CORBA 개방형 분산 환경을 위한 공유 객체 명세 언어 시스템

박 양 수[†] · 김 현 규[††] · 이 명 준[†] · 한 상 영[†††]

요 약

클라이언트/서버로 구성된 분산 응용 시스템에서 공유 자료를 처리하는 서버는 유효 응답 시간(valid response-time)과 효율성(efficiency)을 위해 여러 클라이언트로부터의 요청에 대한 병행 수행을 지원할 수 있도록 설계하는 것이 바람직하다. 그러나 병행성 지원과 그에 따른 동기화 문제는 난이도가 높은 프로그래밍 기법이므로, 필요시마다 개발자가 이를 구현하는 것은 신뢰성 있는 시스템의 제작을 어렵게 한다.

이러한 문제를 체계적으로 지원하기 위한 노력의 일환으로, 본 논문에서는 CORBA 서버 객체의 구현에 있어서 *공유 객체*를 명세하기 위한 언어인 ***IDL/SSO***를 설계하고 구현하였다. 공유 객체(*Shared object*)는 여러 클라이언트의 요청에 대해 다양한 형태의 병행 수행을 지원하며, 병행 수행에 필요한 동기화 문제를 지원하는 자료 구조이다. IDL/SSO는 공유 객체의 명세에 있어서 CORBA IDL을 확장한 형태로 설계되었으며, IDL/SSO로 작성된 객체의 명세는 IDL/SSO 컴파일러를 통해 CORBA 서버로서 클라이언트의 요청에 대한 병행성을 지원해 주는 공유 객체의 구현 부분과, 공유 객체의 클라이언트를 위한 IDL 화일로 변환된다. IDL/SSO의 전체 시스템은 Java로 작성되었으며, Sun사의 CORBA 지원 제품인 JavaIDL상에서 실험되었다.

A Specification Language System for Shared object under CORBA Open Distributed Computing Environment

Yang-Su Park[†] · Hyun-Gyu Kim[††] · Myung-Joon Lee[†] · Sang-Yong Han[†††]

ABSTRACT

In distributed client/server applications, servers which handle shared resources are required to support concurrent execution of a lot of clients' requests for efficiency and rapid response. However, since programming techniques for controlling concurrency and synchronization between clients' requests are somewhat complex, it is not straightforward to develop reliable distributed applications through resolving such concurrency control and synchronization problem with hand-written codes whenever it is required.

As an effort for systematic resolution of such problems, in this paper, we present a language called IDL/SSO, which specifies shared objects for implementation of CORBA server objects. A shared object is an object which supports concurrent processing of clients' messages (requests) with appropriate synchronization between them. The IDL/SSO is designed as an extension of CORBA IDL to the specification of shared objects. The developed

※이 논문은 1996년도 한국학술진흥재단의 공모 과제 연구비에 의해 연구되었음.

† 정 회 원:울산대학교 전자계산학과
†† 준 회 원:울산대학교 전자계산학과
††† 정 회 원:서울대학교 계산통계학과

논문접수:1997년 9월 23일, 심사완료:1997년 11월 24일

IDL/SSO compiler generates the code to control concurrency and synchronization as a part of the implementation of the specified shared object in association with the related CORBA environment, while generating the CORBA IDL file for the clients of the shared object. The overall system is written in Java and has been tested under JavaIDL CORBA environment announced by Sun.

1. 서 론

분산 환경은 네트워크로 연결된 이기종 시스템들 간의 자원 공유를 가능하게 하여 정보 이용의 효율을 극대화시키며, 다양한 분산 환경 서비스의 투명성을 제공하고, 일부 시스템의 결함에 대처하여 높은 신뢰도를 제공할 수 있는 등의 장점을 가지고 있다[1]. 그러나 분산 환경을 구성하는 각 시스템간의 이질성과 원거리 통신에 대한 세부 구현 사항은 분산 응용 프로그램을 구축하는 개발자들에게 상당한 어려움을 주게 되었다. 이를 극복하기 위하여 90년대에 이르러 분산 객체 컴퓨팅(Distributed Object-Oriented Computing) 기법이 소개되었으며, 이는 분산 시스템을 구축하기 위한 방법론으로서 분산 시스템의 작성에 클라이언트-서버(Client-Server) 기법과 객체지향 기술을 도입하여 클라이언트-서버(Client-Server)의 다운사이징(Down-sizing)과 객체 지향 기술의 재사용성(Reusing)의 장점을 충분히 활용할 수 있도록 하였다[2].

이러한 분산 객체 컴퓨팅의 표준 구조로서 OMG에서는 CORBA[3, 4, 5]를 제안하였다. CORBA는 클라이언트와 서버 객체 간의 요청과 응답을 위한 세부 구현 사항을 개발자로부터 추상화시킬 수 있도록 구성되었으며, 서비스의 확장이나 재사용을 위해 서로 다른 CORBA 시스템과의 상호 운용이나 호환이 가능하도록 하였다. CORBA는 현재 Digital, IBM, Sun Microsystems, HP 등 100여 개의 회사에서 표준으로 채택하고 있으며 많은 객체 기술 개발사들도 이에 동조하고 있다.

CORBA를 포함한 여러 분산 응용 시스템은 유효 응답 시간(valid response-time)과 효율성(efficiency) 측면에서 여러 프로세스의 요청에 대하여 병행 수행을 지원할 수 있도록 설계하는 것이 바람직하다[6]. 그러나 병행성에 대한 지원과 그에 따른 동기화 구현 문제는 난이도를 요구하는 프로그래밍 기법 중 하나이므로 개발자가 효율적인 분산 응용 시스템을 구현하는데 있어서 상당한 부담으로 작용하게 된다[7]. 따라서 분산 시스템의 기반 구축에 있어서 여러 유용한 특성을 제공하고 있는 CORBA에서 서버 객체의 구현에 대해 병행성 문제를 체계적으로 지원할 수 있게 된다면, 효율적인 분산 응용 시스템의 구축에 있어서 개발자에게 편의를 제공할 수 있다.

본 논문에서는 CORBA 서버 객체의 구현에 있어서 병행성을 효과적으로 지원하기 위해 개발된 공유 객체 명세 언어 시스템인 IDL/SSO (IDL for Specifying Shared object)에 대하여 기술하고자 한다. IDL/SSO는 서버 객체의 명세에 있어서 공유 객체의 특성을 도입함으로써, 병행성과 그에 따른 동기화 문제를 구현하고자 하는 개발자의 노력을 최소화할 수 있도록 하였다. 공유 객체(Shared object)는 여러 프로세스에 의해 공유되어질 수 있는 자료 구조로서 각 프로세스의 요청에 대해 다양한 형태로 병행 수행이 가능하도록 지원하며, 병행 수행에 필요한 동기화 문제 역시 기본적으로 해결해 준다. 공유 객체의 각 멤버 오퍼레이션에는 병행 수행의 형태를 정의할 수 있으며, 각 멤버 오퍼레이션은 정의에 의해 상호 배제적인 수행을 하는 연산, 병행 수행이 가능한 연산, 그리고 조건부 수행을 제공하는 연산 등으로 구분되어 수행된다[8, 9].

IDL/SSO는 순수한 명세 언어로서 공유 객체의 명세를 위해 기본적으로 CORBA의 객체 인터페이스 정의 언어인 IDL[3, 10]을 수용하였으며, 이에 공유 객체의 특성인 멤버 오퍼레이션의 병행 수행 형태를 기술하기 위한 언어 구조를 추가하였다. IDL/SSO 컴파일러는 객체의 명세로부터 IDL 화일과 공유 객체를 구현한 화일을 생성하게 되며, IDL로부터는 CORBA의 Stub과 Skeleton 코드를 생성하여 통신을 위한 하부 구조로 사용될 수 있도록 구성하였고, 공유 객체의 구현 부분은 CORBA서버로 동작하게 함으로서 다중 클라이언트의 요청에 대한 병행 수행을 자연스럽게 처리할 수 있도록 하였다.

본 논문의 구성은 다음과 같다. 2장에서는 IDL/SSO 객체를 기술하기 위한 언어 구조에 대해 소개한다. 3장에서는 공유 객체의 구현 방법에 대해 다루며, 4장에서는 CORBA 환경에서 공유 객체의 특성을 지원하기 위한 방법론에 대해 기술한다. 마지막으로 5장에서는 IDL/SSO 설계 방법에 대한 요약과 병행성의 지원 확장에 대한 추후 연구 방향을 제시한다.

2. IDL/SSO 명세를 위한 언어 구조

IDL/SSO는 CORBA 서버 객체의 구현에 대해 공유 객체의 특성을 지원하기 위한 명세 언어로서 객체의 인터페이스 기술에 대해 기본적으로 CORBA IDL을 수용하고 있으며, 객체의 각 멤버 오퍼레이션에는 공유 객체의 특성인 병행 수행의 형태를 추가적으로 정의할 수 있도록 하였다.

이 장에서는 IDL/SSO 객체의 명세를 위한 언어 구조에 대해 다룬다. 먼저 2.1 절에서는 객체의 일반적인 구조 기술을 위한 CORBA IDL에 대해 소개하고 2.2 절에서는 공유 객체의 동기화 정의를 위한 구조에 대해 살펴본다.

2.1 CORBA IDL

IDL(Interface Definition Language)은 CORBA 객체의 인터페이스를 정의하기 위한 순수 명세 언어로서 데이터 형으로만 구성되어 있으며[3, 10], 구현을 위한 제어 구조에 대한 부분은 가지지 않는다. 또한 IDL은 C++나 Java와 유사한 구문 형태로 정의되어 있으므로 기존 사용자들이 사용하기 쉽다는 장점이 있다. 다음은 IDL에서 사용 가능한 데이터 형의 종류와 그에 대한 간략한 설명이다.

가. module : Java package, C++ namespace
나. interface : Java interface, C++ abstract class
다. operation : Java method, C++ member function
라. and other built-in data types ...

- Basic Types : (unsigned) short, (unsigned) long, float, double, char, boolean, octet, array, string ...

  octet형은 통신에서 호환성을 위한 단위 타입으로서 모든 시스템에서 8-bit로 처리하도록 하고 있다. 그 외에 기본형은 C++나 Java와 같은 해당 호스트 언어에서 대부분 제공하는 기본형과 일치한다.

- Constructed Types : const, enum, typedef, exception, struct, (discriminated) union, sequence, any ...

  sequence형은 크기가 가변적인 배열에 해당하며, any형은 모든 타입으로의 변환이 가능한 데이터 형이다. 이들 복합형에 대해서는 대부분 호스트 언어에서 직접적으로 지원하지 않으므로, IDL 컴파일러에서 각 데이터 형의 성질을 만족하는 클래스 형태로 구현하게 된다.

IDL 컴파일러는 IDL로 정의된 명세의 각 자료형을 해당 호스트 언어로 기술된 타입으로 변환하여 객체의 구현에 사용될 수 있도록 하며, 인터페이스 정의는 클라이언트와 서버 객체 간의 오퍼레이션의 요청과 응답을 위해 Stub과 Skeleton코드를 생성한다. Stub과 Skeleton 코드는 분산 환경에서 발생할 수 있는 데이터 형 사이의 이질성 문제와 통신을 위한 세부 코드를 포함한다. 개발자는 분산 응용 프로그램의 작성 시에 IDL 컴파일러를 통해 생성된 Stub과 Skeleton을 이용함으로써 분산 환경의 세부 구현에 대한 부담을 줄일 수 있다.

2.2 동기화 정의를 위한 언어 구조

공유 객체(Shared object)는 여러 프로세스에 의해 공유되어질 수 있는 자료 구조로서 각 프로세스의 요청에 대한 수행을 병행적으로 처리할 수 있도록 하며, 병행 수행에 필요한 동기화 문제 역시 기본적으로 해결해 준다. 공유 객체의 각 멤버 오퍼레이션에는 병행 수행의 형태를 정의할 수 있으며, 각 멤버 오퍼레이션은 정의될 수 있는 형태는 procedure, function, guard중 하나이다. procedure는 모든 오퍼레이션과 상호 배제적으로 동작하며 주로 쓰기 가능한 연산에 쓰인다. function은 procedure와 guard와는 상호 배제적이며 function 타입의 오퍼레이션과는 병행 수행이 가능하다. function은 주로 읽기 연산에 사용된다. 그리고 guard는 procedure와 같이 모든 타입의 오퍼레이션과 상호 배제적이다. 그러나 guard는 조건부 수행을 제공하며 procedure에 비해 더욱 세밀한 제어를 할 수 있다[8, 9].

공유 객체를 제공하는 언어의 대표적인 예로는 Orca의 Shared-data object와 Ada95의 Protected type등이 있다[9]. Orca의 Shared-data object에서는 동기화에 대한 정의가 명시적으로 이루어지지 않으며, 파라미터를 가지는 오퍼레이션은 procedure로, 리턴 값을 가지는 오퍼레이션은 function으로 동작한다. 그리고 조건부 수행은 오퍼레이션의 구현 내에서 guard 연산으로 정의하도록 한다. 이에 반해 Ada95의 Protected type에서는 객체의 선언부에서 모든 오퍼레이션에 대해 동기화 정의를 명확히 명세하도록 하고 있다(그림 1).

```
object specification Counter;                protected type Counter is
    operation Value(): integer;                  function Value() return integer;
    operation SetValue(V: integer);              procedure SetValue(V: integer);
    operation Decrement();                       procedure Decrement;
    operation AwaitValue(V: integer);            entry AwaitValue;
end;                                         end;
    [ Orca's Shared-Data Object ]                [ Ada95's Protected Type ]
```

(그림 1) Orca & Ada95's example
(Fig. 1) Orca vs. Ada95 Examples for interface definition

IDL/SSO는 Ada95의 Protected type의 방식처럼 동기화 정의에 대해 명시적으로 정의하는 방식을 수용하였다. IDL/SSO 객체의 오퍼레이션에 정의될 수 있는 동기화 구문은 다음과 같다.

```
with concurrency : [function | procedure | guard(boolean)]
```

IDL/SSO의 동기화 정의는 with절로 시작하며, concurrency는 공유 객체의 동기화에 대한 정의임을 의미한다. concurrency 이외에도 메세지 필터[11, 12]를 사용한 병행성 지원의 확장을 위한 구문으로서 precondition이나 redirection 등을 정의할 수 있으며, precondition과 redirection등에 대한 지원은 계속 연구 중이다.

concurrency의 값으로는 공유 객체의 동기화 형태인 function, procedure, guard 중 하나가 될 수 있다. 그 중 guard는 조건부 수행을 위한 boolean 함수를 인자로 가진다. (그림 2)는 IDL/SSO로 명세한 공유 객체의 예이다.

이 객체는 현재 카운터를 알려주는 Value, 인자를 통해 새로운 카운터 값으로 설정해 주는 SetValue, 카운터를 하나 감소시키는 Decrement, 그리고 어떤 일

```
class Counter {
    void Counter( );
    int Value( ) with
        concurrency :  function;
    void SetValue(int pValue)  with
        concurrency :  procedure;
    void Decrement( )  with
//..     precondition :  NotZero( );
//..     redirection :  ExtCnt.Decrement( );
        concurrency :  procedure;
    void AwaitValue(int pValue)  with
        concurrency :  guard( IsValue(pValue) );
}
```

(그림 2) IDL/SSO객체의 명세
(Fig. 2) Interface definition of IDL/SSO

정한 조건이 만족될 때까지 기다리는 AwaitValue를 서비스로 제공한다. 우선 Value는 객체의 내부 변수의 변경이 없으므로 function의 형태로 기술한다. 이에 반해 SetValue나 Decrement와 같은 서비스는 내부 변수에 영향을 줄 수 있으므로, procedure의 형태로 기술한다. AwaitValue는 내부 함수 IsValue가 참이 될 때까지 계속 진입 부분에서 기다리게 된다.

3. IDL/SSO 공유 객체의 동기화 구조

IDL/SSO시스템은 IDL/SSO 객체의 명세로부터 공유 객체의 특성을 지원하는 서버 객체의 구현 코드를 생성한다. IDL/SSO객체의 명세에서 각 오퍼레이션에 정의될 수 있는 병행 수행의 형태는 procedure, function, guard등이 있으며, 이 장에서는 이러한 병행 수행의 형태를 만족시키기 위해 동기화 구현이 어떠한 방향으로 지원되어야 하는지에 대해 살펴본다. 본 연구에서는 procedure, function, guard의 동기화 구현을 위해 세마포어와 조건 변수를 사용하며, 이러한 동기화 도구는 IDL/SSO의 목적 언어인 Java에서 직접적으로 지원하지 않으므로 본 연구에서 Java의 스레드에 대한 지연과 재수행 기능을 이용하여 자체적으로 개발하여 사용하였다.

3.1 Procedure의 지원

procedure로 정의된 오퍼레이션의 인스턴스(스레드)는 수행되는 모든 인스턴스와 상호 배제적으로 동작한다. 상호 배제는 세마포어를 이용하여 구현할 수 있으며, 다른 타입의 오퍼레이션과 procedure와의 완전한 상호 배제를 위해 모든 타입의 오퍼레이션의 Entry와 Exit 부분에서 (그림 3)과 같이 정의한다. En-

```
static semaphore sp;

void SetValue()                    void Value()                    void AwaitValue()
{                                  {                               {
   //.. Entry                        //.. Entry                      //.. Entry
   sp.wait(this);                   sp.wait(this);                  sp.wait(this);

   //.. User code will be here ...   //.. User code will be here ...  //.. User code will be here ...

   //.. Exit                         //.. Exit                       //.. Exit
   sp.signal();                     sp.signal();                    sp.signal();
}                                  }                               }
```

(그림 3) procedure, function, guard간의 완전한 상호 배제를 위한 구현

(Fig. 3) Implementation of mutual exclusion among procedure, function and guard operations

try는 오퍼레이션의 진입 부분의 구현을 나타내며, Exit는 진출 부분을 나타낸다고 가정한다.

세마포어 SP는 모든 인스턴스에서 상호 배제를 위해 공유될 수 있도록 static으로 선언된다.

위 구현에서 모든 오퍼레이션의 Entry에서 정의된 세마포어의 wait() 연산은 이미 수행되고 있는 스레드가 있을 경우에는 현재 자신의 수행을 지연시키며, 그렇지 않으면 다른 스레드가 수행될 수 없는 상태로 만들고 자신은 수행을 계속한다. wait() 연산의 인자로 사용되는 this는 현재 스레드의 핸들(handle)로서, 세마포어의 wait() 연산 내에서 이 핸들에 대해 suspend() 함수를 호출함으로써 해당 스레드의 수행을 지연시킨다. 이와는 대조적으로 Exit의 signal() 연산은 Entry에서 대기 상태의 스레드를 재수행할 수 있도록 해 주며, 내부적으로 resume() 함수를 사용하고 있다.

procedure는 디폴트 옵션(Default option)으로서 객체의 명세에서 오퍼레이션에 대해 명확한 동기화 정의가 없으면 IDL/SSO는 procedure로 간주한다.

3.2 Function의 지원

function으로 정의된 오퍼레이션의 인스턴스(스레드)는 procedure와 guard 인스턴스와는 상호 배제적이며, function 인스턴스와는 병행적으로 수행된다. 예를 들어 수행되고 있는 procedure나 guard 스레드가 존재할 경우 function을 포함한 모든 스레드가 Entry에서 블록(대기 상태로)된다. 이에 반해 수행되고 있는 스레드가 function일 경우에는 procedure나 guard의 요청은 블록되며, function의 요청은 병행적으로 수행될 수 있다.

(그림 4)의 구현에서 function Entry의 무조건적인 sp.wait()은 function 스레드 간에도 상호 배제적으로

```
static int cntFunc = 0;
static Semaphore sf;

void Value()
{
      //.. Entry
      sf.wait(this);
      if(cntFunc==0) sp.wait(this);
      cntFunc++;
      sf.signal();

      //.. User code ...
      //.. Exit
      sf.wait(this);
      cntFunc--;
      if(cntFunc==0) sp.signal();
      sf.signal();
}
```

(그림 4) function의 병행 수행을 위한 수정. 2

(Fig. 4) Implementation for parallel execution of function operations

수행할 수 있으므로 수정을 필요로 한다. 이를 위해 처음 수행되는 function 스레드에서만 sp.wait()을 호출하도록 수정하였으며, 수행되고 있는 스레드의 갯수에 대한 참조를 위해 cntFunc을 선언하였다. 그러나 cntFunc에 대한 변형과 참조는 function 스레드간에 병행적으로 수행될 수 있으므로 임계 영역(Critical section)으로서 보호되어야 하며, 이를 구현하기 위해 새로운 세마포어 sf를 도입하였다. function간의 병행 수행을 위한 전체적인 구현은 (그림 5)와 같다.

(그림 4)의 수정된 구현에서는 모든 function 스레드에서 sp.wait()을 호출하지 않고 처음 수행되는 function 스레드에서만 sp.wait()을 호출하도록 한다. 따라서 그 후 수행되는 function 스레드는 sp.wait()을 만나지 않으므로 function 스레드 간의 병행 수행이 가능해진다. 그리고 수행하려는 procedure나 guard 스레드는 처음 수행되는 function 스레드에서 이미 sp.wait()을 호출함으로서 Entry에서 블록된다. 이미 수행되고 있는 procedure나 guard 스레드가 존재한다

면 모든 function 스레드가 위 구현의 if ~ 이하의 조건을 만족하므로 역시 procedure나 guard간의 상호 배제가 이루어지게 된다.

3.3 Guard의 지원

guard로 정의된 오퍼레이션의 스레드는 모든 타입의 스레드와 상호 배제적이며, 추가적으로 조건부 수행을 제공한다. 조건부 수행에서는 오퍼레이션의 선행 조건이 만족되면 오퍼레이션을 수행하고 그렇지 않으면 만족될 때까지 대기 상태로 있게 된다.

```
static CondVar c1;

void AwaitValue1()
{
      //..  Entry
      sp.wait(this);
      if(!precond1()) {
            sp.singal();
            c1.wait();
      }

      //..  User code will be here ...
      //..  Exit
      sp.signal();
}
```

(그림 5) guard의 조건부 수행을 위한 수정. 1
(Fig. 5) Implementation for pre-conditional execution of guard operations [1]

guard의 조건부 수행을 처리하기 위해 guard 오퍼레이션의 Entry에서는 선행 조건인 precond1()이 만족되지 않았을 때 다른 스레드가 수행될 수 있도록 sp.signal()을 호출한 후, 자신은 AwaitValue1()의 선행 조건을 위한 조건 변수 c1의 대기열에서 대기할 수 있도록 (그림 4)의 구현을 수정하였다(그림 5). 각 guard 오퍼레이션은 자신의 조건 변수를 가지게 된다.

guard의 구현에서 주의할 점은 대기 상태로부터 재수행을 위해 조건에 대한 재검사가 반복적으로 이루어져야 한다는 점이다. 그렇지 않으면 guard 스레드는 영원히 대기 상태가 된다. 이 재검사는 객체의 내부 상태가 변경된 후에 이루어지는 것이 바람직하다. 따라서 쓰기 가능한 연산으로 쓰이는 procedure나 guard 스레드의 Exit에서 재검사를 수행하도록 (그림 6)처럼 작성한다.

재검사에 대한 코드는 reevaluation()에서 구현하고 있다(그림 7). reevaluation()에서는 procedure나 guard 스레드의 수행 이후 바뀐 상태를 만족하는 guard 스레드를 깨워 주어야 한다. 그리고 만족하는 guard 스레드가 없으면 다음 스레드가 수행될 수 있도록 sp.signal()을 호출한다.

```
void SetValue()                          void AwaitValue()
{                                        {
    //..  Entry                            //..  Entry
    //..  User code will be here ...       //..  User code will be here ...

    //..  Exit                             //..  Exit
    reevaluation();                        reevaluation();
}                                        }
```

(그림 6) guard의 조건부 수행을 위한 수정. 2
(Fig. 6) Implementation for pre-conditional execution of guard operations [2]

```
void reevaluation()
{
      if(precond1())   c1.signal();
      else if(precond2())   c2.signal();
      else if(precond3())   c3.signal();
      ...
      else sp.signal();
}
```

(그림 7) guard의 재수행을 위한 오퍼레이션 Reevaluation()
(Fig. 7) Reevaluation() to resume guard operations

3.4 Semaphore의 지원

IDL/SSO에서는 공유 객체의 동기화를 구현하기 위해 세마포어나 조건 변수를 사용한다. IDL/SSO에서 각 오퍼레이션의 수행은 스레드로 구현되므로 세마포어나 조건 변수는 조건에 따라 스레드를 지연시키거나 재수행할 수 있도록 설계되었다. 이를 지원하기 위해 IDL/SSO의 목적 언어인 Java의 suspend(), resume()연산을 사용하였다.

세마포어의 정의는 (그림 8)과 같다[14].

```
Semaphore s = 1;
wait(s) :       while s<=0   do no-op;
                s := s - 1;
signal(s) :     s := s + 1;
```

(그림 8) 세마포어의 정의
(Fig. 8) The definition of semaphore

세마포어 변수 s에 대해 행해질 수 있는 연산은 wait()과 signal() 뿐이다. wait(s)은 s가 0보다 작거나 같다면 계속 대기 상태로 있게 되고, 그렇지 않으면 s를 1만큼 감소시킨다. signal(s)는 wait(s)을 호출한 오퍼레이션이 수행될 수 있도록 1만큼 증가시킨다.

Java세마포어의 구현은 다음과 같다. wait()가 호출되었을 때 각 스레드에서 세마포어 변수 s에 대해 busy waiting을 하기 보다는 스레드의 suspend() 함수를 호출해서 수행을 지연시킨다. 그리고 signal() 시에

는 스레드의 resume() 함수를 통해 수행을 재개한다. 또한 Java version의 세마포어는 여러 스레드의 수행의 지연과 재개를 위해 내부적으로 스레드의 핸들(handle)을 인자로 하는 큐를 포함하게 된다. 세마포어의 정의에 맞게 Java로 구현하면 (그림 9)와 같다.

```
int s = 1;
ThreadQueue q;

void synchronized wait(Thread t)
{
    if(s<=0) {
        q.put(t);
        t.suspend();
    }
    else s--;
}

void synchronized signal()
{
    Thread t = q.get();
    t.resume();

    s++;
}
```

(그림 9) Java로 구현한 세마포어
(Fig. 9) Semaphore written in java

wait()과 signal()의 수정자로 쓰여진 synchronized는 이들 함수가 수행될 때 다른 스레드의 함수와 완전한 상호 배재가 이루어지도록 보장한다. Java 세마포어는 스레드의 핸들을 사용하기 때문에 스레드로 구현된 클래스에서만 사용가능하다.

4. IDL/SSO 시스템

IDL/SSO 시스템은 Sun사의 CORBA 지원 제품인 JavaIDL상에서 설계되었으며, IDL/SSO 객체의 명세로부터 공유 객체의 특성을 지원하는 서버 객체의 구현 코드를 생성한다. 이 장에서는 IDL/SSO의 CORBA 플랫폼인 JavaIDL에 대한 간략한 설명과 생성된 서버 객체의 구현 코드가 어떠한 형태로 공유 객체의 특성을 지원하는지에 대한 전반적인 면을 살펴본다.

4.1 JavaIDL 시스템

JavaIDL은 Sun사에서 CORBA를 지원하기 위해 만든 제품으로서 IDL 컴파일러와 DoorORB로 구성되어 있다[15]. JavaIDL의 IDL 컴파일러는 idlgen이라는 하나의 화일로 구성되어 있으며, IDL로 정의된 명세로부터 Java 언어로 작성된 Stub 코드와 Skeleton 코드를 생성시킨다. 그리고 DoorORB는 CORBA ORBCore에 해당하는 부분으로서 Java 패키지로 구성되어 있으며, 대부분 Stub과 Skeleton 코드에서 원거리 요청과 응답을 처리하기 위한 부분으로서 사용된다.

현재 JavaIDL이 제공하는 기능은 CORBA에 명세된 기능 중 가장 기본적인 구조로서 IDL 컴파일러를 이용한 Stub과 Skeleton 코드의 작성과 ORB core부분에 한정되어 있으며, CORBA에 정의된 Dynamic invocation, Interface repository, Implementation repository, Interoperablity 등에 대한 기능이 추가되어야 할 것으로 보인다.

4.2 IDL/SSO 서버 객체

IDL/SSO시스템은 CORBA에서 클라이언트간의 병행성을 지원하기 위해 JavaIDL의 서버 객체 구현 부분을 공유 객체의 형태로 구성한다. IDL/SSO시스템은 IDL/SSO로 정의된 객체의 명세로부터 컴파일러를 통해 공유 객체 형태의 서버 객체 구현에 해당하는 코드를 생성하며, 이 코드는 멤버 오퍼레이션의 병행 수행을 처리하기 위해 스레드[13]를 생성하고 수행시키는 클래스 ~Impl과, 요청된 오퍼레이션의 실제 구현과 공유 객체의 동기화를 구현하기 위한 코드를 포함한 스레드 클래스 ~ImplThread로 구성되어 있다.

가. 스레드의 생성과 수행

~Impl에서는 ~ImplThread의 인스턴스를 생성함으로서 클라이언트로부터 요청된 오퍼레이션을 수행하기 위한 스레드를 생성하며, ~ImplThread의 start() 함수를 통해 스레드로서의 병행 수행을 시작한다. start()는 ~ImplThread의 상위 클래스인 Java Thread의 멤버 함수로서 Thread 인스턴스의 수행을 시작하도록 하며, 이는 스레드 내의 run() 함수에 대응된다. 개발자는 스레드 고유의 동작을 구현하기 위해 run() 함수 내에 코드를 삽입하게 된다. IDL/SSO 역시 run() 함수 내에서 클라이언트의 요청에 해당하는 오퍼레이션을 호출하는 코드를 작성한다(그림 10).

그러나 start()만으로는 ~ImplThread의 run()에서 어떤 오퍼레이션을 수행해야 하는지 알 수 없다. 이를 위해 IDL/SSO에서는 start()로 스레드의 수행을 시작하기 전에 오퍼레이션을 식별할 수 있는 추가 정보들을 ~ImplThread의 생성자를 통해 전달한다.

오퍼레이션을 식별하기 위해 ~ImplThread에 전달되는 정보는 요청된 오퍼레이션의 이름과 리턴값 그

```
public class CounterImpl implements CounterServant {
   //..  Data member
   CounterImplThread aThread;
   ...
   public String SetValue(String msg) {
        //..  Create SetValue()'s thread and Execute
        aThread = new CounterImplThread("SetValue", paramList);
        aThread.start();
   }
}
```

(그림 10) IDL/SSO 명세로부터 생성된 ~Impl.java 화일 [1]
(Fig. 10) The file "~Impl.java" produced from the IDL/SSO specification [1]

리고 파라미터에 대한 정보이다. 데이터 중 오퍼레이션의 이름은 하나의 인자로 처리 가능하지만, 리턴값과 파라미터는 inout, out에 의한 데이터 변경을 고려해야 하며, 파라미터의 경우 크기가 가변적이므로 이를 처리할 수 있는 자료 구조가 필요하다. IDL/SSO에서는 리턴 값과 파라미터를 처리하기 위해 Java 유틸리티 클래스 중 하나인 Vector를 사용한다. Vector는 Object를 인자로 가지는 크기가 가변적인 배열이다. Vector는 Object 타입을 인자로 정의함으로써 모든 타입의 데이터를 인자로 가질 수 있으며, 또한 inout, out에 의한 데이터의 변경도 자연스럽게 처리할 수 있다. IDL/SSO는 Vector의 0번째 인자에 리턴 타입에 대한 정보를 가지도록 하며, 그 다음의 1부터 n까지의 인자에는 파라미터의 갯수만큼 정보를 가지도록 한다(그림 11).

```
public String SetValue(String msg) {
     Vector paramList = new Vector();

     //..  Parameter list :  0th element for return type
     paramList.addElement(null);
     //..  Parameter list :  1th element for 1th parameter
     Param p = new Param(msg);
     paramList.addElement(p);

     //..  Create SetValue()'s thread and Execute
     aThread = new CounterImplThread("SetValue", paramList);
     aThread.start();

     //..  Return value handling ...
}
```

(그림 11) IDL/SSO 명세로부터 생성된 ~Impl.java 화일 [2]
(Fig. 11) The file "~Impl.java" produced from the IDL/SSO specification [2]

나. 스레드 구현 부분

클래스 ~ImplThread는 클라이언트로부터 요청된 오퍼레이션의 병행 수행을 위해 스레드를 구현하는 부분이며, ~ImplThread의 각 멤버 오퍼레이션은 ~Impl에서 정의된 각 오퍼레이션에 대응된다. 그리고 ~ImplThread의 각 멤버 오퍼레이션은 명세를 구현하기 위한 고유 코드와 동기화 정의를 처리하기 위한 코드를 포함한다(그림 12).

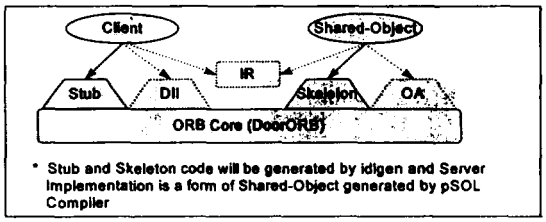
```
public class CounterImplThread extends Thread {
   //..  Data members
   Vector paramList;
   String service;
   static Semaphore SP = new Semaphore();

   public CounterImplThread(String pOpName, Vector pParamList) {
        service = new String(pOpName);
        paramList = pParamList;
   }
   public void run() {
        if(service.equals("SetValue")) SetValueSync();
   }
   void SetValueSync() {
        try {
             //.. Prolog code for procedure's synchronization
             SP.Wait(this);
             //..  Extract return type and parameters
             Param p = new Param();
             p = (Param)paramList.elementAt(1);
             ParamString p0 = (ParamString) p.getObject();
             paramList.removeElement(p);

             //..  Real implementation of SetValue()
             String r1 = SetValue(p0.getValue());

             //..  Return type handling
             p = new Param(r1);
             paramList.insertElementAt(p, 0);
             //.. Epilog code for procedure's synchronization
             Reevaluation();
        } catch(Exception e) { ... }
   }
   String SetValue(String msg) {
        //.. Do something :
   }
}
```

(그림 12) IDL/SSO 명세로부터 생성된 ~ImplThread.java 화일
(Fig. 12) The file "~ImplThread.java" produced from the IDL/SSO specification

위에서 언급했듯이 클라이언트의 ~Impl의 요청은 start()를 통해 ~ImplThread의 run() 함수로 전달되며, IDL/SSO는 run()내에서 해당 오퍼레이션을 호출한다. 그러나 해당 오퍼레이션은 실제 구현 내용을 수행하기에 앞서 ~Impl에서 Vector로 함축된 파라미터와 동기화 정의에 대한 처리를 해 주어야 한다. IDL/SSO는 이를 위해 각 오퍼레이션마다 부가적인 오퍼레이션을 정의하며, 명세에서 정의된 이름에 Sync를 추가하여 명명한다. 예를 들어 클라이언트로부터 SetValue()의 호출은 run()에서 SetValueSync()에 대응되며, SetValueSync()에서 파라미터와 동기화 정의에 대한 처리가 이루어진 후 실제 구현 내용을 수행하기 위한 SetValue()가 호출된다.

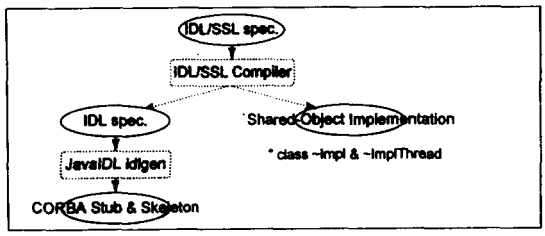
4.3 IDL/SSO 시스템의 전체적인 구조

IDL/SSO 시스템은 Sun사의 CORBA 지원 제품인 JavaIDL상에서 설계되었으며, IDL/SSO 객체의 명세로부터 공유 객체의 특성을 지원하는 서버 객체의 구현 코드를 생성한다(그림 13). 개발자는 이러한 서버 객체의 구현 부분을 이용함으로써, 클라이언트의 요청에 대한 병행 수행과 그에 따른 동기화 구현을 위한 노력을 줄일 수 있다.



(그림 13) IDL/SSO 시스템 구성도
(Fig. 13) The IDL/SSO system architecture

IDL/SSO 컴파일러는 명세로부터 동기화 정의를 제외한 순수 IDL 코드와 공유 객체를 구현한 Java 클래스를 생성한다(그림 14). IDL 코드로부터는 JavaIDL의 컴파일러 idlgen을 사용하여 CORBA Stub과 Skeleton을 얻을 수 있다. JavaIDL의 Stub과 Skeleton의 사용으로 개발자는 원거리 통신을 위한 세부 구현에 대해 고려할 필요가 없다. 그리고 공유 객체 클래스는 JavaIDL에서 서버 객체로 동작할 수 있도록 JavaIDL의 Skeleton 코드를 사용하며, 병행 수행과 동기화를 처리할 수 있도록 설계하였다.



(그림 14) IDL/SSO 컴파일러
(Fig. 14) The IDL/SSO compiler

IDL/SSO는 공유 객체 명세에 대한 컴파일러를 구성하기 위해 어휘 분석기로 JavaLex[16]를 사용하였고, 구문 분석기로는 Cup Parser[17]를 사용하였다.

5. 결 론

본 논문에서는 CORBA 서버 객체의 구현에 있어서 공유 객체의 특성을 지원하기 위한 명세 언어 시스템인 IDL/SSO (IDL for Specifying Shared object)의 설계 및 구현에 대하여 기술하였다. IDL/SSO 시스템은 Sun사의 CORBA 지원 제품인 JavaIDL상에서 설계되었으며. IDL/SSO 객체의 명세로부터 공유 객체의 특성을 지원하는 서버 객체의 구현 코드를 생성한다.

IDL/SSO시스템은 개발자가 이러한 서버 객체의 구현 부분을 이용함으로써, 클라이언트의 요청에 대한 병행 수행과 그에 따른 동기화 구현을 위한 노력을 최소화할 수 있도록 하였다.

IDL/SSO는 객체의 인터페이스 기술에 대해 기본적으로 CORBA IDL을 수용하고 있으며, 객체의 각 멤버 오퍼레이션에는 공유 객체의 특성인 병행 수행의 형태를 추가적으로 정의할 수 있도록 하였다. IDL/SSO 객체의 멤버 오퍼레이션에 대해 정의 가능한 병행 수행의 형태는 procedure, function, guard중 하나로서, procedure는 상호 배제적인 수행을 하는 연산에 사용되며, function은 병행 수행을 위한 연산에, 그리고 guard는 조건부 수행을 제공하는 연산을 위해 사용된다.

IDL/SSO로 명세된 객체는 공유 객체의 특성인 멤버 오퍼레이션의 병행 수행과 그에 따른 동기화 문제를 처리하기 위해 Java의 스레드 클래스로 구현된다. 멤버 오퍼레이션 간의 동기화는 세마포어와 조건 변수를 사용하여 구현하고 있으며, 이를 위하여 스레드의 지연과 재수행 기능을 이용하여 세마포어와 조건 변수를 위한 클래스가 자체 제작되었다.

IDL/SSO 컴파일러는 객체의 명세로부터 IDL 화일과 공유 객체를 구현한 화일을 생성한다. IDL로부터는 CORBA의 Stub과 Skeleton 코드를 생성하여 통신을 위한 하부 구조로 사용될 수 있도록 하며, 공유 객체의 구현 부분은 CORBA서버로 동작하게 함으로써 다중 클라이언트의 요청에 대한 병행 수행을 처리할 수 있도록 한다.

앞으로 병행성 지원에 있어서 공유 객체의 특성 외에 메세지 필터 개념을 도입하여 메세지 수행에 있어서 Precondition이나, Redirection 등의 기능을 지원할

수 있도록 하여 분산 객체를 이용한 응용 서비스가 보다 정교하면서도 신뢰성 있게 개발될 수 있도록 할 계획이다.

참 고 문 헌

[1] A. S. Tanenbaum, "Computer Networks", 2nd Edition, Prentice-Hall, Englewood Cliffs, NJ (1988).

[2] O. Hagsand, H. Herzog, K. Berman, R. Cooper, "Object-Oriented Reliable Distributed Programming", 2nd International Workshop on Object-Orientation in Operating Systems.

[3] OMG, "The Common Object Request Broker Architecture and Specification", OMG document Number 93-12-29, Dec 1993.

[4] Orbix-distributed Object Technology, "The Orbix Architecture", IONA Technologies Ltd, August 1993.

[5] Zhonghua Yang, Keith Duddy, "Distributed Object Computing with CORBA", DSTC Technical Report 23, June 1995.

[6] F. Ranno, S. K. Shrivastava and S. M. Wheater, "A System for Specifing and Coordinating the Execution of Reliable Distributed Applications," (DAIS'97), Cottbus, Germany, September 30-October 2, 1997.

[7] Douglas C. Schmidt, Steve Vinoski, "Object Interconnections, Column 5-6", C++ Report Magazine, 1995-1996.

[8] Henrie E. Bal, "Comparing data synchronization in Ada95 and Orca", ACM Ada letters, Vol. 15, No. 1, pp. 50-63, Jan/Feb. 1995.

[9] S. B. Hassen, H. E. Bal, "Integrating Task and Data Parallelism Using Shared objects", Vrije University, 1994.

[10] JavaSoft Inc., "Mapping IDL to Java", (alpha.1) Feb 1996.

[11] Lodewijk M. J. Bergmans, "Concurrency and Synchronization", Ph.D Thesis, Universiteit Twente, pp. 82-148, Jun 1994.

[12] L. Bergmans, M. Aksit & J. Bosch, "Composition-Filters: Extended Expressiveness for OOPLs", OOPSLA, 92 Workshop on Object-Oriented Programming Languages: The Next Generation, October 8, 1992.

[13] Aaron E. Walsh., "Java Programming for the World Wide Web", IDG Books World Wide, pp. 607-645. 1996.

[14] E. W. Dijstra, "Cooperating Sequential Processes", Technical Report EWD-123, Technological University, Eindhoven, the Netherlands(1965).

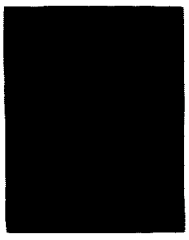
[15] JavaSoft Inc., "JavaIDL: An Integrating Java with CORBA", (alpha.1) Feb 1996.

[16] Elliot Joel Berk, "JavaLex: A Lexical Analyzer Generator for Java", Princeton University, Version 1.1, August 15, 1996.

[17] Scott E. Hudson, "Cup Parser: LALR Parser Generator for Java", Princeton University, Version 10, November, 1996.

박 양 수

1978년 2월 울산대학교 전자계산학과 졸업(학사)
1981년 2월 서울대학교 계산통계학과 졸업(석사)
1986년 3월~현재 서울대학교 계산통계학과 박사과정
1980년 3월 울산대학교 전자계산학과 근무(현재 부교수)
관심분야: 분산처리, 컴퓨터알고리즘등

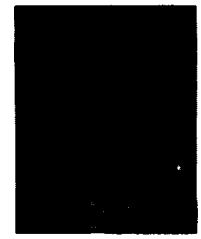


김 현 규

1997년 울산대학교 전자계산학과 공학사
1997년~현재 울산대학교 전자계산학과 대학원 석사과정
관심분야: 분산처리

이 명 준

1980년 2월 서울대학교 수학과 졸업(학사)
1982년 2월 한국과학기술원 전산학과 졸업(석사)
1991년 8월 한국과학기술원 전산학과 졸업(박사)
1982년 3월 울산대학교 전자계산학과 근무(현재 교수)
1993년 8월~1994년 7월 미국 버지니아대학교 교환교수
관심분야: 프로그래밍언어, 분산객체 프로그래밍시스템, 병행실시간 컴퓨팅, 인터넷 프로그래밍 시스템등



한 상 영

1972년 서울대학교 공과대학 응용수학과 졸업(공학사)
1977년 서울대학교 대학원 계산통계학과(이학석사)
1977년 3월~1978년 3월 울산대학교 공과대학 전임강사
1984년 3월~현재 서울대학교 자연과학대학 전산과학과 교수
관심분야: 병렬처리