SDL을 이용한 TCP 혼잡제어 알고리즘의 구현 및 검증

이재훈* · 조성현** · 이태오*** · 임재홍****

Implementation and Verification of TCP Congestion Control Algorithm using SDL

Jae-Hoon Yi* · Sung-Hyoun Cho** · Tae-Oh Lee*** · Jae-Hong Yim****

요 약

어플리케이션을 개발하는 데 있어서 전통적인 텍스트 기반의 방법으로는 정확한 요구사항을 파악하기 어렵다. 그리고 각 개발 단계에서의 분석과 확인에 관한 문제점이 있다. 그러므로 만약 검증에서 에러와 요구사항의 교환이 요구된다면 모든 시스템에 나쁜 효력이 발생할 수 있다. 이 경우 개발비용과 기간에 불리하게 작용한다. 따라서 분석과 검증은 에러가 자주 일어나는 것을 방지하기 위해서 수행한다.

본 논문에서는 각 개발 단계에 대한 분석과 확인 그리고 그래픽 인터페이스를 제공하는 SDL을 이용한 TCP/IP 혼잡제어 알고리즘에 대해서 논한다. SDL을 사용함으로써 각 개발 단계에 대한 명확한 표현과 정확한 에러 또는 요구사항 교환에 검증을 쉽게 할 수 있다. 추가적으로 프로토콜의 단계는 MSC의 확인을 통해서 시뮬레이션으로 확인하고 결론에서는 TCP/IP 프로토콜보다 개발의 가능성을 보인다.

ABSTRACT

Developing an application, it is difficult to catch an exact requirement with the conventional text-based method. It has also problems in verification and analysis at each developing stage. Therefore, if an adjustment is required with an error and change of requirement, a bad effect happen in the whole system. In this case, it also affect adversely on the developing cost and period. Meanwhile, if an analysis or verification is performed, the possibility of an error frequency reduces. Thus, not only is it easier to correct the error but also add an new requirement.

This thesis embody a TCP/IP congestion control algorithm with SDL which provides automatically graphic interface, verification and analysis to each developing stage. Using SDL gave a clear representation embodiment in each developing stage and easiness of adjustment due to changing requirements or correcting errors.

In addition, the stages of protocol have been certified in a simulation by verification of MSC and the results showed a possibility of developing a better TCP/IP protocol.

키워드

SDL(Specification and Description Language), MSC(Message Sequence Chart), TCP-Tahoe, TCP-Reno, CCAlgorithm

Ⅰ. 서 론

현재 컴퓨터 기술의 성장과 함께 이전과는 비교할 수 없을 정도로 인터넷과 통신 기술이 널리 보급되어 쓰이고 있으며, 그 수요는 날로 증가하고 있다. 이러한 추세와 더불어 멀티미디어와 고속 데이터 서비스에 대한 요구도 급증하여, 사용자들은 초고속 네트워크와 함께 높은 서비스 품질(QoS : Quality of

*한국해양대학교 전자통신공학과
**(주)유정시스템
***동명정보대학교 정보공학부 컴퓨터공학과
***한국해양대학교 전파 · 정보통신공학부

접수일자 : 2003. 1. 8

Service)을 제공받기 원하며, 각각의 인터넷 서비스 제공업체(ISP : Internet Service Provider)와 통신 서비스 제공업체들은 이런 요구에 부응하기 위해 더 높은 대역과 고품질을 제공할 수 있는 네트워크와 통신 기법을 개발하고 있다. 그러나 네트워크 또는 다른 유·무선 통신 환경에서도 통신의 기본적인 기능과 규약을 정의하는 통신 프로토콜이 있어야만 통신이 가능하며, 현재 사용되고 있는 여러 프로토콜보다 성능이 향상된 프로토콜을 개발하려는 움직임도 활발히 진행되고 있다.

이러한 통신 프로토콜 개발 시에, 기존의 수기적인 프로그래밍 방식에서는 구현하려는 시스템이나 프로토콜의 요구사항 분석, 디자인, 코딩, 시험, 유지관리 등의 작업이 각각 개별적으로 진행되며, 각 작업을 진행 시에 선행되었던 작업 혹은 이후에 진행될 작업에 대한 분석과 검증과정이 없다. 그러므로 이러한 방법으로 프로토콜을 개발한다면 개발 중에 일어날 예기치 못한 오류의 정정, 개발자나 사용자의 요구사항이 변경될 때의 수정작업, 개발 마지막 단계에서 초기 단계의 문제점이 발견되는 경우의 문제점 수정, 그리고 완성 후 디버깅 작업이 상당히 까다로워진다. 또한 검증과정 없이 프로토콜을 개발하게 되면 통신 신호처리 자체가 불안정 할 수 있으며, 기본 통신기능 뿐 아니라 프로토콜을 사용하는 장치간의 연동시험도 어려워지며, 프로토콜 자체의 구조에 오류가 발생하는 경우도 생긴다. 그러므로 철도제어, 공항관제, 의료기 제어 등 극히 안전성이 요구되는 시스템을 구현하는 경우에는 이러한 개발과정에서의 문제점들이 극히 치명적인 결과를 가져올 수도 있다[1].

반면에, 시스템이나 프로토콜 개발 시에 계획수립 단계부터 시스템 완성 및 테스트까지 각 단계를 진행함에 있어서, 철저한 요구사항의 분석과 검증을 수행하며, 이러한 과정을 정형화된 수단을 통하여 자동적으로 수행할 수 있다면 전체 개발 단계에서의 엄격한 일관성을 보장받을 수 있으며, 처음에 의도했던 개발기간이나 비용을 넘어서지 않으며, 한 단계의 문제점이 발견될 시 수정도 용이하게 된다. 대표적으로 ITU-T(International Telecommunications Union-Tele-communication Standardization Sector)에서 제안된 명세 기술 언어(SDL : Specification and Description Language)는 이러한 개발환경을 지원하며, 1972년 이후로 여러 개발 단계를 거쳐서 SDL-92와 SDL-2000에 와서는 실시간 시스템이나 대화형 시스템, 분산 시스템 개발 시에 많은 장점이 있다[2].

본 논문에서는 SDL을 이용하여 TCP/IP 프로토콜에서의 혼잡제어 알고리즘인 Reno 알고리즘을 구현하고 SDL Suite에서 제공하는 시뮬레이터와 MSC(Message Sequence Chart)를 통하여 검증함으로서 SDL을 이용한 통신 프로토콜의 개발방법을 제시하였다. 그리고 프로토콜이나 소프트웨어 개발 시, 요구사항의 변경 때 이미 구현한 내용이 전면적으로 수정되는 기존의 텍스트 기반의 개발 방법보다 요구사항의 분석이 선행되고 개발과정에서의 검증과 일관성을 제공하는 개발방법의 장점과 중요성에 대해서 논한다.

II. SDL 및 MSC 개요

2.1 SDL

SDL은 시스템의 명세와 동작을 설명하는 표준 언어로써, ITU-T에 의해 개발되어 권고안 Z.100으로 표준화되었다[3]. 현재 통신 시스템들이 복잡해짐에 따라 시스템의 동작을 표현하기 위한 명세 언어의 필요성이 증대되고 있는 가운데, SDL은 현재 전기통신 분야에서 많이 사용되는 명세 언어로써, 시스템을 계층적으로 표현하면서 많은 동시 작업과 상호동작이 필요한 사건 중심의 실시간 시스템을 기술하며, 시스템의 구조, 동작, 기능 및 데이터를 표시하며 검증하는데 적합하다. 기존의 개발 방법과 SDL을 비교하면 다음과 같다. 하나의 시스템을 설계하고 제작하는데 있어서 기존의 텍스트에 의존적인 개발 방법은 아래 그림 1의 워터 폴(water-fall) 모델과 같은 방식이다.

워터 폴 모델에서는 시스템을 설계할 때 각각의 단계별로 진행을 하는 과정에서 한 단계와 그 다음 단계와의 연관성만 고려할 뿐 두 단계 혹은 전체적인 시스템 설계시의 고려사항이나 에러 수정, 그리고 검증과정을 수행하지 않는다. 그리고 각 단계별로 작업을 진행할 때 요구사항의 변경이나 오류 발생 시 전체 개발 과정에 큰 영향을 미치게 된다. 예를 들어 마지막 단계에서 초기 단계의 문제점이 발견된다면 그것을 수정하기가 쉽지 않으며, 또한 수정으로 인해 수

정한 부분의 전 단계와 다음 단계에 영향을 미치게 되므로 결국 전체적인 시스템의 수정이 요구된다, 따라서 이로 인한 시스템의 안정성, 개발 기간, 개발비용 등이 개발자가 처음에 의도했던 것 보다 많은 차이가 날 수 있다.

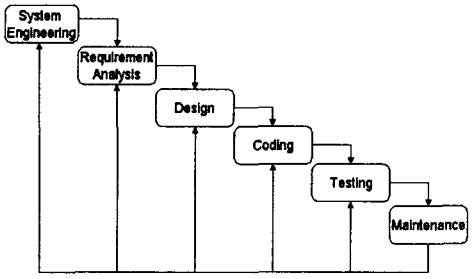


그림 1. 워터 폴 모델
Fig. 1 Water-fall model

반면에, SDL에서 사용하는 방식인 나선형 모델과 같이 개발하는 경우를 그림 2에 나타내었다. 나선형 모델은 개발시의 초기 계획, 요구사항 분석, 설계 및 디자인, 테스팅(testing)의 모든 과정에서 각 단계간의 상호 검증과 분석이 이루어지며 이런 과정을 통하여 전체 시스템의 안정성을 가져오며, 문제점 발견시에도 수정이 용이하다.

그림 3은 SDL의 계층구조를 나타내며, SDL에서의 시스템 디자인은 크게 시스템 레벨, 블록 레벨, 프로세스 레벨로 나눌 수 있다.

첫 번째로 시스템 레벨은 서로간에 연결된 블록들로 구성되고, 블록들은 정보 전달 통로인 채널(channel)에 의해 연결된다. 블록은 사각형으로, 채널은 화살표가 있는 선으로 표시되어 정보흐름의 방향을 나타낸다. 채널들은 무한대의 FI-FO(First In First Out) 큐를 가지며 채널 내에 블록의 하위 단위인 프로세스에서 통신할 때 쓰는 신호인 시그널(signal)들을 포함한다.

두 번째로 블록레벨은 다시 여러 하위 블록들로 구성될 수 있으며, 시스템 레벨에서의 블록과 채널은 다시 여러 블록과 채널로 구성되는 블록 다이어그램으로 나타나는 하향식 순환적 표현 방법을 취하고 있다. 하지만 블록과 프로세스가 같은 계층에 존재할 수는 없고, 프로세스 하위에 다시 블록을 정의할 수도 없다.

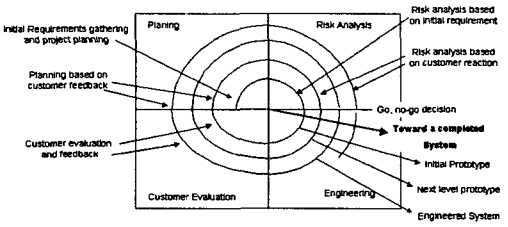


그림 2. 나선형 모델
Fig. 2 spiral model

세 번째로 프로세스 레벨은 블록이 몇 번에 걸쳐 하위 단계의 블록 다이어그램으로 쪼개진 후 실제 동작을 수행하고 자신의 지역 속성을 가지는 동적 객체이다. 프로세스간의 통신에는 위에서 설명한 것처럼 신호(signal)를 사용하므로, 모든 의존 관계는 프로세스 사이에 주고받는 신호로 표시되며, 프로세스의 동작과 주고받는 신호는 모두 불연속적이며 시간에 따른 연속적인 변화를 고려하지 않고 신호 입력 순서만을 생ㅊ각한다[4]. 즉, 하나의 블록은 여러 프로세스 사이의 정보 전달 통로인 신호의 집합으로 표현될 수 있으며, 한 블록 안에 존재하는 프로세스들 사이에는 신호 경로를 통해, 그리고 다른 블록에 존재하는 프로세스들 사이에는 블록을 연결하는 채널을 통해 신호를 주고받는다.

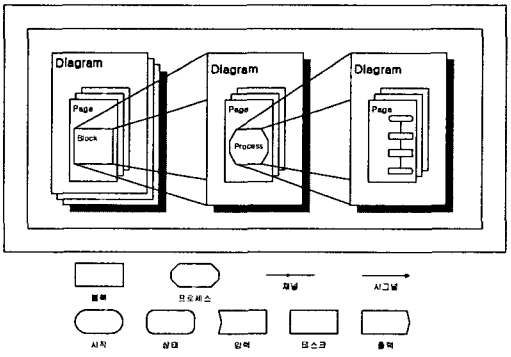


그림 3. SDL의 계층구조 및 심볼
Fig. 3 Hierarchy of SDL and symbols

본 논문에서 사용한 Telelogic Tau SDL Suite의 전체 구성을 그림 4에 나타내었다.

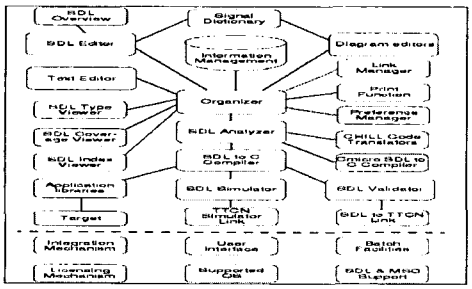


그림 4. SDL Suite의 전체구조
Fig. 4 Overview of SDL Suite

대략적인 구조는 Organizer에서 여러 부가 요소들을 사용해 디자인을 하며, SDL Analyzer로 제작한 시스템을 논리적으로 분석한다. 분석과정에서 이상이 없으면 C 컴파일러와 연동하여 C 파일을 생성해 내고, 시뮬레이터로 그 내용을 다시 검증한다. 마지막으로 TTCN(the Tree and Tabular Com- bined Notation) 시뮬레이션을 수행하여 검증과정을 마치게 된다[5],[6].

2.2 MSC(Message Sequence Chart)

MSC는 1992년에 ITU-T의 권고안 Z.120으로 표준화된 언어로서, SDL 시스템의 동적인 행동을 표현한다. 즉, 시스템을 구성하는 객체사이에 발생할 수 있는 동적인 경우를 모두 나타냄으로서 SDL 표현을 보다 수월하게 해준다. MSC는 사건 추적도와 비슷한 기본적인 구성요소(객체, 사건, 메시지)를 가지고 있다. 그러나 사건 추적도는 사건 발생의 시간적 관점보다는 발생하는 순서를 나타냄으로 상태 변화도를 보다 쉽게 그리기 위한 보조적인 전 단계인 반면, MSC는 SDL을 쉽게 그리기 위한 보조단계일 뿐만 아니라 타이머 객체를 두어서 시간을 고려하며 SDL 시스템의 시뮬레이션과 무결성 검증에도 사용된다[7].

MSC는 기본적으로 시스템 인스턴스(SDL에서의 블록, 프로세스, 또는 서비스의 객체), 환경(외부세계), 이들 사이의 메시지들로 구성된다. 객체(인스턴스)들은 수직선으로 나타내고 객체들 사이의 정보흐름은 메시지로 나타내며, 각 객체는 객체 시작기호로 시작하여 메시지들의 흐름을 나타내고 객체 종료기호에 의해 활동을 끝내게 된다. 간단한 MSC의 예를 그림 5에 나타내었다[5],[8].

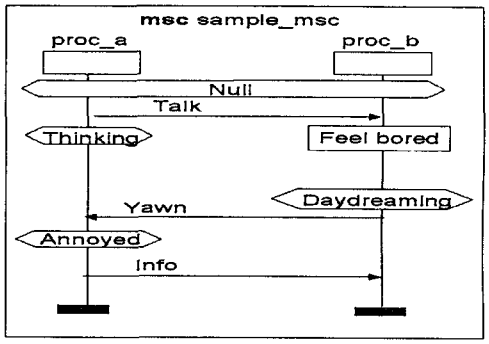


그림 5. MSC의 예
Fig. 5 Example of MSC

III. TCP 혼잡제어 알고리즘

본 논문에서는 현재 가장 많이 쓰이고 있는 TCP/IP 프로토콜에서의 혼잡제어 알고리즘을 SDL을 이용하여 설계하고 구현하며, 시뮬레이션을 통해 그것을 검증하여 보았다.

TCP/IP는 현재 데이터 전송 프로토콜로 가장 많이 사용되는 프로토콜이며, 데이터를 신뢰성 있게 전송하기 위하여 window 방식을 이용한다. 혼잡을 피하기 위해 slow-start, 혼잡회피, fast retransmit를 수행하는 Tahoe와 Tahoe의 세 가지 기능에 fast recovery를 추가시킨 Reno 알고리즘이 있다[9].

3.1 TCP-Tahoe

Tahoe는 초기에 slow-start를 시작하여 패킷을 송·수신한다. slow-start는 송신 측의 TCP에 cwnd (congestion window)라고 불리는 또 하나의 window를 사용하며(연결 설정 시 cwnd=1) 수신 측이 알려주는 윈도우 크기와 cwnd중 작은 값으로 전송을 시작한다. 전송 시작 후 보낸 패킷에 대한 Ack(Acknowledgement)을 받을 때마다 cwnd 크기를 1씩 증가시켜서 지수적으로 전송률을 증가시켜

나가다가 임계치인 ssthresh (slow-start threshold)에 도달하면 혼잡회피로 동작한다. 이 과정 전에 혼잡이 발생하여 송신 측에서 재전송 Ack을 받았을 때 이것이 손실인지 아니면 단순히 순서가 틀렸는지 알 수 없으므로 일단은 대기하다가 retransmit threshold 즉, Duplicate Ack threshold(예: 3개)이상의 중복 Ack가 연속으로 수신된다면 그 세그먼트를 손실로 보고 재전송 타이머가 만료되는 것에 상관없이 바로 재전송을 하도록 한다[10].

예를 들어, 현재의 cwnd 크기를 CW라하고 cwnd 임계치(ssthresh)를 CWt라고 가정한다면, 첫 번째로 송신한 데이터가 무사히 도착하여 정상적인 Ack응답이 오는 경우, 현재 CW < CWt인 상태라면 CW =CW+1씩 지수적으로 증가시켜 slow-start 단계를 수행한다. 그리고 이렇게 계속 전송률을 증가시키다가 CW = CWt로 같아진 이후에는 CW = CW + 1/CW씩 선형적으로 증가시켜서 혼잡회피 단계를 수행한다. 이러한 과정에서 혼잡이 발생하여 TCP/IP의 재전송 타이머가 만료된 경우나 하나 또는 다수의 데이터가 손상되어 그 데이터에 대한 중복 Ack (duplicate Ack, ex : 3개 연속)가 수신되는 경우에는 CWt를 현재 CW 크기의 1/2로 설정한다. 그리고 CW =1 로 설정하여 다시 slow-start로 복귀하여서 전송을 계속한다.

위에서 혼잡회피 단계는 CWt를 넘어도 혼잡이 발생하지 않으면 매 Ack마다 cwnd의 값을 1/CW 만큼 증가시켜 선형적으로 전송률이 증가하게 하여 slow-start보다 대역을 조금씩 늘려가며 혼잡을 피하는 방식이다. Tahoe의 경우에는 한 윈도우 내에 처음으로 손실된 패킷을 재전송 한 다음 차례로 slow-start를 실행하면서 이후의 모든 패킷들을 재전송 하므로 패킷손실이 발생하지 않아 안전성이 높다고 볼 수 있지만 결과적으로 수신 측에서 이미 수신한 패킷을 다시 수신하게 되는 문제점이 있으며, 모든 경우의 혼잡에 대해 CW=1로 설정되는 문제점이 있다[11].

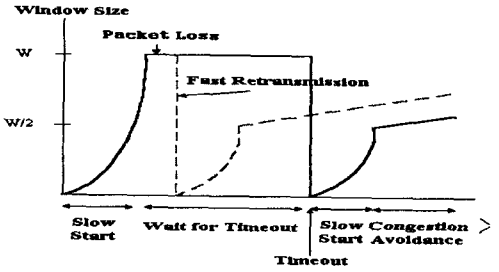


그림 6. Tahoe의 윈도우 크기 변화
Fig. 6 Valiation of Tahoe's window size

표. 1 Tahoe의 동작원리
Table. 1 Tahoe's operation

1) 정상적인 ACK의 경우	CW < CWt 이면 CW = CW + 1	slow-start
	CW = CWt 로 같아진 이후, CW=CW+1/CW	혼잡회피 단계
2) 타임아웃 또는 duplicate ACK가 수신된 경우	CWt = CW/2, CW = 1로 설정	다시 slow-start

Tahoe 알고리즘의 동작을 표 1에 나타내었으며, 그림 6에 윈도우 크기 변화를 나타내었다.

3.2 TCP-Reno

Reno는 Tahoe와 마찬가지로 초기에 slow- start를 시작하여 패킷을 송신하며 망에서 패킷 손실이 발생할 경우, 즉 특정 세그먼트에 대하여 duplicate Ack가 3개 이상 수신이 된 경우에는 그 세그먼트를 제외한 나머지 세그먼트들은 문제없이 전송되었다는 것을 의미함으로 다시 sl- ow-start로 돌아갈 필요없이 fast retransmit한 이후에 바로 혼잡회피로 동작해서 fast recovery를 수행한다. fast retransmit할 때의 cwnd는 (ssthresh + 3 x 세그먼트)이다. 이는 수신 측의 버퍼(buffer)에 3개의 패킷이 들어가는 동안 송신 측은 전송을 하지 않으므로 이를 보상하기 위해서이다. Tahoe에서는 cwnd = 1로 하고 slow-start를 실행했었다. time-out 시는 Tahoe와 마찬가지로 ssthresh = cwnd/2로 설정하고 cwnd = 1의 slow-start로 동작하다 혼잡회피 단계로 넘어가게 된다.

결과적으로 reno는 fast-recovery를 실행하는 점이 Tahoe와 다르다고 할 수 있다[11][12]. 예를 들어, 정상적인 Ack의 경우는 Taeho와 같이 cwnd를 CW,

cwnd thrreshold를 CWt라고 가정하였을 때, 초기에 CW < CWt인 상태에서는 CW = (CW + 1)씩 증가시켜서 slow start 단계로 동작하며, CWt값을 넘어가는 경우 CW = (CW + 1/CW)씩 증가시켜서 혼잡회피 단계로 동작한다. 중복 Ack가 수신된 경우 해당 패킷을 재전송하고 CWt = CW/2로 설정하고, CW = CWt로 설정하여 전송율 증가를 slow-start 과정으로 복귀하지 않고 혼잡회피 단계로 동작하여 fast recovery 과정을 수행한다. 이러한 reno의 윈도우 크기 변화를 그림 7에 나타내었다. 그리고 Tahoe Reno 알고리즘의 동작의 비교를 그림 8에 나타내었다.

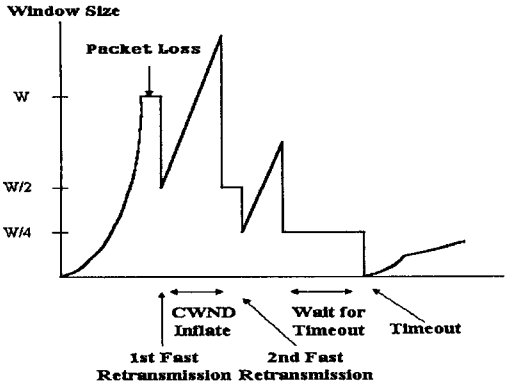


그림 7. Reno의 윈도우 크기 변화
Fig. 7 Validation of Reno's window size

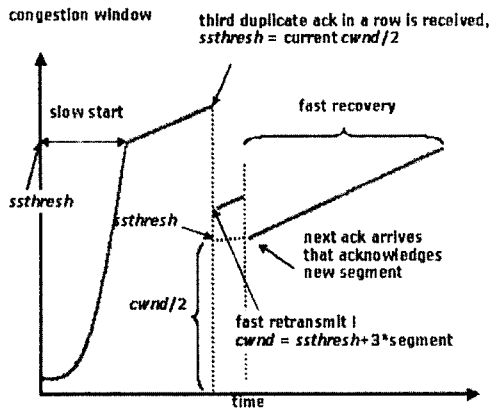


그림 8. Tahoe의 윈도우 크기 변화
Fig. 8 Comparison of Tahoe's window size

IV. 알고리즘 설계 및 구현

본 장에서는 SDL을 이용하여 3장에서 설명하였던 TCP/IP의 Reno 알고리즘을 SDL로 구현하였다. SDL의 계층구조인 시스템 레벨, 블록 레벨, 프로세스 레벨에서 알고리즘을 구현 할 때의 각 레벨에서의 역할과 동작을 살펴보겠다. 먼저 알고리즘을 구현하는데 사용한 Telelogic사의 SDL Tau Suite의 Organizer Window와 설계한 전체 시스템 구조를 그림 9에 나타내었다.

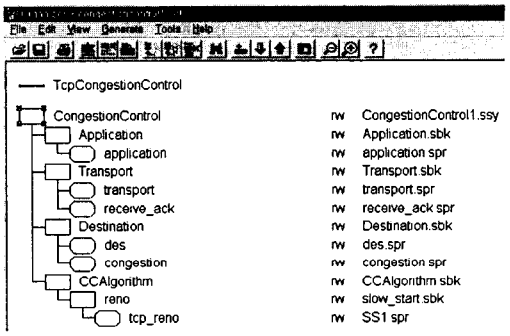


그림 9. 전체 시스템 구성
Fig. 9 System Configuration

전체적인 구성은 Application, CCAlgorithm, Transport, Destination의 네 부분으로 구성하였으며, 각 블록에 대한 대략적인 설명은 다음과 같다. Application 블록에서는 사용자와 직접 통신하는 부분으로, 사용자의 전송 요청을 받아들여서 CCAlgorithm 블록에 요청하여 자신이 전송할 데이터의 전송률을 결정하고 데이터를 전송 계층으로 넘기는 역할을 수행한다. CCAlgo-rithm 블록은 실제 전송률을 결정하는 알고리즘 부분으로 보내야 할 데이터 전송률을 계산하여 Application 계층으로 알리며, 수신측에서 보내는 Ack를 전송계층인 Transport 블록을 통해 받아들여 그것을 토대로 전송률 가감을 결정한다. Transport 블록은 상위의 Application에서 내려온 PDU(Protocol Data Unit)를 받아 목적지로 전송하고 목적지로부터의 확인 응답을 수신하는 직접 목적지와 통신하는 부분이다. 그리고 Destination 블록은 데이터를 받고 거기에 응답을 보내

며, 데이터를 수신하지 못하였거나 오류가 있는 경우 망의 혼잡으로 인식하고 혼잡 신호를 보내는 역할을 수행한다. OSI 7 Layer 계층구조에 비추어 생각해 볼 때, Application과 CC- Algorithm블록은 Transport층 이상의 상위 계층으로 볼 수가 있으며, Transport 블록은 Trans-port 계층 이하의 하위 계층으로 볼 수 있다.

시스템 레벨에서는 위에서 설명한 4개의 블록을 정의하고 이들간 주고받는 시그널을 보여줌으로서 블록을 통해 개괄적인 전체 전송 구조와 그 내부에서 알고리즘과 다른 구성 요소들이 어떻게 상호 동작하는지 개괄적으로 보여준다. 블록레벨에서는 상위 시스템 레벨에서 정의한 블록간의 시그널이 세분화되며, 각각의 블록들이 수행하는 기능들을 프로세스 단위로 분류하고 있다. 마지막으로 프로세스 레벨에서는 실제의 각 부분의 동작이 상세하게 명시되는 부분으로서 SDL 다이어그램을 통하여 처리 동작 과정을 상세하게 나타내었다.

4.1 시스템 레벨

시스템 레벨을 살펴보면 다음과 같다. 그림 10의 시스템 레벨은 SDL의 최상위 레벨로, 시스템의 모든 블록을 표현함으로서 전체 시스템을 쉽게 이해할 수 있도록 해준다. 시스템 레벨에서 블록간 혹은 외부 환경과 통신하는 패스를 채널이라고 하며, 그림의 c1부터 c8까지가 채널에 해당된다. 각 채널에서는 각 블록단위의 하부구조인 프로세스간의 통신하는 패스인 시그널들이 각 괄호('[]')속에 포함되어 나타내어진다. 그리고 시스템 레벨에서는 전체 시스템에서 사용하는 시그널들이 정의된다.

시스템 레벨에서의 논리적인 구성은, Appli- cation 블록이 user_request 시그널을 받아서 in- itialize 시그널로 CCAlgorithm 블록에 Conges- tion window 크기를 요청하고, CCAlgorithm 부분은 초기에는 cwnd 시그널로 전송률을 알리고 첫 번째 Ack가 도착한 이후부터는 cwndcurrent 시그널을 통하여 Application 블록에 전송률을 알리는데, 이에 대해서는 프로세스 레벨에서 설명한다. 그리고 혼잡 상황이 발생하여 재전송이 필요한 경우 retrans_request 시그널을 통하여 Application 블록으로 알린다. Application 블록에서는 CCAlgorithm에서 지정해 주는 전송률대로 데이터를 frame1, currentframesize등의 시그널을 통해 Transport 블록으로 전달하며, 재전송이 필요한 경우 retrans_frame 시그널을 통해 재전송 할 데이터를 Transport 블록으로 전달한다.

Transport 블록은 상위에서 받은 데이터를 out_frame 시그널을 통해 목적지로 전달하는 역할을 수행하며, 위에서와 마찬가지로 재전송 데이터는 retrans_out으로 전달한다. 목적지로부터의 수신응답을 ack_des 시그널로 받으며, 혼잡 상황이 발생한 경우 congestion 시그널을 받아 CCAlgorithm 블록으로 혼잡상황을 알린다. De-stination 블록은 가상적인 데이터 수신지로서, 받은 데이터에 대한 Ack를 보내고, 혼잡 시에는 congestion 시그널을 보낸다. 그리고 혼잡시의 동작을 구현하기 위하여 no_data와 break_data 시그널을 사용자 입력으로 삽입하였다.

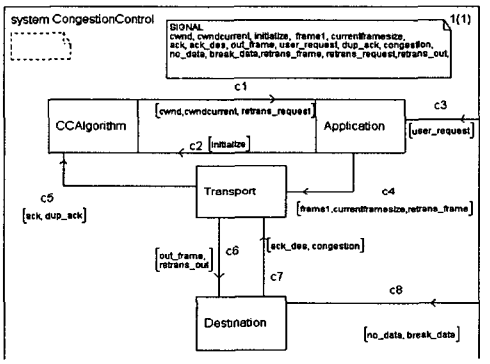


그림 10. 시스템 레벨
Fig. 10 System level

4.2 블록 레벨

블록레벨에서는 실제적인 동작을 명시하는 여러 프로세스들을 포함하여서 프로세스를 보다 편리하게 관리하도록 하고 있다. 블록은 다시 다른 하위 블록을 포함할 수 있으며, 한 계층 내에서 블록과 프로세스가 공존하는 것은 허용되지 않는다. 그리고 시스템 레벨에서도 프로세스가 포함될 수 없다.

첫 번째로 그림 11에서 CCAlgorithm 이라는 블록 하위에 다시 reno라는 블록을 정의된 형태이며, 그 아래 그림 12에 tcp_reno라는 프로세스 블록이

존재한다. 시스템 레벨에서의 c1, c2 채널이 하위 블록이 통신하는 boundary에 표시되어 있으며, 블록레벨에서는 채널의 이름을 식별하기 쉽게 indicate_window_size, request_ window_size로 변경하여 사용하였으며 그림 12에서는 s1, s2로 다시 변경되며, 두 경우 모두 상위 채널에서 정의된 시그널들이 그대로 사용된다.

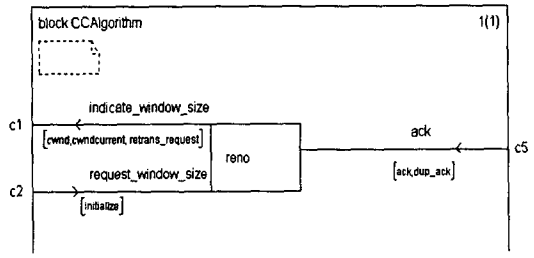


그림 11. 블록레벨 - reno 블록
Fig. 11 Block level - block reno

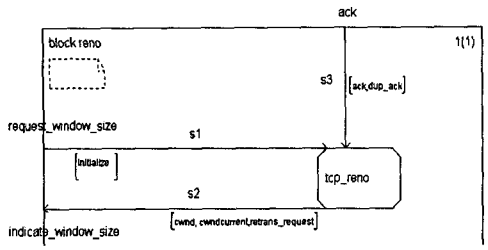


그림 12. 블록레벨 - tcp_reno 블록
Fig. 12 Block level - block tcp_reno

그림 13에서는 Application 블록의 내부 프로세스 블록을 보여 주고 있다. 상위 블록에서 정의된 c1, c2, c3, c6 채널이 boundary에 나타나 있으며, 각 채널의 이름을 시그널의 속성에 맞게 재정의 하였다. 그리고 역시 상위 레벨에서 사용된 시그널들이 그대로 사용된다.

그림 14에서는 Transport 블록의 내부 프로세스 블록들을 나타내었다. 여기서는 transport와 receive_ack 두 개의 프로세스 블록을 사용하였으며, transport 프로세스는 Application에서 오는 데이터를 받아 수신측으로 전달하는 역할을 수행하며, receive_ack 프로세스는 송신한 데이터에 대한 확인응답 시그널들을 받아 CCAlgori- thm 블록으로 전달하는 역할을 수행한다.

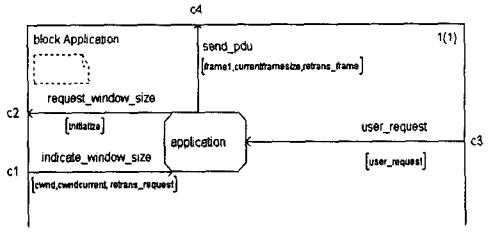


그림 13. 블록레벨 - application 블록
Fig. 13 Block level - block application

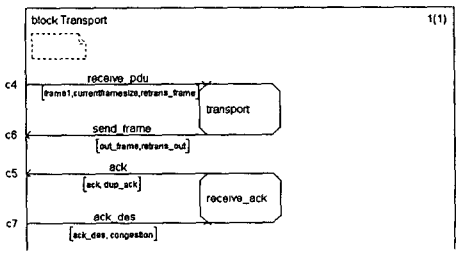


그림 14. 블록레벨-transport, receive_ack 블록
Fig. 14 Block level-block transport, receive_ack

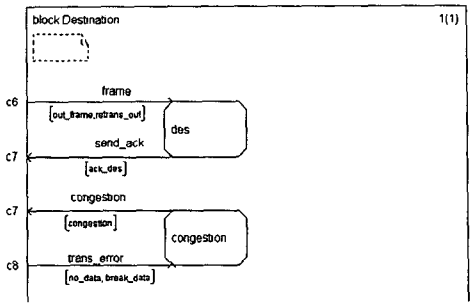


그림 15. 블록레벨 - des, congestion 블록
Fig. 15 Block level - block des, congestion

그림 15에서는 Destination 블록의 프로세스 블록을 나타내었다. des와 congestion이라는 두 개의 프로세스 블록으로 구성되어 있으며, des 프로세스는 송신지, 즉 Transport 블록에서 오는 데이터를 받아서 그것에 대한 확인응답을 수행하며, congestion 프

로세스는 데이터를 받지 못하였을 때나 받은 데이터가 손상되었을 때 그것을 망의 혼잡으로 인한 것으로 인식하여 혼잡을 알리는 시그널을 Transport 블록으로 전송하는 역할을 수행한다.

4.3 프로세스 레벨

프로세스 레벨에서는 시스템의 동작을 실제로 나타내며, 각 프로세스 블록 내부에 존재하며, 상위에서 정의된 시그널들이 실제로 어떻게 사용되어 어떻게 동작하는지 다이어그램으로서 명시화되는 부분이다. 2장에서 설명하였던 것처럼 현재 프로세스의 상태, 실제 시그널의 입력, 시그널을 받아 수행하는 작업과 출력 시그널, 그리고 필요한 연산과정들이 상세하게 나타난다.

첫 번째로 그림 16에 블록 CCAlgorithm의 하위계층인 tcp_reno 프로세스의 동작을 나타내었다. 여기서는 초기에 listen 상태로 Application 블록의 요청을 기다리다가, Application 블록에서 initialize 시그널로 전송율을 요청해 오면 slow- start 알고리즘에 따라 초기에 1인 cwnd 시그널을 보냄으로서 Application 블록이 전송하는 윈도우 크기를 1로 지정하고, 확인 응답을 기다리는 wait_ack 상태로 전이된다.

그 다음 Application 블록에서 지시한 전송 윈도우 크기대로 전송하여 Transport 블록을 거쳐서 목적지인 Destination 블록에 도착한 후 다시 Destination 블록에서 보내오는 Ack 시그널이 Transport 블록을 경유하여 도착하면, 현재 Ack와 threshold값을 비교하여, threshold 값을 초과하기 이전에는 slow-start 알고리즘과 같이 수신한 ACK수 x 2씩 지수적인 혼잡 윈도우 증가를 수행하며, threshold 값을 초과한 경우에는 수신되는 Ack수 x (1/threshold) 씩을 더하여 선형적으로 혼잡 윈도우 크기를 증가시키므로 혼잡회피로 동작하게 된다. 그리고 망에서 혼잡이 발생하여 혼잡을 알리는 신호인 dup_ack신호가 수신될 때, 이 입력이 정상적인 동작보다 우선 순위를 가지도록 priority input 심벌로 처리하였으며, 이런 경우 즉시 Application 블록으로 retrans_ request 신호를 보냄으로서 재전송을 요청하는데, 이것이 fast-retransmission 동작을 나타낸다. 그리고 혼잡 상황이 발생하였으므로 threshold 값을 조절하고 현재 혼잡 윈도우 크기를 조절하게 되는데, 혼잡 윈도우 크기가 임계치를 넘지 않은 과정에서 혼잡이 발생한 경우는 현재 임계치를 반으로 줄인 값을 새로운 임계치로 설정하고 혼잡윈도우 크기를 1로 줄여 slow-start로 복귀한다. 그리고 혼잡 윈도우 크기가 임계치를 초과한 상태에서 혼잡이 발생한 경우는 임계치를 현재 혼잡 윈도우 크기의 1/2로 설정하고 이 새로 설정된 임계치로부터 앞으로 수신될 Ack에 대해 Ack수 x 1/임계치 만큼 전송율을 증가시킴으로서 혼잡회피로 동작한다.

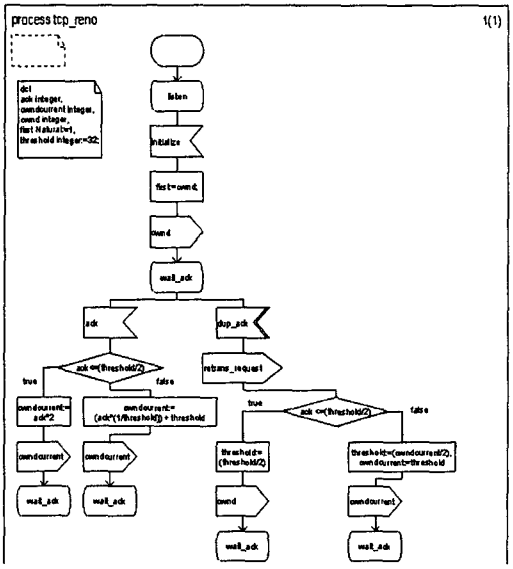


그림 16. 프로세스레벨 - tcp_reno 프로세스
Fig. 16 Process level - process tcp_reno

그림 17의 application 프로세스는 초기 listen_ user 상태로 사용자 입력을 기다리다가 사용자가 전송을 요청하는 user_request 시그널을 받으면 CCApplication으로 initialize 시그널을 보내어 혼잡 윈도우 크기를 요청하여 지정 받은대로 데이터를 전송한다.

초기에는 크기가 1인 혼잡 윈도우 사이즈를 지정받아 자신의 데이터를 전송하며, 이후부터 trans 상태가 계속 루프를 돌면서 혼잡 윈도우 크기를 받아들여 currentframesize 시그널로 데이터를 Transport 블록으로 전송하고, 혼잡 상황 시에는 tcp_reno

프로세스와 마찬가지로 priority input으로 혼잡 상황을 알리는 retrans_request 시그널을 받아들이고 손실이나 손상된 데이터를 재전송 한다.

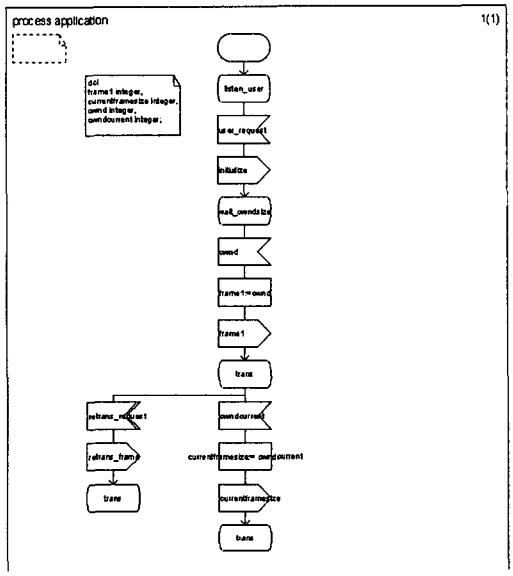


그림 17. 프로세스레벨 - application 프로세스
Fig. 17 Process level - process application

위의 블록레벨에서 설명하였던 것과 같이 Transport 블록에는 두 개의 프로세스 다이어그램이 있다. 첫 번째로 그림 18의 transport 프로세스는 application 프로세스로부터 PDU를 받아들여서 out_frame 시그널로 목적지로 전송한다. 앞에서 설명한 다른 프로세스와 마찬가지로 현재 자신의 전송하는 데이터 크기를 상위에서 지정한 크기와 같게 전송하고 있으며, transporting 상태가 계속 루프백 되면서 연속적인 전송을 수행한다. 재전송 요청인 retrans_frame 시그널을 받았을 때는 retrans_out 시그널을 통하여 데이터를 재전송 한다.

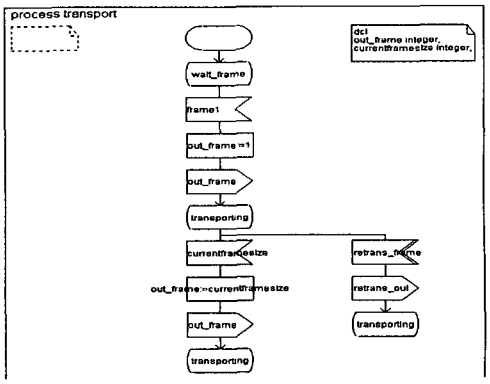


그림 18. 프로세스레벨 - transport 프로세스
Fig. 18 Process level - process transport

두 번째로 그림 19의 receive_ack 프로세스는 목적지로부터의 수신확인 응답을 받는 부분이다. ack_des 시그널로 정상적인 Ack를 받으며 받은 Ack를 CCAlgorithm 블록으로 전달하여 tcp_ reno 프로세스가 Ack 수를 보고 다음 혼잡 윈도우 사이즈를 결정할 수 있게 한다. 혼잡 상황이 발생하여 congestion 시그널을 받은 경우에는 dup_ack 시그널을 보내어 혼잡상황을 알린다. 그리고 waiting 상태가 루프백 되면서 전송 상태를 유지한다.

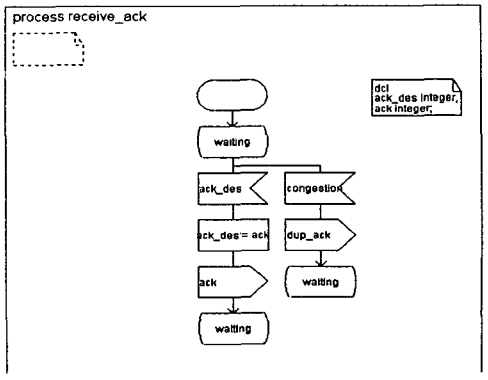


그림 19. 프로세스레벨 - receive_ack 프로세스
Fig. 19 Process level - process receive_ack

Destination 블록도 두 개의 프로세스로 구성되어 있는데, 첫째로 그림 20의 des 프로세스는 Transport 블록으로부터 데이터를 받아서 받은 데이터의 수만큼 ack_des 시그널로 수신 확인 응답을 보낸다. 혼잡상황으로 인한 재전송인 경우에는 역시 재전송에 대한 수신 확인 응답을 보낸다. 그리고 그림 21의 congestion 프로세스는 혼잡상황 시에 보내는 시그널들을 정의하고 있는데, 여기서는 실제 TCP/IP 통신이 아니므로 혼잡상황을 유도해야 하기 때문에 사용자 입력 시그널로 처리하였다.

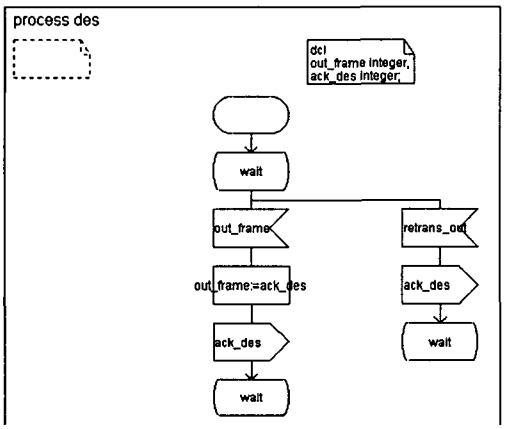


그림 20. 프로세스레벨 - des 프로세스
Fig. 20 Process level - process des

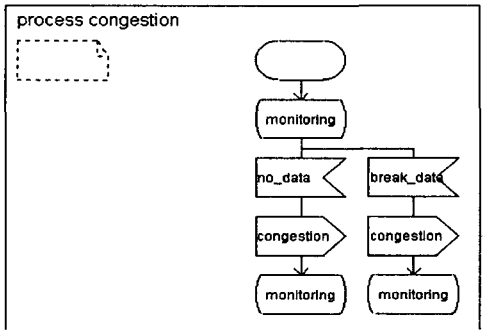


그림 21. 프로세스레벨 - congestion 프로세스
Fig. 21 Process level - process congestion

V. 시뮬레이션 및 고찰

본 장은 4장에서 구현한 내용을 SDL tau Suite를 통하여 C 컴파일러와 연동시켜 생성된 실제 C 코드와 환경 변수, 여러 구현에 관계되는 파일들을 생성해 내고, 그 결과로 생성된 파일을 그림 22에 나타내었으며, 이것을 이용하여 SDL에서 제공하는 시뮬레이터에서 구현한 과정을 검증하는 단계를 수행하여 보았다. 검증 및 분석은 SDL 시뮬레이터에서 제공되는 MSC Trace 기능을 이용하여 이해하기 쉽게 도식적으로 나타내었다.

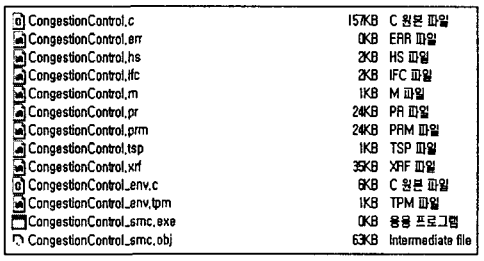


그림 22. SDL에서 생성된 파일
Fig. 22 Generated files by SDL

먼저 초기에는 그림 23과 같이 모든 프로세스가 첫 번째 상태에서 입력을 기다리게 된다.

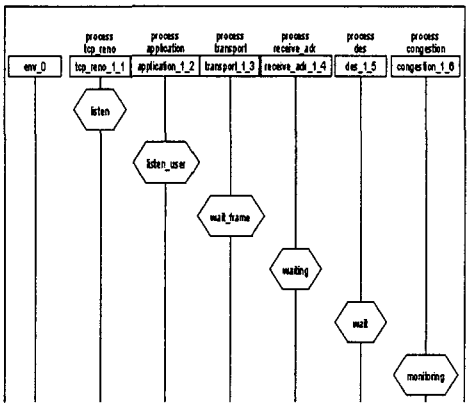


그림 23. MSC - 입력대기
Fig. 23 MSC - waiting signal

그 후 아래 그림 24와 25에서 user_request 시그널이 도착하면 순차적으로 트랜잭션이 일어나게 되는데 4장에서 구현한대로 처음에는 크기가 1인 혼잡윈도우를 보내는 시그널인 cwnd가 전송되고 있으며, 수신측까지 데이터가 전송되어 ack 시그널이 수신된 경우에는 현재 임계치와 비교하여 전송율을 결정하여 cwndcurrent시그널을 전송하고 있다. 연산 과정은 여기서는 보이지 않으며, 시간에 따라 순차적인 이벤트와 주고받는 시그널, 그리고 상태 전이를 표현하고 있다.

그리고 혼잡시의 동작을 살펴보기 위하여 no_data나 break_data 시그널이 전송된 경우를 아래 그림 26과 그림 27에 나타내었다. 그림 26은 임계치를 넘기 전 no_data 시그널이 수신된 경우이며, 그림 27은 임계치를 초과하였을 경우 break_data 시그널이 수신된 경우인데, 반대의 경우라도 두 시그널이 수행하는 기능은 같으므로 결과는 같다.

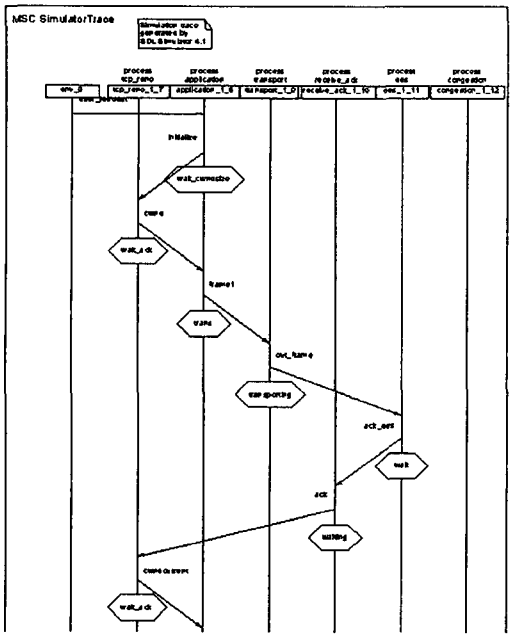


그림 24. MSC - 윈도우 크기 증가(1)
Fig. 24 MSC - Increase Window size(1)

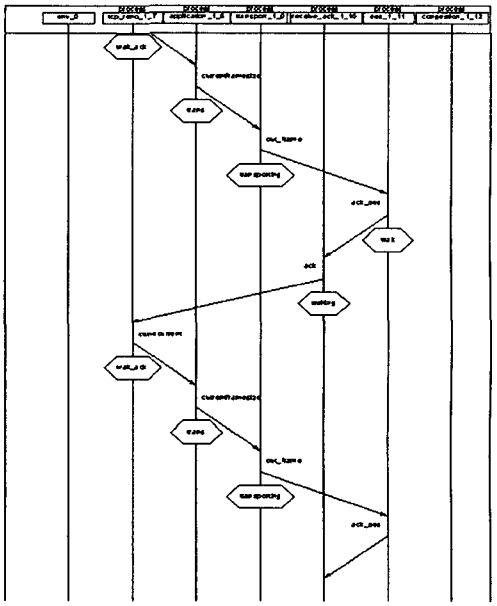


그림 25. MSC - 윈도우 크기 증가(2)
Fig. 25 MSC - Increase Window size(2)

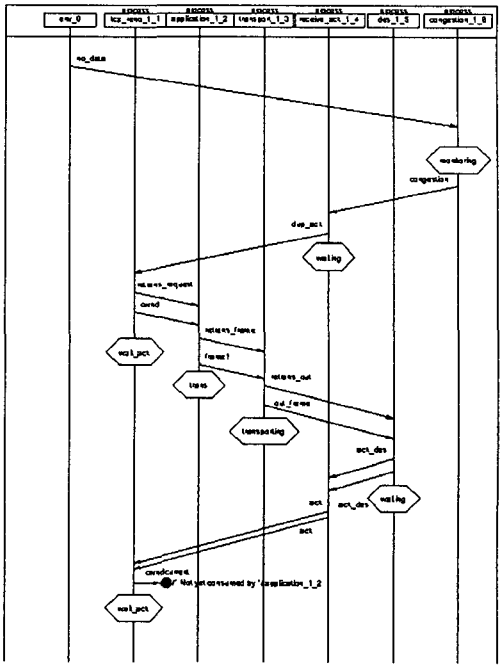


그림 26. MSC - 임계치 넘기전 혼잡 발생
Fig. 26 MSC - Congestion happen before threshold

그림 26에서 살펴보면, no_data 시그널을 받으면 congestion 프로세스에서 congestion 시그널을 Transport 블록의 receive_ack 프로세스로 보낸다. receive_ack 프로세스는 congestion 시그널을 받으면 CCAlg- orithm 블록의 tcp_reno 프로세스로 dup_ack 시그널을 보내어 혼잡을 알리며, tcp_reno 프로세스는 dup_ack 시그널을 받으면 재전송을 요청하는 retrans_request 시그널과 함께 현재 임계치를 넘지 않은 상태에서 혼잡이 발생하였기 때문에 새로운 임계치를 현재 임계치의 반으로 줄이고 cwnd 시그널을 보냄으로서 혼잡 윈도우 크기를 1로 초기화한다. 그림의 MSC에서는 두 개의 시그널이 같이 전송되는 것으로 표현되어 있는데 실제로는 한 프로세스당 동시에 두 개의 인스턴스가 수행 될 수 없기 때문에 순차적으로 엇갈려 전송되는 것을 이해하기 쉽게 정리하여 표현한 것이다.

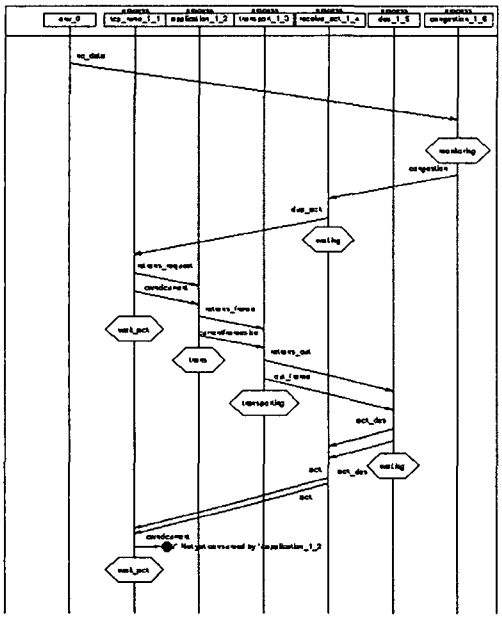


그림 27 MSC - 임계치 넘은후 혼잡 발생
Fig. 27 MSC - Congestion happen after threshold

그림 27의 경우는 현재 혼잡 윈도우 크기가 임계치를 넘어서 전송을 하고 있는 경우에 혼잡이 발생한 것을 나타낸 것으로 위의 그림 26과 수행하는 과정은 거의 같지만 혼잡이 발생하였을 때 임계치를 현재 전송되고 있는 혼잡 윈도우 크기의 반으로 줄이고 그 반으로 줄인 임계치로부터 전송을 혼잡회피로 동작시킴으로 fast- recovery 과정을 수행함을 보여준다.

VI. 결 론

본 논문에서는 SDL을 이용하여 TCP/IP혼잡제어 알고리즘을 구현하고 검증하여 보았다. 기존의 text 기반의 개발 방법과는 달리 SDL을 이용하여 프로토콜을 구현할 때에는 구현내역을 명확하게 표시하는 것이 가능하며, 명시한 내용이 사용자나 개발자의 새로운 요구사항에 의해 바뀌더라도 요구되는 기능에 대한 블록과 다이어그램의 추가로 간편하게 변경 내용을 수정할 수 있으며 각 단계적인 분석 및 검증이 가능하다.

그리고 계층 구조를 그래픽으로 표현함으로서 타 언어로 구현하는 것보다 쉽게 이해되며 이것은 초보 개발자나 숙련된 개발자에게도 개발시의 편의를 제공한다. 또한 시뮬레이션과 검증도구를 포함하고 있어 실제의 어플리케이션으로 구현하기 전에 구현한 내용을 확인할 수 있으므로 완성도를 높일 수 있다.

또한 본 논문에서 구현한 TCP/IP 혼잡제어 알고리즘 뿐 아니라 다른 언어를 써서 어플리케이션을 개발할 때에도 정확한 요구사항의 분석과 각 개발단계에서의 분석 및 검증 작업이 이루어진다면 얻을 수 있는 이점이 훨씬 많을 것이다. 그리고 이렇게 구현된 TCP/IP 혼잡제어 알고리즘을 토대로 하여 SDL이 가지는 손쉬운 추가기능으로 보다 향상된 TCP/IP프로토콜 개발도 가능할 것이다.

따라서 향후 연구 과제로는 여기서 시뮬레이션하고 검증하였던 내용을 실제 TCP/IP 소켓 통신에 적용하여 보는 것과 SDL을 통한 알고리즘 기능향상을 수행할 것이다.

참고문헌

[1] F. Hessel, P. Coste, P. LeMarrec, N. Zergainoh, Jm. Daveau, A. A. Jerraya,

"Communication Interface Synthesis for Multi-language Specifications", Proceedings of the 10th IEEE International Workshop on Rapid System Prototyping, pp.15-20, 1999. 6.

[2] Hintelmann J. Hofmann R. Lemmen F. Mitschele-Thiel A. Muller-Clostermann B. "Applying techniques and tools for the performance engineering of SDL systems", Computer Networks - The International Journal of Computer & Telecommunications Networking, V.35, N.6, pp.647-665, 2001. 5

[3] Jan Ellsberger, Dieter Hogrefe, Amardeo Sarma, "SDL Formal Object-oriented Language for Communication Systems", Prentice Hall, 1998

[4] 임지영, 김희정, 임수정, 채기준, 이미정, 최길영, 강훈 "SDL을 이용한 MPOA 설계 및 구현", 정보과학회 논문지 : 컴퓨팅의 실제 제6권 제6호 2000. 12

[5] Copyright by Telelogic AB "Simulation & C Generation Using SDL", 1999

[6] Copyright by Telelogic AB, "Introduction to SDL and SDT" Revision 3.02, 1995

[7] Frederic Boutet, Gilles Rieux, Yves Lejeune and Eric Choveau, "Scheduling in SDL simula- tion : Application to Future Air Navigation Systems", 1998

[8] J.M. Alvarez, M. Diaz, L.M. Llopis, E. Pimentel, J.M. Troya, "SDL and Hard Real- Time System : New Design and Analysis Techniques", 1998

[9] W. Stevens, "TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms", RFC 2001, Jan., 1997.

[10] T. V. Lakshman, U. Madhow, B. Suter, "Window-based error recovery and flow control with a slow acknowledgment channel : a study of TCP/IP performance", Proc. Infocom 1997, Apr., 1997.

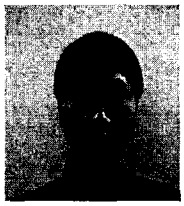
[11] Kevin Fall, Sally Floyd. "Comparisons of Tahoe, Reno, and Sack TCP", Dec., 1995.

[12] Jacobson,V. "Congestion Avoidance and Control", In Proceeding of SIGCOMM, 1998.

저자소개

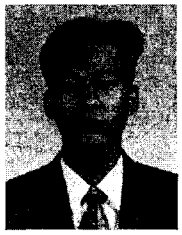
이재훈(Jae-Hoon Yi)

1981년 동아대학교 전자공학과(공학사)
1985년 한양대학교 산업대학원(공학석사)
2003년 한국해양대학교 전자통신공학과(박사과정)
1985년~1994년 선경매그네틱(주) 개발실장
1996년~2000년 유정정보시스템 대표이사
2001~현재 유정시스템(주) 대표이사
※관심분야: 네트워크, 정보통신, 무선인터넷



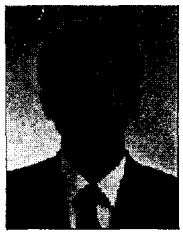
조성현(Sung-Hyoun Cho)

2000년 한국해양대학교 전자통신공학과(공학사)
2002년 한국해양대학교 전자통신공학과(공학석사)
2001년~현재 유정시스템(주) 연구원
※관심분야: 이동통신 단말기, 무선인터넷



이태오(Tae-Oh Lee)

1997년 한국해양대학교 전자통신공학과(공학사)
1999년 한국해양대학교 전자통신공학과(공학석사)
2001년 한국해양대학교 전자통신공학과(박사수료)
1997년~1999년 한국해양대학교 전자통신공학과 조교
2000년~현재 동명정보대학교 정보공학부 컴퓨터공학과 기간제 전임강사
※관심분야: 위성/선박통신, 무선인터넷, GPS



임재홍(Jae-Hong Yim)

1986년 서강대학교 전자공학과(공학사)
1988년 한양대학교 대학원 전자공학과(공학석사)
1995년 한양대학교 대학원 전자공학과(공학박사)
1995년~현재 한국해양대학교 전파·정보통신공학과 부교수
※관심분야: 컴퓨터 네트워크, 통신 프로토콜, 임베디드 시스템, 분산 컴퓨팅