안정적인 대용량 I/O거래 처리를 위한 Peak Load Control(PLC) 기반의 Worker-Linker 패턴

(A Peak Load Control-Based Worker-Linker Pattern for Stably Processing Massive I/O Transactions)

이 용 환 † 민 덕 기 ††
(Yong hwan Lee) (Dug Ki Min)

요 약 EAI나 B2BI와 같은 통합 어플리케이션들은 짧은 시간 동안의 서비스 요청 폭주로 인한 과부하시에도 대용량의 I/O기반 트랜잭션들을 안정적으로 처리할 수 있는 신뢰성 있는 시스템을 필요로 한다. 본 논문에서는 I/O기반 시스템에서 대용량의 거래를 효과적으로 처리할 수 있으며 짧은 시간 동안의 거래 요청 폭주로 인한 과부하시에도 안정적으로 서비스를 제공할 수 있는 Peak Load Control(PLC)기반의 Worker-Linker 패턴을 제시한다. 본 논문에서는 PLC 기법을 위해 지체시간(Delay Time) 알고리즘을 사용한다. 또한 제안한 알고리즘이 가지는 과부하 시의 성능 안정성을 증명하기 위해 본 논문에서는 Worker-Linker 패턴을 실제 B2BI 시스템에 적용해서 성능 안정성을 증명하기 위한 실험 결과를 제시한다. 실험 결과에 의하면 본 논문에서 제안한 지체시간 알고리즘은 과부하 상태를 안정적으로 통제하는데 효과가 있다.

키워드 : 최대부하조절, 기업간 통합, 통합 시스템, 성능, 통합 프레임워크, 성능 안정성, 신뢰성

Abstract Integration applications, such as EAI, B2Bi, need stable massive data processing systems during overload state cause by service request congestion in a short period time. In this paper, we propose the PLC (Peak Load Control)-based Worker-Linker pattern, which can effectively and stably process massive I/O transactions in spite of overlaod state generated by service request congestion. This pattern uses the delay time algorithm for the PLC mechanism. In this paper, we also show the example of applying the pattern to business-business integration framework and the experimental result for proving the stability of performance. According to our experiment result, the proposed delay time algorithm can stably control the heavy overload after the saturation point and has an effect on the controlling peak load.

Key words : Peak Load Control, B2Bi, Integration System, Performance, Integration Framework, Performance Stability, Reliability

1. 서 론

인터넷의 진화는 기업들이 그들의 파트너와 상호작용할 수 있는 새로운 길을 제시했다. 기업들은 인터넷을 기반으로 비즈니스나 부가가치 서비스를 확장할 수 있도록 하기 위해 많은 인프라들과 정보시스템들을 개발했으며 기업간 통합 시스템을 통해 외부 기업과 연동함으로서 새로운 부가가치 서비스들을 제공하고 있다. 이러한 인터넷의 발전과 함께 통합 시스템은 짧은 시간 동안의 대량의 서비스 요청 폭주로 인해 불안정성을 초래하는 사례가 많아지고 있다[1].

짧은 기간 동안의 대량의 거래 폭주를 안정적으로 처리하기 위해 많은 기업에서는 분산 클러스터링이나 하드웨어 증설과 같은 방법을 사용한다. 하지만 이 방식은 비용 효과적인 측면에서 1년에 단 몇번 혹은 몇일만 이러한 현상이 발생하는 시스템에 너무 많은 비용을 지불해야 하기 때문에 좋은 방법이 아니다. 이에 대한 방안으로 하드웨어 벤더에서 제공하는 On-Demand 서비스를

† 정 회 원 : 동덕여자대학교 컴퓨터학과 교수
yhlee@konkuk.ac.kr
†† 정 회 원 : 건국대학교 소프트웨어학과 교수
dkmin@konkuk.ac.kr
논문접수 : 2006년 5월 2일
심사완료 : 2006년 7월 21일

사용할 수 있다. 이들 서비스는 필요한 만큼만 사용하고 사용한 부분에 대해서만 요금을 지불하는 방식이라는 측면에서 좋은 대안이 될 수 있지만 관리적 포인트의 증가가 예상외로 크고, 장애발생 시 원인 규명이 쉽지 않다. 또한 소프트웨어 가격 책정이 CPU 개수에 의해서 책정되기 때문에 가격이 상승하게 되는 단점들이 있다.

특점 시점에 서비스 요청 폭주로 인해 시스템 과부하가 발생할 때 비용 효과적인 측면에서 처리 속도는 약간 희생하더라도 시스템이 다운되거나 불능사태에 빠지는 것을 예방하는 것이 바로 Peak Load Control(PLC)이다. PLC는 현재 구성된 시스템이 최대로 수용할 수 있는 한도까지는 정상적으로 서비스 되지만, 그 이상을 넘어설 경우는 서비스를 제한하거나 처리하는 속도를 조절함으로 부하를 조절하는 기법이다[2].

본 논문에서는 대용량의 I/O기반 거래들을 효과적으로 처리하며 특점 시점에 요청 폭주로 인한 시스템 과부하시에 부하를 조절할 수 있는 Worker-Linker 패턴을 제시한다. 제시한 Worker-Linker 패턴은 I/O관련 거래 처리 효과성을 위해 기존의 Connector-Acceptor 패턴[3], Worker 패턴[2], Command 패턴, Reactor 패턴[4], 그리고 Proactor 패턴[5]등을 조합해서 만들었으며 기존 Worker 패턴에 PLC 메커니즘을 추가했다. 특점 시점에 서비스 폭주로 인한 과부하를 통제하기 위해 기존의 다른 PLC 방법은 쓰레드의 수를 제한하거나 분석 모델링 기법을 사용하였지만[6,7] 본 논문에서는 각 Worker 쓰레드가 수행해야 할 지체 시간(Delay Time) 기법을 사용하여 과부하를 통제하고 있다. 본 논문에서는 몇 개의 인자를 가지고 이러한 지체 시간을 계산하는 알고리즘과 이를 설계 및 구현하는 방법에 대해서 제시한다. 더욱이 본 논문에서 제시한 Worker-Linker 패턴은 Command 패턴 형태로 UTL기반 비즈니스 컴포넌트들을 실행할수 있어 비즈니스 로직에 대한 기능 확장성과 유연성을 달성할 수도 있다. 제시한 PLC기반의 Worker-Linker 패턴이 가지는 성능상의 안정성을 증명하기 위해 본 논문에서는 대용량의 I/O 처리 효과성과 안정성을 필요로 하는 기업간 통합 시스템 프레임워크에 이 패턴을 적용했으면 이를 기반으로 성능 실험을 수행하였다. 성능 실험 결과에 의하면 본 논문에서 제시한 PLC기반의 Worker-Linker 패턴은 특점 시점에 서비스 폭주로 인한 과부하시에 안정적으로 거래 처리를 처리할 수 있게 해준다[8].

본 논문은 다음과 같이 구성된다. 먼저 2장에서는 본 논문에서 제시한 Worker-Linker 패턴이 사용하고 있는 기존 패턴들을 설명하고 3장에서는 본 논문에서 제안하는 Worker-Linker 패턴에 대해 기술한다. 4장에서는 Worker-Linker 패턴을 대용량 I/O처리를 안정적 제공해야 하는 기업간 통합 프레임워크에 적용한 예제를 제시하며 5장에서는 제시한 프레임워크의 처리량과 통합 프레임워크의 안정성에 대한 성능결과를 보여준다. 마지막으로 6장에서는 결론을 기술한다.

2. 관련연구

2.1 Acceptor-Connector 패턴

Acceptor-Connector패턴[9]은 소켓(Socket)과 커넥션(Connection)과 같은 네트워크에 관한 세부 사항을 분리함으로서 복잡한 네트워크 세부사항으로부터 어플리케이션을 독립시키기 위한 패턴이다. 그림 1은 이러한 Acceptor-Connector 패턴에 대한 구조이다. Transport

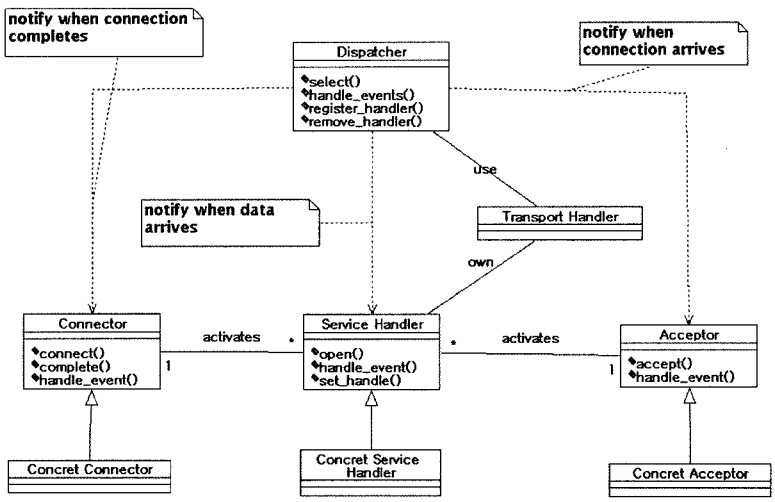

그림 1 Acceptor-Connector 패턴 구조

Handle은 통신을 위한 정보를 캡슐화한 클래이며 Acceptor는 Dipatcher로부터 커넥션이 들어올 경우 수동적으로 해당 커넥션을 받아서 처리한다. Dispatcher는 클라이언트로부터 통신 이벤트를 받아서 통신 관련 I/O 이벤트들을 역다중화(Demultiplexer)하는 책임을 가지고 있다. 반면에Service Handler는 Acceptor나 Connector의 서비스 초기화 후에 사용자가 구현한 작업 처리 루틴을 수행하는 책임을 가진다.

그림 2는 Acceptor-Connector에서 Acceptor에 대한 순차 다이어그램이다. 이러한 Acceptor는 3가지 국면(Phase)[9]으로 나누어 볼 수 있는데 첫번째 국면은 종점(Endpoint) 초기화로서 Acceptor는 Dispatcher에 자신을 등록하고 클라이언트로부터 연결요청을 동기적으로 기다리는 단계이다. 두번째 국면은 서비스 초기화 국면으로서 클라이언트로부터 커넥션 요청이 있을 경우 Dispatcher에 등록되어 있는 Acceptor에게 커넥션 요청이 왔다는 것을 알려준다. 그런 후 Acceptor는 각 통신 프로토콜에 맞는 Transport Handler객체와 서비스를 수행할 Service Handler객체를 생성해서 Dipatcher에게 등록함으로서 서비스 초기화를 수행한다. 세번째 국면은 커넥션 완료 후 데이타 송수신에 해당하는 서비스 처리 국면으로 Dispatcher는 클라이언트로부터 데이타를 받아 등록된 Service Handler를 통해 사용자가 구현한 루틴을 처리한다.

그림 3은 Connector의 비동기적인 연결을 위한 순차 다이어그램이다.

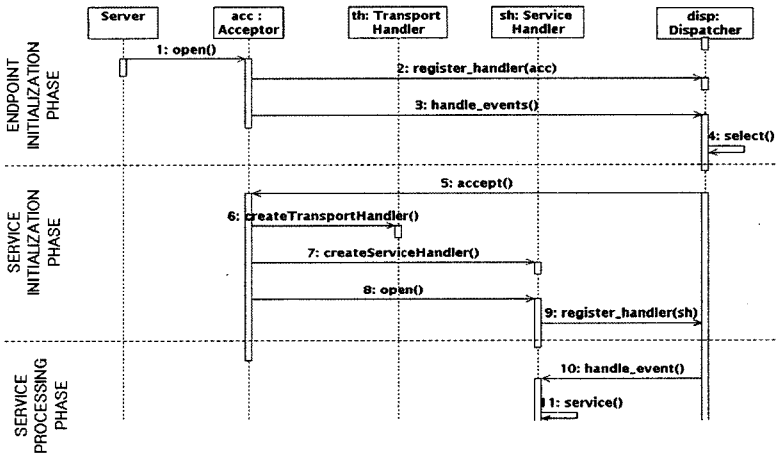

그림 2 Acceptor의 순차 다이어그램

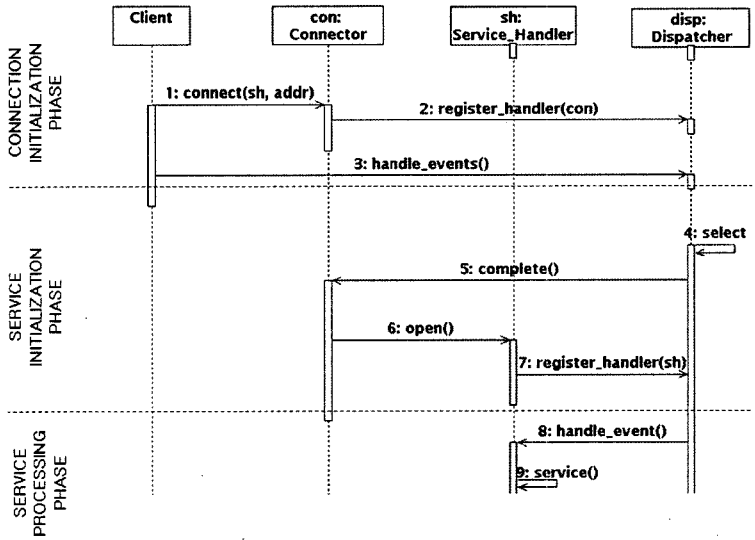

그림 3 Connector의 비동기적인 연결을 위한 순차다이어그램

클라이언트가 Connector에 연결 요청을 할 때 Connector는 Service Handler와 원격지 객체간의 연결을 동기적 혹은 비동기적으로 수행할 수 있다. 커넥션 초기화 국면에서 Connector는 자신을 Acceptor에 등록한 후에 커넥션이 완료되기를 기다린다. 서비스 초기화 국면에서 커넥션이 완료 된 후에 Dispatcher는 Connector의 'complete'라는 메소드를 호출해 해당 Service Handler를 Dispatcher에 등록한다. 서비스 처리 국면은 통신 당사자 간에 데이타를 주고 받을 때 처리하는 것으로서 Dispatcher는 해당 데이타를 받은 후에 등록된 Service Handler에게 'handle_event' 메소드를 호출해 해당 데이타를 처리하도록 한다.

2.2 Reactor와 Proactor패턴

Dispatcher부분의 I/O이벤트들에 대한 역다중화 방식에 따라 Acceptor-Connector 패턴은 Reactor 패턴과 Proactor 패턴으로 분류할 수 있다. 그림 4는 Reactor 패턴 방식의 순차 다이어그램이다.

Acceptor-Connector 패턴의 Dispatcher에 해당하는 부분이 바로 Reactor, Event Demultiplex, 그리고 Event Queue이다. Event Demultiplex는 Event Queue에 다중화 되어 있는 이벤트들을 역다중화(Demultiplex)한다. Reactor는 이벤트 타입과 이벤트 핸들러 쌍의 형태로 역다중화 매핑 테이블을 가지고 있어서 실제 해당 이벤트가 발생할 때 그 이벤트 타입에 해당하는 이벤트 핸들러를 호출한다.

Proactor패턴은 Event Demultiplex로부터 핸들(Handle)을 받을 때 Reactor처럼 동기적으로 받지 않고 비동기적으로 받는다. Proactor는 Reactor처럼 해당 핸들을 받을 때까지 블록(Block)상태에 빠지지 않고 다른 작업을 수행하다. 즉, Event Demultiplex는 역다중화하는 동안에 해당 이벤트 타입을 발견하면 Reactor에게 알려주는 비동기적인 방식으로 이벤트를 처리한다.

Reactor 패턴이 이벤트 접수와 처리를 각각 분리할 수 있다는 장점이 있다면 Proactor 패턴은 Event Queue에서 I/O이벤트 처리를 비동기적으로 처리할 수 있다. Proactor 패턴에서 Event Queue에 다중화되어 있는 I/O 이벤트들을 처리하기 위해서는 핸들(Handle), 핸들러(Handler), 그리고 이벤트 타입(Event Type)가 같은 칼럼(Column)을 가지는 역다중화 테이블이 필요하다. Proactor 패턴에서는 이러한 역다중화 테이블을 사용해 비동기적 방식으로 Proactor에게 해당 핸들을 전송한다. Proactor 패턴이 Reactor 패턴보다 통신관련 자원 소비량이 적기 때문에 일반적으로 좀더 높은 성능을 제공할 수 있지만 Proactor 패턴은 공유자원들에 대한 동기화를 제공해야 한다는 단점도 있다.

2.3 Worker 쓰레드 패턴

쓰레드를 분류하면 태스크(Task)기반의 쓰레드와 액터(Actor)기반의 쓰레드로 분류해 볼 수 있다[10]. 태스크 기반의 쓰레드들은 엔진(Engine)과 같이 외부 이벤트에 반응해서 다시 다른 액터와 상호 작용하는것에 관점이 있으며 태스크 기반의 쓰레드들은 작업, 서비스, 그리고 계산과 같은 비동기적인 방식으로 작업들을 효율적으로 처리하는 것에 관점이 있다.

그림 5는 액터 기반 쓰레드 예제로서 Thread-Per-Message 모델[10]이다. 클라이언트로부터 각각의 요청 메시지에 하나의 호스트 쓰레드를 할당해 커넥션을 처리하고 Helper를 사용해서 실제 요청에 대한 로직을 처리한다. 이 방식은 Helper에서 처리하는 로직의 처리시

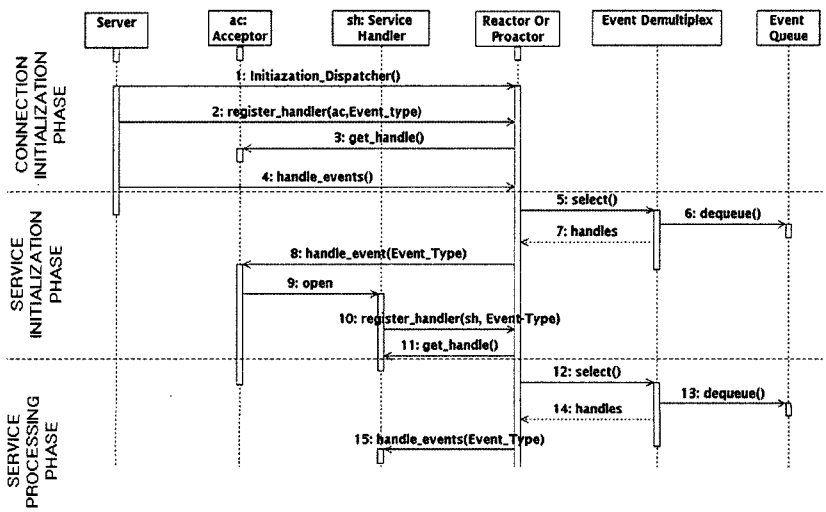

그림 4 Reactor패턴의 순차 다이어그램

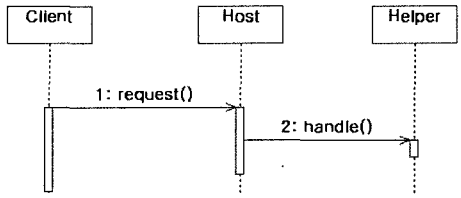

그림 5 Thread-Per-Message모델

간이 많이 필요할 때 적합하며 원격 메소드가 실행되는 동안에 다음 메소드를 실행하기 위해 대기하는 문제가 발생하지 않는다. 하지만 메소드가 동시에 여러 개 실행되면서 다루어지는 공유자원에 대한 동기화를 보장해야 한다.

그림 6은 태스크 기반 쓰레드 예제로서 Worker Thread 패턴이다. 호스트 쓰레드는 클라이언트로터 명령을 받아 버퍼(Buffer)나 큐(Queue)와 같은 채널에 넣고 Worker 쓰레드는 채널에서 명령들을 가져가 Helper를 통해 로직을 처리한다.

이 패턴은 호스트 쓰레드는 작업을 만들고 Worker 쓰레드는 작업들을 실행한다는 측면에서 고전적인 Producer-Consumer 패턴과 관계가 있으며 작업을 큐에 넣는 것이 새로운 쓰레드를 생성해서 처리하는 것보다 성능면에서 우수하다는 개념에 기반을 두고 있다. 이 방식은 Worker 쓰레드의 수를 제어할 수 있기 때문에 시스템의 자원이 고갈되는 것을 막을 수 있으며 또한 문맥 교환(Context-Switching)으로 인한 시스템 과부하를 막을 수 있는 장점이 있다[11]. 더불어 명령어가 큐에 있기 때문에 큐 구현에 따라 명령의 우선순위에 따른 스케쥴링을 수행할 수 있다.

3. Worker-Linker패턴

3.1 개요(Synopsis)

Worker-Linker 패턴의 용도는 EAI, B2BI와 같이 I/O기반의 네트워크 컴퓨팅 환경에서 대용량 I/O거래들을 안정적으로 처리하고 싶을 때 사용할 수 있다. 즉, 명절때의 열차표 예매와 같이1년중 특정 시간대에만 거래가 폭주하는 시스템을 위해 비용을 과다하게 지불할 수 없을 때 성능을 어느 정도 감소하더라도 안정적으로 서비스를 제공하기를 원할 때 사용한다. 또한 I/O기반 네트워크 컴퓨팅 환경에서 비즈니스 로직 개발의 용이성, 확장성을 강화하고자 할 때 사용된다.

3.2 배경(Context)

최근에 EAI(Enterprise Application Integration), B2BI (Business-To-Business Integration), 그리고 ESB (Enterprise Service Bus)와 같은 많은 I/O기반 통신 미들웨어 인프라들은 대용량의 I/O거래들을 효과적이면 안정적으로 처리해야 한다. 더욱이 특정 시점에 특정 서비스에 대한 폭주로 인해 시스템의 불안정성을 초래하는 경우가 많다. 본 논문에서 제안한 Worker-Linker패턴에서 해결하려 하는 첫번째 문제는 복잡한 I/O관련 모듈들을 Worker와 Linker 쌍으로 간단하게 쉽고 설계하는 것이다. 두번째 문제는 대용량의 I/O거래를 효과적으로 처리할 수 있으며 세번째는 특정 시점에 서비스 폭주로 인해 과부하가 발생할 때 Peak Load Control (PLC)로 인해 어느정도 성능을 희생하더라도 안정적으로 서비스할 수 있는 시스템을 만드는 것이다. 네번째는 I/O기반 통신 미들웨어와 관련된 비즈니스 로직을 쉽게 구현할 수 있고 변경과 확장을 용이하게 할 수 있도록 하는 것이다.

3.3 고려사항들(Forces)

본 논문에서 제시한 Worker-Linker 패턴이 제시한 문제에 대한 솔루션을 내기 위해서는 몇가지 제약 사항이 있다. 첫번째는 Worker-Linker 패턴은 새롭게 창조한 패턴이 아니고 기존 Acceptor-Connector, Reactor,

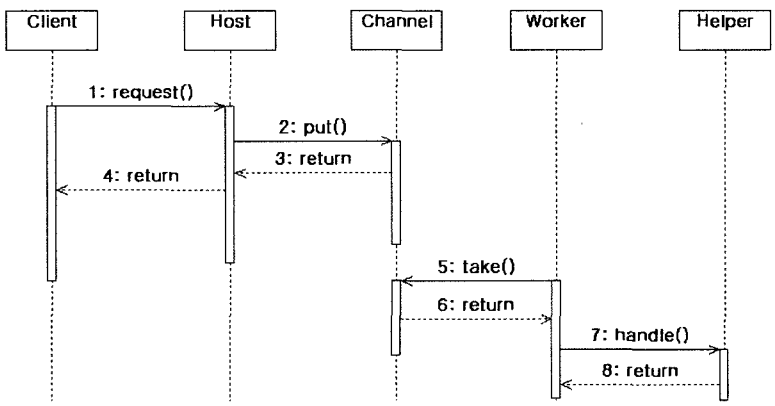

그림 6 Worker Thread패턴

Proactor, Worker Thread, Command 패턴들을 조합해서 생성한 패턴이다. 따라서 Worker-Linker 패턴을 위해서는 이들 패턴에 대한 이해가 필수적이다. 두번째는 특정 시점에 특정 서비스에 대한 거래 폭주 시 PLC기능을 수행하기 때문에 서비스를 안정적으로 제공할 수 있지만 이로 인해 약간의 성능저하가 생길 수 있다. 세번째는 특정 시점에 특정 서비스에 대한 거래 폭주로 인한 시스템 불안정성을 해결하기 위해 분산 클러스터링, 하드웨에 증설, On-Demand서비스와 같은 다양한 방법들을 사용할 수 있다. 본 논문에서 제안한 Worker-Linker 패턴은 특정 시점에만 발생하는 이러한 문제점을 해결하기 위해 과다한 비용을 지불할 수 없으며 On-Demaind 서비스에 맡기지 않고 시스템 문제에 대한 원인 규명이나 관리를 직접하고자 할 경우에 적용할 수 있는 패턴이다. 네번째는 I/O기반 통신 미들웨어 관련 비즈니스 로직 구현 용이성과 확장성을 위해 UTL이라는 데이타 포맷을 정의하기 위한 XML언어를 사용하고 있다. 데이타 포맷을 정의하기 위한 다른 언어를 사용할 경우에도 Command 패턴을 사용하는 것은 동일하지만 세부 구현에서는 달라질 수 있다.

3.4 해결책(Solution)

본 논문에서 제안한 기업간 통합 시스템의 주요 모듈들은 다음 그림7과 같은 Worker-Linker 패턴 기반으로 설계되었다. 제안한 패턴은 기존 Acceptor-Connector을 응용하여 Linker-Worker 쌍으로 구성했다. 또한 기존 Worker Thread패턴의 안정성과 고효율성에 Peak Load Control(PLC)기능을 추가함으로서 짧은 순간에 폭주로 인한 성능상의 불안정성을 제거했다. 더욱이 기업간 통합 시스템의 기능 확장성을 용이하게 하기위해 Command 패턴 형태로 UTL(Unified Transaction Language) 기반으로 작성된 Commnad 객체를 실행하고 있다. 그림 7에서 Linker는 Acceptor-Connector패턴의 Connector의 역할을 수행한다. Acceptor 쓰레드가 수행하는 작업은 Linker로부터 요청 메시지를 받아 그것을 큐에 넣고 Worker 쓰레드는 큐에서 해당 작업(Job)을 얻은 후에 WorkerManager가 주기적으로 계산한 지체 시간만큼 수면 상태에 빠진 후에 Command 패턴 형태로 로직을 처리한다.

그림 8은 이러한 Worker 쓰레드들에 대한 활동 다이어그램(Activity Diagram)이다.

Worker 쓰레드는 먼저 큐에 락을 건 후에 해당 작업을 가져온다. 큐에 처리할 작업이 없으며 Worker Manager가 관리하는 대기큐에 해당 Worker 쓰레드를 넣고 대기(Wait) 상태로 전환한다. 만일 큐에 처리할 작업이 있다면 WorkerManager로부터 Delay값을 가져온 후에 해당 값만큼 수면(Sleep)상태로 들어간다. 지정된 수면 상태 후에 비즈니스 로직을 수행하고 Worker Manager에 현재 로드 수를 증가시키고 종료한다.

그림 9는 이러한 지체 시간(Delay Time)을 계산하기 위한 알고리즘에 대한 슈도 코드이다. 이 알고리즘은 WorkerManager가 수행한다.

WorkerManager쓰레드는 먼저 부하 검사 주기만큼 수면 상태에 들어간 후에 수면 시간 동안 처리된 거래 처리 건수를 가져온다. 검사주기와 검사주기동안 처리된 처리 건수를 사용해서 TPMS(Transaction Per Milliseconds)을 계산한다. 계산된 TPMS값은 현재 시스템의 거래 처리 속도를 나타내는 값으로서 이값과 시스템에 설정된 최대 속도와의 차이가 바로 제한속도 초과값(Over Speed)이다.

만일 시스템이 제한 속도를 초과한 경우는 시스템에

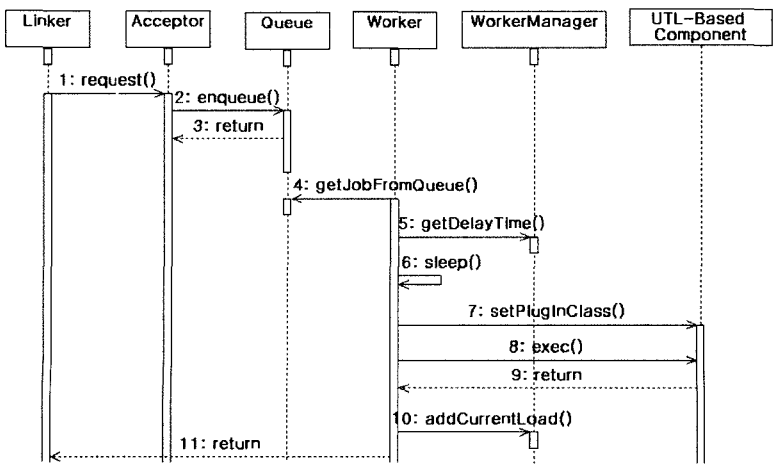

그림 7 Worker-Linker패턴의 순차 다이어 그램

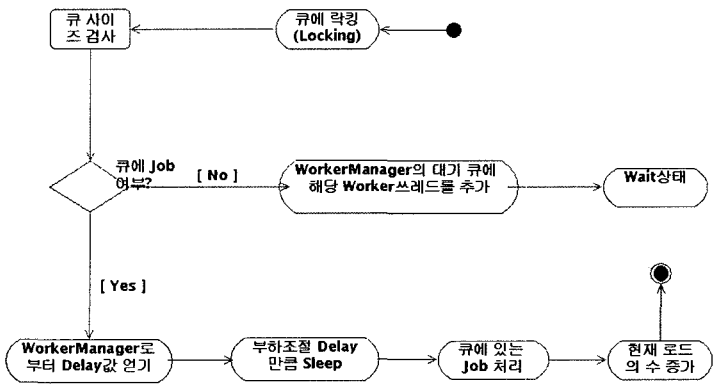

그림 8 Worker쓰레드의 활동 다이어그램

```
1.while run_flag equals "true" do
2.  get interval time for checking load
3.  sleep for the interval time
4.  get the number of transactions processed during the interval time
5.  get the configured maximum speed
6.  TPMS := number of transactions / interval time
7.  over speed := TPMS – maximum speed
8.  If over speed >0 then
      8.1 get the previous delay time
      8.2 if previous delay time = 0
       8.2.1 previous delay time := 1
      8.3 get number of active worker thread
      8.4 new delay time:= over speed / number of active worker * previous delay time
9   else
      9.1 get current delay
      9.2 if current delay > δ
        9.2.1 new delay time := current delay * β
      9.3 else
        9.3.1 new delay time := 0
      9.4 end if
10. end if
11.end while
```

그림 9 지체시간 계산 알고리즘을 위한 슈도(Pseudo) 코드

과부하가 걸린 상태로 부하 조절이 필요한 경우이다. 최대 부하 조절을 위해 본 논문에서 제안한 기본 방법은 각 Worker쓰레들의 수면 시간을 의미하는 지체 시간 값을 사용하는 것이다. 아래 수식1은 밀리세컨드 당 Worker쓰레드들이 처리한 평균 거래처리 건수를 나태내는 TPMS의 계산이고 수식 2는 제한 속도를 초과한 경우에WorkerManager 쓰레드가 과부하를 조절하기 위해 수면시간 값인 지체시간을 계산하는 수식이다. 이들 수식에서 MaxLoad값은 시스템에서 설정된 최대 처리 속도를, ActiveWorker쓰레드의 수는 현재 활동중인 Worker쓰레드의 수를 의미하며, Current Delay는 바로 전 단계의 최근 지체시간 값으로 만일 값이 0이면 기본 값인 1로 설정해서 계산한다. 수식2에서 분자값은 제한 속도 초과값으로서 이 값이 0보다 크면 시스템에 규정된 최대 제한 속도를 초과해서 처리하고 있는 과부하 상태이고 0이거나 0보다 작으면 시스템에서 설정한 최대 처리속도에 미치지 못하는 저부하 상태이다. 과부하 상태이면 처리속도를 줄이기 위해 지체 시간값을 상향 조정함으로서 각 Worker쓰레드의 수면 시간을 길게 잡아 처리 속도를 줄여야 한다.

$$TPMS = \frac{Current\ Load}{Interval\ Time}$$

수식 1 TPMS 계산

$$New\ Delay = \frac{TPMS - MaxLoad}{Numberf\ of\ ActiveWorker \times Current\ Delay}$$

수식 2 새로운 지체 시간값 계산

만일 시스템에 설정한 최대 처리속도에 미치지 못하는 저부하 상황이라면 일단 그 원인을 두가지로 분류한다. 첫번째는 이전 시스템의 상태가 심한 과부하로 인해서 WorkerManager가 이전 지체 시간을 특정 기준시간보다 높게 설정해서 Worker 쓰레드의 처리속도를 줄였지만 현재 상태는 높게 설정된 지체시간 때문에 오히려 저부하 상태에 있는 경우이다. 두번째는 이전 시스템의 상태가 심한 과부하가 아니여서 WorkerManager가 이전 지체시간을 특정 기준시간보다 낮게 설정했으며 현재는 저부하 상태인 경우다. 짧은 시간 동안에 대량의 서비스 요청으로 인한 과부하 상태의 경우에 계속해서 과부하가 발생할 가능성이 많기 때문에 점차적으로 줄여야한다. 그렇지 않고 다음 지체 시간을 갑자기 0으로 설정하게 되면 바로 이어서 제한속도 초과가 반복적으로 발생하게 된다. 하지만 지체 시간과 처리 효율성의 관계는 반비례이기 때문에 특정 기준 시간대까지 어떤 기울기로 지체 시간을 하향 조절할 것인지를 결정해야 한다. 그림 9의 슈도코드에서 δ가 바로 기준 시간대이며 β는 지체 시간 하향 조절에 대한 기울기이다. 즉, δ 이전까지는 다음 지체 시간을 점차적으로 하향조정하며 δ 기준 시간 이후부터는 점차적으로 하향조정하지 않고

바로 0으로 조정하는 것이다. 이러한 δ와 β 값에 대한 결정은 서비스를 사용하는 동시 사용자의 수나 사용자들의 패턴을 예측하거나 혹은 실제 시스템 운영에 대한 상태를 모니터링하는 로그 파일을 분석해서 결정한다. 수식 2에서 보는 것과 같이 제한속도 초과의 경우에는 활동중인Active Worker 쓰레드의 개수가 적을수록, 최근 Delay값이 적을수록, 그리고 제한 속도 초과값이 클수록 새로운 지체시간 값은 커지게 되고 각 Worker 쓰레드의 수면 시간이 길어지게 된다.

3.5 중요성(Consequences)

Worker-Linker 패턴은 I/O관련 모듈들을 Worker-Linker쌍의 형태로 간단하고 쉽게 설계할 수 있으며 또한 Worker 쓰레드가 가지는 I/O관련 처리 효율성을 달성할 수 있다. 또한 기존 Worker 쓰레드 패턴에 PLC 기능을 추가할 수 있기 때문에 저렴한 비용으로 특정 시점에 특정 서비스 폭주시에도 안정적으로 서비스를 제공할 수 있는 장점이 있다. 또한 포맷 변환과 같은 I/O 관련된 비즈니스 로직 추가, 변경, 그리고 확장이 용이하다. 약점은 시스템 과부하 시 지체 시간만큼 수면 시간으로 인한 약간의 성능 저하를 감수해야 한다.

3.6 구현(Implementation)

그림 10은 본 논문에서 PLC를 구현하기 위해서 사용한 PLC관련 클래스 다이어그램이다. PLC기능을 추가하고자 하는 모든 Worker들은 PLCWorker 쓰레드 클래스를 상속받으면 된다. PLCWorker들은 Worker 쓰레드들 상속받고 있으며 Worker 쓰레드는 기본적으로 큐를 검사해 해당 작업이 없으면 WorkerManager가 관리하는 대기큐로 들어가고 작업이 있는 경우는 큐에서 해당 작업을 가져가 'acquire' 메소드를 통해 전처리 작업을 수행하고 'exec' 안에 Command 패턴 형태로 UTL 기반 컴포넌트를 실행한다. 마지막으로 'restore' 메소드를 통해 후처리 작업을 처리한다.

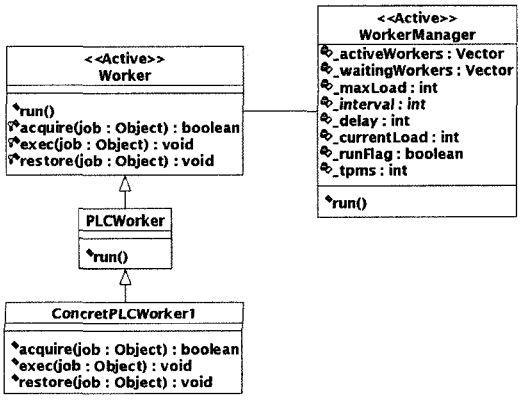

그림 10 PLC구현을 위한 클래스 다이어그램

Worker-Linker 패턴은 기업간 통합 비즈니스 로직의 유연성과 재사용성을 향상하기 위해 비즈니스 로직을 가지고 있는 UTL 기반의 컴포넌트를 Command 패턴을 사용해 실행하고 있다. 따라서 통합 프레임워크는 단지 Command 객체를 구현함으로서 새로운 비즈니스 로직을 쉽게 추가할 수 있다. 유틸(UTL)이란 메시지 내부에 존재하는 헤더(Header), 요청(Request), 그리고 응답(Response) 블록의 데이타 포맷을 정의하기 위한 XML기반의 언어이다.

그림 11은 UTL 기반으로 한 기본 템플릿 포맷에 대한 예시이다. UTL 엘리멘트 내부의 Header, Reqeust, 그리고 Response 엘리멘트(Element)는 각 블록에 필요한 정보와 데이타 포맷들을 정의할 수 있으며 엘리멘트나 속성들을 확장해서 사용할 수 있다. code엘리멘트 내의 Java 코드는 기업간 거래와 관련된 로직을 구현한 클래스로서 프레임워크에서 설정한 훅 메소드에 데이타 포맷 변환과 같은 사용자 정의 로직을 작성한 것으로 이 클래스 파일이 바로 유틸(UTL) 컴포넌트가 된다. UTL 사용해서 작성된 각 파일의 확장자는 *. utl형태로서 유틸서버에 의해 유틸 저장소(Repository)에 저장되며 Compile 엘리멘트는 자바 클래스 파일에 대한 컴파일 명령어로서 이를 통해 클래스 파일을 생성하고 해당 객체를 로딩한다.

그림 12는 그림 11에서 정의한 각 데이타 포맷과 관련 자바 코드들을 구현하기 위한 클래스 다이어그램이다. 그림 11의 UTL클래스 내부에는 그림 10의 Header, Request, 그리고 Response 블록에 정의된 데이타 포맷 정보를 담기 위한 속성들이 존재한다. 또한 그림 11의

```
<UTL>
    <HEADER>
        <DATA_SIZE_LEN default="0000000000">10</DATA_SIZE_LEN>
        <MESSAGETYPE>1</MESSAGETYPE>
        <SVCCODE>6</SVCCODE> <SVCTYPE>6</SVCTYPE>
        <DATE>8</DATE><TIME>9</TIME>
        <USERID>20</USERID><USERPASSWORD>20</USERPASSWORD>
        <RETURN_CODE default="000000">6</RETURN_CODE>
        <CLIENTNAME>10</CLIENTNAME>
        <ERRORMSG> 29<ERRORMSG>
    </HEADER>
    <REQUEST>
        <USERID>20</USERID>
        <PASSWORD>20</PASSWORD>
        <HOST>50</HOST>
        <BOXCODE>40</BOXCODE>
    </REQUEST>
    <RESPONSE>
        <Mail_Count> 5</Mail_Count>
        <Remain_Count>5</Remain_Count>
        <BOXCode>40</BOXCode>
    </RESPONSE>
    <SCRIPT>
        <COMPILE>javac Webmai0001.java </COMPILE>
        <code>
            public class Webmai0001 implements UTLScript {
                public void setPluginClass(String name, Object plugin) {
                // write code
                }
                public String exec(String input) {
                // write java code
                }
            }
        </code>
    </SCRIPT>
</UTL>
```

그림 11 UTL을 사용한 데이타 포맷 예제

'code' 내부에 있는 Java 소스들을 표현하고 있는 것이 UTLScript 타입의 객체이다. 그림 12와 같은 모든 UTL 컴포넌트들은 UTLScript 인터페이스를 구현해야 하며 그림 12와 같이 UTL 클래스 내부의 UTLScript 속성 형태로 표현된다. UTL 컴포넌트를 Command 패턴 형태로 실행하기 위해 UTLScript 인터페이스에 정의된 훅 메소드 'setPluginClass'와 'exec'를 차례대로 호출한다.

3.7 관련 패턴들(Related Pattern)

Worker-Linker패턴과 관련된 패턴은 먼저 Worker-Linker 쌍 형태로 구성된다는 측면에서 Connector-Acceptor패턴과 유사하다. Acceptor는 큐에 해당 작업을 만들고 Worker는 큐에서 해당 작업을 가져와서 수행하는 것은 Consumer-Producer패턴과 유사하다[10]. I/O 거래의 효과적인 처리를 위해 Worker쓰레드 패턴을 사용했다. 또한 비즈니스 로직의 구현의 용이성과 확장성을 위해 Command패턴을 사용하고 있다.

4. Worker-Linker패턴 적용 사례

본 장에서는 본 논문에서 제안한 Worker-Linker패턴을 기업간 통합 시스템 프레임워크에 적용한 사례를 제시한다. 본 논문에서 제안한 기업간 통합 시스템 프레임워크는 소스 시스템들과 대상 시스템들 사이에 안정적이고 신뢰적인 통합 거래들을 제공하기 위한 통신 미들웨어이다[12,13]. 그림 13은 기업간 통합 시스템의 소프트웨어 아키텍처이다. 기업간 통합 프레임워크가 가지는 아키텍처가 주요 기능은 내부 어플리케이션 기업 외부 어플리케이션간에 연동을 제공하는 것이며 주요 비기능적인 품질 속성은 대학교의 원서접수나 명절때의 철도표 예매와 같은 일년중 특정 몇일동안 시스템 심한 과부하시에 안정적인 서비스를 제공하는 것을 목표로 한다. 본 논문에서 예제로 든 기업간 통합 시스템에서는 이러한 비기능적인 요구사항을 만족하기 위해 소프트웨어 아키텍처 설계시에 Worker-Linker패턴을 어떻게 적용하는지에 대해서 기술한다.

통합 프레임워크는 크게 Admin Console, System Management, Gateway, UTLProcessor, 그리고 UTL Server로 구성된다. Admin Console은 코더(Coder)들과 시스템 관리자들을 위한 개발 및 관리 툴킷(Toolkit)들을 제공한다. 개발 툴킷 관점에서 관리자 콘솔은 관련된 업무 단위별로 소스코드들과 다양한 설정 파라메타(Parameters)들을 관리한다. 다시 말해 관리자 콘솔은 트리형태로 업무를 구분해서 각 업무에 관련된 소스코드(예: 유틸코드)들을 제공해 소스 개발 및 수정, 컴파일, 패키징, 배포, 그리소 테스트를 수행한다. 관리 툴킷 관점에서 관리자 콘솔은 프레임워크에 관련된 모든 리소스를 관리하고 모니터링한다. Gateway는 네트워크 관련된 세부 사항들, 프로토콜 변환, 그리고 메시지 라우팅과 같은 통합 프레임워크의 핵심적인 기능들을 수행한다. UTLProcessor와 UTL Server는 통합에 관련된 비즈니스 로직을 처리하기 위한 UTL기반 컴포넌트들을 생성(Creating), 캐싱(Caching), 배포(Deploying), 그리고 실행(Executing)하는 책임을 가진다.

그림 13의 Gateway는 Adapter, Message Router, 그리고 Dispatcher로 구성된다. Adapter는 내부 및 외부 시스템과 연결(Connection), 리스닝(Listening), 그리고 프로토콜 변환과 같은 통신에 관련된 세부적인 사항들을 처리한다. Message Router는 라우팅 테이블을 기반으로 메시지의 라우팅을 수행한다. 소스나 타켓 어댑터들로부터 받은 헤더 메시지 내부에는 대상시스템 이름(TargetSystem Name)이 존재하며 이 값을 기반으로 라우팅 테이블의 정보를 기반으로 대상 시스템과 연동한다. 대상 시스템에는 Dispatcher, UTL Server, 그리고 Adapter들이 가능하다. 만일 데이타포맷 변환이나 데이타 타당성 검증과 같은 비즈니스 로직이 필요한 경우에는 UTL기반의 컴포넌트를 실행하기 위해 Dispatcher를 호출한다. 비즈니스 로직 실행을 위해 Dispatcher는 UTLProcessor로부터 UTL기반 컴포넌트를 가져와 실행한다. UTLProcessor는 UTL기반 컴포넌트를 생성하고 캐싱(Caching)하는 역할을 수행한다. 만일

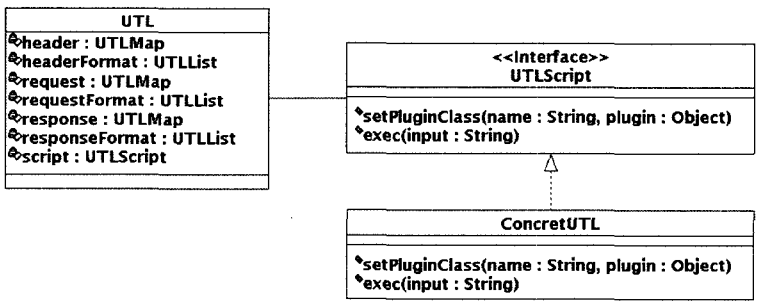

그림 12 UTL 관련 클래스 다이어그램

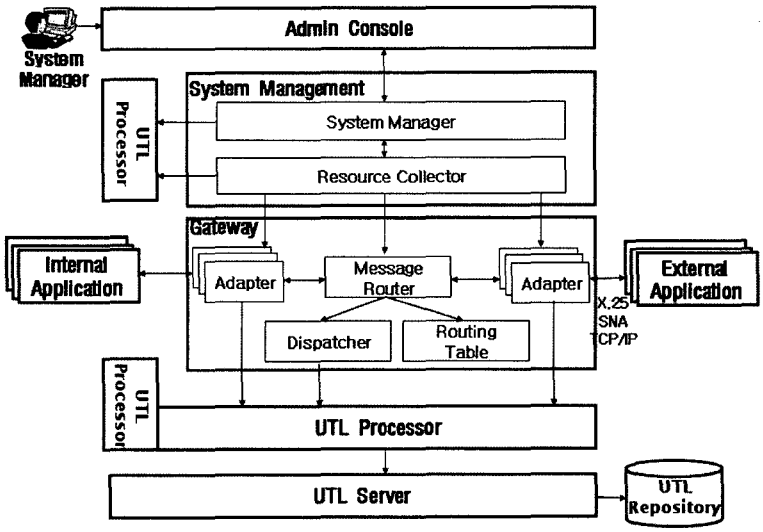

그림 13 기업간 통합 시스템의 소프트웨어 아키텍처

컴포넌트가 UTLProcessor에 존재하지 않으면 UTL-Processor는 UTL기반 컴포넌트를 생성하기 위해 UTL Server로부터 필요한 정보를 가져와 해당 컴포넌트들을 생성한다.

System Management는 시스템과 관련된 자원들을 모니터링하고 관리하는 역할을 가지고 System Manager와 Resource Collector모듈로 구성된다. System Manager는 프레임워크와 관련된 자원들을 스케줄링하고 관리하는 역할을 수행한다. 성능이나 확장성과 같은 품질 향상을 위해 Adapter, Gateway, Dispatcher와 같은 메인 모듈들은 분산 시스템의 다른 하드웨어 박스나 다른 프로세스들에 배포될 수 있다. 이들 메인 모듈에는 State Manager라고 부르는 경량 쓰레드가 있어서 프로세스 상태, CPU, 메모리, 입출력과 같은 리소스 관련 정보를 수집해서 Resource Collector에게 전송한다. Resource Collector는 메인 모듈로부터 모든 리소스 관련 정보를 모아서 System Manager에게 전송하며 System Manager는 각 메인 모듈의 로드 발란싱(Load Balancing)이나 시스템 관리를 위해 사용한다. 그림 14는 내부 어플리케이션에서 외부 어플리케이션과 연동하기 위해 소프트웨어 아키텍처 상의 주요 모듈간의 흐름을 보여주는 상호작용 다이어그램이다.

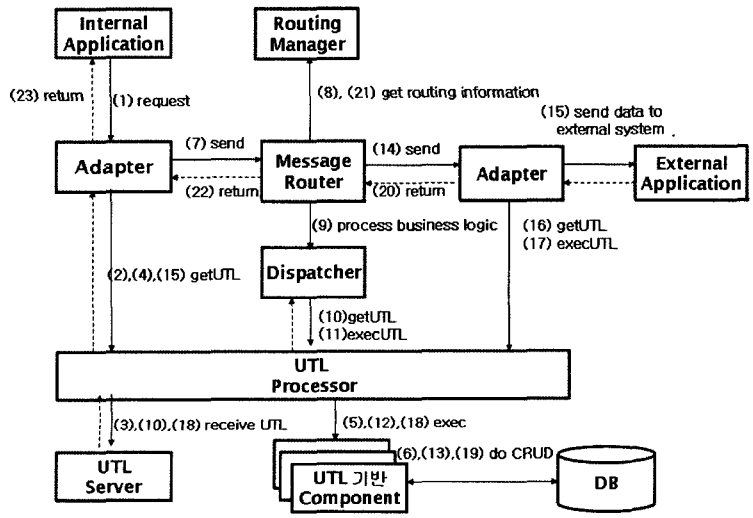

그림 14 내부에서 외부 어플리케이션과의 연동을 위한 상호작용 다이어그램

Adapter가 클라이언트로부터 요청을 받았을 때 Adapter는 데이타 무결성이나 포맷 변환과 같은 사전 처리 작업이 있는지를 검사한다. 사전 처리 작업이 설정되어 있다면 Adapter는 UTL Processor를 호출해 사전 처리 작업을 수행한다. 사전처리 작업을 수행한 후에 Adapter는 Message Router와 바인딩(Binding)한다. Message Router는 Adapter로부터 받은 메시지의 헤더 정보중에 다음대상 시스템 이름을 보고 라우팅 테이블에서 대상 시스템과 바인딩하기 위한 정보를 가져와 연동한다. 만일 UTL기반 컴포넌트 형태의 비즈니스 로직 실행이 필요하면 다음 대상 시스템은 Dispatcher가 된다. Dispatcher는 UTLProcessor를 호출해 실행하고자 하는 UTL 컴포넌트를 가져와 Command패턴 형태로 해당 컴포넌트를 실행시킨다. 비즈니스 로직 처리 후에 Message Router는 라우팅 테이블에 있는 외부 어플리케이션과 연동하기 위한 Adapter의 IP나 포트번호를 기반으로 해당 Adapter와 바인딩을 수행한다. Adapter가 외부 대상 어플리케이션으로부터 응답 메시지를 받는다면 Adapter는 데이타 포맷 변동이나 타당성 검사와 같은 사후 처리 작업을 위해 관리자 콘솔에서 설정된 UTL 컴포넌트들을 호출한다.

그림 15는 본 논문에서 제안한 Worker-Linker패턴이 가지는 Worker와 Linker쌍의 형태로 기업간 통합 시스템을 어떻게 설계 했는지를 설명하기 위한 그림이다.

대부분 Linker에서 다른 모듈의 Acceptor로 연결요청을 하고 실제 데이타를 주고 받는 부분은 Linker와 Worker모듈 쌍을 통해 주고 받는다. 그림 15에서는 이러한 실제 데이타를 주고 받는 부분에 대한 Linker부분은 생략했다. 그림 15와 같이 기업간 통합 시스템의 Adapter, Message Router, Dispatcher, UTL Server 들은 모두 Worker-Linker 패턴에 따라 Acceptor, Worker, Queue 서브 모듈들을 가지고 있다. Acceptor 쓰레드는 연결요청을 받아 Socket Queue에 넣고 Worker 쓰레드는 큐에서 클라이언트 Socket을 가져간 후에 해당 작업을 처리한 후 다른 모듈과 연동하기 위해 Linker 쓰레드를 사용하고 있다. 특히 Message Router와 연동하거나 혹은 Message Router에서 다른 대상 시스템과 연동하기 위해서는 MRLinker를 사용하는데 특히, Message Router에서 MRLinker를 통해 다른 대상 시스템과 연동 시 라우팅 테이블에 있는 정보를 기반으로 연동한다.

Worker-Linker 패턴이 가지는 있는 PLC 기능을 적용하기에 좋은 부분은 바로 내부와 외부 Adapter 모듈의 Worker 쓰레드와 Message Router 모듈의 Worker 쓰레드 모듈에 적용했다. 따라 이들 Worker 쓰레드들은 PeakPointControlWorker 객체를 상속받아야 한다. 따라서 WorkerManager 쓰레드가 계산한 지체시간만큼 이들 Worker 쓰레들은 수면 시간(Sleep Time)을 조정함으로서 부하를 조절한다.

5. Worker-Linker패턴 성능실험

최근에 Worker-Linker 패턴의 거래 처리 효율성과 안정성 확인을 위해 본 논문에서 제시한 기업간 통합 시스템을 한국의 'K' 은행에 적용해 보았다. 평균 하루에 500백만개의 거래들을 처리할 수 있으며 더욱이 특정시점에 요청 폭주로 인해 발생하는 심한 과부하 상황

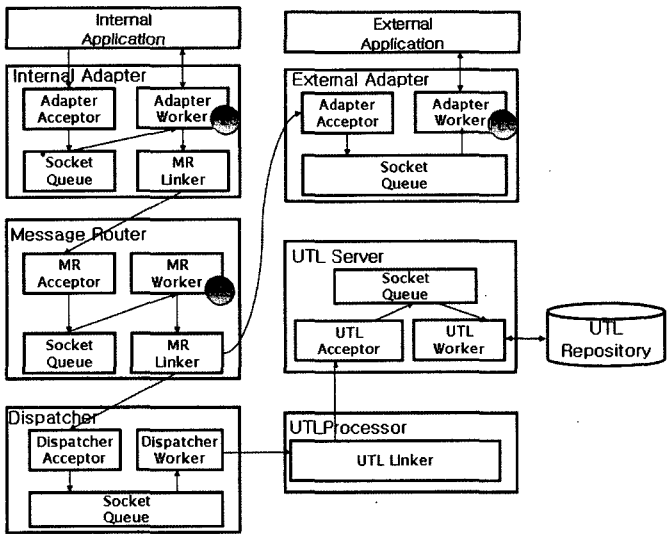

그림 15 기업간 통합 시스템 거래 안정성과 효율성을 위한 Worker-Linker패턴 적용

에서도 안정적인 성능을 제공하는 것을 확인할 수 있었다. 본장에서는 기업간 통합 시스템의 대용량 거래처리와 성능 안정성을 위한 실험 결과를 제시한다.

그림 16은 성능 분석을 위한 실험 환경이다. 워크로드(Workload)생성을 위해 Load Runner 8.0 툴[14]을 사용했으며 성능분석을 위한 메트릭(Metrics)으로 TPS(Transaction Per Seconds)을 사용한다.

성능 실험을 위한 목적이 다른 기업간 통합 시스템과 성능을 비교하기 위한 목적이 아니고 제안한 통합 시스템의 대용량 거래 처리와 폭주시의 성능 안정성을 제시하는 것이 목적이기 때문에 실험을 위한 워크로드(Work Load)서 은행 원장 거래조회와 같은 하나의 어플리케이션을 사용했다. 또한 PLCWorkerManager가 수행하는 지체시간 계산 알고리즘에서 사용하는 δ와 β 값은 특정 서비스에 대해 폭주가 발생하는 특정 시점의 로그 파일을 분석해서 각각 100ms와 0.75로 고정했다.

Load Generator가 J2EE의 JSP 페이지에 요청을 보내면 JSP페이지는 TCP 프로토콜을 사용해 본 논문에서 제안한 기업간 통합 시스템에 요청을 보낸다. 통합 시스템은 SNA 프로토콜[15]을 사용해 IBM 메인 프레임 호스트로부터 데이타들을 가져온다. IBM 호스트로부터 가져온 데이타 사이즈는 475 바이트이고 동시 사용자 수는 300유저이고 각 요청간 간격인 Think Time값은 0으로 설정했다. 또한 기업간 통합 시스템과 관련된 여러 Worker 쓰레드 들 중에서 그림 15와 같이 Dispatcher와 Message Router쪽 Worker 쓰레드에서만 PLC 기능을 넣었다.

그림 17은 통합 시스템의 성능 실험 결과이다. 본 논문에서 제안한 통합 시스템은 초당 최대 450개의 거래를 처리할 수 있으며 평균 초당 381개의 거래를 처리할 수 있다. 제안한 통합 시스템은 대용량의 거래를 효과적으로 처리하기 위해 I/O의 효율성이 좋은 기존 Worker 패턴 사용, 관리 프로세스와 거래처리 프로세스의 분리, 쓰레드 풀(Thread Pool) 등과 같은 성능을 향상시키기 위한 여러 메커니즘을 사용하고 있다.

그림 18은 기존 Worker 패턴과 본 논문에서 제안한 Worker-Linker 패턴의 성능 안정성을 비교한 것이다. 동시 사용자가 300명일 경우 Worker-Linker 패턴은 TPS 관점에서 Worker 패턴과 유사하며 양쪽 패턴의 최대 TPS는 388이다. 하지만 동시 사용자의 수가 300명이 넘을경우 양쪽 패턴은 다른 양상을 보이고 있다. Worker-Linker 패턴은 PPC 매커니즘 때문에 평균 386 TPS을 유지하고 있지만 기존 Worker 패턴은 동시 사용자 550까지 TPS가 계속 떨어지다 동시 사용자 550 이상이 되면 120 TPS를 유지하는 증상을 보이고 있다.

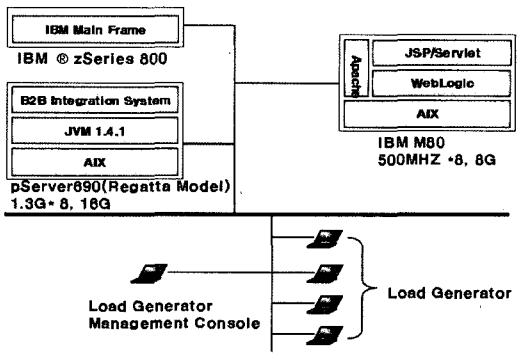

그림 16 기업간 통합 프레임워크 실험환경

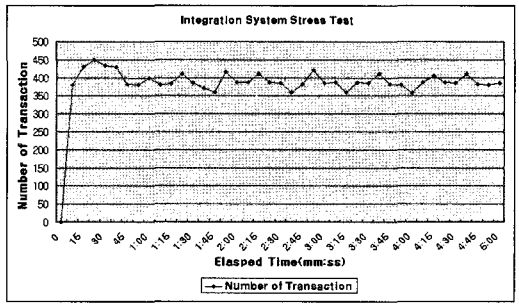

그림 17 기업간 통합 프레임워크의 처리량

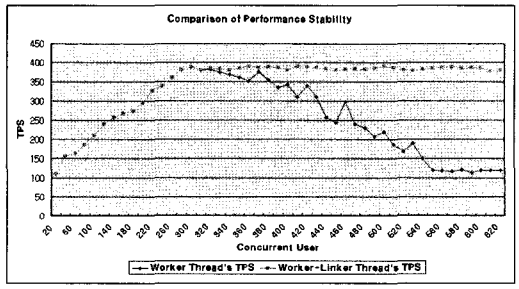

그림 18 Worker 패턴과 Workr-Linker 패턴 사이의 성능 안정성 비교

그림 19는 Worker 패턴과 Worker-Linker 패턴 사이의 다른 증상에 대한 이유를 설명하고 있다. 동시 사용자 수가 300명이 초과할경우 Worker 패턴은 CPU 사용율이 100%에 도달할 만큼 과부하 상태에 있다. 하지만 Worker-Linker 패턴은 PPC 메커니즘을 사용하기 때문에 92%의 CPU 사용을 유지한다. 본 논문에서 제안한 Worker-Linker 패턴은 입출력 효과를 제공하는 기존 Worker 패턴에 PLC를 추가함으로서 과부하시에도 성능의 안정성을 달성하고 있다.

그림 20은 최고 TPS인 동시 사용자 300명을 초과해 계속해서 동시 사용자 수를 증가할 경우에 제한 속도와

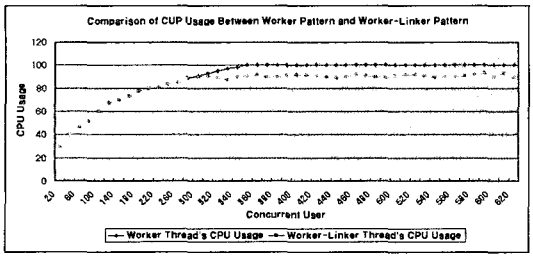

그림 19 Worker 패턴과 Worker-Linker 패턴 사이의 CPU 사용률

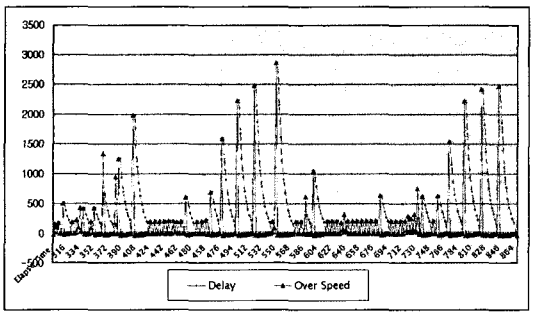

그림 20 지체 시간 알고리즘을 사용한 제한속도 조절

지체시간 사이의 관계를 보여주고 있다. 그림 20은 본 논문에서 제안한 지체 시간 알고리즘을 기반한 PLC 메커니즘이 요청 폭주로 인한 과부하로 인해서 발생하는 제한속도 초과를 통제하는데 효과적이라는 것을 증명한다.

동시 사용자 수가 300명이 초과되면서 제한 속도 초과가 발생한다. 제한 속도 초과가 발생하면 각 Worker 쓰레드는 WorkerManager가 계산한 지체시간만큼 수면 상태에 들어간다. 그림 20은 제한 속도 초과값이 높으면 높을수록 그 만큼 지체 시간 값도 커지게 되며 따라서 제한 속도는 급격하게 떨어지고 있다는 것을 확인할 수 있다. 하지만 다음 지체 시간 값은 기준 시간값 인 δ와 지체시간 하강에 대한 기울기 값인 β에 따라 조정되기 때문에 기준 시간 값 까지는 급격하게 떨어지지 않고 β 기울기만큼 떨어지고 있다. 즉, 기준 시간 값인 100ms 까지는 β기울기 값인 0.75에 따라 점차적으로 하강하며 기준선 100ms을 넘어서는 순간 다음 지체 시간은 바로 0으로 설정된다. 그림 20에서 보듯이 제한 속도 초과가 발생한 이후부터 다음 지체 시간이 0으로 설정되기 전까지 제한 속도 초과 현상은 발생하지 않고 있다. 하지만 지체 시간이 0으로 설정되자 마자 바로 제한 속도 초과 현상이 다시 발생하고 있으며 다음 지체시간 값이 이러한 제한속도 초과를 다시 통제한다.

6. 결론 및 향후 연구

EAI, B2Bi, 홈게이트웨이, 그리고 유비쿼터스 환경에서 이질적인 디바이스간 통합과 같이 입출력 기반으로 하는 통합 어플리케이션들은 높은 성능과 신뢰성을 가지고 입출력 기반의 대용량 거래들을 신뢰성 있게 처리하기 위한 통합 프레임워크를 필요로 한다. 본 논문에서는 안정적인 대용량 I/O기반 거래를 처리하기 위한 Worker-Linker 패턴을 기술했다. Worker-Linker 패턴은 Worker와 Linker의 쌍 형태로 I/O관련 주요 모듈을 설계할 수 있으며 입출력의 효율성을 제공하는 기존 Worker 패턴에 PLC 메커니즘을 추가함으로서 특정 시점에 특정 서비스에 심한 과부하가 걸릴 경우에도 안정적으로 서비스를 제공할 수 있다. 또한 데이타포맷 변동과 같은 I/O관련 비즈니스 로직 추가, 변경, 그리고 확장을 용이하게 하기 위해서 Command 패턴 형태로 UTL 기반의 컴포넌트들을 실행하고 있다.

본 논문에서는 기업간 통합 시스템에 Worker-Linker 패턴을 적용해서 성능 분석을 해본 결과 특정 동시 사용자가 300명이 초과되는 시점에 Worker 쓰레드 패턴은 TPS가 저하되다가 550이상이 되면 일정하게 TPS을 유지한다. 반면에 Worker-Linker 패턴은 아무리 동시 사용자 수가 많아 PLC 기능을 가지고 있기 때문에 안정적으로 서비스를 제공한다.

향후 연구에서는 본 논문의 알고리즘에서 제시한 설정값인 최대 속도, 기준 시간 값인 δ, 그리고 지체시간 하향 곡선의 기술에 대한 기울기 값인 β에 대한 최적값을 동적으로 설정하는 방법에 대해 연구할 것이다.

참 고 문 헌

[1] I. Ahmad and A. Ghafoor, "Semi-Distributed Load Balancing for Massively Parallel Multicomputer Systems," IEEE Trans. Software Eng., vol. 17, no. 10, pp. 987-1004, Oct. 1991.

[2] Robert Steinke, Micah Clark, Elihu Mcmahon, "A new pattern for flexible worker threads with in-place consumption message queues," Volume 39, Issue 2 (April 2005) table of contents Pages: 71-73 Year of Publication: 2005.

[3] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in Pattern Languages of Program Design (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[4] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in Pattern Languages of Program Design (J. O. Coplien and D. C. Schmidt, eds.), pp. 529-545, Reading, MA: Addison-Wesley, 1995.

[5] J. Hu, I. Pyarali, and D. C. Schmidt, "Applying

the Proactor Pattern to High-Performance Web Servers," in Proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems, IASTED, Oct. 1998.
[6] Tay, Y.C, Goodman, N, Suri, R, "Locking Performance with Dynamic Locking," ACM TODS 10, 4: 415-462, 1985.
[7] Iyer, B.R., "Limits in Transaction Throughput-Why Big is Better," IBM Research Report No. RJ6584, IBM Res. Div., Yorktown Heights, NY10598, 1988.
[8] Performance Stability. http://www.performance-stability.com/
[9] Douglas C. Schmidt, Michael Stal, Hans Rohert, and Frank Buschmann, "Pattern-Oriented Software Architecture: Concurrent and Networked Objects," John Wiley and Sons, 2000.
[10] Doug Lea, Concurrent Programming in Java, Second Edition, Addison-Wesley, November, 1999.
[11] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in Proceedings of the 2nd Annual Conference on the Pattern Languages of Programs, (Monticello, Illinois), pp. 1-7, September 1995.
[12] C. Bussler, "B2B Protocol Standards and their Role in Semantic B2B Integration Engines," EEE *Bulletin of the Technical Committee on Data* Engineering, Special Issue on Infrastructure for Advanced E-Services, 2001, vol. 24 no. 1, pp.3-11.
[13] Bussler, C, "The Role of B2B Protocols in Inter-enterprise Process Execution," In: Proceedings of the Workshop on Technologies for E-Services (TES 2001), Rome, Italy, September 2001.
[14] Mercury RoadRunner. http://www.mercury.com/us/products/performance-center/loadrunner/
[15] SNA. http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/ibmsna.htm

이 용 환
정보과학회논문지 : 컴퓨팅의 실제
제 12 권 제 2호 참조

민 덕 기
정보과학회논문지 : 컴퓨팅의 실제
제 12 권 제 2 호 참조