CORBA 기반 네트워크 관리 시스템에서 MIB/MIT 구현

정회원 조 행 래*, 김 충 수**, 김 영 탁*

Implementation of MIB/MIT in CORBA based Network Management System

Haeng-Rae Cho*, Chung-Su Kim**, Young-Tak Kim* Regular Members

요 약

네트워크 기술의 발전과 더불어 새로운 멀티미디어 서비스가 급속히 늘어남에 따라 네트워크 관리는 더욱 복잡해지고 있다. 새로운 네트워크 관리 시스템의 플랫폼을 정의하려는 노력으로 분산 처리와 객체 지향 모델링 등의 정보 기술을 이용한 TINA 시스템이 출현하였다. 특히, TINA는 DPE(Distributed Processing Environment)를 기반으로 하므로 개방된 통신 시장의 범주에서 네트워크 및 서비스를 관리할 수 있다는 장점을 갖는다. 본 논문에서는 CORBA를 이용하여 TINA DPE를 구현한 네트워크 관리 시스템에서 다양한 CORBA 객체들을 연동하기 위한 MIB/MIT의 구현 방안을 제안한다. 제안한 구현 방안은 분산 컴포넌트 기반의 TINA 시스템을 위한 MIB/MIT의 분산 관리 방안을 제공하며, MIT에 대해 CORBA Naming Service를 이용하여 Scoping과 Filtering 기능을 지원할 수 있다는 장점을 갖는다.

ABSTRACT

The network management becomes more complicated due to the growth of network technology and introduction of a large number of new multimedia services. TINA has appeared as a concept for the advanced network management system platform using the information technology such as distributed processing and object oriented modeling. Since TINA is on the basis of DPE (Distributed Processing Environment), it can manage networks and services for open telecommunications. In this paper, we propose an implementation strategy of the MIB/MIT to federate various CORBA objects in CORBA based network management system implementing TINA DPE. The proposed strategy is novel in the sense that it can support the distribution of MIB/MIT that is well matched with the distributed component architecture of TINA DPE, and it can also support the scoping and filtering services on the MIT using CORBA Naming Service.

Ⅰ. 서 론

네트워크의 발전과 다양한 멀티미디어 서비스들이 등장하면서 네트워크의 규모와 구조의 복잡성이 급속히 증가하고 있으며, 대부분의 응용 소프트웨어도 분산 환경 모델로 변화하고 있다. 이러한 추세에 따라 네트워크를 효율적으로 관리하고 QoS(Quality of Service)를 보장하기 위해서 보다 진보된 네트워크 관리 시스템의 필요성이 증가하고 있다. 이러한 배경에서 분산 컴포넌트 기반의 네트워크 서비스 환경인 TINA(Telecommunications Information Net working Architecture)가 등장하였다[2,14]. TINA 서비스는 여러 개의 서비스 컴포넌트들로 구성되며, 각 서비스 컴포넌트들은 서비스 객체로 정의되는

* 영남대학교 전자정보공학부 (hrcho@yu.ac.kr) ** (주)네이버시스템
논문번호 : 020270-0607 접수일자 : 2002년 6월 11일
※ 본 연구는 학술진흥재단 중점연구소지원사업(1998-005-E0018)의 지원으로 수행되었습니다.

여러 개의 소프트웨어 모듈들로 구성된다. 이때 서비스 컴포넌트들과 서비스 객체들 간의 연동은 TINA DPE(Distributed Processing Environment)에 의해 정의된다. TINA DPE는 OMG(Object Management Group)의 CORBA(Common Object Request Broker Architecture)[9]를 기반으로 하고 있으며, 스트림 통신이나 다중 인터페이스 객체 등에 대한 기능들이 추가되어 있다[1].

네트워크 관리 시스템의 요소와 자원들을 효율적으로 관리하기 위해서는 데이터베이스를 구축하는 작업이 요구된다. OSI(Open Systems Interconnection) 네트워크 관리 표준에서는 네트워크 관리에 관련된 대규모의 데이터를 네트워크 관리 정보(MIB: Management Information Base)로 정의한다[4]. 그리고 MIB를 구성하는 관리 객체간의 포함 관계를 나타내기 위하여 관리 정보 트리(MIT: Management Information Tree)를 사용한다[5,6]. 이때, 네트워크를 구성하는 자원들은 관리 객체(MO: Managed Object)로 추상화하여 정의된다. 즉, MO는 네트워크 요소들 내에 존재하는 보드나 포트와 같은 물리적인 장치뿐만 아니라, 연결과 같은 논리적인 장치들을 시스템의 자원으로 관리하기 위해 각 자원들의 특성 및 관련 동작을 추상화하여 표현한 것이다.

본 논문에서는 CORBA를 이용하여 TINA DPE를 구현한 네트워크 관리 시스템에서 다양한 CORBA 객체들을 연동시키기 위한 MIB/MIT의 구현 방안을 제안한다. 본 논문에서 제안한 CORBA 기반 MIB/MIT 구현 방안의 특징은 다음과 같다.

- OSI 네트워크 관리 표준의 경우 관리자/에이전트 모델에 기반을 두어 MO들은 에이전트에 의해 통합 관리되며, 에이전트는 관리자의 요청에 따라 MO에 대한 연산을 수행하고 MO들의 상태를 관리한다. 이와는 달리 TINA 명세에 따른 CORBA 기반 네트워크 관리 시스템의 경우 관리자와 에이전트의 기능이 여러 서버에 의해 분산되며, 그 결과 MIB와 MIT에 대한 분산 관리가 요구된다[11]. 본 논문에서 구현한 MIB/MIT 모델의 경우, 전체 MIB는 관리 기능에 따라 MO 들을 분산하여 저장한 다양한 컴포넌트들로 구성하며 각 컴포넌트들은 여러 서버에 분산되어 저장될 수 있도록 한다. 그리고 MIT Handler라는 특수한 컴포넌트를 개발하여 분산된 MO들에 대한 검색/등록/삭제 연산을 수행할 수 있는 인터페이스를 제공하도록 한다.
- OSI 네트워크 관리 표준의 경우 CMIS/P 프로토콜을 이용하여 여러 MO들이 계층적으로 구성된 MIT에 대한 Scoping 기능과 Filtering 기능을 지원하고 있지만, CORBA 모델은 단일 객체에 대한 타입 기반 검색 서비스를 최적화하는 것이 주요 목적이다. 따라서 CORBA 기반 네트워크 관리 시스템을 개발할 경우, CORBA 객체들을 계층적으로 연결할 수 있는 MIT 구축 방법론과 관리 서비스에 소요되는 네트워크 트래픽을 최소화하기 위한 Scoping이나 Filtering과 같은 강력한 검색 기능들을 지원하여야 한다[12]. 특히 TINA와 같은 분산 컴포넌트 기반 모델의 경우 MIT는 여러 서버에 분산된 형태로 구성되어야 한다. 이런 관점에서 본 논문에서는 여러 서버에 분산되어 저장된 MO들을 논리적으로 통합할 수 있는 분산 MIT 구축 방안을 제안하였으며, CORBA의 Naming Service를 이용하여 MIT에 대한 Scoping과 Filtering 기능을 지원하도록 한다.

MIB/MIT에 관한 대부분의 기존 연구들은 OSI 네트워크 관리 표준을 가정하였으며, 본 논문과 같이 분산 컴포넌트 기반 네트워크 관리 시스템에서의 MIB/MIT 적용 방안에 대한 연구는 제안되지 않고 있다. 구체적으로 대부분의 기존 연구들은 CMIS/P 프로토콜에서 Scoping과 Filtering의 성능을 향상시키기 위한 MIB/MIT의 구성 방안을 제안하고 있는데, [7]의 경우 객체지향 데이터베이스의 개념을 확장한 다양한 인덱스 구조를 제안하였으며, [3]의 경우 균형이진 검색 트리와 이중 연결 리스트, 그리고 MO 클래스에 따른 MIT의 분할 저장 등의 방법을 통하여 Scoping과 Filtering의 성능을 향상시키고자 하였다.

본 논문의 구성은 다음과 같다. 2절에서는 관련 연구로 TINA와 CORBA의 기본 개념에 대해 살펴보고, 3절에서 CORBA 기반 네트워크 관리 시스템의 구현 모델에 대해서 설명한다. 4절에서는 MIB/MIT 구현을 위해 제안한 컴포넌트의 구조 및 상세 알고리즘을 기술하고, 분산 MIT를 지원하기 위한 구현 방안을 설명한다. 마지막으로 5절에서 결론 및 앞으로의 연구방향을 제시한다.

Ⅱ. 관련 연구

본 절에서는 TINA 시스템의 네트워크 관리 구조에 대해 살펴보고, CORBA 기반 네트워크 관리 시스템에서 MIB/MIT를 구현함에 있어 기반이 되는 CORBA Naming Service에 대해서 살펴본다. 본 논문과 다소 유사한 연구로는 [10]을 들 수 있는데, 이 연구에서는 CORBA 프로토콜을 이용하여 TMN 시스템을 구축할 때 필요한 OSI 네트워크 관리 표준의 GDMO 명세를 CORBA의 IDL로 변환하는 방법을 제안하고 있다. 그러나 관리자/에이전트 모델에 기반을 둔 TMN 시스템을 가정하고 있는 만큼, 본 논문에서 제안하고 있는 CORBA 모델 상에서의 MIT Handler나 분산 MIT 등에 관한 구축 방안은 고려되지 않았다.

2.1 TINA 시스템의 네트워크 관리 구조

TINA의 구조는 컴퓨팅 구조와 서비스 구조, 그리고 관리 구조의 3 부분으로 구성되는데, 본 절에서는 관리 구조 중에서 통신망 관리를 위한 네트워크 관리 구조에 대해 설명한다[2]. TINA 시스템의 네트워크 관리 구조는 DPE를 기반으로 하부 전송망 기술에 독립적인 분산 컴포넌트 기반 구조를 정의하고 있으며, 네트워크 계층화 및 분할 개념에 따라 각 계층에서 관리해야 하는 논리적/물리적 자원들을 정보 객체로 나타내고 있다. 정보 객체들은 TMN 체계의 MO에 해당하며 GDMO와 유사한 Quasi-GDMO로 기술된다. 정보 객체를 정의하고 있는 TINA 컨소시움의 NRIM(Network Resource Information Model Specification)에서는 정보 객체와 관계를 Network, Connectivity, Termination Point, Domain and Management Support, Resource Configuration, Fault Management, Accounting Management Fragment로 분류하여 정의하고 있다[13]. 뿐만 아니라, 기존의 네트워크 관리 체계의 장점들을 수용하기 위하여 TMN 체계의 5대 관리 계층(Business Management, Service Management, Network Management, Network Element Management, Network Element)의 개념을 지원하며, OSI의 5가지 네트워크 관리기능 분야(장애, 구성, 과금, 성능, 보안: FCAPS) 및 MO의 개념 등을 포함한다. 단, TINA 시스템에서는 FCAPS의 5가지 관리기능 영역에서 구성관리를 연결 관리와 구성 관리로 다시 세분화하고 있다[13].

TINA 시스템의 네트워크 관리 구조 하에서 응용은 여러 개의 소프트웨어 컴포넌트로 정의되는데, 각각의 컴포넌트는 TMN의 관리 계층과 OSI의 5가지 네트워크 관리 기능 분야의 특정 조합을 나타낸다. 예를 들면, 연결 기능 중에서 Network Management 계층에서 동작하는 기능들은 NML-CP라는 소프트웨어 컴포넌트로 정의되며, Network Element Management 계층에서 동작하는 기능들은 EML-CP라는 소프트웨어 컴포넌트로 정의된다. 각 소프트웨어 컴포넌트의 인터페이스는 OMG의 IDL을 확장한 ODL(Object Definition Language) 명세[14]에 따라 기술된다.

2.2 CORBA Naming Service

CORBA 기반의 네트워크 관리 시스템에서 MIT는 Naming Service를 기반으로 하여 구축된다. CORBA Naming Service[8]는 인터넷에서의 DNS 서비스와 같은 기능을 수행한다. DNS 서비스가 인터넷에서 호스트 식별을 위해 IP와 도메인네임을 관리하는 반면에, CORBA Naming Service는 네트워크에서 객체 단위 식별을 위해 객체 이름과 참조를 관리한다. CORBA 시스템에서 상호 운용 객체 참조자(IOR: Interoperable Object Reference)는 네트워크 상에서 CORBA 객체에 대한 유일한 식별을 위해 사용된다. IOR은 호스트 IP와 Port, POA, Object ID등의 정보로 이루어지며, 클라이언트에서 IOR을 이용해 서버의 CORBA 객체를 접근할 수 있게 된다. CORBA Naming Service는 객체에 대한 Name과 객체 식별자 사이에 하나의 연결을 만들어 트리 형태로 저장을 한다. 따라서 사용자는 논리적으로 잘 분류된 Naming Tree에서 객체 Name을 통해 객체 참조자를 쉽게 제공받을 수 있다.

표 1에 CORBA Naming Service의 다섯 가지 구성 요소가 나타난다. 객체의 Name은 문자열 ID와 KIND로 구성된 NameComponent의 나열로 이루어지고, Context라는 객체에 Name과 객체 참조자가 Binding이라는 형태로 저장된다. Naming Tree는 하나의 Graph로 구성된다. 그림 1은 Naming Service를 구성하는 각 요소의 개념을 Graph를 통해 설명해 놓은 것이다. Graph에서 실제 서비스 객체는 Graph의 리프 노드에 연결된다. 중간 노드의 객체는 NamingContext라는 Naming Service의 관리기능을 제공하는 CORBA 객체가 된다. Naming Tree를 위

의 그림과 같은 Graph로 구성하기 위해 표 2와 같은 연산이 사용된다.

표 1. Naming Service의 구성 요소

NameComponet	String ID and String KIND
Name	Sequence of NameComponet
Binding	A Name to Reference Association
Context	Object that stores name bindings
Graph	Hierarchy of contexts and bindings

표 2. Naming Service의 기본 연산

NamingContext 생성	NamingContext new_context()
Binding 생성, 변경, 해제	bind(in Name n, in object obj)
	bind_context(...)
	rebind(...)
	unbind(...)
Object 참조자 획득	object resolve(in Name n)

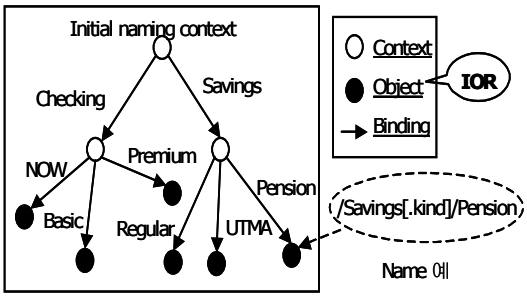

그림 5. Naming Graph의 예

표 2에서 분류한 각 연산은 Naming Service의 NamingContext라는 객체에 의해서 제공되는 연산들이다. 초기 NamingContext의 참조를 획득한 이후에, 해당 Context의 Binding 관리 메소드를 이용해 Naming Tree상에 다른 Context 및 Object에 대한 Binding을 추가/변경/삭제하여 트리를 구성하게 된다. Resolve() 함수는 구축된 Naming Tree에서 객체 참조자를 획득할 때 사용된다.

Ⅲ. CORBA 기반 네트워크 관리 시스템 구현 모델

본 절에서는 CORBA 기반 네트워크 관리 시스템에서 실제 MO들이 연동되는 방법에 따른 시스템 구조를 제시하고 구현 모델을 설명한다. 또한, MIB의 논리적 관계를 표현하여 MO들이 연동할 수 있는 기본 구조를 제공해 주는 MIT가 어떠한 구조를 가지는지 설명한다.

3.1 CORBA 기반 네트워크 관리 시스템 구조

네트워크 관리 시스템은 구성관리, 연결관리, 장애관리, 성능관리, 과금관리, 보안관리의 6가지 관리 기능과 IP, Data Link, Physical의 네트워크 구성 계층에 기반하여 네트워크의 장치 및 관리자원에 대해서 정의된 MO들로 구성된다. MO들은 관리기능과 계층별로 하나의 서버 프로세스로 실행될 컴포넌트로 분류화되어 저장되고 관리된다. MO로 구성된 부분은 네트워크 관리 시스템의 핵심이 되며, 부가적으로 MO들의 연동을 위한 추가 컴포넌트들과 기존의 SNMP나 TMN 기반 네트워크 자원과의 연동을 위한 Agent들이 추가로 CORBA 객체로 구현이 되어 MO로 동작하며, 핵심 컴포넌트들과 함께 연동하여 관리기능을 수행하게 된다.

그림 2는 MIT 관점에서 네트워크 관리 시스템 구조를 나타낸다. 전체 MIB는 각각의 관리기능에 따라 MO들을 분산하여 저장한 다양한 컴포넌트들로 이루어진다. 각 컴포넌트들은 실제로 여러 서버에 분산되어 저장된다. 많은 MO들이 생성되고 사라지면서 전체 네트워크를 관리하는데, 이때 각 MO들의 참조를 원활히 해주기 위해서 존재하는 것이 전체 MIB내의 MO들의 논리적인 구조를 표현한 MIT이다. MIT는 논리적으로 하나의 구조를 가지고 있지만, 실제 여러 서버의 Naming Service에 분산되어 구축될 수 있다.

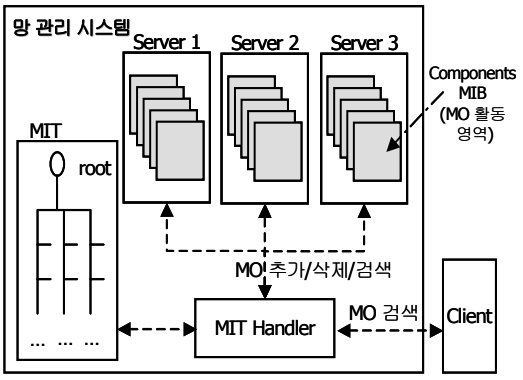

그림 7. MIT 관점에서 네트워크 관리 시스템 구조

MIT Handler는 MIT의 세부구조에 따라 네트워크 관리 시스템의 요구에 맞는 MIT 관리 연산을 수행하는 컴포넌트이다. 관리 시스템을 이용하는 사용자와의 인터페이스를 위해 관리 시스템 외부에는 추가 클라이언트들이 존재한다. 각 클라이언트는 관

리자가 시스템의 상태를 확인하고 관리행위를 수행할 수 있게 해주는 어플리케이션이다. MIT Handler는 CORBA 기반 네트워크 관리 시스템에서 MIT를 관리하기 위해 추가로 구현되는 CORBA 객체로서 별개의 컴포넌트로 저장되고, 네트워크 관리 시스템이 초기화될 때 구동되어 서로 다른 MO들이나 MO와 클라이언트간 연동을 지원한다. 이를 위하여, MIT Handler는 네트워크 관리 시스템 내부에 생성되는 MO들의 위치 정보를 MIT에 등록하고, 삭제하며 검색할 수 있는 인터페이스를 제공한다. 네트워크 관리 시스템의 각 구성 요소는 여러 플랫폼에서 구동되며, 구현에 사용된 개발언어 또한 다양하게 존재한다. MIT Handler는 CORBA 객체로 구현이 이루어져 플랫폼 및 개발언어에 독립적인 접근 인터페이스를 제공해 줄 수 있다.

3.2 MIT 구조

MIB의 논리적인 구조를 저장한 MIT는 하나의 트리로 보여지지만, 여러 서버의 CORBA Naming Service를 통해 분산되어 저장된다. 그리고, MIT는 크게 정적인 부분과 동적인 부분의 두 부분으로 구성된다. 정적인 부분은 Naming 트리의 골격을 형성하는 Context와 관리 시스템이 초기화될 때 구동하여 MIT에 등록되는 Factory MO이다. Factory는 다른 MO들의 인스턴스화를 담당하는 CORBA Factory 객체로 구현되어 있다. Naming Service의 특성상 실제 MO의 IOR은 리프 노드에 저장되기 때문에 Factory는 두 개 노드로 구성된다. 하나는 Factory MO의 IOR을 저장한 노드와 Factory가 인스턴스화 시키는 MO에 대한 IOR을 저장하기 위한 NamingContext 노드로 구성된다. Factory에 의해 생성되는 MO들은 실제 네트워크의 관리정보를 제공하는 객체들로 MIT의 동적인 부분을 구성한다.

그림 3은 본 논문에서 구현한 네트워크 관리 시스템 MIT 구조의 논리적인 형태이다. Intranet이라는 이름을 가진 전체 시스템은 Physical, Datalink, IP 계층의 논리적인 세 분류로 나누어진다. 각 계층은 MIT가 분산될 때 하나의 MIT를 나누는 경계의 대표 척도가 될 수 있다. 각 계층별로 관리기능에 따라 MO들은 Configurator, Connection, Recovery, Performance와 같은 독립된 컴포넌트에 분산되어 위치하게 된다. 전체 MIB는 CORBA Naming Service를 통해 그림과 같은 형태의 MIT로 구축되고, 모든 MO는 어느 위치에서나 일관된 위치 투명성을 제공받게 된다. 실제 MIT에서 Intranet/Physical/Configurator/Router의 하위 노드에는 Router1, Router2

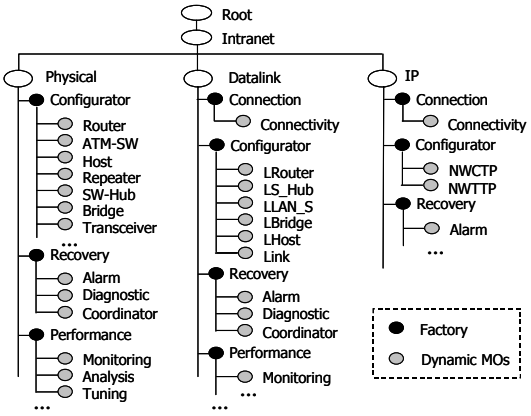

그림 3. MIT 구조

와 같이 인트라넷에 존재하는 라우터들의 MO가 다수 존재할 것이다. 라우터의 관리정보를 얻고자 할 때 해당 라우터를 관리하는 MO의 참조를 요청하여 MO와의 대화로 정보를 획득하거나 동작을 지시하게 된다. 하지만, MO가 존재하지 않을 때 Factory 객체인 Configurator MO에게 원하는 라우터의 MO를 CreateMO()와 같은 함수를 호출하여 요청하게 되면 해당 라우터의 MO가 생성되고 참조를 돌려받아 관리 작업을 수행할 수 있다. 새롭게 생성된 MO들은 하위 노드에 연결되어 다음 요청 시에는 새로운 MO 생성 절차 없이 참조가 이루어질 수 있다.

그림 3은 MIT의 논리적인 구조를 나타내는데, MIT를 구성하는 서브 트리들은 여러 서버에 분산되어 저장될 수 있다는 것에 주의하여야 한다. 예를 들면, 그림 3에서 Physical을 루트로 하는 MIT와 Datalink를 MIT, 그리고 IP를 루트로 하는 MIT들이 각각 다른 서버에 저장될 수 있다. 이 경우 MIT를 참조하는 응용에게 분산 투명성을 지원하기 위하여 분산된 서브 MIT들을 통합하는 과정이 필요한데, 본 논문에서는 CORBA Naming Service의 협동 구조를 이용하여 분산 MIT를 구축할 수 있도록 한다. 분산 MIT에 대한 자세한 설명은 4.3절에 나타난다.

Ⅳ. MIT Handler의 구현

네트워크 관리 시스템에서 MIT와 관련된 연산은 기본적으로 MO에 대한 등록/삭제/검색 연산으로 구성된다. MIT 관련 연산은 MIT Handler에서 구현되

고, MIT Handler는 네트워크 관리 시스템 내부에 CORBA 서버로 구동하여 MO들 및 추가 관리 컴포넌트들의 연동을 위해 이용된다. MO 등록작업은 활성화된 MO에 대해서 MIT에 Name과 IOR을 함께 등록하는 작업이며, 삭제 작업은 MO가 작업을 마치고 없어질 때 해당 IOR과 함께 Name을 MIT에서 삭제하는 것이다. MO 검색은 트리상에 등록된 MO들에 대해서, 간단하게는 Name을 기반으로 해서 하나의 객체를 찾아 IOR을 되돌려 주는 작업을 하며, 복잡하게는 MIT상의 여러 레벨에서 특정 검색 조건을 만족하는 모든 MO의 IOR을 되돌려 주는 작업을 수행한다.

본 절에서는 본 논문에서 구현한 MIT Handler의 구조 및 수행 함수의 알고리즘을 기술한다. 또한, 분산 MIT 지원을 위한 MIT 분산정책을 비교하고 분산 MIT 및 MIT Handler의 분산 지원 알고리즘의 구현 방법을 설명한다.

4.1 MIT Handler IDL 및 구조

MIT Handler는 CORBA 객체로 구현되며, 네트워크 관리 시스템 초기화시 구동되어 Intranet/ Mit Handler의 Name을 가지고 MIT에 등록된다. MIT Handler는 CORBA 객체로 구현되기 때문에 그 구현의 시작은 IDL의 정의에서 시작한다. MIT Handler의 명세는 그림 4의 IDL 정의에 나타난다.

```
module MitHandler {
  typedef sequence<string> NameSeq;
  enum ObjType { mit_Context, mit_Object, mit_Factory };
  enum SyncType {Atomic, bestEffort};

  interface i_MitHandler {
    readonly attribute Object RootCxt;
    short addMO(in string name, in Object obj,
                in ObjType type);
    short removeMO(in string name, in ObjType type);
    short resolveMO(in string name, out Object obj);
    short findMOs(
      in string base_obj_name,
      in long level,      in string filter,
      in SyncType sync, out NameSeq list
    );
};  };
```

그림 4. MIT Handler IDL

MIT를 구성하는 노드는 세 가지 유형을 가지는데, MitHandler::ObjType이 각 유형의 종류를 나타낸다. 첫째는 단순히 골격을 형성하는 mit_Context이며, 둘째는 구동중인 MO의 IOR를 저장한 mit_Object, 셋째는 논리적으로는 하나의 노드로 보이지만 실제로 mit_Object와 mit_Context 타입을 함께 가지는 mit_Factory이다. i_MitHandler는 MIT Handler 객체의 인터페이스이다. IDL은 개별 프로그래밍 언어와 독립적이지만, 인터페이스는 개별 언어의 특성에 맞게끔 IDL Compiler에 의해 자동으로 변환되어진다. C++의 경우 실제 구현되는 MIT Handler 객체의 부모 추상 클래스로 변환되며, Java의 경우 interface로 변환되어 생성된다. i_MitHandler는 개발 언어에 독립적인 MIT Handler에 대한 일반적 명세를 포함하고 있다.

MIT Handler는 Naming Service의 초기 Context인 RootCxt를 참조하기 위한 i_MitHandler:: RootCxt() 함수를 제공하고, 활성화된 MO를 MIT에 등록하는 i_MitHandler::addMO()와 MIT에서 MO 참조를 삭제하는 i_MitHandler::removeMO()를 제공한다. 문자열 타입의 Name을 입력받아 Object 타입의 IOR을 되돌려 주는 i_MitHandler:: resolveMO()와 MIT에서 검색조건에 따라 문자열 타입의 Name을 수집하여 MitHandler::NameSeq 형태로 반환하는 i_MitHandler::findMOs()가 있다.

4.2 Scoping 및 Filtering 구현

네트워크 관리 시스템의 MO나 관리 응용은 특정 관리요소에 대한 정보를 얻기 위하여 MIT Handler를 액세스한 후 원하는 서비스 MO의 IOR을 획득한다. 이때, MO를 획득하는 방법이 해당 MO의 Name을 통해 하나의 IOR만을 획득할 수 있다면, 특정 조건을 만족하는 여러 MO에게 정보를 획득하고 명령을 요청하는 작업을 수행하기 어렵게 된다. MO에 대한 위치정보는 Naming 트리를 구축함으로써 제공할 수 있지만, 사용자에게 관리 작업을 위한 MO 선택의 유연함을 제공해 주기 위해서는 CMIS에서 정의하고 있는 MO 선택을 위한 Scoping과 Filtering의 추가적인 기능을 MIT Handler에서 구현하여야 한다.

Scoping은 Tree상에서 그림 5와 같이 MO를 선택할 수 있도록 해준다. 첫째, Base object에 해당하는 하나의 MO를 선택한다. 둘째, Base object로부터 N번째 위치한 레벨의 전체 MO들을 검색한다. 셋째, Base object로부터 N번째 레벨까지의 모든 MO들을 검색한다. 넷째, Base object로부터 하위레벨 전체 MO들을 검색한다. 위와 같은 네 가지 범주에 따라 MIT를 검색할 수 있게 한다면 MO들은 관리

기능을 수행함에 있어 많은 유연함을 제공받게 된

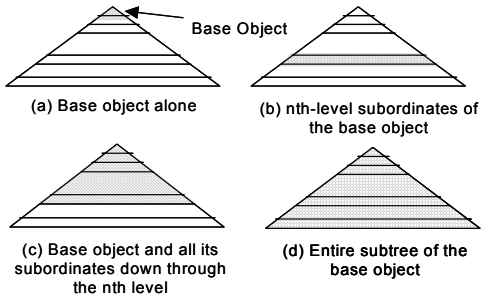

그림 5. Scoping의 종류

다. Filtering은 MO의 속성에 기반한 논리식 형태의 Filter를 통해 개별 MO가 특정 조건을 만족하는지 여부를 확인할 수 있다. Scoping에 추가해서 Filtering을 적용하게 되면 Scoping의 범주에 따른 범위에서 원하는 MO에게만 관리명령을 보내거나 정보를 수집할 수 있다. Synchronization은 Scoping과 Filtering에 의해 선택된 MO들이 특정 명령을 수행할 때, 해당 명령이 모두 수행 가능할 때 선택된 MO들에 실제 명령을 수행하게 하는 Atomic 타입과 선택된 MO들 중 모두가 명령을 수행하지 못해도 처리 가능한 MO들에게 실제 명령을 수행하게 하는 BestEffort 타입이 있다.

본 논문에서는 MIT Handler의 findMOs() 함수를 통해 Scoping과 Filtering을 구현하였다. 그림 6은 findMOs() 함수의 구조 및 작업흐름을 도식화 한 것이다. findMOs()는 Base object의 Name과 Scoping을 위한 레벨조건, 그리고 Filtering을 위한 Filter를 입력으로 받아 Scoping과 Filtering에 의해 선택된 Name 목록을 되돌려 준다. Naming Service 접근을 위한 RootCxt의 유효함을 검사한 후, filter 문자열을 파싱하여 문법 오류를 검사한다. 그리고, Base object로부터 Scoping을 수행하면서 Filtering을 적용하여 조건에 부합하는 MO에 대한 Name 목록을 저장한 후 결과로 반환한다. MIT는 여러 레벨로 많은 노드를 보유한 일반적인 트리 형태의 자료구조를 가지고 있으므로, 모든 레벨에 대한 검사를 수행하기 위해 각 레벨별로 검사를 수행하는 재귀 함수인 _findMOs()를 함께 정의하여 findMOs() 내부에서 사용한다. 즉, findMOs()는 MIT에 존재하는 MO의 검색을 위한 입력의 요구사항을 분석하고 얻어진 조건에 따라 _findMOs()를 호출하게 된다. _findMOs() 함수는 CORBA Naming Service와 실제적인 연동을 통해 MIT를 검색한 후, Filter와 Scoping 조건을 평가하고 MO의 Name 목록을 수집하는 작업을 수행한다.

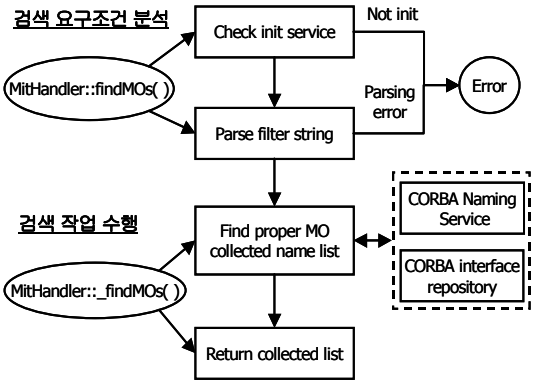

그림 6. findMOs() 함수 흐름도

```
short i_MitHandler::findMOs(...) {
 // 필요한 자료형 선언
 if(CORBA::is_nil(m_RootCxtExt)) return (RET_NOTINIT);
 sc_level = check_scoping( &level );
 filter = parsing_filter(filter_string);
 if( ! fillter ) return (RET_ERR_FILTER);
 list = new MitHandler::NameSeq;
 try {
   tmpObj = m_RootCxtExt->resolve_str(name_str);
   if ( sc_level == SC_BASEOBJECT )
   { ret = chk_filter_and_collect(nameStr, tmpObj, filter, list);}
   else {
     tmpContext =
          CosNaming::NamingContext::_narrow(tmpObj);
     ret = _findMOs(nameStr, sc_level, level, filter,
                    tmpContext, list);
 } }
 catch(...) { // 예외처리 }
 return (ret);
}
```

그림 7. findMOs() 함수 구조 및 알고리즘

그림 7은 findMOs() 함수의 알고리즘을 간략히 표현한 것이다. findMOs()는 검색 조건을 분석하고 _findMOs()의 입력 값을 생성한 후 _findMOs() 함수를 호출하여 Name 목록을 수집한다. findMOs() 함수는 Base object의 name 문자열과 level 값, 그리고 filter 문자열을 입력으로 받은 후, RootCxt의 유효성을 평가하고, Scoping과 Filtering의 요구 조건을 분석한다. 이때, check_scoping()은 정수 값의 level을 입력으로 sc_level을 판별한다. Sc_level은 SC_BASEOBJECT, SC_INDIVIDUALLEVEL, SC_BASETONTHLEVEL, SC_WHOLESUBTREE 의

네 가지 Scoping 범주에 대한 열거형 변수로, level 값에 따른 Scoping 범주는 표 3과 같다.

표 3. Level 값에 따른 Scoping 범주

level 값	Scoping 범주
level == 0	SC_BASEOBJECT
-MAX_LEVEL < level < 0	SC_INDIVIDUALLEVEL
0 < level < MAX_LEVEL	SC_BASETONTHLEVEL
MAX_LEVEL <= 절대값	SC_WHOLESUBTREE

Scoping 입력은 구조체로 구현할 수 있지만, 호출의 단순화를 주기 위해 정수 타입의 level값 하나로 Scoping을 분류하였다. Level 값은 Scoping 범주를 분류한 이후에 양수 값으로 변환시켜 _findMOs() 함수의 재귀 종료 조건으로 사용한다. parsing_filter()는 입력 filter 문자열의 문법을 체크하고, Filtering 수행에 용이한 자료를 구축한다. Name 목록을 수집해서 저장하게 될 list 입력변수는 실제 메모리를 할당받아 초기화된다. Scoping 범주가 SC_BASEOBJECT인 경우는, 바로 filter를 Base object에 적용시켜 조건에 만족한다면 Name 목록에 저장하게 된다. 다른 Scoping 범주인 경우, _findMOs() 함수가 입력받을 NamingContext를 생성한다. _findMOs()는 레벨별로 NamingContext를 입력받아 그 레벨의 검색을 수행하기 때문이다. 이후 생성한 sc_level, level, filter, tmpContext, list를 _findMOs()의 입력으로 넘겨 Scoping과 Filtering을 통한 검색을 수행하는 재귀 작업을 시작한다.

그림 8은 _findMOs() 함수의 구조와 알고리즘을 나타낸다. _findMOs()는 MIT에서 현 레벨의 Name과 NamingContext, Scoping을 위한 sc_level 범주 값과 level 정수값, 그리고 Filtering을 위한 filter를 입력으로 받는다. _findMOs() 함수가 호출되면 재귀 종료 조건을 결정하는 level 값을 감소시킨 후, 감소된 level 값과 Scoping 범주 값을 평가하여 현재 레벨이 Scoping의 검색범주에 따라 종료조건을 만족하는지 판별한다. is_this_level_to_collect()가 현 레벨의 검색종료 여부를 판별하는 루틴이다.

표 4는 Scoping 범주별로 재귀 종료 조건을 level 값으로 표현한 것이다. SC_BASEOBJECT는 findMOs() 자체적으로 작업이 행해져 _findMOs()에는 나타나지 않는다. 재귀 종료 조건에 부합된다면 현 NamingContext의 list() 함수를 통해 자식 노드의 Binding List를 구하여 object 타입에 대해서 Filtering을 평가한다. filter 조건에 부합하는 MO는 Name 목록에 추가된다. 재귀 종료 조건이 만족되지 않는 경우 자식 노드의 Binding List들 중 context 타입에 대해 _findMOs() 함수를 다시 호출하여 하위레벨로 검색해 들어간다.

```
short i_MitHandler::_findMOs(...) {
  // 필요한 변수 선언 ... 생략
  level --;
  Cxt->list(MAX_BINDINGS, bList, bIter);
  if ( is_this_level_to_collect(sc_level, level) ) {
    for(CORBA::ULong i = 0; i < bList->length(); i++)
      if(bList[i].binding_type == CosNaming::nobject)
        chk_filter_and_collect(upName, bList[i], filter, list);
    if(!CORBA::is_nil(bIter)) { // 남은 bList획득 및 추가 }
  } else {
    for(CORBA::ULong i = 0; i < bList->length(); i++)
      if(bList[i].binding_type == CosNaming::ncontext)
        ret = _findMOs(nameStr, sc_level, level, filter,
                       tmpCxt, list);
    if(!CORBA::is_nil(bIter)) { // 남은 bList 획득 및 처리 }
  }
  return (ret);
}
```

그림 8. _findMOs() 함수 구조 및 알고리즘

표 4. Scoping 범주별 재귀 종료조건

Scoping 범주	종료 조건
SC_INDIVIDUALLEVEL	level == 0
SC_BASETONTHLEVEL	level >= 0
SC_WHOLESUBTREE	level >= 0

Scoping은 Naming Service로 구축된 MIT를 _findMOs() 함수가 재귀적으로 검색함으로 이루어진다. 이와는 달리 Filtering은 그림 8의 코드 중 chk_filter_and_collect()에서 평가를 한다. IDL 컴파일러에 의해 생성된 코드에서 MO IDL의 속성은 실제로 액세스 가능한 함수로 생성된다. MIT Handler에서 Filtering을 지원하기 위해서 특정 MO가 가지고 있는 속성은 어떠한 것이 있는지, 속성에 대한 값에 대한 비교연산이 어떠한 형태로 이루어질 수 있는지 알 수 있어야 하지만 순수 CORBA에서 IDL을 통해 생성되는 코드는 충분한 해법을 제공해 주지 못한다. 따라서, 본 논문에서는 각 MO가 자신의 속성정보를 제공할 수 있고, 필터를 평가할 수 있도록 하였다.

그림 9는 MO들이 상속을 받는 i_ManagedObject 인터페이스의 IDL 코드로서 MO가 관리하는 속성

```
struct Attribute_t     {  string attrId;  any attrVal; };
enum GetListError_t { noError, noSuchAttribute };
struct GetAttribute_t {
   string attrId;
   any attrVal;
   GetListError_t error;
};
typedef sequence<string> AttributeIdList_t;
typedef sequence<Attribute_t> AttributeList_t;
typedef sequence<GetAttribute_t> GetAttributeList_t;
enum FilteringType { ftNotFillter, ftEqual, ftBig, ftSmall,
            ftBigOrEqual, ftSmallOrEqual, ftValueExist };
struct Filter_t {
   string attrId;              // 속성 명
   FilteringType filterType; // 연산 종류
   any attrVal;                // 이진 연산의 비교 대상 값
};
interface i_ManagedObject
{
   void getAttributes (in AttributeIdList_t attrIdList,
                       out GetAttributeList_t attrList);
   void getAllAttributes (out AttributeList_t attrList);
   boolean evaluateFilter (in Filter_t filter);
};
```

그림 9. Filtering 지원을 위한 MO 부모 IDL

정보를 확인해 볼 수 있는 getAttributes() 및 getAll Attributes()를 제공한다. getAttributes()는 특정 속성 정보만을 질의해 볼 수 있는 함수이며, getAllAttrib utes()는 MO가 관리하는 전체 속성 정보를 확인할 수 있도록 한다. 속성정보를 확인하는 함수에 추가해서 필터 조건을 평가할 수 있는 evaluateFilter() 함수를 제공하는데 Filtering의 주요 연산은 속성값에 대한 비교로 이루어진다. evaluateFilter()의 입력은 Filter_t 타입으로 '속성명', '연산종류', '이진연산의 비교 대상 값' 등 3가지로 구성되는 구조체로서 findMO()의 시작부에서 필터링 문자열을 파싱하여 구한 자료이다. i_ManagedObject 인터페이스를 구현한 MO들에 대해서는 MO 자체의 종류 및 내부 구현에 관계없이 위 3가지 함수를 통해 쉽게 MO의 Filtering 결과를 얻어낼 수 있다. 복잡한 필터링은 Filter_t의 시퀀스나 트리 구조를 활용함으로 구현 가능하다. evaluateFilter() 함수 내부 동작을 살펴보면, 입력으로 들어온 filter의 attrId에 대해 자신의 getAttributes() 함수를 호출하여 해당 속성을 얻어 filter의 연산에 따라 속성 값 비교를 수행하고 참/거짓을 반환한다. 비교하고자 하는 속성이 MO에 존재하지 않는 경우에는 거짓을 반환한다. Filtering 연산의 종류인 FilteringType은 표 5와 같다.

표 5. Filtering 연산 종류 (FilteringType)

FilteringType	필터 적용 설명
ftNotFillter	필터를 적용하지 않고, 무조건 참이다.
ftEqual	비교 대상값과 일치한다.
ftBig	비교 대상값보다 크다.
ftBigOrEqual	비교 대상값보다 크거나 같다.
ftSmall	비교 대상값보다 작다.
ftSmallOrEqual	비교 대상값보다 작거나 같다.
ftValueExist	속성의 값이 존재한다.

```
CORBA::Object_var tmpObj;
i_ManagedObject_var tmpMO;
tmpObj = resolveMO(nameStr, mit_Object);
tmpMO = i_ManagedObject::_narrow(tmpObj);
if ( tmpMO->evaluateFilter(filter) == true ) {
            add_name_to_list(nameStr, list);
```

그림 10. Filtering 평가 코드

그림 10은 그림 8의 check_filter_and_collect()에서 i_Managed Object를 이용해 Filtering을 평가하는 코드이다. Name을 통해 MO의 참조를 획득한 후 i _ManagedObject로 변환한다. 그리고 evaluateFilter () 함수를 호출하여 필터를 평가하고 조건을 만족하면 Name 목록에 추가한다.

4.3 분산 MIT 지원

복잡한 네트워크 관리 시스템일 경우 MIT 자체도 분산되어 관리될 필요가 있다. 분산 MIT란 하나의 논리적인 MIT 구조를 몇 개의 서버에 분산해서 저장하고 관리하는 것으로 CORBA Naming Service의 협동 구조(federation structure)가 이를 지원하기 위한 기본적인 방법을 제공한다.

그림 11은 Naming Service 협동 구조의 대표적인 두 가지 참조 구조이다[15]. 완전 연결 구조는 각 도메인이 다른 도메인들에 대한 Binding을 모두 가지고 있어, 각 도메인에서 동일한 Name을 가지고 모든 MO를 접근할 수 있다는 장점을 갖는다. 그러나, 부 도메인 추가/삭제 시 연결관리를 위한 부하가 크다는 단점이 존재한다. 계층형 구조는 Root Server라는 대표 서버를 추가로 두어 각 부 도메인은 대표 서버로 'parent' 라는 Binding을 생성하고 대표서버에서 각 도메인으로 하나의 Binding을 가지고 있는 구조이다. 지역 도메인과 원격 도메인의 M

O 접근시 상이한 Name 체계를 가진다는 것과 Roo

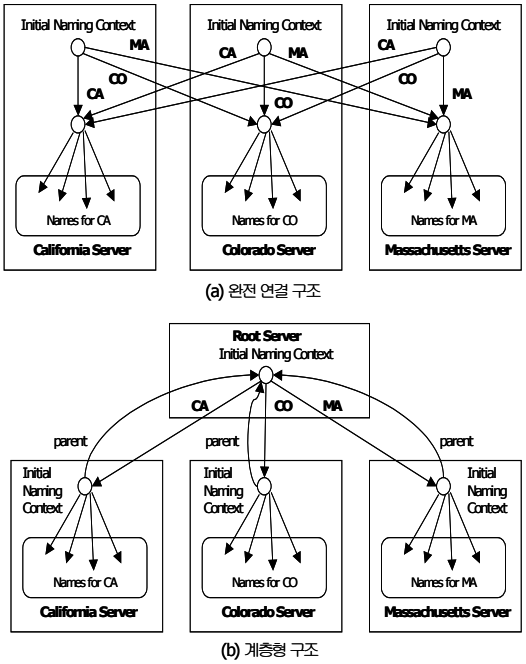

그림 11. CORBA Naming Service의 협동 구조[15]

t Server가 병목지점으로 작용할 수 있는 것이 단점이다. 그러나, 부 도메인 추가/삭제 시 완전 연결 구조보다 관리가 용이하며 Naming Service가 수직의 여러 계층으로 확장 가능하다는 장점이 있다.

본 논문은 주로 인트라넷을 기반으로 한 네트워크 관리 시스템을 고려하고 있어 분산 MIT 자체를 여러 계층의 Naming Service로 구현할 필요성이 없다. 뿐만 아니라, 네트워크 관리 시스템이 초기화될 때 협동구조는 완성되어지고 시스템이 초기화한 이후에 부 도메인의 추가/삭제는 쉽게 발생하지 않기 때문에 완전 연결 구조를 기반으로 하여 분산 MIT를 구현하였다.

그림 3의 MIT 구조를 분산하면 그림 12와 같이 하나의 MIT가 논리적으로 몇 개의 영역으로 나뉘어지며 각각의 영역은 다른 서버의 Naming Service에서 구축된다. 본 논문에서는 MIT에서 Physical, Datalink, IP의 세 개 영역의 수직 분할을 이용하여 분산 MIT를 구현하였다. 위 그림의 경우 Server1에서 Server2의 노드를 액세스하려면 Intranet/Datalink의 Name을 가지는 노드의 NamingContext를 먼저 획득한 이후 그 노드의 Context를 통해 하위 레벨에 존재하는 노드의 참조를 액세스해야 한다. 따라서 분산 MIT를 지원하기 위해서 앞 절에서 구현한 MIT Handler에 추가적인 코드가 필요하다.

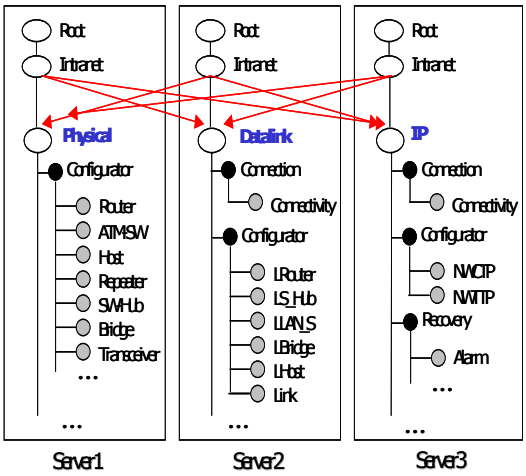

그림 12. 분산 MIT 구조

그림 13은 i_MitHandler에서 분산 MIT를 지원하기 위해 필요한 코드를 표현한 것이다. Naming Service는 Name을 통해 참조획득을 시도할 때 존재하지 않는 경우 NotFound라는 예외를 발생시킨다. 이 예외는 더 이상 접근이 불가능한 위치에서 이후의 Name을 rest_of_name 이라는 멤버 변수에 저장하고 있어 이를 이용하면 다른 서버에 존재하는 NamingContext의 참조를 얻어 그 NamingContext 하위 레벨의 노드를 액세스 할 수 있다. 제시한 코드는 resolveMO() 함수의 추가코드 이지만, addMO()와 deleteMO()의 경우도 비슷한 알고리즘을 가지고 있으므로, ※ 표시된 부분을 함수의 기능에 따라 다른 함수 혹은 코드로 교체하면 된다. findMOs() 함수의 경우 각 레벨별로 NamingContext를 획득하여 _findMOs() 재귀루틴을 이용하므로 추가 코드가 필요하지 않다.

인터넷과 같이 규모가 확대된 네트워크의 관리를 수행하는 시스템이라면 분산 MIT 구조를 확립하기 위해 확장성 및 성능을 위해서 Naming Service를 복수 계층으로 구축하는 구조와 각 서버가 다른 도메인의 참조 캐쉬를 보유한 구조를 고려해 보아야 한다. 본 논문에서 구현한 분산 MIT 구조와 같이 Naming Service간 수평적 연동은 확장에 제한이 따르기 때문에 복수 계층을 지원할 수 있는 계층형 구조를 활용할 필요가 있다. 하지만, 이 구조의 Root Server는 병목지점으로 성능저하를 가져올 수 있

으므로 각 도메인별 Naming Service에서 지역참조 캐쉬를 보유하여 다른 서버에 존재하는 Naming Service의 참조 횟수를 줄여주는 방안이 필요하다.

```
i_MitHandler::resolveMO(...) {
  // 생략 ...
  try {
    CosNaming::Name name =
          m_RootCxtExt->to_name(nameStr);
    retObj = m_RootCxtExt->resolve(name); // - ※
  }
  // 생략 ...
  catch(CosNaming::NamingContext::NotFound& nf) {
    CORBA::ULong l =
          name.length() - nf.rest_of_name.length();
    CosNaming::Name tmpName = name;
    tmpName.length(l);
    tmpObj = m_RootCxtExt->resolve(name);
    tmpContext =
          CosNaming::NamingContext::_narrow(tmpObj);
    retObj = tmpContext->resolve(nf.rest_of_name); // - ※
  }
  // 생략 ...
}
```

그림 13. 분산 MIT 지원 코드

V. 결 론

네트워크 규모의 증가와 복잡화, 그리고 다양한 멀티미디어 서비스의 증가로 보다 진보된 네트워크 관리 시스템이 필요해짐에 따라 새로운 응용 개발 표준인 CORBA를 이용한 네트워크 관리 시스템이 출현하였다. CORBA 기반 네트워크 관리 시스템은 무수히 많은 MO들이 연동하여 이루어지는 시스템이므로 MO들의 연동을 위해 MIB/MIT가 구현되어야 한다. 본 논문에서는 CORBA 기반 네트워크 관리 시스템의 MO 연동을 위한 시스템 개발 모델 및 MIB/MIT 구현 방안을 제시하였다. 순수 CORBA 기반 상용 네트워크 관리 시스템은 현존하지 않으며, 기존 연구는 주로 CORBA 플랫폼을 필요로 하는 서비스 환경과 기존 TMN 기반 네트워크 관리 시스템의 연동을 제안한 연구와 그에 따른 결과들이다. 따라서 순수 CORBA 기반 네트워크 관리 시스템을 위한 개발 모델을 충분히 제시하지 못하고 있다. 본 논문은 순수 CORBA 기반 네트워크 관리 시스템의 MO 연동을 위한 시스템 모델로 MIT Handler 컴포넌트를 추가한 시스템 모델을 제시하고 MIT Handler를 구현하였다. 본 논문에서 개발한 MIB/MIT 구현 전략은 CORBA 기반 네트워크 관리 시스템 및 CORBA 객체를 연동하는 응용 분야의 참조 모델이 될 수 있을 것으로 판단된다.

향후 연구과제는 인터넷을 기반으로 한 네트워크 관리 시스템을 위한 제안 모델의 확장이다. 본 논문의 구현 범위가 인트라넷으로 제한되어 분산 MIT 구현은 Naming Service의 수평적 연동을 이용하였지만, 인터넷 환경에서는 수평적 연동의 계층화가 요구될 것이다. 이를 위한 MIT 구조와 MIT Handler의 확장 방안을 연구할 예정이다.

참고문헌

[1] "Principles of TINA," http://www.tinac.com/ about/principles_of_tinac.htm.

[2] F. Dupuy, G. Nilsson, and Y. Inoue, "The TINA Consortium: Toward Networking Telecommunications Information Services," IEEE Comm. Magazine, vol.33, no.11, 1995.

[3] D. Han et al., "The Design and Implementation of a Network Management Platform for TMN," Proc. Int. Conf. on Computer Comm. and Networks, 1998.

[4] ITU-T Rec. X.701, "Information Technology- Open Systems Interconnection Systems Management Overview," 1992.

[5] ITU-T Rec. X.720, "Information Technology - Open Systems Interconnection Management Information Model," 1992.

[6] ITU-T Rec. X.721, "Information Technology - Open Systems Interconnection, Definition of Management Information," 1992.

[7] M-J. Kim and M. Choi, "Indexing Techniques for MIB Considering the CMIS Operations," Proc. GLOBECOM 96, 1996.

[8] Object Management Group, "Interopera

ble Naming Service Specification," 2000.
[9] Object Management Group, "The Common Object Request Broker: Architecture and Specification," CORBA Version 2.5, 2001.
[10] G. Pavlou, "Using Distributed Object Technologies in Telecommunications Network Management," IEEE J. on Selected Areas in Comm., vol.18, no.5, 2000.
[11] J. Pavon et al., "CORBA for Network and Service Management in the TINA Framework," IEEE Comm. Magazine, vol.36, no.3, 1998.
[12] D. Ranc et al., "A Staged Approach for TMN to TINA Migration," Proc. TINA 97, 1997.
[13] TINA-C Deliverable, "Network Resource Information Model Specification," Ver 3.0, 1997.
[14] TINA-C Deliverable, "Overall Concepts and Principles in TINA," Ver 1.0, 1995.
[15] H. Vinoski, Advanced CORBA Programming with C++, Addison Wesley, 1999.

조 행 래(Haeng-Rae Cho) 정회원

1988년 2월 : 서울대학교 컴퓨터공학과(공학사)
1990년 2월 : 한국과학기술원 전산학과(공학석사)
1995년 2월 : 한국과학기술원 전산학과 (공학박사)
1995년~현재 : 영남대학교 전자정보공학부 부교수

<주관심 분야> 분산/병렬 데이터베이스, DBMS 개발, Mobile Computing, 고성능 트랜잭션 처리, 성능 평가

김 충 수(Choong-Su Kim) 정회원
2000년 2월 : 영남대학교 컴퓨터공학과(공학사)
2002년 2월 : 영남대학교 컴퓨터공학과(공학석사)
2002년~현재 : (주) 네이버시스템 연구원

<주관심 분야> TINA/ CORBA 구조의 차세대 인터넷 망 운용관리

김 영 탁(Young-Tak Kim) 정회원
1988년 2월 : 영남대학교 전자공학과(공학사)
1986년 2월 : 한국과학기술원 전기및전자공학과(공학석사)
1990년 2월 : 한국과학기술원 전기및전자공학과(공학박사)
1990년~1994년: 한국통신 통신망연구소 전송망 연구실장/선임 연구원
1994년~현재 : 영남대학교 전자정보공학부 부교수
2001년~2002년: 미국 국립 표준 기술 연구소 (NIST) Advanced Networking Technology Division 방문연구원

<주관심 분야> DiffServ-aware-MPLS 및 TINA/ CORBA 구조의 차세대 인터넷 트래픽 엔지니어링 및 망운용관리, GMPLS 구조의 차세대 광통신망 제어 프로토콜 설계 및 구현