이블에 접근할 수 있도록 결함허용 중개자 스터브는 FtbAddMesg와 FtbDelMesg의 두 메시지구동 메쏘드를 제공한다. 그림 13은 IDL로 표현한 FtbAddMesg의 API와 알고리즘을 보여준다. FtbAddMesg는 실시간메시지의 호출자와 피호출자, 그리고 RMO의 RID를 인자로 갖는다. 리턴값은 RMO가 제대로 등록(또는 삭제)되었는지를 나타낸다. FtbDelMesg의 API는 FtbAddMesg의 API와 동일하다. 한 응용 실시간객체에서 발생할 실시간메시지는 설계시 미리 알 수 있을 뿐만 아니라 실행시 변경되지 않으므로 실시간메시지 테이블의 호출자와 피호출자 부분은 스터브 생성기에 의해 초기화된다. 따라서, FtbAddMesg와 FtbDelMesg는 실시간메시지 테이블에서 실시간메시지를 찾아서 RMO의 RID를 추가하거나 삭제한다.

```
short  FtbAddMesg(in RID caller_rto, in MID caller_method,
                  in RID callee_rto, in MID callee_method,
                  in RID rmo)
{
  1. 실시간메시지 테이블에서 호출자가 caller_rto.caller_
     method이고 피호출자가 callee_rto.callee_method인
     실시간메시지를 찾는다
  2. 해당하는 엔트리에 rmo를 추가한다
}
```

그림 13 스터브 방식에서 FtdAddMesg의 API와 알고리즘

실시간메시지는 호출자에 있는 인터셉터 메쏘드가 가로챈다. 따라서, RMO는 감시하려는 실시간메시지의 호출자에 있는 실시간메시지 테이블에 자신을 등록하거나 삭제해야 한다. *X.mm1*이 *Y.mm2*를 호출할 때 발생하는 실시간메시지를 감시하는 경우 RMO는 *X.FtbAddMesg*를 동기 호출한다.

실시간메시지 테이블은 응용이 수행되기 전에 초기화되어야 한다. 이를 위하여 시스템은 다음과 같은 초기화 과정을 거친다. 먼저 모든 실시간객체들을 생성한다(CREATED 상태). 그런 다음, 응용 실시간객체들만 초기화한다(INIT 상태). 그러면, 응용 실시간객체의 생성자만 실행되고 시간구동 메쏘드는 실행되지 않는다. 다음으로 RMO들을 초기화한다. 이 때 RMO의 생성자가 실행되는데 이 생성자에는 자신이 감시할 실시간메시지들을 등록하기 위하여 FtbAddMesg를 호출하는 코드가 존재한다. 따라서, RMO가 초기화될 때 각 응용 실시간객체에 붙어 있는 결함허용 중개자 스터브의 실시간메시지 테이블이 초기화된다. 모든 RMO의 생성자 메쏘드가 수행을 마치면 시스템 내의 모든 응용 실시간객체와 RMO를 활성화한다(ACTIVE 상태). 실시간메시지 테이블이 초기화되었으므로 RMO는 실시간메시지를 감시할 수 있다.

5.3 메시지감시 메쏘드

RMO는 인터셉터 메쏘드가 가로챈 실시간메시지를 받아서 자신의 히스토리에 기록하는 메시지감시 메쏘드를 갖는다. 이 메쏘드는 메시지구동 메쏘드이며 RMO가 감시하는 실시간메시지마다 존재한다. 스터브의 인터셉터 메쏘드는 가로챈 실시간메시지를 RMO에게 전달할 때 메시지감시 메쏘드를 비동기 호출한다. 메시지감시 메쏘드의 이름은 "MonMesg_"로 시작하며 그 뒤에는 실시간메시지의 호출자와 피호출자의 이름을 쓴다. 예를 들면, *X.mm1*이 *Y.mm2*를 호출할 때 발생하는 실시간메시지를 감시하는 메시지감시 메쏘드의 이름은 *MonMesg_X_mm1_Y_mm2*가 된다. 메시지감시 메쏘드의 인자로는 호출자, 피호출자, 피호출자의 인자들, 호출 방식(동기 호출 또는 비동기 호출), 실시간쓰레드의 스케줄링 클래스(긴급, 경성, 연성), 실시간메시지의 방향(호출 방향 또는 리턴 방향), 마감시간, 실시간메시지의 발생시간이 있다. 피호출자의 인자는 피호출자의 인터페이스에 따라 달라지므로 메시지감시 메쏘드의 인터페이스도 메쏘드마다 다르다.

5.4 인터셉터 메쏘드의 알고리즘

그림 14는 동기 호출을 가로채는 인터셉터 메쏘드의 간단한 알고리즘이다. 실시간메시지를 감시하는 RMO를 실시간메시지 테이블에서 찾는다. 실시간메시지를 감시하는 RMO가 존재하면, 호출한 실시간쓰레드의 스케줄링 클래스를 알아낸다. 동기호출에서 피호출자의 스케줄링 클래스는 호출자의 스케줄링 클래스가 되므로 호출자의 스케줄링 클래스만 알면 피호출자의 스케줄링 클래스도 알 수 있다. 그런 다음, RMO의 메시지감시 메쏘드를 비동기 호출한다. 인터셉터 메쏘드는 피호출자를 동기 호출한 후 블록되어 있다가 피호출자가 실행을 마치면 다시 깨어나 계속 실행된다. 인터셉터 메쏘드는 리턴된 실

```
1. 실시간메시지 테이블에서 실시간메시지를 감시하는
   RMO를 알아낸다
2. RMO가 있는 경우
   2.1. 호출한 실시간쓰레드의 스케줄링 클래스를 알아낸다
   2.2. RMO의 메시지감시 메쏘드를 비동기 호출한다
   2.3. 피호출자를 동기 호출한다
   2.4. RMO의 메시지감시 메쏘드를 비동기 호출한다
3. RMO가 없는 경우
   3.1. 피호출자를 동기 호출한다
```

그림 14 동기 호출을 가로채는 인터셉터 메쏘드의 알고리즘

시간메시지에 대한 정보를 RMO에게 알리기 위하여 메시지감시 메쏘드를 비동기 호출하고 종료된다.

그림 15는 비동기 호출을 가로채는 인터셉터 메쏘드의 간단한 알고리즘이다. 이 알고리즘은 피호출자에게 먼저 실시간메시지를 전달한다는 점과 리턴되는 실시간메시지를 처리하지 않는다는 것이 동기 호출을 가로채는 인터셉터 메쏘드와 다르다. 비동기 호출 시에는 호출자가 피호출자의 스케줄링 클래스를 지정하므로 인터셉터 메쏘드는 호출자가 지정한 스케줄링 클래스로 피호출자를 비동기 호출한다. 그런 다음, 이 실시간메시지를 감시하는 RMO에게 실시간메시지에 대한 정보를 알리기 위하여 RMO의 메시지감시 메쏘드를 비동기 호출한다.

1. 호출자가 지정한 스케줄링 클래스에 따라 피호출자를 호출한다
2. 실시간메시지 테이블에서 실시간메시지를 감시하는 RMO를 알아낸다
3. RMO가 있는 경우
   3.1. RMO의 메시지감시 메쏘드를 비동기 호출한다

그림 15 비동기 호출을 가로채는 인터셉터 메쏘드의 알고리즘

5.5 예

그림 16은 *X.mm1*이 *Y.mm2*를 동기 호출할 때 발생한 실시간메시지를 RMO *R*이 감시하는 예를 보여준다. RMO *R*의 생성자 *init*은 호출자가 *X.mm1*이고 피호출자가 *Y.mm2*인 실시간메시지를 RMO *R*이 감시한다고 *X*의 실시간메시지 테이블에 등록하기 위하여 *X.FtbAddMesg*를 호출한다(①). 스터브 생성기는 *X.mm1*이 피호출자 *Y.mm2*를 호출하는 코드를 인터셉터 메쏘드 *FtbIntercept_X_mm1_Y_mm2*를 호출하도록 조작하였다. 따라서, 실행시에 *X.mm1*은 *Y.mm2*가 아닌 인터셉터 메쏘드를 호출한다(②). 인터셉터 메쏘드는 실시간메시지 테이블에서 *R*이 이 실시간메시지를 감시하고 있다는 것을 알아낸 후 *R.MonMesg_X_mm1_Y_mm2*를 비동기 호출함으로써 실시간메시지에 대한 정보를 *R*에게 전달한다(③). 그런 다음, 인터셉터 메쏘드는 *Y.mm2*를 동기 호출한다(④). *Y.mm2*가 종료되면 블럭되어 있던 인터셉터 메쏘드가 리턴된 실시간메시지에 대한 정보를 *R*에게 전달하기 위하여 *R.MonMesg_X_mm1_Y_mm2*를 비동기 호출한 후 수행을 마친다(⑤).

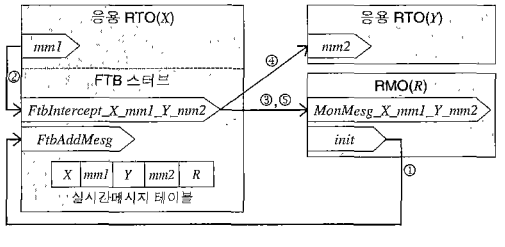


그림 16 스터브를 이용한 실시간메시지 감시의 예

6. 실시간객체의 실시간자료 감시

실시간자료를 감시하기 위한 구조는 결함허용 중개자의 실시간자료 테이블과 FtbGetVal 함수, 그리고 RMO의 자료감시 메쏘드로 구성된다. 그림 17은 실시간자료를 감시하는 구조를 보여준다.

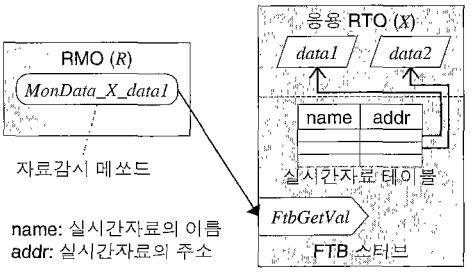


그림 17 스터브를 이용한 실시간자료 감시 구조

실시간자료 테이블은 한 응용 실시간객체에 소속된 실시간자료의 이름과 해당하는 실시간자료를 가리키는 포인터로 구성된다. 한 응용 실시간객체의 실시간자료는 설계시 고정되므로 스터브의 실시간자료 테이블은 스터브 생성기에 의해 만들어지고 초기화된다.

IDL로 표현한 FtbGetVal 메시지구동 메쏘드의 API는 "void FtbGetVal(in string rtdata_name, out any value);"와 같다. FtbGetVal 메쏘드는 첫번째 인자로 받은 실시간자료의 이름을 실시간자료 테이블에서 찾아서 그 주소를 알아낸 다음, 실시간자료의 값을 두번째 인자를 통해 호출자에게 알려준다.

실시간자료를 감시하는 RMO의 자료감시 메쏘드는 시간구동 메쏘드이며 RMO가 감시하는 실시간자료마다 존재한다. 이 메쏘드의 이름은 "MonData_" 접두사와 실시간자료의 이름으로 구성된다. 응용 실시간객체 *X*의 실시간자료 *data1*을 감시하는 경우 RMO는 *MonData_X_data1*이라는 자료감시 메쏘드를 갖는다. 자료감시 메쏘드는 실시간자료가 있는 응용 실시간객체의 FtbGetVal 메쏘드를 동기 호출한다. 예를 들면, 응용 실시간객체 *X*의 실시간자료 *data1*을 감시하는 경우 자료감시 메쏘드 *MonData_X_data1*은 *X.FtbGetVal("data1", X_data_value);*

와 같이 동기 호출한다. 자료감시 메쏘드는 실시간자료의 값을 알아낸 후 이를 자신의 히스토리에 로깅한다.

RobustRTO는 캡슐화한 응용 실시간객체의 실시간자료들을 자신의 실시간자료로 갖는다. RobustRTO는 내장된 결함허용 기법을 이용하여 오류가 없는 올바른 값만 자신의 실시간자료에 기록한다. RobustRTO는 캡슐화한 응용 실시간객체의 이름을 갖으므로 RMO가 응용 실시간객체의 실시간자료를 감시할 때에 실제로는 RobustRTO의 실시간자료를 감시하게 된다. RMO는 응용 실시간객체가 RobustRTO에 캡슐화되었는지 여부에 관계없이 실시간자료를 감시한다.

7. 감시 구조의 구현 예제

본 장에서는 dSource로 구현된 응용 실시간객체와 이를 감시하는 RMO를 보인다. 이들은 dkSim에서 실행되며 2 장에서 설명한 dSource 전처리기에 의해 dkSim에서 실행될 수 있는 C 언어로 변환된다. dkSim에서 시간 단위인 클럭틱은 0.5 ms로 설정되어 있고 스택 크기의 단위는 Kbyte로 설정되어있다.

실험에 사용된 기본적인 응용은 두 개의 실시간객체 *X*와 *Y*로 구성되며 이들을 감시하는 RMO인 *rmoR*이 존재한다. 그림 18은 *rmoR*과 결함허용 중개자 스터브가 추가된 *X*와 *Y*의 전체적인 구조를 보여준다. *X*는 실시간자료로 float 형의 *nping*과 char 형 배열의 *name*을 갖으며 주기가 10 ms인 시간구동 메쏘드 *tmPing*을 갖는다. *Y*는 실시간자료로 int 형의 *count*를 갖고 시간구동 메쏘드 *tmInc*와 메시지구동 메쏘드 *mmGetCnt*를 갖는다.

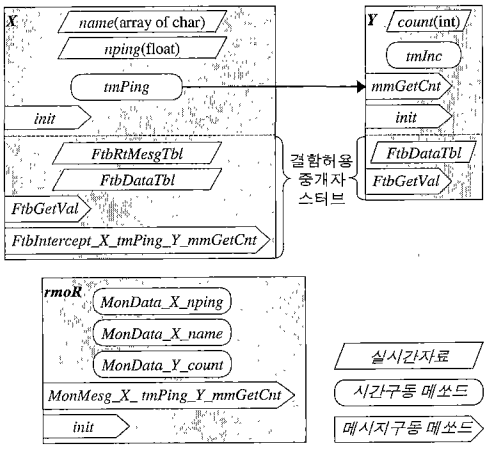


그림 18 응용 실시간객체 X와 Y, 그리고 이들을 감시하는 RMO rmoR

*Y.tmInc*는 2 ms에 한 번씩 *Y.count*의 값을 증가시킨다. *X.tmPing*은 *Y.mmGetCnt*를 호출하여 *Y.count*의 값을 알아낸 후 이를 float형으로 변환하여 *X.nping*에 기록한다. *rmoR*은 *X.nping*, *X.name*, *Y.count*의 세 실시간자료를 감시한다. 이를 위하여 *X*와 *Y*의 결함허용 중개자 스터브에는 실시간자료 테이블(*X.FtbDataTbl*과 *Y.FtbDataTbl*)과 *FtbGetVal* 메시지구동 메쏘드가 있고, *rmoR*에는 세 개의 자료감시 메쏘드(*MonData_X_nping*, *MonData_X_name*, *MonData_Y_count*)가 존재한다.

그림 19는 *rmoR*의 dSource 코드를 보여준다. *rmoR.MonData_X_nping*이 주기적로 *X.FtbGetVal*을 호출하여 *X.nping*의 값을 감시한다. **aac** 부분에 명세된 이 메쏘드의 주기와 마감시간은 각각 20 틱(10 ms)과 2 틱(또는 1 ms)이고 **code** 옆에 명세된 최악수행시간은 1 틱(0.5 ms)이다. 이 메쏘드의 이름 옆에 명세된 스택의 크기는 1 KB이다.

*rmoR*은 *X.tmPing*이 *Y.mmGetCnt*를 동기 호출할 때 발생하는 실시간메시지를 감시한다. 이를 위하여 *X*는 실시간메시지 테이블(*X.FtbRtMesgTbl*)과 인터셉터 메쏘드(*X.FtbIntercept_X_tmPing_Y_mmGetCnt*)를 갖으며 *rmoR*은 메시지감시 메쏘드 (*rmoR.MonMesg_X_tmPing_Y_mmGetCnt*)를 갖는다. 그림 19는 메시지감시 메쏘드의 인터페이스를 보여준다. RMO의 생성자 *rmoR.init*은 *X.FtbAddMesg*를 호출하여 실시간메시지 테이블 *X.FtbRtMesgTbl*에 자신을 등록한다.

그림 20은 실시간객체 *X*의 실시간자료들과 시간구동 메쏘드 *tmPing*을 보여준다. *X.tmPing*이 *Y.mmGetCnt*를 호출하는 코드 "*Y.mmGetCnt(count) by 2;*"는 스터브 생

```
rmo rmoR {
  ⋮
tm:
  MonData_X_nping {
    aac { every 20 by 3 }
    code wcet 1 stack 1 {
      X.FtbGetVal("nping", value);
      /* value의 값을 히스토리에 기록 */
        ⋮
    }
  }
  MonData_X_name   {. . .}
  MonData_Y_count  {. . .}
mm:
  init {
    code wcet 1 stack 1 {
      X.FtbAddMesg(RID_X, MID_tmPing, RID_Y, MID_mmGetCnt, RID_rmoR);
    }
  }
  MonMesg_X_tmPing_Y_mmInc {
    code {incut int count, in RID caller_rid, in MID caller_mid,
          in RID callee_rid, in MID callee_mid,
          in TimeIndex dealdine, in int sched_class,
          in int call_type, in TimeIndex msg_time)
          wcet 1 stack 1 {
      /* 실시간메시지를 히스토리에 기록 */
        ⋮
    }
  }
}
```

그림 19 *rmoR*의 dSource 코드

```
rto X {
  ⋮
ods:
  float nping;
  char name[10];
  rtMesgTbl_t FtbRtMesgTbl;
  dataTbl_t FtbDataTbl[2];
  ⋮
tm:
  hard tmPing {
    aac { every 20 by 8 }
    code wcet 4 stack 1 {
        ⋮
      FtbIntercept_X_tmPing_Y_mmGetCnt(count,3,SCHEDCLASS_UNKNOWN);
        ⋮
    }
  }
mm:
  ⋮
}
```

그림 20 실시간객체 *X*의 실시간자료와 시간구동 메쏘드 *tmPing*

```
FtbIntercept_X_tmPing_Y_mmGetCnt  {
  code (inout int count, in long deadline, in int sched_class)
  wcet 3 stack 1 {
      RID rmoRid[MAX_RMO_PER_RTMESG];
      int num = 0, i, sc;

      if (sched_class == SCHEDCLASS_UNKNOWN)
          sc = dkThread_GetSchedClass();
      else sc = sched_class;

      num = FtbFindRmo(rmoRid, RID_X, MID_tmPing,
                               RID_Y, MID_mmGetCnt);

      for (i=0; i<num; i++) {
        hard RTOID(rmoRid[i]).MonMsg_X_tmPing_Y_mmGetCnt
             (count, RID_X, MID_tmPing, RID_Y, MID_mmGetCnt,
              deadline, sc, BLOCKING_CALL, msg_time) by 1;
      }

      Y.mmGetCnt(count) by deadline-1;

      for (i=0; i<num; i++) {
        hard RTOID(rmoRid[i]).MonMsg_X_tmPing_Y_mmGetCnt
             (count, RID_X, MID_tmPing, RID_Y, MID_mmGetCnt,
              deadline, sc, BLOCKING_CALL_RETURN, msg_time) by 1;
      } /* for */
  } /* code */
} /* Interceptor method */
```

그림 21 실시간객체 *X*의 인터셉터 메쏘드

성기에 의해 인터셉터 메쏘드를 호출하는 코드로 변경되었다.

그림 21은 인터셉터 메쏘드의 코드이다. *rmoRid*는 실시간메시지를 감시하는 RMO의 RID를 저장할 배열이고 MAX_RMO_PER_RTMESG는 한 실시간메시지를 감시하는 RMO의 최대 수이다. dkSim은 호출자의 스케줄링 클래스를 알려주는 *dkThread_GetSchedClass* 프리미티브를 제공한다. 이를 이용하여 스케줄링 클래스를 알아낸 후 RMO를 찾아낸다. *FtbFindRmo*는 호출자와 피호출자를 인자로 받아서 해당하는 실시간메시지를 감시하는 RMO를 실시간메시지 테이블에서 찾아서 마지막 인자로 받은 RID의 배열인 *rmoRid*에 모두 기록한다. 그리고, 실시간메시지를 감시하는 RMO의 수를 리턴한다. 그런 다음, 실시간메시지를 감시하는 모든 RMO를 비동기 호출하여 실시간메시지에 대한 정보를 알린다. RTOID는 dSource 전처리기에 대한 지시자로써 RID가 인자이다. dSource 전처리기는 RTOID를 만나면 인자로 받은 RID에 해당하는 실시간객체의 메시지구동 메쏘드를 호출하도록 조작한다. 실행시에 어떤 실시간메시지를 감시하는 RMO가 추가 및 삭제되어도 이를 감시하는 RMO에게 전달할 수 있다. *Y.mmGetCnt*를 동기 호출한 후 리턴된 실시간메시지를 *num* 개의 RMO에게 알려주고 인터셉터 메쏘드는 종료된다.

8. 결 론

실시간객체 모델에 기반하며 RobustRTO와 RMO의 두 개념을 제공하는 결함허용 실시간 시스템 구축을 위한 모델을 소개하였다. RobustRTO는 자신 안에서 발생하는 결함을 감지하고 처리하는 실시간객체이다. RMO는 여러 실시간객체들의 상태를 감시 및 분석함으로써 시스템의 비정상적인 행위를 감지한 후, 그 증상을 진단하여 결함을 찾아 알맞은 복구 및 재배치 방법을 결정하고 수행하는 실시간객체이다. 이를 위하여 RMO는 응용 실시간객체를 감시하고 제어할 수 있는 권한을 갖는다.

본 논문은 RMO의 권한이 응용의 설계와 응용의 위치에 투명하게 수행될 수 있는 감시 및 제어 구조를 제안하였다. 제안한 구조는 실시간 커널과 결함허용 중개자로 구성된다. 결함허용 중개자는 RMO와 응용 실시간객체의 중간에서 중개자 역할을 하여 RMO가 응용 실시간객체의 위치에 관계없이 감시와 제어를 할 수 있게 해주며, 응용의 코드가 RMO의 감시 기능에 의해 변경되지 않게 한다. 결함허용 중개자의 위치에 따라 스터브 방식과 커널 모듈 방식의 두 가지 구조가 있는데 본 논문에서는 스터브 방식에서 실시간객체를 감시하는 구조를 설계하였다. 실시간자료와 실시간메시지를 감시하는 간단한 예를 dSource 코드로 구현하였으며 dKernel에서 실험하였다.

향후과제로 커널 모듈 방식에서 RMO의 권한을 지원하는 감시 및 제어 구조에 대한 연구가 있다. 그리고, [5]와 [6]에서 제시한 RobustRTOspec과 RMOspec으로부터 자동으로 dSource 형태의 RobustRTO와 RMO를 만들어 내는 RobustRTOgen과 RMOgen, 그리고 결함허용 중개자 스터브(역시 dSource 형태)를 생성하는 스터브 생성기를 개발한다.

참 고 문 헌

[1] G. Booch, 'Object-Oriented Analysis and Design,' 2nd Ed., The Benjamin/Cummings Pub., 1994.

[2] K.H. Kim, "Object Structures for Real-Time Systems and Simulators," IEEE Computer, Aug., 1997.

[3] 이신, 손혁수, 양승민, "실시간 객체 모델 dRTO", 한국정보과학회 논문지: 소프트웨어 및 응용, 제27권, 제3호, pp.300-312, 2000. 3.
[4] J.-C. Fabre and T. Perennou, "A Metaobject Architecture for Fault-Tolerant Distributed Systems: The FRIENDS Approach, " IEEE Trans. on Computers, Jan., pp.78-95, 1998.
[5] 임형택, 양승민, "실시간객체 기반 결함허용 시스템 모델링", 한국정보처리학회 논문지 제6권 제8호, pp. 2233-2244, 1999.
[6] H.T. Lim and S.M. Yang, "A Framework to Model Dependable Real-Time Systems based on Real-Time Object Model," to appear in RTCSA'00, 2000.
[7] B. Randell, "System Structure for Software Fault Tolerance," IEEE Transactions on Software Engineering, pp.220-232, Jun. 1975.
[8] A. Avizienis, "The Methodology of N-version Programming," Software Fault Tolerance, Michael R. Lyu, Wiley, pp.23-46,1995.
[9] N. Storey, Safety-Critical Computer Systems, *Addison- Wesley, 1996.*
[10] K. H. Kim, "Action-Level Fault Tolerance, Advances in Real-time Systems," Advances in Real-Time Systems, S. Son, Prentice-Hall, pp.415-434, 1995.
[11] Rachid Guerraoui and Andre Schiper, "Software-Based Replication for Fault Tolerance," IEEE Computer, Vol. 30, No. 4, pp. 68-74, April 1997.
[12] F.B. Schneider, "Replication Management using the State-Machine Approach," in Distributed System, by S. Mullender, ed., ch.7, pp.169-198, Addison Wesley, 2nd Ed., 1993.
[13] N. Budhiraja, K. Marzullo, F.B. Schneider, and S. *Toueg, "The Primary-Backup Approach," in* Distributed System, by S. Mullender, ed., ch.8, pp.199-216, Addison Wesley, 2nd Ed., 1993.

임 형 택
1994년 숭실대학교 전자계산학과 졸업(학사). 1996년 숭실대학교 대학원 컴퓨터학과 졸업(석사). 1996년 ~ 현재 숭실대학교 대학원 컴퓨터학과 박사과정. 관심분야는 결함허용, 실시간 시스템, 정형 명세 등

양 승 민
1978년 2월서울대학교 전자공학(학사). 1983년 4월 미국 Univ. of South Florida 전산학(석사). 1986년 12월 미국 Univ. of South Florida 전산학(박사). 1988년 ~ 1993년 미국 Univ. of Texas at Arlington 조교수. 1996년 ~ 1998년 국회도서관 정보처리국장. 1993년 ~ 현재 숭실대학교 컴퓨터학과 부교수. 관심분야는 실시간 시스템, 운영체제, 결함허용 시스템 등

SMIL, RDF, WIDL 문서의 통합 객체 모델링

(Integrated Object Modeling for SMIL, RDF, WIDL Documents)

김 상 은 † 하 얀 †† 김 용 성 †††
(Sang-Eun Kim) (Yan Ha) (Yong-Sung Kim)

요 약 XML은 다양하게 응용할 수가 있어 여러 분야에서 널리 사용되고 있다. 그러나 이러한 응용들에 대해서 통합적으로 관리해 주는 시스템은 제안된 바 없어, 각각의 응용에 대해 별개의 언어로 사용되고 있다. 따라서, 본 논문은 XML의 다양한 응용 중에 웹을 기반으로 하는 대표적인 응용인 SMIL, RDF, WIDL에 대해, 이들의 DTD와 문서 인스턴스를 통합하는 객체 모델링을 하고자 한다. 각 XML 응용에 대해 객체 모델링 규칙과 알고리즘을 통합할 수 있는 시스템을 제안한다. 이를 통해 XML 종류에 상관없이 웹 기반 XML 응용의 구조를 쉽게 파악할 수 있으므로 문서 생성을 용이하게 하며, 객체지향 스키마를 쉽게 생성할 수 있으므로 객체지향 데이터베이스 문서관리의 기반이 될 것이다.

Abstract For having various applications, XML is widely used in various fields. But, there have not been proposed a system to integrate and manage XML and its applications together. In this paper, we propose methods to integrate web-based XML applications(SMIL, RDF, WIDL) and for object-oriented modeling of each DTD and document instance. We propose an system to merge object modeling of XML applications. With the proposed integrating algorithm, we can not only easily analysis various web-based XML applications which is represented by complex tags but also generate object-oriented schema for each document and store it to OODBMS.

1. 서 론

XML(eXtensible Markup Language)은 특정 응용 목적에 맞게 엘리먼트, 애트리뷰트, 엔티티를 정의할 수 있는 융통성(flexibility)을 갖고 있으며, 대표적인 예로 SMIL(Synchronized Multimedia Integration Language), RDF(Resource Description Framework), WIDL (Web Interface Definition Language) 등이 있다[1]. 이들 XML 응용들은 웹을 기반으로 한 형태이기 때문에 앞으로 그 사용이 점차 급증하리라 예상된다. 이들 응용에 대해 간략하게 살펴보면, SMIL은 그래픽, 오디오 등 멀티미디어 내용을 웹에 전송할 수 있도록 지원해 주는 표준화된 문법이며[2], RDF는 메타데이타를 처리하기 위한 기반으로 웹에서 기계가 읽을 수 있는 정보를 교환하는 응용들(applications) 간에 상호운영성(interoperability)을 제공하는 규정이다[3]. 그리고, WIDL은 웹 데이타와 서비스를 위한 응용 프로그램 인터페이스(APIs)를 정의하기 위한 메타데이타 문법이다. 지금까지 업무 응용 분야에서 웹 데이타에 직접 접근하는 문제에 대해서는 거의 무시되어 왔으나, 최근에는 WIDL로 인하여 웹 자원을 원격 시스템에서 표준 웹 프로토콜에 접근할 수 있는 인터페이스로 기술되어 사용되고 있다. WIDL의 목적은 표준 웹 프로토콜에 대한 요청/응답 상호작용을 표현하는 일반적인 방법을 제공하고, 웹에서 다양한 통합 플랫폼으로 이용될 수 있는 HTML/XML 문서와 형태를 가진 지원들을 상호작용할 수 있게 자동화하는데 있다. WIDL에서 정의한 서비스는 웹의 자원이 수정 없이 다른 업무 시스템과 잘 통합되도록 웹 내용을 프로그램 변수에 사상되어야 한다[4, 5, 6].

이러한 특징을 갖고 있는 WIDL 역시 객체지향적인

† 종신회원 : 대경대학 컴퓨터통신계열 교수
sekim@tk.ac.kr
†† 정 회 원 : 중앙대학교 정보통신연구소 연구전담교수
hayan@object.cse.cau.ac.kr
††† 종신회원 : 전북대학교 컴퓨터과학과 교수
yskim@moak.chonbuk.ac.kr
논문접수 : 2000년 5월 2일
심사완료 : 2000년 10월 9일