ISSN 1598-0170 (Print)
ISSN 2287-1136 (Online)
http://www.jksii.or.kr

분산 시스템의 기능 및 비기능 검증을 위한 테스트 프레임워크 개발☆

Development of a Test Framework for Functional and Non-functional Verification of Distributed Systems

윤 상 필[1] 서 용 진[2] 민 법 기[2] 김 현 수[2*]
Sangpil Yun Yongjin Seo Bup-Ki Min Hyeon Soo Kim

요 약

분산 시스템은 물리적으로 분산된 컴퓨터들이 네트워크에 의해 유기적으로 연결된 것을 의미한다. 유무선 인터넷의 보편적인 사용으로 인해 사용자는 언제 어디서나 분산 서비스의 이용이 가능하게 되었다. 분산 서비스의 폭발적인 증가는 서비스의 기능적 측면에서의 검증뿐만 아니라 서비스 품질과 관련된 비기능적 요소의 검증도 강하게 요구하고 있다. 분산 서비스를 검증하기 위해서는 분산 시스템에 맞는 테스트 환경을 구축해야 한다. 하지만 분산 시스템은 물리적으로 분산된 노드로 구성되기 때문에 테스트 환경을 구축함에 있어서 단일 시스템의 테스트 환경보다 많은 노력이 요구된다. 이 논문에서 우리는 분산 시스템의 기능 및 비기능 요소의 검증을 위한 테스트 프레임워크를 제안한다. 제안하는 테스트 프레임워크는 메시지 시퀀스 차트(Message Sequence Chart)를 기반으로 테스트 케이스를 자동 생성하며, 물리적으로 분산된 노드를 흉내 낼 수 있는 가상의 분산 노드로 구성된 테스트 드라이버를 포함한다. 테스트 수행 결과는 다양한 그래프와 GUI를 통해서 쉽게 확인할 수 있다. 이 논문에서 제안하는 테스트 프레임워크를 통해 분산 시스템 테스트에 드는 노력을 감소할 수 있고 시스템의 신뢰성을 향상 시킬 수 있을 것이다.

☞ 주제어 : 분산 시스템, 테스트 프레임워크, 기능 테스트, 비기능 테스트, 서비스 품질

ABSTRACT

Distributed systems are collection of physically distributed computers linked by a network. General use of wired/wireless Internet enables users to make use of distributed service anytime and anywhere. The explosive growth of distributed services strongly requires functional verification of services as well as verification of non-functional elements such as service quality. In order to verify distributed services it is necessary to build a test environment for distributed systems. Because, however, distributed systems are composed of physically distributed nodes, efforts to construct a test environment are required more than those in a test environment for a monolithic system. In this paper we propose a test framework to verify functional and non-functional features of distributed systems. The suggested framework automatically generates test cases through the message sequence charts, and includes a test driver composed of the virtual nodes which can simulate the physically distributed nodes. The test result can be checked easily through the various graphs and the graphical user interface (GUI). The test framework can reduce testing efforts for a distributed system and can enhance the reliability of the system.

☞ keyword : Distributed systems, Test Framework, Functional Testing, Non-functional Testing, Quality of Service

1. 서 론

분산 시스템은 물리적으로 분산된 컴퓨터들이 네트워크에 의해 유기적으로 연결된 것을 의미한다. 주변에서 쉽게 볼 수 있는 ATM(Automatic Teller Machines), POS 시스템, 인터넷 뱅킹 시스템 등을 예로 들 수 있다. 유무선 인터넷이 보편적으로 사용됨으로 인해 모바일 컴퓨팅이나 유비쿼터스 컴퓨팅과 같은 기술이 등장하였고, 이런 기술을 바탕으로 사용자는 언제 어디서나 분산 서비스의 이용이 가능하게 되었다[1]. 하지만 한편으로 분산 시스템 애플리케이션을 개발할 때 제공하는 서비스에 대한 기본적인 기능 이외에도 확장성(Scalability), 보안(Security), 이질성(Heterogeneity), 투명성(Transparency) 등

[1] Healthcare Sector, Siemens, Seongnam-si, Gyeonggi-do, 463-847, Korea.
[2] Department of Computer Science & Engineering, Chungnam National University, Daejeon, 305-764, Korea.
* Corresponding author (hskim401@cnu.ac.kr)
[Received 28 March 2014, Reviewed 04 April 2014, Accepted 22 July 2014]
☆ 이 연구는 충남대학교 학술연구비에 의해 지원되었음.

과 같은 비기능적인 요소들에 대한 검증이 중요하게 되었다[1]. 예를 들어, 동영상 스트리밍 서비스를 제공받는 사용자의 경우 영상을 수신하는 기본기능 이외에 영상의 화질이나 끊김과 같은 서비스 품질(Quality of Service: QoS)에 대한 요구사항을 갖는다. 이는 분산 시스템이 제공하는 서비스에 대해서 다수의 클라이언트의 접속, 서비스에 대한 요청 폭주와 같은 예외상황에 대한 검증이 중요함을 의미한다.

분산 시스템의 기능 및 비기능 요구사항을 검증하기 위해서는 기본적으로 테스트 환경을 구축하고 테스트를 수행하면서 요구사항에 명시된 기능이 오류 없이 잘 동작하는지, 사용자에게 만족할 만한 수준의 서비스를 제공하는지 확인해야 한다. 하지만 분산 시스템은 일반 시스템과는 달리 노드(node)라고 하는 물리적으로 분산된 컴퓨터(하드웨어/소프트웨어 컴포넌트 포함)로 구성되기 때문에 테스트의 어려움을 가중 시킨다. 이를 요약하면 다음과 같다.

- 테스트 환경 구축
  분산 시스템은 연결되는 노드의 수와 거리, 그리고 각 노드의 특성이 매우 다양하기 때문에 테스트 환경을 구축하는데 많은 노력이 든다. 예를 들어 분산 시스템의 확장성을 검증하기 위해서는 수많은 노드를 연결해야 하는데, 각 노드를 설치하는데 많은 시간과 비용이 요구된다.

- 테스트 수행 제어 및 테스트 결과 관찰
  분산 시스템은 다수의 노드가 병렬적으로 동작하면서 비동기적인 메시지를 주고받는다. 테스트 수행을 위해서는 먼저 노드들이 주고받는 메시지에 대한 테스트 시나리오를 작성해야 한다. 그리고 테스트 수행 시 작성된 시나리오에 맞게 각 노드들의 동작을 제어해야 하는데, 이 과정이 매우 복잡하다. 또한 오류가 발견되었을 때, 발생한 오류를 반복적으로 관찰하는 테스트 재수행의 어려움이 있다.

이처럼 분산 시스템은 단일 시스템보다 테스트 환경을 구축하고 테스트를 수행하는데 많은 노력이 필요하다. 이 논문에서는 분산 시스템의 기능적인 측면과 서비스 품질을 달성하기 위한 비기능적인 측면을 검증하는 과정에서 테스터가 쉽게 테스트 환경을 구축할 수 있는 테스트 프레임워크를 제안한다. 제안하는 테스트 프레임워크는 분산 시스템이 메시지 전달(Message Passing)에 의해서 제어되고 동작한다는 점에 주목하였다. 즉 분산된 노드들은 테스트 대상 시스템(System Under Test: SUT)과의 메시지 송수신에 의해서 서비스를 요청하거나 서비스를 제공받는다. 이런 관점을 반영하여 SUT의 테스트에 필요한 노드들을 물리적으로 설치하지 않고, 테스트 시나리오를 기반으로 각 노드들의 메시지 송수신 동작을 시뮬레이션 하는 방법을 사용하여 테스트를 수행한다. 제안하는 테스트 프레임워크를 사용하면 테스트 수행에 드는 시간과 비용을 단축하며 분산 시스템의 신뢰성을 높일 수 있다.

2. 관련 연구

분산 시스템은 하나의 머신에서 동작하는 단일 시스템과는 달리, 네트워크에 연결된 원격지의 노드들이 RPC(Remote Procedure Call) 또는 메시지 전달(Message Passing)과 같은 통신에 의해 협업한다. 따라서 분산 시스템을 테스트하기 위해서는 먼저 물리적으로 분산된 다수의 노드(클라이언트)들을 구축해야 한다. 그리고 테스터는 SUT가 각 노드들과 주고받는 메시지에 대한 테스트 시나리오를 작성한 후에 테스트를 수행한다. 노드들과 SUT와의 메시지 송수신을 통해 SUT가 기능 및 비기능적인 요구사항을 만족하는지 확인한다. 이 장에서는 단일 시스템과는 다른 분산 시스템 검증에서의 주요 이슈 및 기존연구를 살펴본다.

2.1 분산 시스템 검증에서의 주요 이슈

분산 시스템은 아래와 같이 세 가지의 검증 이슈를 가진다. 본 논문에서는 나열된 검증 이슈를 고려하여 테스트 프레임워크를 구축한다.

- 노드(클라이언트)의 제어
  분산 시스템은 다수의 노드가 병렬적으로 동작하기 때문에 비결정성(Non-determinism)을 내포하고 있다. 이와 같은 비결정성은 동일한 입력에 대해서 다른 실행 결과를 생성할 수 있다. 비결정성을 고려하여 테스트 케이스를 생성하는 방법에 대한 많은 연구들이 존재한다[2,3,4]. 비결정성을 테스트하기 위해서는 테스트 프레임워크에서 생성된 테스트 케이스 즉, 테스터가 의도한 시나리오(메시지 시퀀스)에 맞게 메시지를 전송하도록 노드를 제어해야 한다. 노드의 동작을 제어하기 위해서는 각 노드에서 Global

Trace를 파악하고 있어야 한다. Global Trace란 SUT가 전체 노드들과 주고받는 메시지 시퀀스를 의미한다. 하지만 하나의 노드는 물리적으로 분산된 다른 노드의 상태를 파악할 수 없기 때문에 제어에 어려움이 있다[5,6].

- 테스트 수행 결과의 관찰
  테스트의 수행 결과를 판단하기 위해서는 SUT가 노드들과 주고받는 메시지를 관찰해야 한다[5,6]. 테스트를 수행하면서 송수신하는 메시지의 내용과 순서에 대한 정보를 바탕으로 테스트의 성공/실패 여부를 파악해야하기 때문이다. 비기능적인 측면을 검증하기 위해서는 다양한 척도(metric)가 필요하다. 특히 송수신되는 메시지에 대해서 시간을 측정해야 하는데, 분산 시스템을 구성하는 각 노드들의 기준 시간이 서로 다르기 때문에 시간 동기화(Clock Synchronization)가 필요하다[7]. 이는 분산 시스템에서는 절대적인 시간인 전역 시간(Global Time)이 존재하지 않기 때문이다. 이러한 문제들을 고려하여 척도를 바탕으로 수집된 다양한 데이터들을 취합하여 테스트 결과를 판단해야 한다.

- 노드(클라이언트)의 설치 및 관리
  노드를 제어하여 테스트를 수행하고 테스트 결과를 관찰하기에 앞서, 노드를 설치하여 테스트 환경을 구축해야 한다[8]. 특히 비기능적인 요소를 검증하기 위해서는 많은 노드들을 설치해야 하는데 이에 대한 시간과 비용이 많이 요구된다. 또한 테스트 시나리오마다 서비스를 요청하기 위한 클라이언트의 동작 코드(예: 테스트 스크립트, 메시지 시퀀스)가 다르기 때문에, 각 클라이언트에게 테스트 코드를 분배하는 과정이 필요하다. 따라서 테스트 프레임워크는 노드의 설치 및 상태를 업데이트하는 데에 있어서 효율적인 방법이 필요하다.

2.2 기존 연구와의 비교

이번 절에서는 분산 시스템의 테스트 프레임워크와 관련된 논문을 몇 가지 살펴본다. 먼저, Jata라고 하는 테스트 프레임워크를 바탕으로 테스트를 수행하는 논문 [9], [10]이 있다. Jata는 JUnit[11]과 TTCN-3(Testing and Test Control Notation version 3)[12]을 통합하여 설계되었다. 여기서 JUnit은 자바의 단위 테스트 프레임워크를 말하며, TTCN-3은 통신 프로토콜을 바탕으로 테스트 케이스를 기술할 수 있는 프로그래밍 언어를 의미한다. Jata의 장점은 분산 시스템의 테스트를 단일 시스템의 컴포넌트를 테스트하는 것과 유사한 환경에서 수행할 수 있다는 것이다. 그러나 단위 테스트 프레임워크를 기반으로 설계되었기 때문에, Global Trace를 파악할 수 없는 단점을 갖는다.

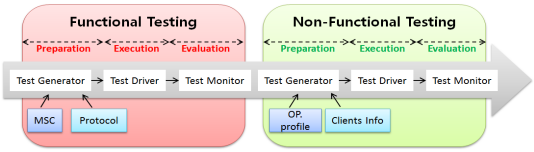
다음으로 논문 [7]이 존재한다. 논문 [7]은 분산 시스템의 비기능적인 요소를 검증하기 위한 테스트 환경을 제안한다. 논문 [7]에서는 노드에게 테스트 코드를 분배한 뒤, 테스트를 수행하는 방법을 사용하며, 테스트의 결과를 취합하여 그래프로 나타낸다. 노드를 설치 및 분배하는 방법을 제공하였으나, Global Trace를 파악할 수 없다는 단점이 존재한다. 또한 시간 동기화를 위한 주기적인 통신은 실제 행위에서 존재하지 않는 오버헤드를 발생시킨다. 마지막으로 단일 서비스에 대해서만 성능 측정이 가능하기 때문에, 다수의 서비스를 제공하는 시스템을 검증하기에 적절하지 않다. 논문 [13] 역시 노드의 설치 및 관리에 대한 부분을 고려한 연구이다. 논문 [13]에서는 시뮬레이터를 통해 노드를 생성하고 관리한다. 모든 노드는 동일한 시뮬레이터 위에서 동작하기 때문에 논문 [7]과 달리 시간 동기화 문제는 발생하지 않는다. 또한 테스트 서버를 통해 노드가 관리되기 때문에 Global Trace를 파악하기에 용이하다. 그러나 논문 [13]의 방법은 시뮬레이터의 노드와 테스트 서버가 서로 네트워크(TCP/IP)를 통해 정보를 전송하기 때문에, 노드와 테스트 서버 사이에 높은 지연율이 존재한다. 이는 테스트의 결과를 분석하는데 문제가 될 수 있다.

논문 [14]에서는 가상화 시스템을 이용하여 분산 시스템을 모사하는 방법을 제안한다. 제한된 물리 장치로 인해 실제 상황과 유사한 환경을 만들기 어렵기 때문에, 물리 장치를 가상화 시스템으로 대체함으로써 실제 환경과 유사한 테스트 환경을 구축하는 방법을 사용한다. 그러나 가상화 시스템을 통해 구성된 노드는 실제 환경의 노드와 유사한 동작을 갖기 때문에 Global Trace를 파악하기 어렵다는 단점을 갖는다.

본 논문에서는 소프트웨어 레벨의 환경을 제공하여 기능과 비기능에 대한 테스트를 모두 수행할 수 있다. 본 논문의 테스트 프레임워크에서는 테스터로부터 입력받은 테스트 시나리오를 바탕으로 자동으로 테스트 케이스를 생성한다. 테스트 케이스에는 SUT와 노드들 간의 Global Trace 정보가 담겨 있으며, 테스트를 케이스를 실행하는 테스트 드라이버가 이 정보를 유지한다. 테스트 드라이버는 노드에 대한 설치, 제어 등과 같은 관리 기능

을 갖는다. 마지막으로 비기능적 요소에 대한 테스트 결과는 처리량, 반응시간, 지연시간, 손실률 등과 같은 척도를 통해 확인하기 때문에, 비기능적 요소에 영향을 미치는 요소를 쉽게 분석할 수 있다.

3. 분산 시스템의 기능 및 비기능 검증을 위한 테스트 기법



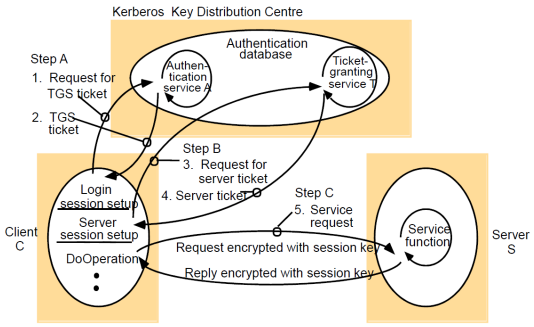
(그림 1) 테스팅 과정
(Figure 1) Testing Process

이 논문에서 제안하는 테스트 프레임워크는 그림 1과 같은 과정으로 테스트를 수행한다. 테스트 수행은 크게 세 단계로 구분된다. 테스트에 필요한 정보를 입력하여 테스트 케이스를 생성하는 준비(Preparation), 테스트 드라이버가 테스트 케이스를 실행하는 실행(Execution), 수집한 정보를 바탕으로 테스트 수행결과를 판단하는 평가(Evaluation) 단계이다. 테스트 수행은 SUT의 기능 검증을 완료한 후에 비기능성을 검증하는 순서로 진행된다. 이 장에서는 준비단계에서 기능 및 비기능성 검증을 위한 테스트 케이스 생성 방법에 대해서 설명한다.

3.1 기능 검증의 위한 테스트 케이스 생성

기능 검증이란 요구사항에 명시된 시스템의 기능이 오류 없이 동작하는지 확인하는 것을 의미한다. 충분한 기능 검증이 되지 않은 시스템은 오작동(malfunction)을 유발할 가능성이 높으며 결과적으로 분산 시스템의 상호 운영성을 감소시킨다. 시스템의 기능적인 측면을 검증하기 위해서는 테스트를 통해서 시스템이 주어진 명세를 만족하는지 확인해야 한다[15]. 이 절에서는 분산 시스템의 기능 검증을 위한 테스트 케이스 생성 방법을 Kerberos 예제[1]로 설명한다.

네트워크에서 메시지를 주고받을 때 여러 가지 보안 공격에 대비해야 한다. 커버로스는 KDC(Key Distribution Center)라는 키 분배 시스템으로부터 서로 간의 통신을 위한 세션키(Session Key)를 제공한다. KDC는 인증(Authentication) 서비스를 제공하는 AS(Authentication Server)와 세션키를 분배하는 TGS(Ticket Granting Service)로 구성된다. KDC를 통해서 클라이언트와 서버가 세션키를 제공받는 단계는 그림 2와 같다. 클라이언트가 AS에게 인증을 요청(①)하면, AS는 일정시간 TGS와 통신할 수 있는 인증키를 제공(②)한다. 클라이언트가 인증키를 사용하여 TGS에게 서버 S와의 통신을 요청(③)하면, TGS는 클라이언트의 인증키를 확인하여 유효한 경우 일정시간 통신이 가능한 세션키를 제공(④)한다. 세션키를 획득한 클라이언트는 서버에게 세션키를 전달(⑤)한다. 서버는 전달받은 세션키가 유효한 경우, 키를 획득했음을 클라이언트에게 알린다(⑥).



(그림 2) 커버로스 프로토콜
(Figure 2) Kerberos Protocol

커버로스 프로토콜에서 KDC를 테스트 대상 시스템(SUT)으로 하여 기능 검증을 위한 테스트 케이스를 생성하는 절차는 다음과 같다.

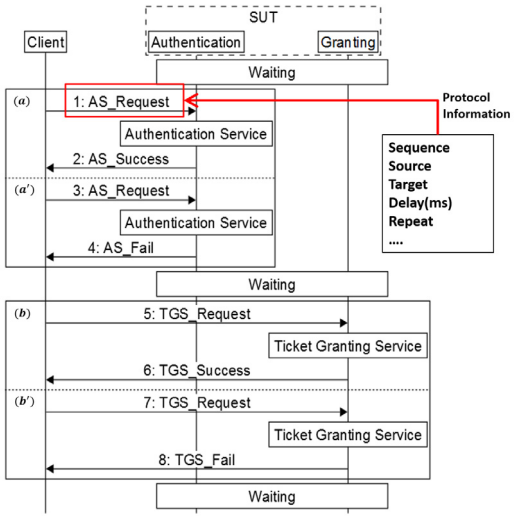
첫째, 테스터는 검증할 KDC의 기능에 대한 테스트 시나리오를 메시지 시퀀스 차트(Message Sequence Chart: MSC)를 이용하여 작성한다. 그림 3과 같이 기능을 수행하기 위해서 필요한 메시지 시퀀스를 기술하는데, 이때 하나의 메시지에 대해서 존재하는 모든 예외 상황을 포함하여 기술한다. 예를 들어 클라이언트가 인증을 요청한다면 인증에 성공한 경우와 인증에 실패한 경우에 대해서 기술할 수 있다. 그리고 메시지를 주고받으면서 검증하고자 하는 SUT의 상태 변화도 함께 기술한다. 본 논문에서는 MSC-generator[16]라는 도구를 이용하여 메시지 시퀀스 차트를 작성한다. MSC-generator에서는 별도의 명세 언어[17]를 제공하며, 그 중 일부의 요소를 이용하여 표 1과 같이 메시지의 송수신, SUT의 상태, 대체 흐름을 표기한다.

그림 3은 표 1의 표기법을 기반으로 커버로스 프로토콜의 흐름을 표기한 것이다. SUT의 상태를 위한 표기법

은 메시지 시퀀스 차트 상에서 네모 박스 안의 label이 표기된 형태로 표현된다. 즉, 그림 3에서는 "Waiting", "Authentication Service", "Ticket Granting Service"가 SUT의 상태를 의미한다.

(표 1) MSC의 표기법
(Table 1) Notation of MSC

Notation	Meaning
[entity]--[entity]: [label]	State of SUT (label is State name)
[entity]--[entity]: [label] { [typical sequence] } ..: [label] { [alternative sequence] }	Alternative sequence
[entity]->[entity]: [label]	Message (label is Message name)
[entity]<-[entity]: [label]	



(그림 3) 테스트 시나리오의 생성 및 프로토콜 정보
(Figure 3) Creation of Test Scenario and Protocol Information

둘째, 테스터는 작성한 테스트 시나리오에서 프로토콜과 관련된 메시지 속성 정보들을 입력한다. 예를 들어 그림 3에서 1번과 3번 시퀀스는 동일한 AS_Request 메시지이지만 메시지의 내용이나 지연 시간과 같은 정보가 다를 수 있다. 따라서 메시지의 지연시간, 반복 횟수 등의 프로토콜 정보들을 입력한다.

셋째, 테스트 시나리오의 메시지와 SUT 상태 정보를 기반으로 테스트 트리를 생성한다. 테스트 트리 생성 알고리즘은 그림 4와 같으며, 표 1의 표기법으로 작성된 메시지 시퀀스 차트를 파스(Parse)하여 테스트 트리를 생성한다.

Algorithm GenerateTestTree
Input : statements of message sequence chart St
Input : initial state of SUT s_i
Input : a root node n_p
Input : a test tree $T=(N,E)$

```
1  procedure GenerateTestTree
2    Q ← ∅ // initialize a queue
3    s_c ← s_i // initialize a SUT's state
4    for each st ∈ St do
5      if st is a statement for SUT's state then
6        L(n_t) ← label, N ← N ∪ {n_t}
7        L(e_t = (n_p, n_t)) ← Q, E ← E ∪ {e_t}
8        n_p ← n_t
9        Q ← ∅
10     else if st is a statement for alternative seq. then
11       if Q ≠ ∅ then
12         L(n_t) ← ∅, N ← N ∪ {n_t}
13         L(e_t = (n_p, n_t)) ← Q, E ← E ∪ {e_t}
14         n_p ← n_t
15         Q ← ∅
16       end if
17       for each block statement st_b in st do
           // St_r is the rest of statements
           // St_r ⊂ St
18         GenerateTestTree (st_b ∪ St_r, s_c, n_p, T)
19       done
20       break a loop
21     else if st is a statement for message then
22       Q ← Q ∪ {label}
23     end if
24   end for
25 end procedure
```

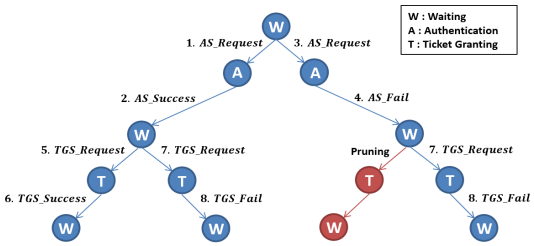
(그림 4) 테스트 트리 생성 알고리즘
(Figure 4) Algorithm of Test Tree Generation

그림 4의 알고리즘은 명세 언어로 작성된 메시지 시퀀스 차트의 코드를 입력받아 동작한다. 본 알고리즘은 재귀적으로 동작하며, 이 과정에서 메시지 시퀀스 차트에 대한 테스트 트리를 생성하며, 생성되는 테스트 트리 $T=(N,E)$는 SUT의 상태를 노드 N로 표현하며, SUT의 상태 전이를 유도하는 메시지를 엣지 E로 표현한다.

그림 4의 알고리즘은 세 가지의 표기법에 따라 알맞은 동작을 하도록 구현된다. 먼저, 메시지와 관련된 문자을

만났을 때는 해당 메시지의 이름을 큐에 저장한다. 이렇게 저장된 메시지의 이름은 엣지의 라벨로 활용된다. SUT의 상태는 메시지를 전송할 때마다 변경되는 것이 아니므로, 하나의 엣지에 다수의 메시지가 라벨링될 수 있다. 따라서 이를 효과적으로 표현하기 위해 큐를 사용한다. 6번째 줄부터 9번째 줄까지는 SUT의 상태에 대한 문장을 만났을 때의 동작을 수행한다. 문장 내부의 label이 곧 SUT의 상태를 의미하므로, 새로운 노드(n_t)를 생성하고, 해당 노드를 SUT의 상태로 라벨링한다. 여기서 라벨링 함수 $L()$은 테스트 트리를 구성하는 노드와 엣지의 라벨 값을 지정하기 위해 사용하는 함수이다. 노드를 생성하고 난 뒤에는 부모 노드 n_p와 생성된 노드 n_t를 연결하는 엣지 e_t를 생성하고, 엣지 e_t의 라벨에 큐 내부에 존재하는 모든 메시지를 추가시킨다. 마지막으로 대체 흐름을 표현하는 문장을 만나면, 각 블록 문장(st_b)에 대해서 테스트 트리 알고리즘을 새로 호출한다. 여기서부터는 흐름이 나뉘기 때문에 각 블록 문장은 서로 다른 흐름으로 판단하고 새로 테스트 트리 알고리즘을 수행하여야 한다. 그전에 큐 내부에 처리되지 않은 메시지가 남아 있다면, 빈 노드를 이용하여 처리한 뒤에 알고리즘을 수행하여야 한다.

그림 4의 알고리즘을 그림 3의 메시지 시퀀스 차트에 적용하면 그림 5와 같은 형태의 테스트 트리를 얻을 수 있다. 다만, 그림 3과 같이 대체 흐름이 구성되는 경우, 즉 대체 흐름이 연속하여 구성되는 경우(단, nested 형태인 경우는 제외)에는 실행 불가능한 조합이 발생할 수 있다. 실행 불가능한 조합은 제거하여야 하지만, 메시지 시퀀스 차트는 단순히 메시지의 흐름만을 다루기 때문에 자동으로 실행 불가능한 조합을 판별할 수 없다. 다만, 그림 3과 같이 대체 흐름이 구성되는 경우에만 발생하기 때문에 이 점을 이용하여 실행 불가능한 조합을 가지치기(Pruning)할 수 있는 지점은 파악할 수 있다. 가지치기가 수행되어야 하는 위치는 연속된 대체 흐름이 존재할 때, 첫 번째 대체 흐름을 제외한 나머지 대체 흐름의 분기가 시작되는 곳이다. 그림 4의 알고리즘에서는 연속된 대체 흐름이 존재할 때 모든 조합을 생성하여 두 번째 대체 흐름부터는 불가능한 조합이 생성될 수 있기 때문이다. 하지만, 실제 가지치기는 테스터가 판별하여 수행하여야 한다. 그림 3의 경우에는 a 또는 a', b 또는 b'의 대체 실행이 있으며, 이로부터 (a, b), (a, b'), (a', b), (a', b')의 조합을 생성된다. 여기서 깊이가 3인 위치가 가지치기가 수행되어야 하는 위치이다. (a', b)의 조합은 첫 번째 대체 흐름에서 인증에 실패하였음에도 불구하고 두 번째 대체 흐름에서 정상적인 동작을 수행하므로 실행 불가능한 조합이 된다. 따라서 그림 5의 붉은 부분을 가지치기하면 테스트 트리를 완성할 수 있다.



TC	Message Seq.	Description
TC1	1->2->5->6 (a, b)	Authentication Success and session key acquisition
TC2	1->2->7->8 (a, b')	Authentication success, but session key acquisition is failed
TC3	3->4->7->8 (a', b')	Session key request under unauthenticated state

(그림 5) 기능 검증을 위한 테스트 케이스의 예
(Figure 5) Example of Test Cases for Functional Testing

넷째, 테스터가 입력한 정보들을 바탕으로 테스트 케이스를 생성한다. 테스트 트리의 루트 노드로부터 리프 노드까지의 하나의 경로(path)가 하나의 테스트 케이스가 된다. 테스트 케이스에서 입력은 SUT에게 송신하는 메시지이며, 출력(관찰 결과)은 SUT의 상태 변화와 반환하는 메시지가 된다. 테스트 결과는 명세인 MSC와 비교하여 판단한다. 그림 5의 트리에서는 3개의 테스트 케이스가 생성되었으며, 메시지 번호는 MSC에서의 번호를 의미한다. 생성한 테스트 케이스를 수행하면 클라이언트가 KDC로부터 세션 키를 제공받는 기능에 대해서 정상적인 상황과 예외적인 상황에 대해서 검증할 수 있다.

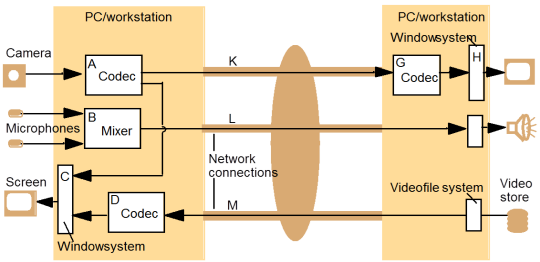
3.2 비기능성 검증을 위한 테스트 케이스 생성

분산 시스템은 확장성(Scalability), 이질성(Heterogeneity), 신뢰성(Reliability), 가용성(Availability), 성능(Performance), 보안(Security), 투명성(Transparency) 등과 같은 다양한 비기능적 요구사항을 갖는다[1]. 하지만 분산 시스템이 이러한 속성들을 모두 충족시키는 것은 쉽지 않다. 왜냐하면 서로 상충(tradeoff) 관계에 있는 요소들이 존재하며, 시스템의 특성에 따라서 달성해야 할 품질 속성이 각각 다르기 때문이다. 따라서 우리는 QoS와 밀접한 관련이 있는 요소들을 검증하는 것을 목표로 한다. 예를 들어, 비디오 스트리밍 서비스를 제공 받는 사용자는 버퍼링에 걸리거나 영상의 깨짐 혹은 재생이 어려운 상황이 생길 수 있다. 왜냐하면 시스템의 자원(CPU, Memory, 네트워

크의 Bandwidth)이 부족하거나 패킷의 손실, 메시지 도착 지연과 같은 다양한 상황이 발생할 수 있기 때문이다. 패킷 손실이나 지연이 발생하는 이유는 다수의 클라이언트 요청이 일시적으로 증가하기 때문이다. 네트워크에서의 패킷 손실이나 지연도 발생하지만, 시스템에서 모든 패킷을 처리하지 못하거나(packet drop) 자원 부족으로 패킷 전송이 늦어질 수 있다. 또한 QoS에서 사용자의 서비스 요청에 대한 응답 시간(Response Time)은 매우 중요하며 다수의 요청을 처리하기 위해서는 높은 처리량(Throughput)과 효율적인 자원의 사용(Utilization)이 필요하다. 따라서 QoS를 달성하기 위한 비기능적인 품질 속성을 확장성과 성능으로 한정한다.

확장성과 성능을 검증하기 위해서는 SUT가 제공하는 서비스(컴포넌트)에 대한 QoS 명세를 기준으로 테스트를 수행해야 한다. 즉 다수 사용자들의 요청에도 SUT가 QoS 명세를 만족하며 서비스를 제공하는지에 대해서 확인해야 한다.

사례 시스템인 원격 화상 회의를 위한 멀티미디어 회의 시스템(Multimedia Conferencing System)은 그림 6과 같은 컴포넌트들로 구성된다[1].



(그림 6) 멀티미디어 회의 시스템의 아키텍처 [1]
(Figure 6) Multimedia Conferencing System Architecture

멀티미디어 데이터는 매우 크기 때문에 전송하는데 큰 대역폭을 요구한다. 하지만 네트워크의 대역 용량은 한계가 있기 때문에 전송하기 전에 압축(Compression)하고 수신하여 이것을 해제(Decompression)하는 코덱(Codec)과 같은 컴포넌트가 필요하다. 코덱을 통해서 대역은 줄일 수 있지만, 한편으로 압축과 해제의 과정이 멀티미디어 스트림의 실시간성에 영향을 줄 수 있다. 또한 클라이언트가 증가하면 다수의 영상, 음성 스트림을 처리해야하기 때문에 그림 6의 각 컴포넌트는 높은 처리량이 요구된다. 따라서 각 컴포넌트에 대해서 표 2와 같은 QoS 명세가 필요하다[1].

(표 2) 멀티미니어 회의 시스템 컴포넌트의 QoS 명세
(Table 2) QoS Specification of Multimedia Conferencing System Components

component	Bandwidth	Latency	Loss Rate	Resources required
A	In : 10frames/sec Out : MPEG-1	Inter-active	Low	10ms CPU each 100ms, 10 Mbytes RAM
B	In : 2 44 kbps Out : 1 44 kbps	Inter-active	Very Low	1ms CPU each 100ms, 1 Mbytes RAM
H	In : various Out : 50frames/sec	Inter-active	Low	5ms CPU each 100ms, 5 Mbytes RAM

각 컴포넌트는 메시지 지연, 손실에 대한 허용 범위와 자원(입출력 대역폭, CPU, Memory)에 대한 요구사항이 있다. 따라서 확장성과 성능을 검증하기 위해서는 다수의 클라이언트가 서비스를 이용할 때에도 요구하는 QoS의 명세를 만족하는지 확인해야 한다. 또한 명세를 만족하지 않으면 시스템 자체의 문제인지 네트워크의 문제인지 그 원인을 알아야 한다. 예를 들어, 코덱의 작업 처리량은 높은데 네트워크에서의 메시지 지연 시간이 긴 경우에는 코덱의 압축률을 높여서 데이터의 대역을 낮춰야 한다. 반대의 경우에는 압축률을 낮춰서 코덱의 처리량을 높여야 한다.

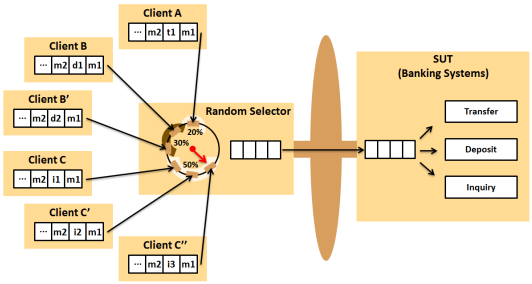
QoS 명세를 기반으로 하여 분산 시스템의 비기능 측면의 검증을 위한 테스트 케이스 생성 방법을 이체(transfer)와 입금(deposit), 조회(inquiry)의 세 가지 서비스를 제공하는 뱅킹 시스템을 예로 들어 설명한다.

첫째 테스터는 클라이언트 그룹을 생성한다. 클라이언트 그룹은 동일한 서비스를 요청하는 클라이언트 집합을 의미한다. 우리는 비기능 요소를 검증하기 위해서 다수의 클라이언트가 여러 가지 서비스를 무작위로 요청하는 테스트 기법을 사용한다. 뱅킹 시스템의 경우 이체와 입금, 조회를 요청할 3가지 그룹(A, B, C)을 생성한다.

둘째 테스터는 SUT가 제공하는 서비스(컴포넌트)에 대한 운영 프로파일(Operational Profile)을 작성한다. 운영 프로파일은 서비스를 요청할 클라이언트 그룹 내의 클라이언트 수와 일반적으로 사용되는 각 서비스의 요청 비율 정보가 담겨 있다. 뱅킹 시스템의 경우 표 3과 같이 작성된다. 따라서 그림 7과 같이 SUT에게 랜덤으로 메시지 전송 시 프로파일 정보를 바탕으로 테스트를 수행한다.

(표 3) 운영 프로파일 기술의 예
(Table 3) Example of Operational Profile Description

Group ID	Service	Request Rate	# of clients
A	Transfer	20%	100
B	Deposit	30%	100
C	Inquiry	50%	100



(그림 7) 테스트 케이스 실행의 예
(Figure 7) Example of Test Case Execution

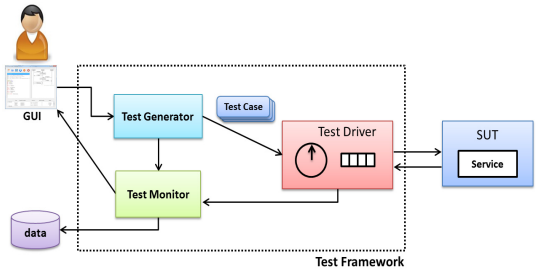
셋째 테스터가 입력한 정보들을 바탕으로 표 4와 같은 테스트 케이스를 생성한다. Test Case 1은 해당 시스템을 사용하는 운영 프로파일 정보에 근거한 비율로 서비스를 요청하며, 나머지 Test Case 2, 3, 4는 특정 서비스의 요청 비율을 급격히 증가하여 테스트를 수행한다. 앞에서 설명한 것처럼 비기능 검증의 목적은 클라이언트의 수와 서비스 요청의 횟수를 증가시키면서 SUT가 QoS 명세를 만족하는 임계치(Threshold)를 확인하고 SUT를 개선하는 데 있다. 따라서 우리는 정상적인 패턴의 서비스 요청보다는 다양한 상황을 발생 시키는 스트레스 테스트 기법을 사용한다. 예를 들어, 뱅킹 서비스에서 가장 많은 리소스를 사용하는 이체(Transfer)에 대한 요청 비율이 높은 상황을 고려할 수 있다. 따라서 단일 서비스에 대해서만 고려했던 기존 연구보다 다양한 상황에 대해서 검증할 수 있다.

(표 4) 비기능 검증을 위한 테스트 케이스의 예
(Table 4) Example of Test Cases for Non-functional Testing

ID	Transfer	Deposit	Inquiry
Test Case1	20%	30%	50%
Test Case2	40%	20%	40%
Test Case3	5%	60%	35%
Test Case4	0%	0%	100%

4. 테스트 프레임워크

서론에서 언급한 것처럼 분산 시스템의 검증을 위해서는 테스트 환경을 구축하고 테스트를 수행하는 데에 많은 노력이 요구된다. 따라서 우리는 테스트 전반적인 과정에서 테스터가 쉽게 테스트 환경을 구축하고, 테스트 수행을 제어하며, 테스트 결과를 확인할 수 있는 테스트 프레임워크를 제안한다. 테스트 프레임워크의 아키텍처는 그림 8과 같다. 이 장에서는 테스트 프레임워크의 컴포넌트가 각 테스트 단계에서 어떻게 동작하는지 설명한다.

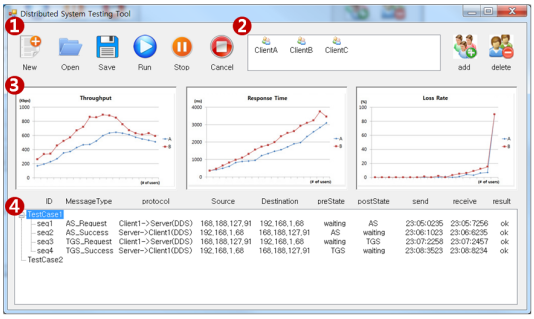


(그림 8) 테스트 프레임워크 아키텍처
(Figure 8) Test Framework Architecture

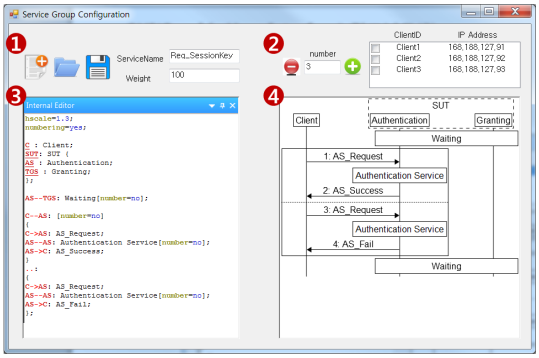
4.1 테스트 준비(Preparation)

테스트 준비는 3장의 방법을 사용하여 테스트 케이스를 생성하는 단계이다. 테스트 케이스를 생성하기에 앞서, 테스터는 그림 9와 같은 GUI(Graphic User Interface)를 통해서 테스트에 필요한 데이터를 입력한다.

테스터는 테스트를 실행, 중지, 종료, 저장 그리고 재수행이 가능하며(①) 동일한 서비스를 요청하는 클라이언트 그룹을 생성, 삭제할 수 있다(②). 테스트를 수행한 결과는 그래프(③)로 표현되며 로그 정보(④)를 통해서 확인할 수 있다. 그림 9에서 생성한 클라이언트 그룹에 대한 정보는 그림 10과 같은 새로운 창을 통해 입력한다. 이 창을 통해 운영 프로파일 정보를 입력(①, ②)할 수 있다. 그리고 기능 테스트를 위한 테스트 시나리오를 MSC를 이용하여 작성하는데 이 부분(③, ④)은 MSC-Generator를 이용하였다.

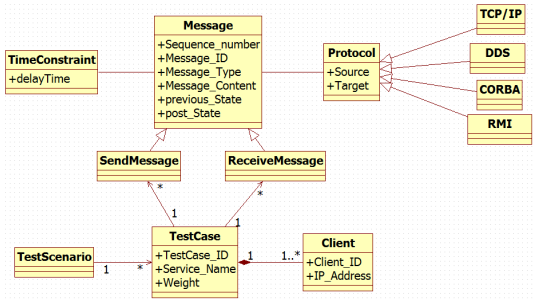


(그림 9) 테스트 프레임워크 GUI 1
(Figure 9) Test Framework GUI 1



(그림 10) 테스트 프레임워크 GUI 2
(Figure 10) Test Framework GUI 2

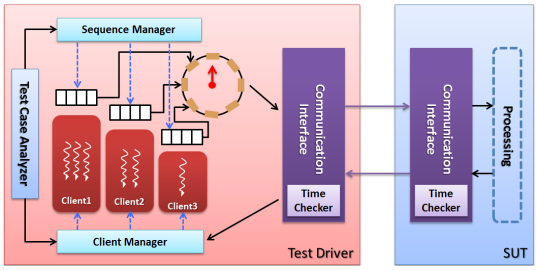
테스트 생성 컴포넌트(Test Generator)는 테스터가 GUI를 통해 입력한 정보들을 바탕으로 테스트 케이스를 생성한다. 테스터가 입력하는 클라이언트 그룹별 테스트 시나리오 정보들을 분석하여 그림 11과 같은 구조의 테스트 케이스를 생성한다. 테스트 시나리오를 구성하는 테스트 케이스는 클라이언트 정보와 MSC로 입력한 송수신 메시지의 내용을 유지한다. 메시지의 타입은 SUT에게 전송하는 메시지(Send Message)와 SUT로부터 수신하는 메시지(Receive Message)로 나뉜다. 공통적으로 MSC에서의 시퀀스 번호, 메시지의 타입과 내용 그리고 메시지 전송 전후의 SUT 상태에 대한 정보를 갖는다. 또한 각 메시지는 지연 시간 설정을 위한 시간제약 정보와 송신자와 수신자간의 통신을 위한 프로토콜 정보를 유지한다.



(그림 11) 테스트 케이스의 구조
(Figure 11) Structure of Test Case

4.2 테스트 실행(Execution)

테스트 드라이버(Test Driver) 컴포넌트는 테스트의 수행을 제어하고, 테스트 결과에 대한 정보를 테스트 관측(Test Monitor) 컴포넌트에게 전달하는 역할을 한다. 우리는 테스트 환경을 구축하기 위해서 실제 클라이언트를 설치하지 않고, SUT의 테스트에 필요한 클라이언트의 메시지 송수신을 시뮬레이션 하는 방법을 사용한다. 이와 같은 방법을 사용하는 이유는 Local Trace 뿐만 아니라 Global Trace도 함께 관리하기 위함이다. 서론에서 언급한 것처럼 분산 시스템은 각 노드의 동작을 제어하는 것이 매우 어렵기 때문에, 테스트 재현의 문제를 갖는다. 이를 해결하기 위해서는 (1)분산 시스템에서 발생할 수 있는 메시지의 흐름을 결정론적인 경로로 도출하여야 하며 (2)분산 시스템을 구성하는 모든 노드에 대한 제어가 가능하여야 한다.



(그림 12) 테스트 드라이버 아키텍처
(Figure 12) Test Driver Architecture

본 논문에서는 테스트 트리를 통해 결정론적인 경로를 도출하고 테스트 드라이버가 모든 노드를 제어할 수 있도록 그림 12와 같은 형태의 테스트 드라이버를 구성한다. 이를 통해서 분산 시스템의 Global Trace를 관리할

수 있기 때문에 분산 시스템이 갖는 비결정적 문제를 해결할 수 있다. 테스트 케이스 분석기(Test Case Analyzer)는 테스트 케이스를 분석하여 메시지 시퀀스와 클라이언트에 대한 정보들을 각각 시퀀스 매니저와 클라이언트 매니저에게 전달한다. 주요 모듈에 대한 설명은 다음과 같다.

- 클라이언트 쓰레드
  클라이언트 쓰레드는 클라이언트의 메시지 송수신 동작을 시뮬레이션 한다. 먼저 송수신 하는 메시지에 대한 시나리오와 메시지 내용은 시퀀스 매니저를 통해서 얻는다. 클라이언트 매니저는 클라이언트 쓰레드를 생성, 관리하며 SUT로부터 수신한 메시지를 해당 쓰레드에게 전달한다. 따라서 각 클라이언트는 Global Trace 정보를 파악하여 SUT와 통신을 수행할 수 있다.

- Random Selector
  비기능 검증 시 스트레스 테스트를 수행하기 위한 모듈이다. 3장 2절의 표 4와 같이 정의된 서비스 요청 비율을 기반으로 동작한다. 즉 일정한 비율로 SUT에게 서비스 요청이 가능하도록 한다.

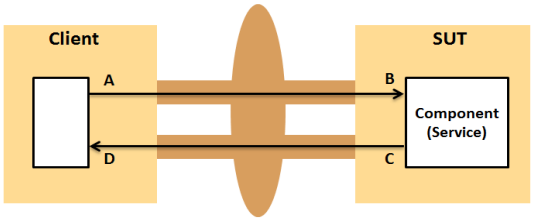
- Time Checker
  메시지의 송신과 수신에 대한 타임스탬프를 기록한다. 기록한 정보들은 메시지의 손실 여부를 확인하고, 메시지 지연과 응답시간, 처리량을 계산할 때 사용된다.

4.3 테스트 평가(Ecaluation)

테스트 관측(Test Monitor) 컴포넌트는 테스트의 수행 결과를 판단하기 위해 데이터를 수집한다. 테스트 케이스에 대한 정보를 바탕으로 실제 송수신되는 메시지에 대한 정보들을 수집한다. 수집한 데이터를 분석하여 GUI에 반영하거나 DB에 저장하며, 기능 테스트의 경우 수행 결과의 성공 실패 유무를 판단하는 오라클(Oracle)의 역할을 수행한다. 분산 시스템의 성능과 확장성과 같은 비기능적 속성을 검증하기 위해서는 기본적으로 많은 노드들을 설치하여 SUT에게 서비스를 요청해야 한다. 그리고 서비스에 대한 응답시간(Response Time)이나 처리량(Throughput)과 같은 요소에 대해 측정해야 한다. 테스트 수행을 통해서 SUT가 QoS의 명세를 만족하는지 확인하고 성능과 확장성을 개선하기 위해서는 다양한 QoS 척도가 필요하다. 우리는 표 5와 같은 네 가지 척도를 사용한다. 그림 13은 클라이언트가 SUT의 서비스(컴포넌트)를 이용하는 상황을 나타낸다. 클라이언트가 SUT에게 서비스를 요청(A->B)하면 SUT는 요청을 처리(B->C)를 하고 처리 결과를 응답(C->D)한다. 멀티미디어 시스템과 같은 경우는 지속적으로 영상을 압축하고 전송하는 과정(B->C->D)가 반복된다.

(표 5) QoS 척도
(Table 5) QoS Metric

Metric	Measurement
Throughput (Kbps, KOps)	Throughput means amount of works performed by the SUT component per unit time. It is measured as the transmission rate at C point.
Loss Rate (%)	Message loss happens in two cases. First, message is lost in the network due to increment of traffic. Second, system cannot send the message due to system malfuction or lack of resources. It is measured between B and D points.
Response Time (ms)	Response time is measured as time interval from the user request to arrival of the response. It is measured between A and D points.
Latency (ms)	Latency means the time delay experienced by a network. It is measured between C and D points.

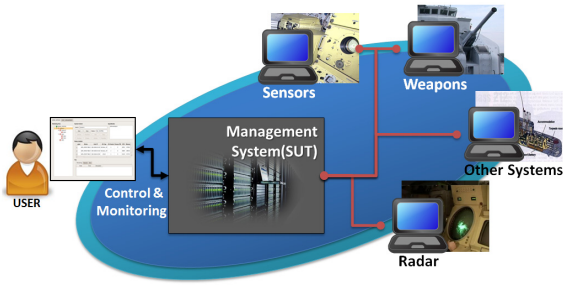


(그림 13) 클라이언트의 서비스 요청 과정
(Figure 13) Service Request Process of Client

5. 테스트 수행 사례

대형 플랜트, 공항, 철도, 전투 체계와 같이 규모가 큰 시스템은 수많은 하위 시스템을 네트워크로 연결하여 전체 시스템을 구성한다. 관리 시스템은 이러한 대규모 분산 환경에서 네트워크에 연결된 하위 시스템의 H/W 장비관리, 동작하는 S/W에 대한 제어 및 감시를 통해 전체

시스템의 운영 효율성을 향상 시킨다. 본 연구에서 제안하는 분산 시스템 테스트프레임워크의 활용을 보이기 위해 테스트 대상 SUT로 분산 환경에서의 관리 시스템을 선정하였다.



(그림 14) 전투 관리 시스템의 예
(Figure 14) Example of Combat Management System

그림 14는 해군 함정에서의 전투 관리 시스템 사례이다. 해군 함정의 전투 관리 시스템은 레이더, 센서, 무기체계 등 다양한 하위 시스템으로 구성되며, 이런 하위 시스템의 정보들을 취합하여 관리하는 역할을 수행한다. 테스트 대상인 관리 시스템은 OMG(Object Management Group)에서 제시한 AMSM(Application Management and System Monitoring for CMS Systems) 명세[18]를 기반으로 개발되었으며 기본적인 요구사항은 다음과 같다.

① 애플리케이션 제어(Control): 관리 시스템은 각 하위 시스템에서 동작하는 S/W를 제어한다. 실행, 종료, 중지, 메모리 적재와 같은 제어 명령을 전송하고 명령을 수행한 결과인 응답 메시지를 수신하여 정보를 유지한다.

② 모니터링(Monitoring): 관리 시스템은 관리 대상의 애플리케이션 및 하드웨어의 감시 기능을 갖는다. 각 노드로부터 S/W 및 H/W의 정보를 주기적으로 수신하여 분석 및 처리한다.

③ 부하 분산(Load Balancing): 전투체계와 같이 전체 분산 시스템의 안전성을 보장하기 위해서는 각 노드의 부하를 관리하여야 한다. 관리 시스템은 애플리케이션을 실행할 노드를 선택할 때, 예측되는 부하를 계산하여 과부하가 발생되지 않도록 조절한다.

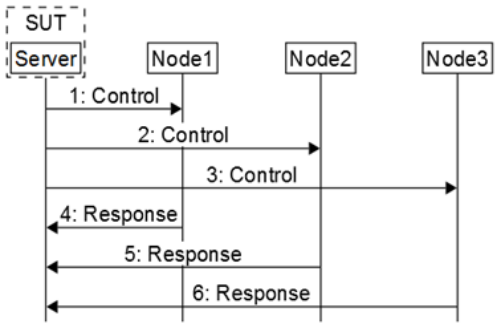
④ 고장 허용(Fault Tolerance): 신뢰성이 요구되는 S/W의 경우 지속적인 서비스를 제공하기 위해서 고장에 대처하는 방법이 필요하다. 관리 시스템은 중복 컴포넌트를 관리하여 고장 시 컴포넌트를 대체하는 방법을 사용한다.

위에서 설명한 각 요구사항을 검증하기 위해서는 많은 세부 기능들을 테스트해야 한다. 각 기능별로 다양한 테스트 시나리오가 존재하며 예외상황을 고려하면 더욱 많은 테스트 케이스가 생성된다. 예를 들어, 애플리케이션 제어에 대한 테스트 시나리오는 표 6과 같다.

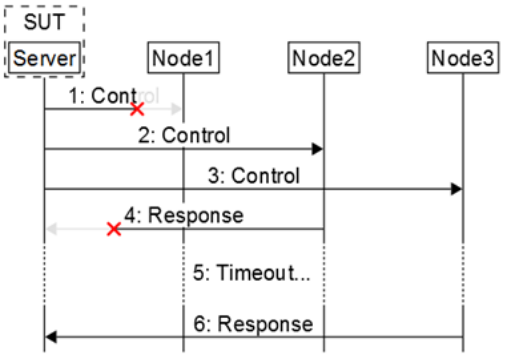
(표 6) 애플리케이션 제어 검증을 위한 테스트 시나리오
(Table 6) Test Scenario to Verify Application Control

Function	Typical Scenario	Alternative Scenario
Execution	1. Transfer execution command to node	Failed to transfer
	2. Execute application on node	Failed to execute
	3. Respond execution result	Failed to respond
Termination	1. Transfer termination command to node	Failed to transfer
	2. Terminate application on node	Failed to execute
	3.Respond termination result	Failed to respond

위와 같은 시나리오들을 테스트하기 위해서 SUT는 노드들과 다양한 메시지들을 주고받아야 한다. 특히 부하 분산이나 고장 허용 같은 경우 더욱 복잡한 메시지 시퀀스를 생성한다. 모든 테스트 시나리오에 대해서 제안한 테스트 기법을 적용하여 테스트를 수행하였으며, 그림 15와 그림 16은 애플리케이션 실행을 테스트하기 위해 작성한 MSC 사례이다.



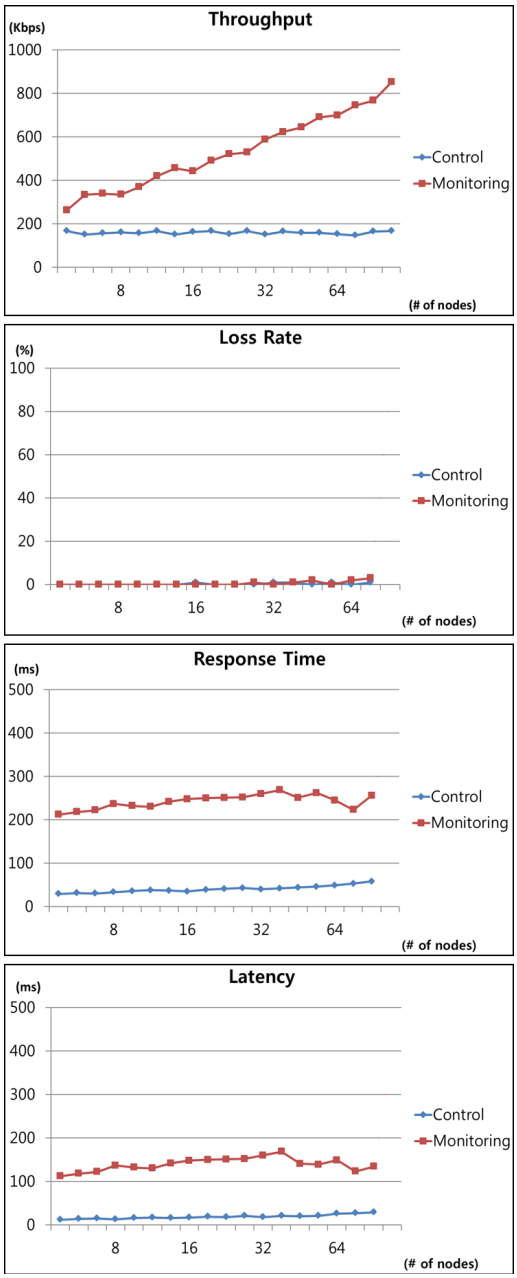
(그림 15) 애플리케이션 실행 시나리오에 대한 MSC 1
(Figure 15) MSC 1 for Application Execution Scenario



(그림 16) 애플리케이션 실행 시나리오에 대한 MSC 2
(Figure 16) MSC 2 for Application Execution Scenario

관리 시스템(SUT)은 각 노드에게 제어 메시지를 전송한다. 그림 15는 정상적으로 제어 메시지를 전달하여 각 노드는 명령을 수행하고 응답 메시지를 전송하는 시나리오이다. 그림 16의 시나리오는 제어 메시지 또는 응답 메시지의 전송이 실패하거나 노드에서 명령 수행을 실패하는 내용이다. 이와 같은 방법으로 관리 시스템의 모든 기능에 대해서 검증을 수행하였다. 100여개의 테스트 케이스를 수행하여 28개의 오류를 검출하였다.

검출된 오류를 수정하여 기능 검증을 완료한 후에, 비기능 검증을 위한 테스트를 수행하였다. 관리 시스템(SUT)은 수많은 노드와 맞물려 동작하여야 하므로, 다수의 노드와 함께 동작하더라도 문제가 없음을 보장하여야 한다. 따라서 본 논문에서는 노드의 개수를 최소 1개부터 최대 128개까지 증가시켜 테스트를 수행하였다. 만일 관리 시스템이 비기능적 측면에서 아무런 문제가 없다면, 몇 개의 노드와 함께 동작하더라도 처리량, 손실률, 지연시간, 응답 시간이 크게 변하지 않아야 한다. 테스트를 수행한 결과는 그림 17의 그래프와 같다. 손실률, 지연시간, 응답 시간을 보면 제어와 모니터링 모두 노드의 개수와 관계없이 일정한 값을 갖는 것을 알 수 있다. 다만, 모니터링에 대한 지연시간과 응답시간의 경우에는 거의 일정한 값을 갖기는 하지만, 제어에 비해 매우 높은 값을 갖는다. 또한 처리량에 대해서도 제어의 경우에는 노드 수에 관계없이 일정한 것에 비해 모니터링은 노드 수가 증가함에 따라 처리량도 함께 증가하고 있음을 알 수 있다. 이는 두 기능의 차이로 인한 결과이다. 제어 기능은 각 노드에 명령을 내린 뒤, 그 처리 결과만을 확인하는데 비해, 모니터링 기능은 주기적으로 각 노드의 상태 정보를 받고 이를 사용자에게 알리기 위해 노드의 상태 정보를 업데이트하는 작업을 수행한다. 즉, 모니터링을 처리하는데 더 많은 리소스를 사용하기 때문에, 그림 17과 같은 결과를 얻은 것이다.



(그림 17) 전투 관리 시스템의 QoS 척도 측정 결과
(Figure 17) Measurement Result of QoS Metric for Combat Management System

비기능 테스트의 수행을 통해서 모니터링의 비기능적 측면의 문제를 확인할 수 있었다. 다만, 손실률의 증가가 없었던 것으로 보아, 모니터링의 지연시간과 응답시간이 높은 값을 유지하고 있는 이유는 관리 시스템에서 이를 처리하는데 많은 시간을 보내기 때문임을 파악할 수 있다. 다른 관점에서 보면, 처리량이 증가함에도 불구하고 지연시간과 응답시간이 일정한 것으로 보아, 테스트가 수행된 환경이 128개의 노드에 대한 작업을 충분히 감당할 수 있는 환경이라 판단할 수 있다. 즉, 비기능 테스트에 대한 충분한 테스트를 수행하기 위해서는 더 많은 노드에 대한 테스트가 필요함을 시사한다. 또한 노드의 수가 적더라도 높은 지연시간과 응답시간을 갖는 것은 모니터링의 처리 작업 자체가 오버헤드가 높다는 것을 의미한다. 따라서 모니터링의 처리 작업에 대한 최적화 작업이 필요하다는 것을 비기능 테스트를 통해서 파악할 수 있다. 반면, 제어의 경우에는 128개까지의 클라이언트에 대해서는 서비스의 요청 비율에 관계없이 안정적으로 동작하는 것을 검증하였다.

6. 결론 및 향후 연구

이 논문에서는 분산 시스템의 기능 및 비기능 검증을 위한 테스트 기법을 제안하고, 제안한 기법을 바탕으로 테스트 프레임워크를 개발하였다. 테스트 프레임워크 설계 시 분산 시스템은 메시지 전달에 의해서 제어되고 동작한다는 점에 주목하였다. 따라서 SUT의 테스트에 필요한 노드들을 실제로 설치하지 않고, 테스트 시나리오대로 각 노드들의 메시지 송수신 동작을 시뮬레이션 할 수 있는 환경을 제안하였다. 이를 통해 본 논문의 테스트 프레임워크는 2장 1절에서 제시한 분산 시스템이 갖는 검증 이슈에 대해서 표 7과 같은 만족도를 갖는다.

“Global Trace의 파악”과 “제어 가능한 노드”는 첫 번째 검증 이슈인 노드의 제어와 관련된 속성이며, “Global Trace의 파악”과 “제어 가능한 노드” 그리고 “테스트 케이스의 자동생성”은 두 번째 검증 이슈인 테스트 수행 결과의 관찰과 관련된 속성이다. 세 번째 검증 이슈인 노드의 설치 및 관리와 관련된 속성은 “SUT와 독립적인 테스트 환경”, “쉬운 테스트 코드의 분배” 그리고 “테스트 케이스의 자동생성”이다. 본 논문의 테스트 프레임워크는 대부분의 속성을 만족한다. 다만, 확장성과 성능에 대해서만 비기능 테스트를 수행하기 때문에 세모로 표시하였다.

(표 7) 테스트 이슈 커버리지
(Table 7) Test Issue Coverage

	[9]	[10]	[7]	[13]	[14]	This Paper
Recognition of Global Trace	X	X	X	O	X	O
Controllable Nodes	X	X	X	O	X	O
SUT-dependent Test Environment	X	X	O	O	O	O
Easy of Test codes Distribution	O	O	O	O	X	O
Automated Generation of Test Cases	X	X	O	X	X	O
Functionality Test	O	O	X	△	X	O
Non-functionality Test	X	X	O	O	O	△

따라서 향후에는 시스템의 가용성 측면을 검증 대상의 품질속성에 추가하여 테스트 프레임워크를 확장하고자 한다. 시스템의 가용성이란 특정 시점에 시스템이 가용될 확률을 의미하며, 시스템의 지속적인 서비스 제공과 밀접한 관련이 있다. 분산 시스템의 가용성을 방해하는 요인으로는 DeadLock, LiveLock, RaceCondition 등이 있다. 분산 시스템의 가용성을 보장하기 위한 여러 가지 기법들이 존재하며 우리는 테스트를 통해서 가용성을 검증하고자 한다.

참 고 문 헌 (Reference)

[1] Coulouris, G., “Distributed Systems: Concepts and Design“, 5th Ed, Addison-Wesley, 2012.

[2] Abbas, H., Hoxha, B., Fainekos, G., ″Conformance Testing as Falsification for Cyber-Physical Systems″, In Proceedings of CoRR, 2014.

[3] Nogueira, S., Sampaio, A., Mota, A., “Test generation from state based use case models“, Formal Aspects Computing, no. 1, pp. 1 – 50, 2012.

[4] Guo, H.-F., Subramaniam, M., “Model-based test generation using extended symbolic grammars“, Int. J. Softw. Tools Technol. Transf., 2014.

[5] Hierons, R., ″Oracles for Distributed Testing″, IEEE Transaction on Software Engineering, Vol. 38, No. 3, pp. 629-641, 2012.

[6] Hierons, R., Ural, H., ″The effect of the distributed test architecture on the power of testing″, The Computer Journal, Vol. 51, No. 4, pp. 497-510, 2008.

[7] Dumitrescu, C., Raicu, I., Ripeanu, M., Foster, I., ″DiPerF: an automated distributed performance testing framework″, Proc. of the 5th IEEE/ACM Int’l Workshop on Grid Computing, pp. 289-296, 2004.

[8] Andre, P., Mottu, J., Ardourel, G., ″Building Test Harness from Service-based Component Models″, MoDeVVa 2013 Workshop on Model Driven Engineering, Verification and Validation, pp. 11-20, 2013.

[9] Wu, J., Yang, L., Luo, X., ″Jata: A Language for Distributed Component Testing″, Proc. of the 15th Asia-Pacific Software Engineering Conference (APSEC), pp. 145-152, 2008.

[10] Tómasson, H., Neukirchen, H., ″Distributed testing of cloud computing applications using the TTCN-3-based Jata test framework″, Proc. of the 2nd Nordic Symposium on Cloud Computing & Internet Technologies, pp. 22-29, 2013.

[11] JUnit [Online], Available:https://github.com/junit-team/junit/wiki, 2014.

[12] ETSI ES 201 873-1, “Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language”, 2014.

[13] Caspar, M., Lippmann, M., Hardt, W., “Automated system testing using dynamic and resource restricted clients”, In Proc. of the conference on Design, Automation & Test in Europe, pp. 1-4, 2014.

[14] Gupta, D., Vishwanath, K., McNett, M., et al, “DIECAST: Testing Distributed Systems with an Accurate Scale Model”, Journal ACM Transactions on Computer Systems (TOCS), vol. 29, no. 2, Article 4, 2011.

[15] Mathur, A., “Foundations of Software Testing“, Pearson Education, 2008.

[16] MSC-Generator, “Tool for Message Sequence Charts“ [Online], Available: https://sites.google.com/site/mscgen2393/, 2013.

[17] Turanyi, Z., “MSC-Generator Manual“ [Online], Available: http://msc-generator.sourceforge.net/help/4.0/msc-gen.pdf, 2014.

[18] OMG, “Application Management and System Monitoring for CMS Systems(AMSM)“ [Online], Available: http://www.omg.org/spec/AMSM, 2010.

◐ 저 자 소 개 ◑

윤 상 필 (Sangpil Yun)
2012년 충남대학교 컴퓨터공학과 졸업(학사)
2014년 충남대학교 컴퓨터공학과 졸업(석사)
2014년~현재 지멘스 코리아 헬스케어 부분 연구원
관심분야 : 소프트웨어 테스팅, 분산 컴퓨팅 개발 및 검증
E-mail : sangpil.yoon@siemens.com

서 용 진 (Yongjin Seo)
2011년 충남대학교 컴퓨터공학과 졸업(학사)
2011년~현재 충남대학교 컴퓨터공학과 (박사 과정, 석박사통합)
관심분야 : 소프트웨어 테스팅, 스마트폰, UX/UI
E-mail : yjseo082@cnu.ac.kr

민 법 기 (Bup-Ki Min)
2009년 공주대학교 산업정보학과 졸업(학사)
2012년 충남대학교 컴퓨터공학과 졸업(석사)
2012년~현재 충남대학교 컴퓨터공학과 (박사 과정)
관심분야 : 소프트웨어 테스팅, 소프트웨어 신뢰성, 분산 시스템
E-mail : bkmin@cnu.ac.kr

김 현 수 (Hyeon Soo Kim)
1988년 서울대학교 계산통계학과 졸업(학사)
1991년 한국과학기술원 전산학과 졸업(석사)
1995년 한국과학기술원 전산학과 졸업(박사)
1995년~1995년 한국전자통신연구원 Post Doc.
1996년~2001년 금오공과대학교 조교수
2001년~현재 충남대학교 컴퓨터공학과 교수
관심분야 : 소프트웨어 공학, 소프트웨어 테스팅, 소프트웨어 아키텍처
E-mail : hskim401@cnu.ac.kr