임베디드 시스템을 위한 동적 CORBA 특성 측정

이경우°, 이인환
한양대학교 전자통신컴퓨터공학과
kwlee@csl.hanyang.co.kr°, ihlee@hanyang.ac.kr

Measuring Dynamic CORBA Characteristics for Embedded System

Kyoungwoo Lee°, Inhwan Lee
Dept. of Electronics and Computer Engineering, Hanyang University

요 약

최근 임베디드 분산 객체 응용 기술로 CORBA 규격이 사용되고 있다. CORBA규격은 엔터프라이즈 응용을 위해서 만들어진 규격이지만, 임베디드 시스템 환경에 적합하게 수정, 이식을 하고 있다. 그러나 주로 예측을 통한 CORBA의 동적인 기능 삭제와 CORBA 서비스를 최소화 한 메모리 사용량 제한에 중점을 둔 규격을 만들어 왔다. 실제 표준 CORBA 규격의 임베디드 시스템에 이식을 통한 실험적인 데이터를 근거 설정된 규격 설정이 필요하다. 따라서 이 논문에서는 표준 규격의 CORBA를 실제 임베디드 시스템에 이식을 하고, 동적인 응용프로그램을 실행함으로써 임베디드 시스템에서 발생 할 수 있는 특징을 조사 해 보고자 한다. 이러한 특징을 근거로 임베디드 시스템 CORBA 규격 설정에 근거가 될 수 있을 것이다.

1. 서 론

최근의 정보통신 기술 산업의 발전과 통신 네트워크의 발달로 임베디드 시스템의 중요성이 더 높아지고 있다. 또한 임베디드 시스템 관련한 하드웨어와 소프트웨어 관련 기술이 다양해지고, 빠른 속도로 발전하고 있다. 따라서 다양한 목적의 임베디드 시스템이 공존하게 되었고, 이 시스템들을 통합하려는 새로운 요구가 생겨나기 시작했다. 임베디드은 시스템은 특성상 제한된 기능을 가지고 있기 때문에, 다양한 임베디드 시스템의 통합을 통해 새로운 기능의 확장과 다양한 서비스를 제공 받을 수 있다. 이를 위해 이기종간 상호 연동을 통한 새로운 서비스를 제공할 수 있는 미들웨어 기반의 개방형 구조가 필요하게 되었다.

CORBA(Common Object Requset Broker Architecture)는 하드웨어와 소프트웨어에 독립적인 객체 지향언어를 기반으로 하는 미들웨어 규격이다[1]. CORBA 규격은 현재 800개 이상의 산업체가 참여하고 있다. CORBA는 구현물이 아닌 규격을 명시 하였기 때문에, 다양한 프로그래밍 언어로 구현가능하다. 특히 임베디드 시스템에 적합한 프로그램 구조와 언어를 선택 다양한 구현을 할 수 있다.

또한, 다른 CORBA 규격으로 임베디드 시스템을 위한 minimumCORBA 규격과 실시간 시스템을 위한 Real-Time CORBA가 있다[2][3][4]. minimumCORBA 규격은 제한된 자원을 가진 임베디드용 응용 프로그램을 위한 CORBA 축소 버전이다. 이 규격은 동적 CORBA 응용을 위한 동적 호출 인터페이스(Dynamic Invocation Interface, DII), 동적 스켈레톤 인터페이스(Dynamic Skeleton Interface, DSI)와 동적 any(Dynamic any)부분이 자원의 효율성의 이유로 생략되었다. 이와 함께 ORB(Object Request Broker)와 POA(Portable Object Adapter의 동적 CORBA 지원을 위한 오퍼레이션이 생략이 되었다. 이 동적 기능은 호출할 객체와 메소드가 컴파일하는 시점이 아니라 프로그램 실행 시에 결정된다. 이런 동적 기능과 달리 컴파일 시 객체와 메소드가 결정되어 있는 정적 CORBA 기능이 있다. 즉, minimumCORBA는 정적 기능을 지원함으로써 임베디드 시스템에서 발생할 수 있는 메모리와 성능의 한계를 최소화하는 규격이다. 하지만 고성능 임베디드 시스템에서의 다양한 응용을 위해서는 정적인 기능 이 외에 동적인 기능이 필요하다. 더욱이 임베디스 시스템의 성능과 메모리등 관련 하드웨어가 발전되고 있기 때문에 제한적인 기능을 제공하는 minimumCORBA 규격은 사용자의 요구를 만족 할 수 없다. 다양한 임베디드 시스템의 통합을 위한 CORAB의 기능들은 실험을 통한 결과를 바탕으로 CORBA의 기능을 설정해야한다.

따라서, 이 논문에서는 임베디드 시스템 환경에 표준 CORBA 규격을 이식하고, 위에서 언급한 동적 CORBA기능인 DII와 DSI를 정적 CORBA 기능과 비교 측정 한다. 동일 동작을 하는 응용 프로그램을 설정 하고, 실제 임베디드 시스템에서의 메모리 사용량과 성능을 측정 시스템에 응용프로그램을 측정함으로써 실제 임베디드 시스템에서 발생할 수 있는 특징들을 알아보고자 한다. 측정 데이터를 기반으로 다양한 임베디드 시스템에서의 CORBA 기능을 선택 사용하는 근거가 될 것이다.

2. 관련 연구

본 장에서는 CORBA의 기능 중 기본적인 정적 CORBA 기능과 동적 CORBA인 DII, DSI, 동적 any에 대해서 살펴보고, 실제 임베디드 시스템에서 동적인 CORBA기능이 시스템의 메모리와 성능에 영향을 주는지 , 간단한 CORBA 응용프로그램 측정을 통해 분석 할 것이다.

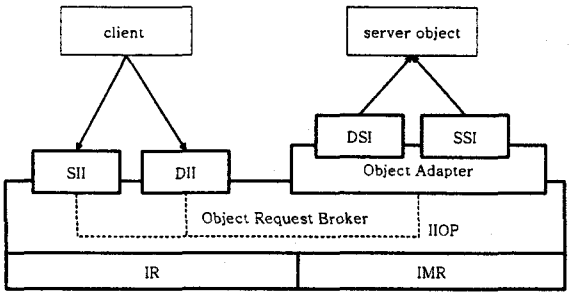

그림 1 CORBA 호출 구조

2.1 CORBA

CORBA는 분산 객체들 간의 통신을 가능하게 하기 위하여 OMG(Object Management Group)에서 제안한 공개
시스템 규격이다. CORBA는 호출 인터페이스(Invocation Inteface), ORB(Object Request Broker), OA(Object Adatper), SI(Skeleton Interface)의 구조로 되어 있다. 특히 CORBA의 핵심 요소인 ORB는 클라이언트/서버 시스템에서 객체들 간의 관계를 투명하게 해준다. 클라이언트 프로그램은 서버 객체의 위치, 운영체제, 통신 프로토콜, 구현 언어에 대한 정보 없이 서버 객체의 메소드를 호출할 수 있다. 원격 객체를 호출하기 위해서 클라이언트 객체가 제공하는 인터페이스에 대해서 알고 있어야 한다. 그림 1에서 동적 호출과 정적호출 2가지 방법으로 메소드를 호출 할 수 있다.

2.2 정적 CORBA

정적 CORBA는 컴파일시 고정된 객체의 인터페이스를 이용하여 메서드에 대해 요청하고, 요청을 받아들이는 기능이다. 인터페이스 정의 언어를 통해 컴파일 시에 언어 명세적인 스텁과 스켈레턴을 만들어 낸다. 하지만 사용되는 스텁, 스켈레톤 코드는 IDL 인터페이스가 변경될 때 마다 재 컴파일이 되어야 한고, 응용 프로그램에 스텁과 스켈레톤코드로 인한 정적 메모리 사용율이 동적 CORBA보다 더 증가한다.

스텁은 클라이언트 함수로서 클라이언트 프로그램 대신 ORB에 요청을 보낸다. 이를 정적호출 인터페이스(Static invocation interface,SII)라고 한다. 반대로 ORB로부터 응답을 받는 스텁 루틴을 서버측 구현객체마다 제공하고 있다. 이를 정적 스켈레턴 인터페이스 (Static skeleton interface,SSI)라고 한다. 스켈레톤은 클라이언트 측의 요청을 받아들이고 구현객체에 포함되는 메서드를 동작시키기 위한 루틴을 가지고 있다.

2.3 동적 CORBA

고정된 인터페이스를 사용하는 정적 CORBA 기능과는 달리 동적 CORBA는 프로그램 실행시 객체의 인터페이스에 관한 서술을 통해서 객체의 메서드에 요구를 하고 요청을 받아들이는 기능이다. 하지만 이 방법은 프로그램을 복잡하게 만들고, 특히 동적으로 정보를 얻기 때문에 특히 임베디드 시스템에서의 성능에 영향을 미친다.

클라이언트부분의 동적호출 인터페이스는 프로그램 실행 시에 프로그램에서 객체 참조, 메소드 이름, 전달될 매개 변수를 서술하여 동적 호출을 가능하게 한다. 정적 호출에 사용하는 스텁코드에 대해서 유사한 프로그램이 필요하다. 서버는 새로운 인터페이스를 준비되는 시점에 제공하고 클라이언트는 동적호출로 실행 시에 새로운 인터페이스를 사용하게 된다. 동적 스켈레톤 인터페이스는 인터페이스 정보를 정적으로 갖고 있지 않아도 임의의 인터페이스 오브젝트에 대한 호출을 처리할 수 있는 구현객체를 작성할 수 있다. 임의의 오브젝트에 대한 호출을 위한 응용프로그램에 사용될 수 있다. 동적 any 형은 컴파일 시점에 고정되지 않은 자료형을 전송하거나 수신할 수 있게 한다.

2.4 정적, 동적호출 성능 평가

CORBA 동적 기능은 대부분 사전에 객체나 메서드에 대해서 정해지지 않은 인터페이스를 지원하는 기능이다. 객체를 호출 하기 위해서는 ORB의 공통의 데이타 구조에 정보를 삽입, 추출을 통해 호출이 일어난다. 이 방법은 ORB에 직접 호출을 하게 되고, ORB에서는 이러한 호출을 지원하기 위해 추가적인 동적 호출 오퍼레이션이 필요하게 되어 실행시간이 증가 될 것이다. minimumCORBA규격에서는 이러한 동적 기능이 메모리 문제로 제거되었다. 실제 임베디드 시스템에서 동적인 기능과 정적인 기능의 메모리 사용량과 호출에 걸리는 시간을 비교함으로써 동적 기능의 성능을 평가 할 수 있을 것이다.

표준 CORBA 규격을 임베디드 플랫폼에 맞게 이식을 하고, 메모리와 성능을 평가하기 위한 프로파일링을 하게 될 것이다. 테스트를 할 응용 프로그램으로 동일한 기능의 서버 클라이언트 프로그램을 각각 정적 기능과 동적 기능을 사용하여 실행시 임베디드 시시템에서 발생하는 모든 부분에 대한 정보를 가지고 성능 평가를 하게 된다.

3. 평가 환경 및 측정 결과

본 장에서는 측정 환경 구축과 측정에 필요한 도구와 설정에 대해서 살펴보고, 측정 결과에 대해서 분석 하고자 한다. 정적 호출에 필요한 메모리와 성능이 어떻게 차이가 나는지 알아 볼 것이다.

3.1 측정 방법

표준 CORBA를 임베디드 시스템에 맞게 컴파일 하고, 동일한 기능을 가지고 있는 프로그램을 정적 기능과 동적 기능을 사용한다. 타겟 시스템에 각각 서버와 클라이언트 응용프로그램을 실행시키고, 동적메모리, 동적 메모리 사용율과 실행 시간을 측정한다.

3.2 하드웨어 구성

타겟 시스템으로는 고성능 임베디드 시스템으로 인텔사의 pxa255-400Mhz와 32MB 플래쉬메모리, 128MB SRAM의 주기억 장치를 가지고 있는 시스템을 선택했다. 성능을 측정을 위해서 고성능의 임베디드 시스템을 선택하였다. 타겟 시스템과 호스트는 LAN으로 연결이 되어 있다.

3.3 소프트웨어 구성

타겟 시스템의 운영체제는 임베디드 리눅스 커널 2.4.18을 사용했다. 사용된 CORBA 구현물로는 MICO 2.3.12를

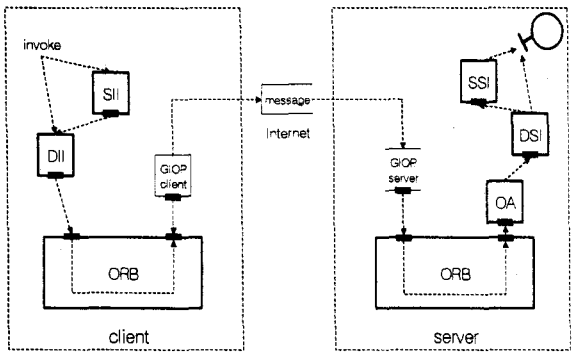

그림 2 MICO 호출 구조

사용하였다. PC와 임베디드 시스템을 서버와 클라이언트로 구성하고, 각각 호출을 한다.

MICO는 'MICO IS COrba의 약자이다. MICO 프로젝트는 GPL에 따라 자유롭게 사용할 수 있는 완벽한 CORBA 2.3을 제공한다. 표준 유닉스 API를 이용하여 C++로 개발되었다. 표준 API를 사용함으로써 다양한 시스템에 이식하기에 유용하다. CORBA를 다양한 임베디드 시스템에 이식 할 수 있다[5].이 CORBA는 라이브러리로 존재하게 된다. 크로스 컴파일을 통해 임베디스 시스템에 맞게 운영체제에 맞게 이식을 했다. 특히 마이크로 커널로 만들어져 있어 최소한의 ORB만을 두고 각각의 기능을 외부 인터페이스로부터 기능을 제공 받고 있다. 따라서 CORBA의 기능을 측정하는데 있어서 각 모듈의 측정에 용이하다. MICO의 호출 구조는 그림2과 같이 서버와 클라이언트 호출이 이루어진다.

```
//IDL 코드
interface Calc {                          ------------(1)
   long add (in long x, in long y);
}
//서버 구현 클래스 코드
class Calc_impl : virtual public POA_Calc{
public:
   long add (CORBA::Long x ,CORBA::Long y){
                    return x+y;  }
}
// 클라이언트 구현 코드
int main (int argc, char *argv[]){
   CORBA::ORB_var orb =
                CORBA::ORB_init (argc, argv);
     CORBA::Object_var obj =
           orb->string_to_object (ior.c_str());
      Calc_var calc = Calc::_narrow (obj);
     calc->add (12,14)    return 0;
     }
```

그림 3 정적 호출 예제 프로그램

FunctionCheck를 사용하여 측정 한다[7]. FunctionCheck는 라이브러리로 존재하고 실행 시 모든 메서드의 메모리 사용량과 일반적인 프로파일정보를 제공해 준다. 기존의 gcc에서 지원하는 프로파이링 정보 보다는 좀더 세밀한 정보를 얻을 수 있기 때문에 선택을 했다. MICO 라이브러리를 컴파일 할 때 이 라이브러리와 동작을 할 수 있게 크로스 컴파일을 했다. 측정을 하기 위해서는 FucntionCheck 라이브러리가 임베디드 리눅스 용으로 크로스 컴파일되어, 시스템 라이브러리로 존재해야 한다.

측정 프로그램은 클라이언트가 덧셈을 서버에게 요구를 하면 서버는 계산된 결과를 클라이언트 쪽으로 전달해 주는 프로그램이다. 이 프로그램의 입력 값은 long형으로 고정했다. 각각 타겟 시스템을 클라이언트와 서버 프로그램으로 설정되면 이에 대응 하는 프로그램을 호스트 시스템에서 실행시킨다. 그림 3의 정적 호출 예제 프로그램 (1)에서 객체 인터페이스를 정의 하고 있다. 정의된 인터페이스로 클라이언트 프로그램과 서버 프로그램을 작성 하지만 그림 4의 동적 호출 부분에는 이 부분이 생략 되어 있다.

```
//서버 구현 클래스 코드
class CalcImpl : public DynamicImplementation
{
public:
   CalcImpl(ORB_ptr orb, POA_ptr poa);
   virtual void invoke(ServerRequest_ptr svreq);
   virtual Long add(Long x, Long y);
   virtual char* _primary_interface(
     const PortableServer::ObjectId&
           oid, PortableServer::POA_ptr poa);
   virtual PortableServer::POA_ptr _default_POA();
  private:
    CORBA::ORB_var orb_;
    PortableServer::POA_var poa_;
}
//클라이언트 구현 코드
int main (int argc, char *argv[]){
     CORBA::ORB_ptr orb = CORBA::ORB_init
     (argc, argv, "mico-local-orb");
     ifstream in("diiserver.ior");
     string ref = "";
     in >> ref;
     CORBA::Object_var obj =
     orb->string_to_object(ref.c_str());
     req->contexts()->add ("a*");

     req->add_in_arg() <<= (CORBA::Long)12;
     req->add_in_arg() <<= (CORBA::Long)14;
     req->add_out_arg ();
     req->result()->value()->set_type
        req->invoke ();
     }
```

그림 4 동적 호출 예제 프로그램

표 1 동적, 정적 클라이언트 프로그램 비교

영역 \ 타입	정적 클라이언트	동적 클라이언트
할당된 스택영역	16 kB	16 kB
실행 코드영역	28 kB	12 kB
최대 힙 사용량	61.855 kB	63.765 kB
총 힙 사용량	65.572 kB	67.728 kB
실행 시간	0.42401 sec	0.46503 sec

표 2 동적, 정적 서버 프로그램 비교

영역 \ 타입	정적 서버	동적 서버
할당된 스택영역	16 kB	16 kB
실행 코드영역	28 kB	16 kB
최대 힙 사용량	57.498 kB	67.204 kB
총 힙 사용량	61.564 kB	71.416 kB

3.3 결과 및 분석

타겟보드에서 동적, 정적 기능의 클라이언트, 서버 프로그램을 실행시킨 결과는 다음과 같다.

클라이언트 프로그램은 정적호출에 쓰이는 스텁 코드 때문에 동적 호출에 비해 35%로 정적 메모리가 증가했고, 동적 호출은 정적 호출에 비해 동적 메모리 3%증가를 보였다. 실행시간은 9%증가가 되었다. 동적 클라이언트 응용프로그램이 동적메모리 부분에서의 사용율이 높지만 정적 메모리는 정적 클라이언트 응용프로그램의 스텁 코드로 인하여 사용율은 상대적으로 낮다. 실제 정적 클라이언트 응용프로그램의 크기도 50%이상 증가를 보였다. 서버 프로그램에서도 역시 클라이언트 프로그램과 마찬가지로 정적 서버가 동적서버가 정적 서버에 비해 스켈레턴 코드의 영향으로 정적인 메모리 정적 메모리가 75%정도 증가 된다. 동적 메모리는 정적 서버보다 동적 서버의 사용율이 17%증가를 보이고 있다. 응용프로그램 크기 역시 50%이상 크게 되었다.

위 결과에서 정적 호출은 동적 호출에 비해 정적메모리를 더 많이 소모하고 있다. 동적 호출이 정적호출보다 동적메모리 사용량이 많기는 하지만 실제 프로그램 실행 시 정적 호출이 더 많은 메모리를 사용하고 있었다. 전체적인 메모리 사용율은 동적 호출의 경우 동적메모리 사용율 보다 20% 증가량을 보이고 있다. 실제 간단한 호출 프로그램에서는 정적 기능이 동적기능 보다 메모리 사용량이 더 크게 사용되고 있다. 특히 스텁과 스켈레톤 코드에 의해서 정적메모리 사용량이 더 많아지고 있다. 이 결과는 CORBA 구현물에 따라 달라질 것이다. 또한 스텁과 스켈레턴의 코드가 만들어지는 형태에 따라서 CORBA시스템의 성능이 달라질 수 있다.

이 논문에서 사용된 단일 메소드 호출 프로그램에서는 동적 기능에 대한 정적기능의 메모리 사용율에 있어서 이득이 있다고 보기 힘들다.

4.결론

본 논문에서는 표준 규격의 CORBA를 임베디드 시스템에 이식하고, CORBA 동적 기능과 정적 기능을 실제 응용프로그램을 통해 측정하고, 메모리사용과 성능비교를 통해 동적기능이 임베디드 시스템에서의 자원 문제에 대해서 알아보았다. 실제 간단한 단일 호출 프로그램에 있어서 메모리 증가량이 동적 기능의 응용 프로그램보다 오히려 정적 기능의 응용프로그램의 메모리 사용율이 20% 정도 증가 되었다. 이에 임베디드 시스템에서 간단한 동적 기능과 정적기능을 사용하는 응용프로그램의 자원사용은 실행되는 응용프로그램에 따라서 각각 차이를 보여 줄 것이다. 환경에 맞게 다양한 응용을 할 수 있을 것이다. 응용프로그램 마다 실질적인 데이터를 통해 CORBA 기능을 선택적으로 사용해야 할 것이다. 동적인 호출을 이용한 응용프로그램을 실험적 데이타를 근거로 사용한다면 다양한 사용자 요구에 대한 서비스를 제공할 수 있을 것이다.

현재는 간단한 프로그램을 가지고 실행했지만 다양한 데이타 타입과 호출 인터페이스를 구현 했을 때 정적기능과 동적기능의 CORBA응용프로그램이 사용하는 메모리와 성능의 증가량을 측정함 으로서 임베디드 시스템에서의 동적기능 사용에 대한 자세한 근거를 제시 하겠다.

참 고 문 헌

[1] Common Object Request Broker Architecture (CORBA/IIOP) Specification, Version 3.0.2. The Object Management Group. December 2002

[2] Minimum CORBA, Version 1.0. The Object Management Group, August 2002.

[3] Real-time CORBA, Version 1.1 The Object Management Group, August 2002

[4] Cathy Hrustich, " CORBA for Real-Time , High performance and Embedded Systems", Fourth IEEE International Symposium on Object Oriented Real-Time Distributed Computing, pp345-349,May 2001

[5] Sergio Perez, "A CORBA Based Architecture for Distriubed Embedded systems using the RTLinux-GPL Platform", Seventh IEEE International Symposium on Object Oriented Real-Time Distributed Computing,

[6] The MICO Project.http//www.mico.org

[7]The FunctionCheck Project.http ://sourceforge. net/projects/fnccheck