유비쿼터스 시스템의 중단 없는 서비스를 위한 경량화된 결함복구 유틸리티

현재명° 최창열, 김성수
아주대학교 정보통신전문대학원
{jmhyun78°, clchoi, sskim}@ajou.ac.kr

Light-Weight Recovery Utilities for Seamless Service in Ubiquitous System

Jaemyung Hyun°, Changyeol Choi, Sungsoo Kim
Graduate School of Information and Communication, Ajou University

요 약

유비쿼터스 시스템(Ubiquitous System)이 가지는 특징인 무선 네트워크, 제한된 자원 등으로 인하여 결함이 발생할 가능성이 높다. 이런 상황에서 결함이 발생하여도 계속적으로 사용자에게 서비스를 제공하기 위해서는 결함 복구방법이 필요하며 가장 많이 쓰이는 방법으로는 마이그레이션(Migration)이 있으나 마이그레이션 자체가 가지는 오버헤드(Overhead)로 인하여 유비쿼터스 시스템에서는 사용하기 어려운 문제점을 가지고 있다. 본 논문에서는 이러한 문제점을 해결하기 위하여 경량화되어 시스템에 적은 오버헤드를 가지며 빠르게 결함을 복구할 수 있는 AHU(Autonomic Healling Utilites)를 개발하였다. AHU는 AHUServer, AHUClient, AHUBackup으로 구성되어 있으며 JDI(Java Debugging Interface)를 사용하여 사용자 어플리케이션(Application)의 필요 정보만을 추출하여 저장하고 최소한의 노력으로 사용자 시스템 또는 어플리케이션에 결함이 발생하여도 사용자의 통제없이 빠른 시간내에 서비스가 중단된 시점부터 계속적으로 서비스를 제공한다.

1. 서 론

유비쿼터스 시스템이 점차 발전하고 이에 대한 사용자가 점차 증가함에 따라 많은 사람들이 유비쿼터스 시스템의 혜택을 받고 있으며 우리 주위에서 쉽게 접할 수 있는 기술이 되었다[1]. 우리는 이 기술을 통하여 언제 어디서나 우리가 원하는 서비스를 받을 수 있는 편리함을 누리고 있지만 기술 자체가 가지는 복잡성으로 인하여 이러한 기술을 많이 접해보지 못한 사람들에게는 오히려 큰 혼란을 야기할 수 있다. 그러므로 이러한 기술은 전문적인 교육을 받지 않은 일반인들도 쉽게 사용할 수 있도록 단순함을 가져야 하며 사용하기 간편해야 한다. 또한 사용자는 빠르고 즉각적인 서비스를 원한다. 어떤 이유를 불문하고 서비스 지연은 사용자로 하여금 서비스가 제대로 동작하지 않고 있다고 생각하게 할 수 있으며 이로 인하여 서비스에 대한 신뢰성을 잃을 수 있기 때문이다.

그러나 유비쿼터스 시스템은 무선 통신 환경과 제한된 자원으로 인하여 결함이 발생할 확률이 존재한다. 만약 결함이 발생하였을 경우 사용자가 원하는 것은 단지 빠른 시간 내에 다시 서비스를 사용하는 것이다. 사용자는 결함을 복구하는 과정이나 메커니즘에 대해서는 관심이 없으며 또한 복구를 위해서 사용자 자신이 어떠한 처리를 하는 것을 원하지 않는다. 그러므로 유비쿼터스 시스템에 결함이 발생하였을 경우 가장 빠른 시간내에 사용자의 참여없이 사용중인 서비스를 계속적으로 제공할 수 있는 방법이 필요하다.

본 연구는 21세기 프론티어 연구개발사업의 일환으로 추진되고 있는 정보통신부의 유비쿼터스컴퓨팅 및 네트워크 원천기반기술개발 사업의 지원에 의한 것임.

계속적으로 서비스를 사용자에게 제공하기 위한 기본적인 방법으로 마이그레이션이 있으며 이는 결함이 발생한 시스템에서의 프로세스를 다른 시스템으로 이동시켜 계속적으로 사용자에게 서비스를 제공하는 것이다. 그러나 마이그레이션에서의 가장 큰 문제점은 메커니즘의 복잡성과 시스템의 오버헤드이다[2,3]. 많은 종류의 마이그레이션 방법이 제한되었지만 자원이 제한된 유비쿼터스 시스템에서 사용되기에는 부적합하다. 마이그레이션의 대안으로 일반 마이그레이션 방법보다 오버헤드가 적고 빠르게 재 시작할 수 있는 원격 실행(remote execution) 방법이 있으나 이 방법은 사용자의 서비스 제공 시점을 기억하지 못하므로 계속적인 서비스를 제공할 수 없다[2]. 그러므로 본 논문에서는 유비쿼터스 시스템에서 결함이 발생할 경우 사용자의 참여없이 빠르게 사용자의 서비스를 계속적으로 제공할 수 있는 유틸리티를 제안하고자 한다.

본 논문에서 제안하는 유틸리티는 사용자가 사용 중인 시스템을 항시 감시하고 결함이 발생할 경우 사용 중인 서비스 어플리케이션의 쓰레드(Thread) 정보를 저장하고 이를 새로운 시스템에서 다시 시작한 사용자 어플리케이션에 재저장하여 사용자로 하여금 서비스가 중단된 시점부터 계속적으로 서비스를 받을 수 있게 한다. 일반 마이그레이션 방법과 달리 사용자 어플리케이션의 모든 정보를 이동시키는 것이 아니라 쓰레드 정보만을 이동시키므로 일반적인 마이그레이션 방법보다 오버헤드가 적으며 원격 실행 방법과 달리 사용자의 서비스 중단 시점을 기억하므로 계속적인 서비스를 사용자에게 제공할 수 있다. 본 논문에서는 이 유틸리티를 AHU(Autonomic Healing Utilities)라 부르고자 한다.

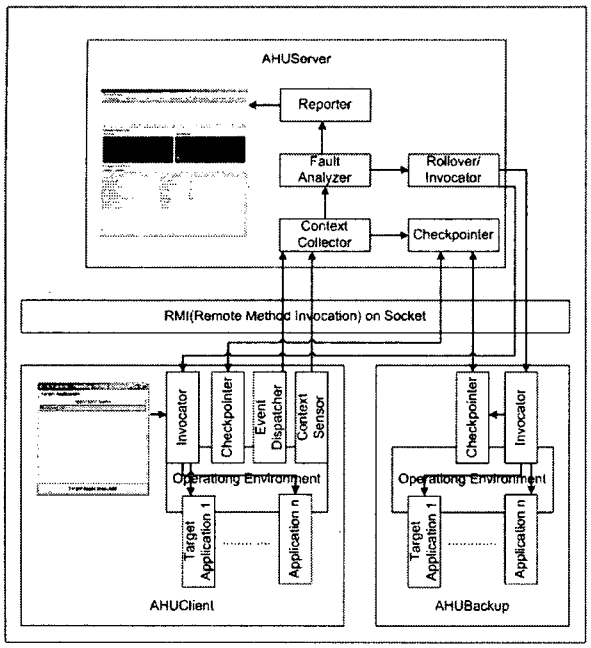

그림 1. AHU 구조도

2. AHU(Autonomic Healing Uitilites) 구조

그림 1은 AHU의 구조를 나타내며, 크게 AHUServer, AHUClient 그리고 AHUBackup으로 구성되어 있다. AHUServer는 사용자의 시스템과 서비스 어플리케이션을 감시하고 결함이 발생할 경우 이를 감지한다. AHUServer가 결함을 감지하면 주기적으로 저장된 사용자 어플리케이션의 쓰레드 정보를 AHUBackup으로 이동시켜 사용자가 계속적으로 서비스를 사용할 수 있도록 한다. AHUClient는 사용자 시스템에서 사용되며 사용자 시스템과 어플리케이션의 정보를 수집하여 주기적으로 AHUServer로 전송하는 역할을 한다. AHUBackup은 추가적인 서버로 사용자의 시스템이나 어플리케이션에 결함이 발생할 경우 사용자는 AHUBackup를 통해 계속적으로 서비스를 사용할 수 있다.

2.1 AHUServer

AHUServer는 다음과 같이 구성되어 있다
Context Collector : 이 컴퍼넌트는 AHUClient의 Event Dispatcher와 Context Sensor로부터 전송되는 정보(CPU 사용량, 메모리 사용량, 디스크 사용량, 네트워크 사용량)를 수집하여 정리한다.
Fault Analyzer : Context Collector가 수집한 정보를 이용하여 현재 사용자의 시스템과 사용자 어플리케이션의 상태를 파악하고 결함을 감지하는 역할을 한다.
Rollover/Invocator : 사용자 시스템 또는 사용자 어플리케이션에서 결함이 발견되었을 경우 AHUBackup에서 사용자 어플리케이션을 다시 시작하게 하여 계속적으로 사용자가 서비스를 사용할 수 있도록 한다.
Checkpointer : AHUClient의 checkpoint가 수집한 사용자 어플리케이션 정보를 전송받아 저장하며 결함 발생시 이 정보를 AHUBackup로 전송하여 사용자가 계속적인 서비스를 제공할 수 있도록 한다.
Reporter : AHUClient로부터 수집된 사용자 시스템과 어플리케이션 상태정보를 관리자가 볼 수 있도록 표시한다.

2.2 AHUClient

AHUClinet는 다음과 같이 구성되어 있다
Invocator : 최초 사용자가 계속적인 서비스를 받길 원하는 어플리케이션을 등록하여 실행시키며 복구과정에서도 AHUServer의 Rollover/Invocator는 이 컴퍼넌트와의 교신을 통하여 복구를 완료시킨다.
Event Dispatcher : 사용자가 지정한 어플리케이션의 등록 여부를 검사하여 AHUServer로 전송한다.
Checkpointer : 주기적으로 사용자 어플레케이션의 정보를 저장하여 결함이 발생하였을 경우 사용자의 어플리케이션이 중단된 시점부터 다시 시작할 수 있도록 한다.
Context Sensor : 지정된 어플리케이션에 관한 정보를 수집하여 AHUServer의 Context Collector로 전송한다.

2.3 AHUBackup

AHUBackup은 다음과 같이 구성되어 있다
Invocator : AHUServer가 사용자 시스템에 대한 결함을 감지하면 이 컴퍼넌트를 통하여 사용 중이던 어플리케이션을 실행한다.
Checkpointer : AHUServer에 저장되어 있던 사용자 어플리케이션 정보를 새로 시작된 어플리케이션에 재저장하여 서비스가 중단 되었던 시점부터 다시 서비스를 사용할 수 있게 한다.

```
for (int i=0 ; I < threads.size() ; i++) {
 try {
    threadReference  thread = threads.get(i);
    thread.suspend();
    List frames = thread.frames();
    // read all stack frames
    for(...){
      if(d.threadName.compareTo(thread.name()) == 0
      && d.typeName.compareTo(var.typeName()) ==
      0 && d.name.compareTo(var.name()) == 0){
       // recover the value
      }
      // checkpointing the target thread stack frame
      if(recovered == false){
        Object rObj = makeObject(...);
        if(rObj != null){
          XStream xStream = new XStream();
          rData r = new rData(...); } } } } }
```

의사코드 1. 사용자 어플리케이션 정보 추출 코드

3. 어플리케이션 정보저장

자바를 이용한 많은 마이그레이션 방법에서 가장 문제점으로 지적되는 것이 마이그레이션 하기 위해서는 자바 버추얼 머신(Java Virtual Machine)을 수정하거나 라이브러리(Libray)를 수정하여야 한다는 것이다. 기존의 JDI를 사용하여 마이그레이션 하는 방법 또한 정보를 추출하여 재저장하기 위해서는 자바 버추얼 머신을 수정하여야 한다[4]. 그러나 AHU의 JDI를 이용한 방법은 쓰레드에 대한 정보 일부만을 저장한다. 의사코드 1이 보여주는 것은 AHUClient에서 사용자 어플리케이션 정보를 저장하는 의사코드로 JDI를 통하여 현재 쓰레드에서 사용되는 파라미터에 대한 값을 추출하고 이를 오브젝트(Object) 형태로 저장한다. 그러므로 일반 마이그레이션 방법에서 어플리케이션의 모든 정보를 저장함으로써 발생하는 오버헤드를 줄일 수 있다[5].

4. 성능평가

AHU의 성능을 측정하기 위하여 AHUServer로 일반 데스크톱 PC Pentium(R) 4 CPU 2.80GHz를 512M RAM을 사용하였고 AHUClient로 노트북 PC Pentium(R) Processor 1.7GHz M 496M RAM을 사용하고 무선네트워크를 통하여 AHUServer에 접속하였다. 측정 방법은 AHUClient수를 증가시키면서 AHUBackup으로 복구되는 시간과 다시 원래의 시스템으로 복구되는 시간을 측정한 것이다. AHUBackup에서 복구가 완료되는 시간은 AHUServer에서 AHUClient의 결함을 인지한 시점을 기준으로 시간을 측정한 것이다. 결함이 발생한 후에 시스템 또는 어플리케이션이 바로 서비스를 중지하는 것이 아니라 일정 시간동안 계속적으로 서비스를 수행한다. AHUClient에서 다시 복구가 완료되는 시간은 일정시간(약20초)간 사용자가 AHUBackup에서 서비스를 사용한 후 다시 복구되는 시간이다. 그러므로 그림 2에서 나타내는 그래프는 절대적인 복구 시간을 의미하는 것은 아니며 AHU를 통해서 사용자는 서비스가 중단되는 순간 바로 다른 시스템을 통하여 계속적으로 서비스를 사용할 수 있다.

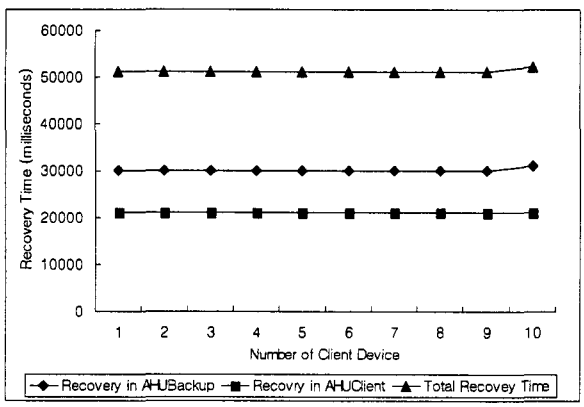

그림 2. 클라이언트 수에 따른 복구 시간

그림 2에서 보여주는 것과 같이 AHUServer에 접속한 AHUClient의 수가 증가하여도 AHU 유틸리티는 시스템에 거의 영향을 미치지 않으므로 복구 속도의 변화가 거의 없는 것을 보여주고 있다. 이것은 AHU가 경량화되어 있으며 오버헤드를 가지지 않는다는 것을 의미한다.

5. 결론

본 논문에서는 사용자의 계속적인 서비스를 위한 방법인 기존 마이그레이션 방법이 가지는 오버헤드로 인한 유비쿼터스 시스템에서의 사용이 어려운 점을 해결하기 위하여 AHU를 개발하였다. AHU는 사용자의 어플리케이션 또는 시스템에서 결함이 발생할 경우 사용자에게 계속적으로 서비스를 제공하기 위하여 사용자 프로세스의 필수적인 정보만을 서버에 저장하여 결함 발생 이후에서 AHUBackup에서 사용자의 중단된 시점부터 계속적으로 서비스를 제공할 수 있게 하였다.

AHU는 경량화된 유틸리티로 시스템에 오버헤드를 만들지 않으므로 유비쿼터스 환경에서 많은 수의 클라이언트가 동시 접속하여도 이를 처리하는데 적은 오버헤드를 가지며 또한 빠르게 사용자의 서비스를 다시 사용할 수 있게 한다. 유비쿼터스 환경이 점차 현대 생활 속에 필수적인 존재로 자리 잡아 가는 현 시점에서 AHU는 사용자의 서비스를 보장하므로 많은 이익을 줄 수 있을 것이라 기대한다.

참고문헌

[1] M. Brugnoli, et al., "User Expectations for Simple Mobile Ubiquitous Computing Environments," Proceedings of the 2nd Workshop on Mobile Commerce and Services, pp. 2-10, July 2005.

[2] D. Milojicic, et al., "Process Migration Survey," Proceedings of the ACM Computing Surveys, pp. 1-49, 2000.

[3] K. Takasugi, et al., "Seamless Service Platform for Following a User Movement in a Dynamic Network Environment," Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications, pp. 71-78, September 2003.

[4] T. Illmann, et al., "Transparent Migration of Mobile Agents Using the Java Platform Debugger Architecture," Proceedings of the 5th IEEE International Conference on Mobile Agents, pp. 198-212, December 2001.

[5] Jon Howell, et al., "Straightforward Java Persistence through Checkpointing In Advances in Persistent Object Systems," Proceedings of the 3rd International Workshop on Persistence and Java, pp. 322-334, January 1999.