분산 실시간 객체 TMO의 실행 지원을 위한 미들웨어 및 실시간/임베디드 커널의 구현 동향†

한국외국어대학교 김정국
건국대학교 김문회

1. 서 론

TMO(Time-triggered Message-triggered Object)는 Kane Kim과 Kopetz[1]에 의해 처음 제안된 분산 실시간 객체 모델로 경성 및 연성 실시간 응용, 병렬 컴퓨팅에 널리 사용되며, 객체 내부 멤버 스레드의 기능적 수행에 대해 시간적 조건을 명세할 수 있는 특징을 가진다. 이러한 TMO 모델의 수행을 위한 엔진들은 기존 운영체제 상의 미들웨어와 공개 소스 기반 커널의 형태로 개발되어 왔으며 윈도우 기반의 WTMOS[2], TMOSM/NT[10], 리눅스 기반의 LTMOS[4]와 TMO-Linux 커널[6] 및 TMO-eCos 커널[9] 등이 그것이다. 윈도우와 리눅스 운영체제 상의 미들웨어 엔진들은 이식성과 호환성의 장점이 있고, 커널 형태의 엔진들은 보다 정확한 정시 보장성을 요구하는 응용을 위한 실시간 특성과 임베디드 응용들을 위한 특성들을 가진다. 현재까지 미들웨어 형태의 TMO 엔진들은 실시간 시뮬레이션 응용[5,8]과 멀티미디어 서비스 응용[3,7]에 주로 이용되고 있으며, 커널 엔진들은 분산 제어나 로봇 제어 등에 주로 활용되고 있다.

본 고에서는 TMO 모델 및 TMO 엔진의 필요 기능 및 구현 동향에 대해 소개한다. 또한 TMO 엔진들의 특성과 적합한 활용 분야에 대해 기술한다.

2. TMO 모델 및 프로그래밍

2.1 TMO 모델

분산 실시간 객체 모델인 TMO(Time-triggered Message-triggered Object)는 객체 기반의 경성/연성 실시간 프로그래밍과 시스템 설계 시부터의 정시 서비스 보장을 고려한 구성을 지향한다. TMO는 다음과 같은 세 가지의 대표적 멤버들로 구성된다.(그림 1)

- ODS(Object Data Store) : 객체 내의 실시간 자료 저장소로 객체 내의 메소드들에게만 접근이 허용된다.
- SpM(Time-triggered Spontaneous Method): 응용 설계 시부터 주어지는 시간 조건에 의해 구동되는 실시간 메소드로 객체의 멤버 스레드이다. 즉 일반적 메소드가 수동적 함수인데 반해 SpM은 동적 스레드로 구현된다. SpM의 시간 조건은 AAC(Autonomous Activation Condition)라 하며 주기, 매 주기적 실행의 종료 데드라인, 시작 및 종료 시간으로 구성된다.
- SvM(Message-triggered Service Method): 분산 환경에서 클라이언트 TMO의 메소드가 보내는 이벤트 전송이나 서비스 요청 메시지에 의해 구동되는 실시간 메소드로, 역시 객체의 멤버 스레드이다. SvM은 네트워크에 투명한 분산 IPC 메시지 수신에 의해 구동되며, 구동 후에는 사전 설정된 데드라인 이전에 실행을 마치도록 스케줄된다.

다음은 이러한 TMO의 특징이다.

- TMO는 분산 컴퓨팅 객체이다. 분산환경에 산재하는 여러 TMO들은 논리적 멀티캐스팅 기반의 채널 방식 메시지 전달에 의해 상호 협력이 가능하다. 논리적 멀티캐스팅이라 함은 네트워크에 투명한 채널 기반 분산 IPC를 말하며, 이러한 기반 위에서 다수의 TMO로 구성되는 실시간 응용은 TMO의 위치 독립성(location independency)을 제공한다.
- TMO는 기존 객체와 달리 SpM과 SvM과 같이 병행 실시간 스레드로 구현되는 동적 멤버들을 갖는다.
- SvM은 SpM의 수행을 방해할 수 없다. 이것은 시간 보장의 개념을 도입한 BCC(Basic Concurrency Constraint) 정책이다. SpM은 SvM 보다 높은

† 연구 내용중 건대와 외대의 연구는 정보통신부 ITRC 지원에 의한 것임.

우선 순위를 가지고 수행된다. 이것은 설계 시 시간 보장의 개념을 도입하기 위해 SpM과 SvM의 시간 선점을 계층화한 것이다.

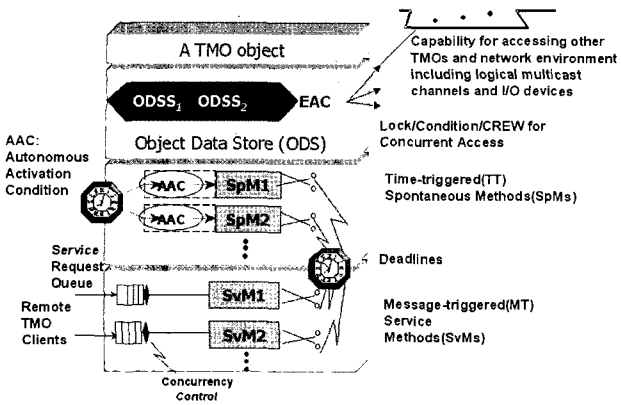

그림 1 분산 실시간 객체 TMO의 구조[1]

2.2 TMO 프로그래밍

TMO 프로그래밍은 현재로선 C++ 기반으로 제공된다. TMO 프로그래밍을 위한 API는 엔진 개발자에(UCI,외대,건대) 따라 약간씩 그 구조를 달리하기 때문에, 현재 그 표준화가 모색되고 있다. 따라서 본고에서는 한국외대에서 개발된 엔진의 API를 기반으로 그 예를 들기로 한다.

다음은 SpM의 전형적 형태이다.

```
SpM_register (AAC constraint, SCHED policy)
while (SpM_wait_invocation ()) { // wait invocation
      do a periodic job within the given deadline;
}
```

위에서 *SpM_register()*는 어떤 운영체제에서의 일반적 스레드를 시간 조건을 갖는 실시간 스레드 즉, SpM으로 변환하는 역할을 하는 함수이다. 시간 조건(AAC class)는 milli-second 단위의 구동 주기, 매 구동 시의 종료 데드라인, 시작 및 종료 구간으로 구성된다. *policy* 인자는 원하는 스케줄러의 형태를 나타내는 것으로 구현에 따라 여러 스케줄링 기법이 제공될 수 있다. 현재로선 EDF(Earliest Deadline First), LLF(Least Laxity First) 및 FIFO들이 제공되는데, 이에 관한 설명은 다음절에서 하기로 한다. *SpM_wait_invocation()* 함수의 호출은 이번 주기의 작업을 마치고 다음 주기에서의 구동을 대기하는 역할을 하며 종료 데드라인은 구동 시작 후, 다음 *SpM_wait_invocation()*까지의 실행에 대해 적용된다.

분산 또는 로컬 IPC 메시지에 의해 구동되는 SvM은 IPC 시스템과 커널의 스케줄러에 의해 관리된다. 다음은 SvM의 전형적인 모습이다.

```
SvM_register (deadline);
alloc_channel (channel_id, channel-attribute);
while( SvM_wait_invocation(channel_id,
  message-pointer) = = TRUE ) {
  do a service job within its given deadline;
}
```

SvM_register()는 일반 스레드를 SvM으로 변환하며, 이때 주어지는 데드라인은 메시지 구동에 의해 깨어나 작업 완료 후, 다음 메시지 대기까지의 최대 시간을 나타낸다. *alloc_channel*() 함수는 해당 SvM이 메시지를 기다리는 분산 채널을 할당하는 것으로, 이 채널을 이용해 메시지를 송수신하는 노드들은 멀티캐스팅 기법의 사용으로 복수 개가 될 수 있다. 클라이언트 TMO가 외부의 SvM에게 서비스 요청을 할 때에는 다음 함수를 사용한다.

```
bool_t send_message(int channel_id,
      Message *pMessage, int mode);
```

다음 〈표 1〉은 TMO 객체의 한 예제이다.

표 1 TMO의 구성 예

```
class TMO_name : public TMO
{
 private :
  void method1(void);
  void method2(void);
  SpM (method1);
  SvM (method2);
  rt_cond Cond_name;
// Condition for sync.
  rt_lock Lock_name;
// Lock for sync.
  time-critical-data;

 public :
  void InitInstance(void);
  :
};
void TMO_name::InitInstance(void)
{
  // SvM, SpM initialization
  SpM_Init (method1);
  SvM_Init (method2, 3);
}
SpM_Body (TMO_name, method1)
void TMO_name::method1(void)
{
  SpM_register (AAC, SCHED_LLF);
  while(SpM_wait_invocation()){
     do something periodic;
     Lock_name.ex_lock();
       update ODS;
     Lock_name.ex_unlock();
  }
  SpM_deregister();
  //returns to a normal thread
}
SvM_Body(TMO_name, method2)
void TMO_name::method2(void)
{
  SvM_register (deadline);
  alloc_channel (channel_id, channel-attribute);
  while(SvM_wait_invocation(
    channel_id, messagep)) {
     do something;
     Lock_name.ex_lock();
       update ODS;
     Lock_name.ex_unlock();
  }
  SvM_deregister();
}
```

위의 예에서 SpM 및 SvM의 클래스 및 몸체 선언과 Lock이나 Condition 등의 동기화 도구들은 C++ 기반의 TMO 프로그래밍 지원을 위한 라이브러리 TMOSL (TMO Support Library)에서 제공되는 매크로 및 기저 객체들로, 이에 관한 자세한 사항은 관련 절에서 기술한다.

3. TMO 지원을 위한 미들웨어 및 커널의 기능

논리적 네트워크의 분산 환경에서 복수 개의 TMO들로 구성된 실시간 응용을 지원하기 위한 미들웨어나 커널 엔진은 TMO의 실행을 위해 다음과 같은 필수적 기능들을 제공하여야 한다.

- SpM의 정시 구동 및 이미 구동된 SpM과 SvM의 데드라인 내 실행을 완료하기 위한 데드라인 스케줄링과 SpM및 SvM의 관리를 위한 API
- SvM의 메시지 기반 구동과 연계된 분산 및 로컬 IPC
- TMO 객체 내의 ODS 접근의 메소드 상호배제 및 동기화 도구
- 분산 클럭 동기화
- 특정 언어(C++ 등) 기반의 TMO 프로그래밍을 위한 변환기

그림 2는 이러한 TMO 엔진의 일반적 구조이다. 그 이외에 추가적으로 제공될 수 있는 도구들에는 다음과 같은 것들이 있을 수 있다.

- TMO 기반 설계를 위한 설계 도구와 스케줄 가능성 분석 도구
- 모니터링 시스템 및 결함 허용 체제를 위한 프레임 워크

본 고에서는 TMO의 실행을 위한 엔진의 기본적 기능 및 그 구현에 대해서만 소개하기로 한다.

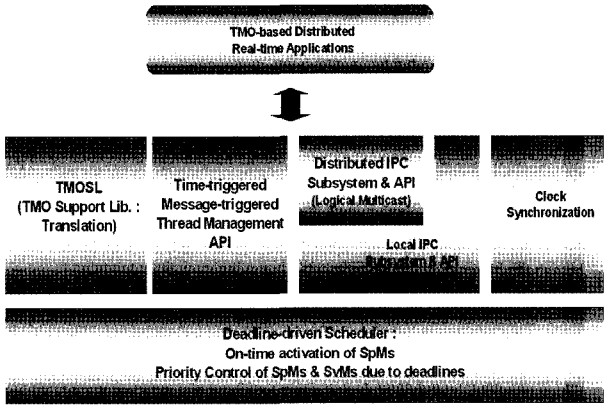

그림 2 TMO 엔진의 구성도

4. 데드라인 스케줄러

SpM과 SvM의 데드라인 기반 실시간 컴퓨팅을 위해서 TMO 엔진은 다음과 같은 기능의 데드라인 스케줄러를 제공하여야 한다.

- 주기의 도래에 의한 SpM의 정시 구동
- 실행 중인 SpM과 SvM들의 잔여 데드라인이나 잔여 수행시간(laxity)에 의한 우선 순위 조정 및 문맥 교환: 스케줄링 정책에 따라 구동 중인 SpM 및 SvM의 우선 순위를 재설정
- 스케줄링 옵션 BCC(basic Concurrency Constraint)의 제공: SvM의 수행이 SpM의 수행을 방해하지 않도록 스케줄하는 것으로 SvM의 선언된 데드라인을 최대 수행시간으로 보고, 이러한 SvM의 수행이 실행 예정인 SpM과 중복되지 않도록 한다.

위에서 스케줄링 정책은 엔진이나 응용의 형태에 따라 여러 가지가 제공될 수 있는데, 경성 실시간 시스템인 경우는 사전 분석이나 시뮬레이션에 의한 SpM의 순차화를 통해 FIFO 기반의 정적 스케줄링을 사용하는 것이 권장된다. 그 이외의 경우에는 선점이 허용되는 EDF나 LLF를 사용하는 것이 보통이다. EDF는 종료 데드라인이 가장 가까운 실시간 스레드를 우선 스케줄링하는 것이고, LLF는 실시간 스레드의 최장수행시간(WCET)을 아는 경우에, 데드라인 대비 잔여 수행 시간이 가장 큰 것을 우선적으로 스케줄하는 기법이다. 일반적인 운영체제는 실시간 스케줄링으로 FIFO와 RR(Round-Robin)만을 제공하는 것이 보통이지만, TMO 엔진의 경우에는 시간 조건의 만족을 위해 EDF나 LLF 등이 제공되어야 한다.

이러한 데드라인 스케줄러는 TMO 엔진이 미들웨어이건, 커널 형태이건, 실시간 스레드들의 감시를 위해 고정밀 클럭에 대한 후킹(hooking)을 바탕으로 주기적으로 동작해야 한다.

4.1 미들웨어 데드라인 스케줄러 구현

어떠한 (윈도우즈나 리눅스) 운영체제 상에서 커널을 수정하지 않고 미들웨어 수준에서 데드라인 스케줄링이 가능하려면, 일단 정밀 클럭에 대한 후킹 기능이 운영체제에서 제공되어야 한다. 이러한 클럭 후킹은 커널 차원의 임계 구역을 파괴하지 않는 수준에서 다른 커널 행위보다 우선하는 것일수록 스케줄링의 정밀도는 높아지게 된다. 윈도우 기반의 미들웨어 엔진인 WTMOS나 TMOSM/NT에서는 윈도우 운영체제의 멀티미디어 타이머 핸들러가 사용되었다. 멀티미디어

타이머는 1/1000초의 정밀도를 가지며 일반적 커널의 행위보다 우선하는 특징을 가진다. 리눅스 기반의 미들웨어인 LTMOS의 경우에는 itimer와 그 시그널 핸들러가 사용되었다. 단 itimer의 실제 정밀도는 일반 리눅스 커널의 1/100초 클럭 틱에 준하게 되므로, 이를 1/1000초 단위 이상으로 올리기 위해서는 커널 자체의 틱을 재설정하고 다시 컴파일하여야 한다. 이러한 커널의 클럭 후킹 서비스를 통한 데드라인 스케줄러의 작동은 우선 순위 조정, 태스크(스레드)의 상태 변환 및 커널에 대한 재스케줄링 요청 등으로 간접적으로 이루어진다. 따라서 응급을 요하는 문맥교환의 경우 커널 내부에 직접적으로 구현한 데드라인 스케줄러 보다는 상대적 지연이 발생하게 된다. 이러한 정밀성의 문제가 있지만 멀티미디어 서비스나 실시간 시뮬레이션과 같은 시간 조건과 동기화를 필요로 하는 연성 실시간 응용의 경우에는 커널을 수정하지 않아 이식성과 호환성에서 우수한 미들웨어 엔진이 선호된다고 할 수 있다.

4.2 커널 내부 데드라인 스케줄러 구현

오픈 소스 커널 내부에의 데드라인 스케줄러의 구현은 일단 대상 커널의 스케줄러 구동 방식과 기능에 따라 달라지게 된다. 리눅스의 경우는 실시간 스레드를 위해 고정 우선 순위 방식의 FIFO와 RR을 제공하고 커널 모드와 사용자 모드를 구분하며, eCos2.0과 같은 경우는 비트맵 방식과 RR 방식의 스케줄러를 선택할 수 있으며 임베디드 커널이므로 커널 모드와 사용자 모드의 구분이 없다. MicroC-OSII와 같은 경우는 간단한 비트맵 스케줄러를 제공하는데 64개의 우선 순위가 있지만 한 우선순위에는 하나의 태스크만이 지정될 수 있고 역시 커널모드와 사용자 모드의 구분은 없다.

커널의 스케줄러은 일반적으로 태스크 블록 시 시스템 호출의 중간 부분에서, 또는 시스템 호출과 인터럽트 처리의 끝 부분에서 재 스케줄링 필요에 의해 호출된다. 클럭 인터럽트는 일반적으로 타임 슬라이스 검사와, 여러 요인에 의한 우선 순위 재조정 작업을 하며 인터럽트 종료 직후 스케줄러를 호출하게 된다. 이러한 스케줄러는 커널에서 일반적으로 스케줄러라는 이름을 사용하지만 사실은 문맥 교환을 의미하며, 실질적인 스케줄링은 커널의 요소요소에서 행해지는 우선순위 변경 및 태스크의 상태변환 시에 적용되는 제반 정책을 의미한다. TMO를 위한 데드라인 스케줄러는 데드라인이나 잔여 수행시간의 감시를 동해 이루어져야 하므로 클럭 인터럽트 처리 과정에 포함되어야 한다. 클럭 인터럽트의 기능 및 처리 과정도 커널에 따라 차이점을 가지는데, 리눅스나 eCos2.0의 경우는 클럭 인터럽트 발생 직후, 인터럽트 불가 모드에서 처음 수행되는 ISR(Interrupt Service Routine)과 인터럽트 가능 모드에서 수행되는 DSR(Deferred Service Routine 또는 Bottom-half)로 나누어 진다. 이와 같은 인터럽트 핸들러의 분리는 인터럽트 처리를 위한 작업이 많은 경우 이를 모두 인터럽트 불가 모드로 처리하면 인터럽트 오버런이 발생할 수 있기 때문이다. 따라서 ISR에서는 시스템의 틱(jiffy)를 수정하는 필수 작업만을 하는 것이 보통이고, 그 이후의 타임 슬라이스 작업이나 기타 클럭 관련 서비스들은 DSR에서 하는 것이 보통이다. 시스템 호출 처리 중 스케줄링 관련 임계 영역에 접근하고 있을 때 클럭 인터럽트가 발생하여 DSR이 수행되면 커널 상호배제의 문제가 발생하므로, 이러한 경우 DSR의 실행은 스케줄러 락(lock)이 해제된 후로 연기된다. 클럭 서비스의 규모가 작은 소형 커널의 경우에는 커널 시스템 호출의 스케줄링 관련 영역은 클럭 인터럽트 불가 모드로 수행되고 이 경우는 추후의 ISR과 DSR의 분리 없이 ISR이 모든 처리를 담당할 수 도 있다. 이와 같은 배경으로 TMO를 위한 데드라인 스케줄러는 DSR에 포함되는 것이 일반적이다. 그림 3은 TMO-Linux 및 TMO-eCos의 데드라인 스케줄러 구현 내용을 나타낸 것이다.

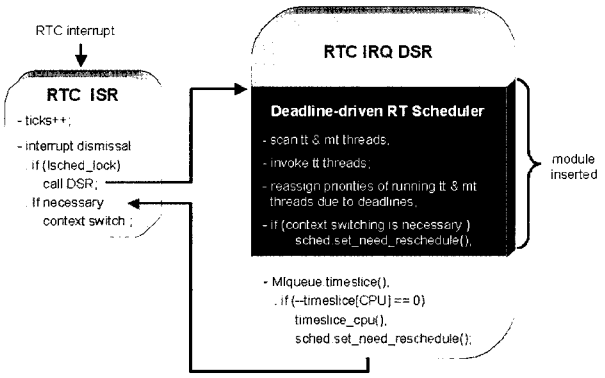

그림 3 커널 데드라인 스케줄러의 구현

4.3 SpM 및 SvM의 관리

데드라인 스케줄링과 SpM 및 SvM의 관리를 위해서는 각 실시간 태스크에 대한 자료 구조체가 필요하다. 이러한 태스크 자료구조체에는 태스크의 형태 및 각종 시간 조건 관련 정보가 포함되어 SpM이나 SvM의 등록 시 초기화되고, 데드라인 스케줄러에 의해 지속적으로 그 변경이 관리된다.(표 2) 태스크의 WCET를 모르는 경우에 적응적 LLF스케줄링을 제공하기 위해서 각 엔진은 메소드의 매번 실행의 최장시간을 기

록하여 사용할 수 도 있다. 이러한 실시간 태스크의 자료 구조체는 미들웨어의 경우에는 사용자 공간의 리스트로 만들고, 항상 현 실행 태스크의 자료 구조체를 포인트하는 current 광역변수는 스레드의 문맥교환이 일어날 때마다 같이 문맥교환에 포함되도록 하는 스레드 로컬 데이터 서비스를 활용하여야 한다. 커널의 경우 이러한 자료구조체는 기존 태스크 구조체를 확장하여 구현한다.

표 2 SpM/SvM 관리를 위한 태스크 자료구조체

. type (time-triggered or message-triggered)
. period (SpM)
. time_left_to_invocation (SpM)
. deadline (SpM/SvM)
. time_left_to_dedline (SpM/SvM)
. current priority (SpM/SvM)
. historic WCET (LLF, SpM/SvM)

5. 논리적 멀티캐스팅 기반의 분산 및 로컬 IPC

TMO를 위한 IPC 시스템은 채널 기반 논리적 멀티캐스팅을 제공하며, 로컬 IPC와 분산 IPC에 대해 같은 API 및 위치독립성(location independency)을 제공하여야 한다. 단 이와 같은 IPC 시스템은 임베디드 시스템에서의 유연한 구성을 위해 그림 4와 같이 로컬 IPC 및 분산 IPC의 2개 계층 구조로 구성하는 것이 바람직하다.

먼저 로컬(intra-node) IPC는 관련 자료 구조와 이벤트 메시지 송수신을 위한 API로 구성된다. 관련 자료구조로는 TMO 응용에 의해 할당된 각 채널 별로 배달된 메시지를 저장하는 메시지 큐와 배달될 메시지를 대기하는 SvM의 태스크 큐로 구성된다. SvM_wait_invacation()에 의해 대기하던 SvM은 메시지 배달과 동시에 스케줄링 큐로 복귀되고 이때부터 데드라인 스케줄링이 적용된다. 이러한 자료 구조와 관련 API의 기본 내용은 미들웨어와 커널의 차이가 없으나, 커널의 경우 API는 확장된 시스템 호출들로 구현되어야 한다.

분산 IPC는 구현된 로컬 IPC를 기반으로 별도의 사용자-모드 라이브러리로 구축되는 것이 구성의 유연성을 위해 바람직하다. 분산 IPC는 내부적으로 기반 프로토콜에 독립적인 부분과 프로토콜에 종속적인 2개의 모듈로 구성된다. 프로토콜에 독립적인 계층은 프로토콜 계층으로부터 외부 메시지를 수신하여 로컬 IPC 계층으로 전달하는 내부 실시간 태스크와, 송출 메시지를 외부로 멀티캐스팅 의뢰하는 send_message() API로 구성된다. 그 이외에 분산 컴퓨팅의 시작, 노드의 참가 및 탈퇴, 종료 등의 취급하는 노드 관리자도 포함된다. 프로토콜 종속 계층은 하위 프로토콜과 프로토콜 독립 계층을 연결하는 역할을 하여 네트워크에 투명한 분산 IPC를 지원한다. TMO의 분산 IPC에 사용되는 기반 프로토콜은 LAN 환경에서 기본적으로 다음과 같은 특성을 지향한다.

- 메시지 전달의 시간 보장성
- 그룹 커뮤니케이션으로서의 통신 신뢰성

메시지 전달의 시간 보장성은 실시간 분산 컴퓨팅에서 데드라인 스케줄링의 제공과 함께 매우 핵심적인 요구 사항이라 할 수 있다. 현재까지 LAN 환경에서 TMO 엔진의 분산 IPC로 사용된 일반적인 프로토콜은 UDP-multicast인데 LAN 버스 및 UDP의 특성상, 신뢰성과 시간 보장성을 제공하는 것은 어려운 일이다. 이러한 문제점을 극복하기 위해서는 기본적으로 각 노드에 전송 시간을 고정 할당하는 시분할(time-triggered) 개념의 프로토콜이 필요한데, 유럽에서는 Kopetz 교수팀이 개발한 TTP(Time-Triggered Protocol)가 많이 사용되고 있으며, 국내에서는 IEEE1394 기반의 분산 IPC가 외대 연구진에 의해 개발되었다[11]. 이들 프로토콜은 고속 통신과 버스 접근에 대한 노드 별 시간 대역을 지정함으로써 분산 통신의 불확실성을 제거한 것이다. 다만 종단 노드간 그룹 통신의 신뢰성은 상위 프로토콜로서 추가 개발이 이루어져야 할 부분이다.

위와 같은 기능 이외에도 분산 IPC 라이브러리는 분산 컴퓨팅이 필수적인 분산 클럭의 동기화를 제공하여야 한다. 분산 클럭 동기화는 분산 노드들이 하나의 글로벌 클럭을 갖도록 하여 컴퓨팅의 동기화, 메시지의 신뢰성 확보 등에 사용된다. 분산 클럭 동기화는 노드 각각이 외부의 정확한 타임 자원을 활용하는 방식과, 하나의 매스터 노드(타임 서버)가 분산 노드들의 클럭을 메시지 기반으로 관리하는 여러 가지 알고리즘과 구현 방식이 있다. 후자의 방식은 정확한 시간의 전달에 통신의 불확실성이 포함되므로 여러 가지 현실적 어려움을 가지게 된다. 근자에는 GPS에 의한 시간 수신 및 동기화 기법이 많이 사용되고 있다.

그 외에 UCI 김광회 교수 팀의 분산 IPC는 다음과 같은 추가적 서비스를 제공하고 있다.

- RMMC(Real-time Multicast & Memory-replication Channel): ODS의 일부를 논리적 멀티캐스트 채널에 연결하여 분산 노드에서 자료를 공유하는 방식으로 내부적으로 분산 IPC 에 의한 동기화로 자료의 일관성을 제공한다. 이는

분산 공유 자료 객체 개념을 채널에 접목한 방식의 서비스이다.

- Official release time: 클라이언트 메소드가 SvM에 서비스 요청 메시지를 전송할 때, 메시지 서비스의 시간을 지정해 보내는 방식으로 SvM에 의한 스케줄링의 불확실성을 없애기 위한 기법이다.

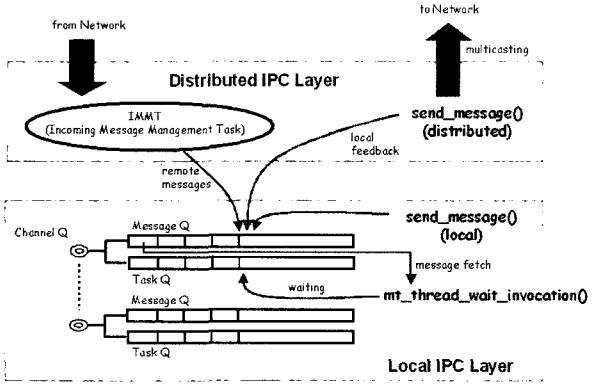

그림 4 로컬/분산 IPC 구조

6. TMO 지원 라이브러리

미들웨어나 커널에서 제공되는 데드라인 스케줄링, SpM/SvM 관련 API 및 분산 IPC 만을 사용하여도 time-triggered/message-triggered 스레드에 의한 분산 실시간 프로그래밍은 가능하지만, 이러한 개념들은 TMO라는 객체 기반 프로그래밍에 적용하려면 C++와 같은 객체 지원 언어와 TMO의 결합을 지원하는 변환기가 필요하다. 궁극적으로 이러한 변환기는 컴파일러에 의해 구현되는 것이 바람직하지만, 객체 메소드의 스레드 변환과 같은 기본적 기능은 매크로와 기저 클래스 라이브러리의 사용으로도 가능한데, 이를 일반적으로 TMOSL(TMO Support Library)이라 한다. TMOSL은 다음과 같은 기능을 제공한다.

- SpM, SvM과 TMO 들을 선언하고 초기화하는 매크로들과 기저 클래스의 제공
- TMO의 객체 멤버 함수를 SpM이나 SvM 스레드로 변환
- SvM 기반의 다중 스레드 워커 모델(worker model) 지원
- 객체 지향 기반의 동기화 도구인 CREW(Concurrent Read Exclusive Write) 모니터의 제공

다음은 앞서 예로든 표 1의 TMO 객체에서 사용된 TMOSL의 구조체들에 대한 설명이다.

- *TMO*는 서브클래스 SpM 과 SvM 을 포함하는 사용자 TMO를 위한 기저 클래스로 관련 자료구조와 API를 제공한다.
- *SpM*과 *SvM* 은 선언된 객체 멤버함수를 멤버 스레드로 선언하기 위한 매크로이다. 이 매크로들은 SpM과 SvM의 서브 클래스들을 생성한다.
- *SpM_Init*과 *SvM_Init* 매크로들은 SpM과 SvM으로 선언된 함수를 스레드로 생성한다.
- *SpM_Body*과 *SvM_Body*는 SpM과 SvM의 함수 내부를 선언하는 매크로들이다.
- *rt_lock*과 *rt_cond*는 CREW 모니터를 생성을 위한 락과 컨디션 구조체들이다. 이들은 배타접근을 위한 *ex_lock()*, *ex_unlock()*, 공유접근을 위한 *sh_lock()*, *sh_unlock()*, 동기화를 위한 *cond_wakeup()*, *cond_wait()* 등과 같은 CREW API를 지원한다. 단 이러한 동기화 도구는 미들웨어나 커널에서 직접 제공하도록 구현하는 것도 가능하다.

예제에서 *SvM_Init*(SvM_name, 3)는 SvM 스레드 풀이 서비스하는 다중 스레드 서버 모델을 구현하기 위해 사용되는 매크로로, 한 채널에서 대기하는 동일한 SvM을 주어진 개수만큼 생성한다.

7. 미들웨어 및 커널의 비교

수 년 동안 개발된 여러 개의 분산 실시간 컴퓨팅을 위한 TMO 엔진들은 커널 형태이건 미들웨어 형태이건 간에 각각의 OS 플랫폼에 따른 장점을 가지고 있다. 표 3은 3개의 오픈 소스 기반의 TMO 엔진들을 사용되는 영역이나 몇 가지 특징적 측면에서 비교한 것이다.

LTMOS는 다른 2개의 엔진에 비해 상대적으로 높은 스케줄링 지연시간을 보인다. 이는 LTMOS가 소프트 클럭 핸들러에 의해 구현된 사용자 모드의 실시간 스케줄러를 사용하기 때문이다. 이러한 단점에도 불구하고 LTMOS는 멀티미디어 서비스나 실시간 시뮬레이션과 같은 연성 실시간 응용에 있어서 높은 이식성과 연동성에 의한 장점을 가진다. 한편, TMO-eCos는 많은 수의 SpM을 가지고 데드라인 위반 테스트를 수행하였을 때, TMO-Linux에 비해 높은 스케줄링 정확도를 보여주었다. 그 이유는 TMO-eCos의 작은 크기와 커널-사용자 모드 간의 인수 전달에 따른 오버헤드가 없기 때문이다.

그림 5는 미들웨어인 LTMOS와 TMO를 지원하는 커널인 TMO-Linux 그리고 TMO-eCos 상에서 SpM을 1개부터 50개까지 동시에 수행시켰을 때 데드라인 위반 테스트 측정결과를 보여주고 있다. 이 실험에서 각 SpM의 주기, 수행시간 및 데드라인은 스케줄

링 오버헤드가 전무한 경우에 데드라인 위반이 없도록 설정되었다. (모든 SpM의 데드라인과 주기값은 (메소드의 수 * 실행시간)으로 설정됨.)

TMO-eCos는 소형 임베디드 장치를 사용하는 제어 시스템에 적용될 때 모듈화 구성이나 용량면에서 장점을 가진다. 한편 TMO-Linux의 경우에는 대규모 분산 시스템에서의 실시간 컴퓨팅 노드로 사용하는 것이 권장되는데, 그 이유는 리눅스의 대중성과 TMO의 시간 특성을 통합할 수 있기 때문이다. 물론, 모든 TMO 엔진들이 동일한 분산 IPC 인터페이스를 제공하므로 이종 TMO 엔진들에 의한 분산 시스템 구성도 가능하다.

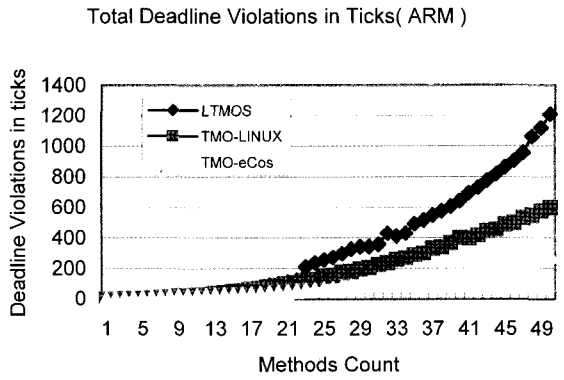

그림 5 데드라인 위반 비교 (ARM-아사벳 보드)

8. 결론 및 향후 과제

실시간 응용의 개발에 있어, 수행 시간 지연이 최소화되는 RTOS를 제공하는 것과 함께, 실시간 응용을 구성하는 컴퓨팅 요소의 시간적인 행동을 쉽게 명세할 수 있는 설계시점에서의 프레임워크를 제공하는 것은 매우 중요한 필수 요소로 부각되고 있다. 이러한 환경에서 TMO 모델은 설계 시점부터의 시간 보장성을 위한 새로운 실시간 프로그래밍 패러다임을 지향하며, 미들웨어 엔진과 함께, TMO-Linux와 TMO-eCos는 이러한 패러다임과 임베디드 시스템 환경의 결합을 위한 새로운 TMO 커널이라 할 수 있다. TMO-Linux가 실시간 시뮬레이션이나 자동 로봇 제어 및 항공 로봇제어 등에서 성공적으로 사용되고 있고, TMO-eCos는 초소형 커널로서 임베디드 기반의 실시간 제어 시스템에 활용이 시작되고 있다. 이러한 연구의 후속 작업으로 센서 노드를 위한 나노급(nano) TMO 커널이 기획되고 있다. 이와 함께, 여러 엔진의 API 표준화, 신뢰성 있는 그룹 통신 등의 개발이 진행되고 있다.

참고문헌

[1] Kim, K.H. and Kopetz, H., "A Real-Time Object Model RTO.k and an Experimental Investigation of Its Potentials", Proc. 18th IEEE Computer Software & Applications Conference, pp.392-402, November 1994.

[2] Kim, J.G., Kim, M.H., Min, B.J., and Im, D.B., "A Soft Real-Time TMO Platform - WTMOS-and Implementation Techniques", Proc. 1st IEEE International Symposium on Object-oriented Real-time Distributed Computing, pp.256-264, April 1997.

[3] Kim, J.G., et al., "Modeling of Multimedia Services Using the TMO Model," Int. Journal of Computer Systems Science and Engineering, Vol. 13, No. 3, May 1998.

[4] Kim, J.G. and Cho, S.Y., "LTMOS: An Execution engine for TMO-Based Real-Time Distributed Objects", Proc. PDPTA'00 Vol. V, pp 2713-2718, Las Vegas, June 2000.

[5] Kim, M.H., et al., "Time-triggered Message-triggered Object Modeling of a Distributed Real-time Control Application for its

표 3 TMO 엔진들의 특성 비교

	LTMOS	TMO-Linux	TMO-eCos
Type	Middleware	Kernel	Kernel
App. mode	User mode	User mode	Kernel mode
Compatibility	Binary compatible	Source patch	Source patch
Portability	Lots of H/W platforms, devices	Lots of H/W platforms, devices	Various but Restricted
Timing Accuracy	Medium to high	High	Higher
Size	Linux + 40KB	600KB ~ 1MB	30KB ~ 470KB
App. domain	Multimedia, Real-time simulation, Soft rt. App.	Networked control, Embedded control, CE devices	Small embedded devices, CE devices, Single-TMO machine

Real-time Simulation, Proc. 20th IEEE Computer Software & Application Conference, pp.549-556, October 2000.
[6] Kim, J. G., et al, " TMO-Linux: A Linux-based Real-time Operating System Supporting Execution of TMO's," Proc. IEEE Int'l Symposium, ISORC2002, Washington DC, Apr. 28, 2002.
[7] Jo, E. H., Kim, M. H., Kim, J. G., "Framework for Development of Multimedia Applications Based on the TMO Structuring Scheme," Proc. IEEE Int'l Symposium, ISORC2003. Hakodate, Hokkaido, Japan, May 14-16, 2003.
[8] Kim, J. G., Kim, M. H., et al, "A Concurrent Event-Driven Simulation Technique for the ROK Army's War Game Model using TMOs," Proc. The 7th World Conference on Integrated Design & Process Technology, Austin, Texas, Dec. 3, 2003.
[9] Kim, J.G., et. al, "TMO-eCos: An eCos-based Real-time Micro Operating System Supporting Execution of TMO's,", ISORC2005, 2005, 4, Seattle.
[10] Kim, K, H, "Oject-Oriented Real-Time Distributed Programming and Support Middleware," Proc. ICPADS 2000, pp.10-20, Iwate, Japan, 2000.
[11] Kim, J.G., et. al, "An IEEE1394-based Real-Time Distributed IPC System for Collaborating TMO's", ISORC2006, 2006, 4, Kyungju, Korea.

김 정 국

1977 서울대학교 계산통계학과(학사)
1979 KAIST 전산학과(석사)
1986 KAIST 전산학과(박사)
1983~현재 한국외국어대학교 컴퓨터공학과 교수
관심분야 : 분산 실시간 시스템, 임베디드 시스템
E-mail : jgkim@hufs.ac.kr

김 문 희

1979 서울대학교 전기공학과(학사)
1981 서울대학교 전기공학과(석사)
1985 Univ. of South Florida, Computer Science and Engineering, MSCS
1991 Univ. of California, Berkeley, Computer Science, Ph.D
1991~현재 건국대학교 컴퓨터공학부 교수
관심분야 : 분산 실시간 시스템, 임베디드 시스템, 결함허용 시스템, 소프트웨어공학
E-mail : mhkim@konkuk.ac.kr