일반적 모델의 분산 교착상태의 신속한 해결 기법
(A Fast Resolution Algorithm for Distributed Deadlocks in the Generalized Model)

이 수 정 †
(Soojung Lee)

요 약 일반적 모델 기반의 분산 교착상태 문제를 해결하기 위한 대부분의 알고리즘들은 diffusing computation이라는 기법을 이용하였는데 이 기법의 주된 특징은 PROBE를 전파하고 그에 따른 응답 메시지에 교착상태 발견에 필요한 정보를 전달하는 것이다. 신속한 교착상태의 발견은 매우 중요하기 때문에 본 연구에서는 응답 메시지 대신 PROBE 상에 교착상태 발견을 위한 정보를 전달하게 한다. 이는 응답 메시지의 역전송 과정을 불필요하게 하기 때문에 기존 알고리즘에 비해 시간을 거의 두 배로 단축시키는 결과를 가져온다. 또한, 기존 알고리즘은 단지 알고리즘이 한번만 실행되는 경우를 고려하였으나 본 연구에서 제시한 알고리즘은 동시 수행하는 경우를 효율적으로 처리하여, 교착상태를 발견하는 시간을 더욱 단축시킬 수 있다. 제안된 알고리즘의 성능은 시뮬레이션을 통하여 타알고리즘들과 비교하였다.

키워드 : 분산시스템, 교착상태, 분산 교착상태, 교착상태 발견, 교착상태 해결, 분산 알고리즘

Abstract Most algorithms for handling distributed deadlock problem in the generalized request model use the diffusing computation technique where propagation of probes and backward propagation of replies carrying dependency information between processes are both required to detect deadlock. Since fast deadlock detection is critical, we propose an algorithm that lets probes rather than replies carry the information required for deadlock detection. This helps to remove the backward propagation of replies and reduce the time cost for deadlock detection to almost half of that of the existing algorithms. Moreover, the proposed algorithm is extended to deal with concurrent executions, which achieves further improvement of deadlock detection time, whereas the current algorithms deal only with a single execution. We compare the performance of the proposed algorithm with that of the other algorithms through simulation experiments.

Key words : distributed systems, deadlocks, distributed deadlocks, deadlock detection, deadlock resolution, distributed algorithms

1. 서 론

분산시스템에서 프로세스들이 서로 간에 리소스에 대한 요청이 허락되길 무한정 기다리는 상태가 발생하면 이를 교착상태라고 일컫는다. 교착상태는 리소스 사용도를 저하시키고 프로세스의 진행을 방해하기 때문에 이의 신속한 발견 및 복구는 매우 중요하다. 분산시스템에서 프로세스들간의 의존 관계는 wait-for graph(대기그래프, WFG)라고 불리는 방향성 그래프(directed graph)로 모델화할 수 있다[1]. 이 그래프에서 각 노드는 프로세스를 의미하고 간선(edge)은 리소스를 대기 중인 프로세스로부터 그 리소스를 사용중인 프로세스로 이어진다.

분산시스템에서 수행 중인 프로세스들을 대상으로 여러 요청 모델들이 정의되었다[1]. AND 요청 모델에서는 임의의 프로세스가 요청한 모든 리소스들이 허락되어야만 그 프로세스는 계속 진행할 수 있다[2-6]. 한편, OR 요청 모델에서는 임의의 한 리소스에 대한 요청만 허락되면 프로세스는 수행을 지속한다[7-9]. 보다 일반적인(Generalized) 요청 모델에서는 대기중인 프로세스가 수행을 재개하는 조건을 AND와 OR 연산자 및 요청한 리소스들을 포함하는 함수로 표현한다[10-12]. 일반적인 모델은 분산 운영체제 시스템과 통신 프로세스 그룹 등 여러 분야에서 찾아볼 수 있다.

· 이 논문은 2003년도 한국학술진흥재단의 지원에 의하여 연구되었음 (KRF-2003-003-D00332)
† 정 회 원 : 경인교육대학교 컴퓨터교육과 교수
sjlee@mail.inue.ac.kr
논문접수 : 2003년 10월 2일
심사완료 : 2004년 1월 27일

교착상태 발견 알고리즘들은 가정하는 요청 모델에 따라 분류될 수 있으며 교착상태는 전제하는 요청 모델에 따라 다르게 정의된다[1]. AND 요청 모델에서 프로세스는 요청한 모든 리소스가 허락되어야만 진행할 수 있기 때문에 이 모델에서 교착상태는 WFG 내 싸이클(cycle) 존재여부로 알 수 있다. OR 요청 모델에서는 WFG 내에 knot가 존재하면 교착상태를 의미한다. Knot는 WFG의 부분 그래프로서 이에 속한 임의의 노드는 같은 knot에 속한 다른 모든 노드들에게로 경로가 존재한다[8]. 일반적 모델에서 교착상태는 WFG 내에 보다 복잡한 위상과 관련되어 있다. 이 모델에서 싸이클은 교착상태의 필요조건이지만 충분조건은 아니다.

AND와 OR 모델 하에서 교착상태의 발견 문제는 광범위하게 연구되었으나 일반적 모델에서는 단지 몇몇의 연구결과만이 알려져 있다[10-14]. 이들 알고리즘의 공통된 특징은 Dijkstra와 Sholten[15]이 제시한 diffusing computation을 이용하는 것이다. 이 계산 방법에서는 알고리즘의 시작노드가 probe라는 특정 메시지를 전파하고 그에 대한 응답 메시지를 수신한다.

Bracha와 Toueg[10]가 제시한 방법에서는 알고리즘의 시작노드가 probe를 전송하고 진행상태에 있는 노드들로부터 응답 메시지가 역으로 전파된다. 시작노드가 대기상태에서 진행상태로 전환할 만큼의 충분한 수의 응답 메시지를 수신하면 시작노드는 교착상태에 속하지 않음을 알 수 있다. 응답 메시지가 수신되는 종료 시점을 파악하기 위하여 이들의 알고리즘은 또다른 형태의 메시지를 필요로 한다. 이에 따라 e를 알고리즘을 실행하는 WFG의 간선 수라고 할 때 4e 메시지가 요구된다. 모든 메시지는 고정 길이를 가졌고 노드 상태에 대한 정보를 전혀 전달하지 않기 때문에, 단지 시작노드가 교착상태에 속하였는지의 여부만을 알 수 있다.

Brzezinski[13]는 프로세스들이 논리적인 환(ring)으로 연결되어 있음을 가정하였다. 프로세스의 상태가 변화하는지를 알기 위하여 토큰을 환 상에서 순환시킨다. 프로세스의 상태가 변화하는 한 토큰을 지속적으로 순환된다. 토큰은 최종적으로 교착상태에 속한 프로세스들의 식별자를 시작노드에게 전달한다. n을 WFG 내의 노드 수라고 할 때 이 알고리즘은 교착상태를 발견하기 위하여 4n의 시간과 O(n) 길이의 메시지 n*n/2 개를 필요로 한다.

Chen의 알고리즘[14]은 probe 전송과 그에 대한 응답 메시지를 수신하는 두 가지 단계를 반복적으로 수행한다. 시작노드는 각 노드에게 probe를 단 한번씩만 전송한다. Probe를 수신하였을 때 노드는 자신과 인접하는 노드들의 식별자를 응답메시지를 통하여 직접 시작노드에게 전송한다. 이러한 정보를 토대로 시작노드는 점차적으로 국부적인 WFG를 구축한다. WFG의 폭을 d라고 했을 때 제시된 알고리즘은 2d의 시간과 2n의 메시지 수를 요구한다. Chen의 알고리즘[14]에서는 시작노드가 알고리즘의 실행을 통제하는 반면에 Kshemkalyani의 알고리즘[11]은 각 노드가 교착상태에 필요한 정보를 유지보관하도록 한다. 이러한 정보는 노드 식별자들과 각각에 해당하는 리소스 요청조건이다. 제시된 알고리즘은 O(e)의 메시지 길이로써 2d 시간과 2e의 메시지 수를 필요로 한다.

분산적이건 중앙집중적이건 간에, 위에서 언급한 알고리즘들은 교착상태를 발견하기 위해 시작노드가 정보를 수집한다. 응답메시지는 기본적으로 프로세스들간의 의존관계에 대한 정보를 전달한다. 그러나, 본 연구에서 제안한 알고리즘은 교착상태를 신속히 발견하기 위하여 응답메시지 대신 probe를 통하여 필요한 정보를 전달한다. Probe가 전파됨에 따라 각 노드의 리소스 요청정보는 통합되어 전달된다. Leaf 노드에서 이제까지 수집된 정보는 시작노드로 직접 전달된다. 결과적으로, 교착상태 발견시간은 기존 알고리즘에 비해 거의 두배로 단축된다.

분산 교착상태를 발견하는 문제의 어려움은 다수의 알고리즘이 각자 독립적으로 진행되는데 있다. 즉, 한 알고리즘의 실행 결과로서 임의의 프로세스가 폐기될 때 이러한 사실을 동시 진행 중인 다른 알고리즘들은 알지 못할 수 있다는 것이다. 이는 교착상태를 오류 판단하는 이유가 될 수 있다. 그럼에도 불구하고, 동시 수행되는 알고리즘에 대한 고려는 거의 이루어지지 않았거나[13] 혹은 우선순위를 사용하여 낮은 우선순위의 알고리즘을 단순히 중지시키는 방식을 취하였다[10,11]. 한편 Chen[14]은 우선순위를 사용하지 않고 교착상태가 발견된 후 또다른 형태의 메시지를 이용하여 진부한 정보를 변경시키도록 하였다.

본 연구에서 제시한 방식은 앞에서 언급한 기존 알고리즘들의 단점을 극복한 것으로, 추가적인 메시지를 필요로 하지 않으며, 우선순위가 낮은 알고리즘을 중지시킴으로써 그의 실행 시간을 낭비하지도 않는다. 가장 주된 관심사는 교착상태를 보다 신속히 해결하는 것이다. 기본적으로, 각 알고리즘의 실행은 우선순위가 부여되고 각 노드는 우선순위가 가장 높은 알고리즘만을 실행하도록 한다. 만약 낮은 우선순위의 알고리즘이 보다 높은 우선순위의 알고리즘을 만나면, 전자는 중지되지 않고 계속 실행하도록 하는데, 이는 높은 우선순위의 알고리즘이 도달할 수 없는 WFG 영역 상에서 존재할지도 모르는 교착상태를 해결하기 위함이다. 그러므로, 교착상태를 보다 신속하게 발견할 수 있다.

제안된 알고리즘은 2e의 메시지 수와 d+2의 시간을

소모한다. 동시 진행되는 다수의 알고리즘에 대한 성능 평가는 프로세스의 역동적인 상태 변화로 인해 가늠하기 어려우므로 시뮬레이션을 통하여 행하였다. 기대했던 대로, 본 연구의 알고리즘이 기존보다 훨씬 신속하게 교착상태를 해결하는 것으로 판명되었다.

논문의 나머지는 다음과 같이 구성되었다. 다음 절에서는 시스템 모델과 정의에 대한 내용을 기술한다. 3절은 본 연구의 알고리즘을 소개하고 4절에서 이의 성능 분석을 제시한다. 5절에서는 논문의 결론을 맺는다.

2. 시스템 모델

분산시스템은 컴퓨터 네트웍으로 연결된 사이트들로 이루어진다. 시스템은 공통 메모리를 갖지 않으며 임의의 두 사이트간에 메시지를 송수신함으로써 통신한다. 본 논문에서는 메시지 전송시간을 예측할 수 없는 비동기 네트웍을 가정한다. 그러나 메시지는 손실되지 않고 오류 없이 목적지에 도달한다. 시스템 내의 메시지는 계산(computation) 메시지와 제어(control) 메시지로 구분한다. 계산 메시지는 프로세스의 수행에 필요한 REQUEST, REPLY 및 CANCEL 메시지이며 제어 메시지는 교착상태 발견을 위한 알고리즘이 발생시킨다.

프로세스의 상태는 대기(blocked) 또는 진행(active) 상태 중 하나이다. 프로세스가 리소스 요청을 위해 REQUEST 메시지를 발생시킬 때 대기상태로 돌입한다. REPLY 메시지는 요청이 허락되었음을 의미한다. 일단 프로세스가 대기상태가 되면 더 이상의 REQUEST 메시지를 보내지 못한다. 일반적 모델에서는 프로세스가 하나의 REPLY를 수신하였다 할지라도 진행상태로 전환하지 못할 수도 있다. 대기 상태의 프로세스가 진행상태가 되기 위한 조건을 요청조건(requesting condition)이라 부르기로 하고 이는 요청한 자원과 AND 및 OR 연산자로 표시한다. 예를 들어, 프로세스 i의 요청조건 RC_i가 $(p \wedge q) \vee r$이라고 하면 프로세스 i는 프로세스 p와 q로부터 승인을 얻거나 또는 r로부터 승인을 얻으면 진행 상태로 전환한다. 이 때 p, q와 r은 i의 계승자(successor)라고 하고, i는 그들의 선행자(predecessor)라고 한다. 프로세스가 진행상태로 전환하면 자신이 요청한 다른 리소스에 대해서 CANCEL 메시지를 보내어 취소한다. 진행 상태의 프로세스는 계산 메시지와 제어 메시지를 모두 보낼 수 있으나 대기상태의 프로세스는 제어 메시지만을 보낼 수 있다.

일반적 모델 하에서 교착상태는 하나 이상의 프로세스의 요청조건이 결코 만족될 수 없을 때 발생한다. 구체적 정의는 다음과 같다.

정의 1. eval(RC_i) = eval(RC_i | $\forall$j$\in$domain(RC_i), j$\leftarrow$eval(RC_j))로 정의한다. 단, 진행상태의 노드 i에 대해서 eval(RC_i)=true이고 domain(RC_i)는 i의 계승자 집합이다. 회귀적 표현식인 eval(RC_i)는 i의 계승자인 각 j가 eval(RC_j)로 대체됨을 의미한다. 집합 D는 다음 두 조건이 만족하면 일반적 교착상태에 속한 것으로 판단한다.

(i) eval(RC_j)=false, $\forall$i$\in$D,

(ii) D 내의 임의의 두 노드간에 계산 메시지(computation message)가 전송되지 않는다.

분산 시스템에서, 노드는 일정 시간 동안 대기 상태를 유지할 때 교착상태를 발견하기 위한 알고리즘을 시작한다. 교착상태가 발견되면 그에 속한 임의의 노드를 선택하여 중지시킴으로써 이를 해결한다. 동적인 환경에서는 알고리즘이 수행되는 동안 WFG 구성이 변경될 수 있다. 따라서 알고리즘이 교착상태의 부재를 선언할 때 마침 시스템 내에 교착상태가 발생하였을 수 있다. 그러나 알고리즘의 분산적 성질에 의거하여 이 교착상태는 추후의 알고리즘 실행에 의해 발견된다. 그러므로 정확한 교착상태 발견 알고리즘은 다음의 두 성질만을 보장하게 된다; (i)(Liveness) 알고리즘 시작시에 교착상태가 존재하면 알고리즘은 이를 발견하며 (ii)(Safety) 알고리즘이 교착상태를 선언하면 알고리즘 종료시에 실제로 교착상태가 존재한다.

3. 교착상태 발견 및 복구 알고리즘

3.1 알고리즘 개요

임의의 노드가 알고리즘을 시작하므로, 다수의 알고리즘이 동시 수행될 수 있으나, 각각은 독립적이고, 시작노드 식별자와 시작시간으로 구별한다. 본 절에서는 하나의 실행에 초점을 맞추어 기술하기로 한다.

알고리즘의 시작노드는 probe를 전파함으로써 distributed spanning tree(분산 신장 트리, DST)를 형성한다. 노드가 만약 처음으로 probe를 받게 되면, DST 상에서 probe 송신자의 자노드(child)가 되고 자신의 계승자에게 probe를 전달한다. Probe는 진행상태의 노드나 이미 트리의 구성원이 된 노드를 만날 때까지 계속 전달된다. 이 때 응답메시지가 시작노드에게 직접 전달된다. 따라서 응답 메시지는 진행상태의 노드가 보내거나 non-tree 간선이 발견되었을 때 보내진다.

Probe는 노드들의 요청조건을 전달한다. 즉, 시작노드는 자신의 요청조건을 임의의 한 계승자에게 전달하고, 후자는 전달받은 정보와 자신의 요청조건을 함께 자신의 계승자에게 또한 전달한다. 이 때 균등한 메시지 길이를 유지하기 위하여 요청조건들은 각 계승자에게 배분된다. 응답 메시지는 자신과 상응하는 probe가 전달한 정보를 그대로 시작노드에게 보낸다. 그러나, 진행상태의 노드는 이에 덧붙여서 자신의 식별자를 시작노드에

게 보낸다. 따라서, 시작노드는 마침내 모든 대기상태의 요청조건과 모든 진행상태인 노드들의 식별자를 알게 된다. 모든 응답 메시지를 수신하면, 시작노드는 각 요청조건을 평가하고, 만약 만족될 수 없는 요청조건이 발견되면 그에 해당되는 노드가 교착상태임을 선언한다.

시작노드에서 응답메시지의 수신완료에 대한 인식은 중량(weight)를 분산하는 기술을 이용한다 [12]; 시작노드는 노드들에게 1의 중량을 분산시키고 응답메시지를 통하여 중량을 수집하며 수집된 중량의 합이 1이 되면 모든 응답메시지를 수신한 것으로 판단한다. 보다 상세히 설명하면, 만약 시작노드가 n개의 계승자를 가졌으면 각 계승자에게 1/n의 중량을 probe를 통하여 송신한다. 이와 마찬가지로 임의의 노드가 probe를 통하여 w의 중량을 받으면 자신의 계승자들에게 probe를 통하여 각기 w/n의 중량을 나눠준다(n개의 계승자를 가정). 응답메시지는 probe가 전달하는 중량을 그대로 시작노드에게 전달한다.

역동적인 환경에서, 노드 i가 자신의 계승자 j에게 probe를 전송 중일 때 마침 동시에 노드 j가 REPLY 메시지를 노드 i에게 전송할 경우가 있다. 이러한 경우 교착상태를 오류 판단할 수 있으므로, 노드 j는 probe를 수신하게 되면 ¬(i, j)의 정보를 시작노드에게 전달한다. 시작노드는 응답 메시지가 전달하는 요청조건 정보들을 R이라는 집합을 통하여 유지 보관한다. 모든 응답 메시지를 받았을 때 시작노드는 R 상에서 replace 연산을 수행하는데 이는 ¬(i, j) 형태의 정보를 반영시키기 위함이다. 이 연산에 대한 구체적 정의는 다음과 같다.

정의 2. replace(R) = {i→RC_i ∈ R | RC_j←rep_j(RC_i) if ¬(i, j)∈R}. 이 때, rep_j(RC_j)는 RC_j에서 j 대신 true로 대체하여 평가함을 의미한다. 즉, RC_i의 임의의 서술식 x에 대해서, x∨j←∅이고 x∧j←x이다.

교착상태가 발견되면, 기존의 여러 기법을 이용하여 희생 노드를 선택할 수 있다. 단, 요청조건이 만족되지 못한 노드들 중에서 선택해야 한다. 이러한 과정은 모든 요청조건이 만족될 때까지 반복된다. 임의의 노드 i에서 실행하는 알고리즘의 pseudo-code는 다음과 같다.

노드 i를 위한 자료 구조(초기값은 괄호 안에 있음)

- father_i: 노드 i에게 probe를 처음 전달한 노드. (0)
- weight_i: 노드 i의 중량 값. (0)
- RC_i: 노드 i의 요청조건.

시작노드 init을 위한 추가적 자료 구조(초기값은 괄호 안에 있음)

- R_init: x→RC_x 형태의 대기 관계를 포함하는 집합. (∅)
- A_init: 진행상태의 노드들의 집합. (∅)

메시지 형식(w 인자는 중량 값을 의미)

- PROBE(init, R, j, w): 노드 j가 보내는 probe 메시지. init은 알고리즘의 시작노드.
- REPORT(R, A, w): probe에 대한 응답 메시지. A 집합은 진행상태의 노드 포함.

```
(I) 노드 i가 알고리즘을 시작할 때:
    proc_probing(i, {i→RC_i}, 1);
(II) 노드 i가 PROBE(init, R, j, w)를 수신할 때:
(II.1) if i≠init and father_i=0 then begin/* tree 간선 */
(II.1.1)  father_i := j;
(II.1.2)  if i is active or i has sent a REPLY message to j then send REPORT(R, {i}, w) to init;
(II.1.3)  else if i is blocked then proc_probing(init,R∪{i→RC_i},w); end if
(II.2) else begin /* non-tree 간선 */
(II.2.1)  if i has sent a REPLY message to j then send REPORT(R∪{¬(j, i)},∅,w) to init;
(II.2.2)  else send REPORT(R, ∅, w) to init;
      end else
(III) 시작노드 init이 REPORT(R, A, w)를 수신할 때:
      weight_init := weight_init + w;
      R_init := R_init ∪ R;
      A_init := A_init ∪ A;
      if weight_init < 1 then return;
      R_init := replace(R_init);
      if R_init≠∅ then evaluation_rtn(R_init, A_init);
      if R_init≠∅ then resolution_rtn(init, R_init, A_init); /* 교착상태 존재 */
(IV) procedure proc_probing(init, R, w)  /* 노드 i에서 수행 */
      n := number of successors;
      /* R을 계승자들에게 분배 */
      send PROBE(init,R_j,i,w/n) to each successor j where R_j⊂R and ∪R_j=R;
    end procedure
(V) procedure evaluation_rtn(R, A)
      /* A와 A_new 집합은 비교착상태의 노드들을 포함하게 됨. */
      A_new := A;
      do
        A := A_new;
        for each i→RC_i∈R,
            if eval(RC_i)=true then begin
                A_new := A_new∪{i};
```

```
                R := R - {i→RC_i};
            end if
        until A_new = A;
    end procedure
(V) procedure resolution_rtn(init, R, A)
        select a victim v among nodes in R
        according to the predefined criteria;
        send ABORT(init) to v;
        A := A∪{v};
        R := R - {v→RC_v};
        if R≠∅ then evaluation_rtn(R, A);
        if R≠∅ then resolution_rtn(init, R, A);
    end procedure
```

알고리즘의 실행은 그림 1에 예시되었다. 그림 1(i)의 WFG에서 RC_a=b∧c, RC_b=(d∧e)∨f, RC_c=e, RC_d =e∨f, RC_e=c∧f를 가정하였다. 그림 1(ii)는 알고리즘 실행 결과인 DST이며 tree 간선과 non-tree 간선은 각기 실선과 점선으로 표시하였다. 다음은 알고리즘의 실행 순서를 예로 든 것이다.

1) 노드 a는 알고리즘을 시작하여 PROBE(a,{a→b∧c},a,1/2)과 PROBE(a,∅,a,1/2)을 노드 b와 c에게 각기 전송한다.
2) 노드 b가 노드 a로부터 PROBE를 받으면, PROBE (a,{a→b∧c},b,1/6), PROBE(a,{b→(d∧e)∨f},b,1/6), 그리고 PROBE(a,∅,b,1/6)를 각기 노드 d, e와 f에게 전송한다.
3) 노드 c가 노드 a로부터 PROBE를 받으면, PROBE (a,{c→e},c,1/2)를 노드 e에게 전송한다.
4) 노드 d가 노드 b로부터 PROBE를 받으면, PROBE (a,{d→e∨f},d,1/12)와 PROBE(a,{a→b∧c},d,1/12)를 노드 e와 f에게 각기 보낸다.
5) 노드 e가 노드 c로부터 PROBE를 받으면, PROBE (a,{e→c∧f},e,1/4)와 PROBE(a,{c→e},e,1/4)를 노드 c와 f에게 각기 전송한다.
6) 노드 f가 노드 d로부터 PROBE를 받으면, REPORT ({a→b∧c},{f},1/12)를 a에게 전송한다.
7) 노드 f가 노드 b로부터 PROBE를 받으면, REPORT (∅,∅,1/6)를 노드 a에게 전송한다.
8) 노드 f가 노드 e로부터 PROBE를 받으면, REPORT ({c→e},∅,1/4)를 a에게 전송한다.
9) 노드 e가 노드 b로부터 PROBE를 받으면, REPORT ({b→(d∧e)∨f},∅,1/6)를 a에게 전송한다.
10) 노드 e가 노드 d로부터 PROBE를 받으면, REPORT({d→e∨f},∅,1/12)를 a에게 전송한다.
11) 노드 c가 노드 e로부터 PROBE를 받으면, REPORT({e→c∧f},∅,1/4)를 a에게 전송한다.

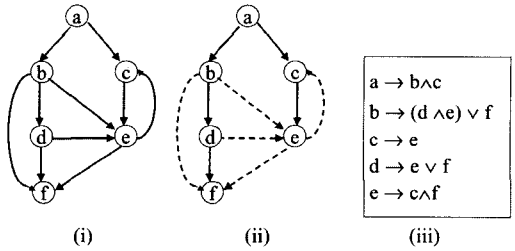

그림 1 알고리즘의 실행 예. (i) WFG (ii) DST (iii) 시작노드에서 수집된 요청조건 정보

알고리즘의 실행 예에서 시작노드가 수집한 중량의 합은 1이 됨을 알 수 있다. 시작노드는 그림 1(iii)과 같은 정보를 수집하며 또한 노드 f가 진행상태임을 알게 된다. 이들 정보로부터 노드 a, c, 및 e가 교착상태임을 판단하게 된다.

3.2 다수 알고리즘의 동시 실행 기법

본 절에서는 다수의 알고리즘이 동시 실행되는 경우, 이를 조정하는 기법을 제안한다. 이러한 기법의 필요성은 앞 절에서 언급하였듯이 교착상태의 오류 판단을 방지하기 위함이다. 기존의 알고리즘에서는 각 알고리즘의 프로세스에 우선순위를 부여함으로써 이를 해결하였다[10,11]. 즉, 우선순위가 낮은 알고리즘은 실행 중지되고 높은 우선순위의 알고리즘만이 계속 진행할 수 있게 하였다. 그림 2는 우선순위를 부여한 알고리즘의 동시적 실행 예이다. 노드 4의 요청조건은 2∧5라고 가정하였다. 그림에서 볼 수 있듯이 두 개의 교착상태를 포함하고 있다. 노드 번호가 클수록 우선순위가 높고 각 노드가 시작한 알고리즘이 동시적으로 진행하고 있음을 가정하자. 그러면 기존의 알고리즘[10,11]에 따르면 노드 1과 2의 알고리즘은 노드 4의 알고리즘으로 인하여 종료할 수 없다. 이는 노드 1과 2가 응답 메시지를 수신하지 못하기 때문이다(노드 4가 probe를 폐기함). 또한 노드 4의 알고리즘도 노드 5의 알고리즘 때문에 종료하지 못한다. 결과적으로, 노드 1, 2와 4로 구성된 교착상태는 발견되지 못한다. 이 문제는 폐기된 알고리즘의 실행이 재시작되면 해결될 수 있으나 새로운 알고리즘의 실행도 종료된다는 보장은 없다. 그러므로, 교착상태의 발견은 지연되거나 이루어지지 못한다.

동시진행하는 알고리즘들을 다루기 위해 우선순위를

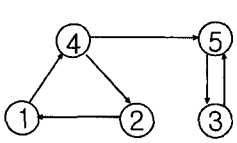

그림 2 동시 실행하는 알고리즘의 예

사용하는 대신, Chen의 알고리즘[14]은 UPDATE라고 불리는 메시지를 전송함으로써 시스템의 상태를 최신으로 유지하고자 하였다. 즉, 교착상태가 발견되면 희생노드가 자신이 중지될 것이라는 사실을 시스템의 다른 노드에게 UPDATE를 통하여 알림으로써 교착상태를 오류 판단하는 일을 방지한다. 그러나, 그럼에도 불구하고 UPDATE 메시지의 전송 지연으로 인하여 교착상태의 오류 판단은 발생할 수 있다.

위의 해결책과 대조적으로, 본 연구의 기법은 낮은 우선순위의 알고리즘을 폐기하는 대신에 계속 진행시킴으로써 보다 신속히 교착상태를 발견 가능토록 하였다. 즉, 낮은 우선순위의 알고리즘은 높은 우선순위의 알고리즘이 도달할 수 없는 WFG 부분을 탐색하여 그 부분에 존재할 지도 모르는 교착상태를 발견한다. 구체적으로는, 임의의 노드가 자신이 실행 중인 알고리즘 보다 낮은 우선순위의 알고리즘의 probe를 수신하게 되면 REPORT를 낮은 우선순위의 알고리즘의 시작노드에게 곧바로 전송한다. 이는 자신이 진행상태가 될 것임을 미리 알리는 역할을 한다. 왜냐하면 자신이 만약 교착상태에 속해 있다면 높은 우선순위의 알고리즘을 실행 중이므로 교착상태로부터 해방될 것이기 때문이다. 제안한 기법의 구체적 pseudo-code는 아래와 같다. 각 메시지는 3.1절에서 기술한 형식 외에 msg_pr이라는 매개변수를 갖는데 이는 우선순위를 나타낸다.

노드 i를 위한 추가 자료 구조

• cur_pr_i: 노드 i가 실행 중인 알고리즘의 우선순위.

노드 i가 PROBE(msg_pr, init, R, j, w)를 수신할 때:

(M.I.1) **if** msg_pr is lower than cur_pr_i **then**
send REPORT(msg_pr, R, {i}, w) to init;

(M.I.2) **else if** msg_pr is higher than cur_pr_i **then begin**
/* 수신한 PROBE를 위한 알고리즘을 새로이 실행함. */
cur_pr_i = msg_pr;
initialize the local data structures;
execute Step (II) in Section 3.1;
end else

(M.I.3) **else** execute Step (II.2) in Section 3.1;
/*PROBE를 전송한 간선은 non-tree임*/

시작노드 init이 REPORT(msg_pr, R, A, w)를 수신할 때:

(M.II.1) **if** msg_pr is lower than cur_pr_i **then**
discard the message; /* init은 그사이에 보다 높은 우선순위의 알고리즘을 실행함.*/

(M.II.2) **else if** msg_pr is higher than cur_pr_i **then**; /* 발생 불가함. */

(M.II.3) **else** execute Step (III) in Section 3.1;

노드 i가 ABORT(msg_pr)를 수신할 때:

if msg_pr=cur_pr_i **then** abort;

else discard the message;

3.3 알고리즘의 정확성

2절에서 제시한 알고리즘의 정확성 조건은 다음 정리에 증명되었다.

정리 1. 알고리즘 시작 시간에 k 노드들로 이루어진 교착상태 집합 D가 존재한다고 가정하자. 그렇다면 시작노드에서 희생자 노드를 선택하기 전의 시점에, D에 속한 임의의 x_i에 대해 eval(RC_x_i)=false 임이 성립한다.

증명. 정리의 반대를 가정하자. 즉, 희생자 노드를 선택하기 전에, D에 속한 노드 x_1이 존재하여 eval(RC_x_1)=true이다. 우선 n을 WFG 노드 수라 할 때 |D|=k<n임을 가정하자. 또한, RC_x_1은 WFG 내에서 교착상태에 속하지 않은 노드들로만 구성되었다고 하자. 그렇다면 교착상태의 정의에 의하여 $x_1 \notin D$이며 이는 $x_1 \in D$라는 가정에 위배된다. 그러므로, $x_2 \in D$인 노드가 반드시 존재하여 $x_2 \in domain(RC_x_1)$이고 x_2의 상태가 x_1의 상태를 결정한다. eval(RC_x_1)=true이기 때문에 eval(RC_x_2) 또한 true이다. x_2에 대하여도 x_1의 경우와 유사한 논리가 전개된다. 즉, $x_3 \in D$가 존재하여 $x_3 \in domain(RC_x_2)$이고 eval(RC_x_3)=true이다. 귀납적으로, $x_k+1 \in D$가 존재하여 $x_k+1 \in domain(RC_x_k)$이고 eval(RC_x_k+1)=true이다. |D|=k이기 때문에, 임의의 i=1,...,k-1에 대하여. $x_k+1=x_i$이다. 따라서, 모든 $x\_i \in D$, i=1,...k,에 대하여 eval(RC_x_i)=true이다. 그러나, evaluation_rtn의 시작시점에 진행상태의 노드만이 eval 값을 true로 갖기 때문에, eval 값을 true로 하는 노드는 존재할 수 없다.

이제 |D|=k=n임을 가정하자. 그렇다면 모든 노드는 교착상태이고 WFG 내에 진행상태인 노드는 없다. 따라서 evaluation_rtn에 의하여 $A_new=A=\emptyset$이다. 그러므로 모든 노드의 요청조건은 false로 평가된다. □

정리 2. 알고리즘은 교착상태를 오류 판단하지 않는다.

증명. 정리의 반대를 가정하자. 즉, 알고리즘이 종료되는 시점에, 교착상태가 존재하지 않음에도 불구하고, 시작노드는 교착상태에 속한 노드 집합 D를 발견하였다고 가정한다. 그렇다면 시스템 내에 실제로 존재하지 않는 간선(들)을 시작노드는 교착상태에 속한 것으로 판단하였음에 틀림없다. D에 속한 이러한 간선들 중에서, e=(a,b)를 시스템에서 최초로 사라진 간선이라고 하자. 만약 노드 a가 시작노드라면, 3.1 절에 의하여 a는 (I) 단계를 실행한다; 그렇지 않은 경우 (II.1) 단계를 실행

한다. 두 경우 모두, proc_probing 절차는 실행되며 노드 a는 a→RC_a를 계승자에게 전달한다. e가 DST에서 tree 간선이라고 가정하자. 노드 b가 a로부터 PROBE를 수신하면, b는 (II.1.2) 또는 (II.1.3) 단계를 실행한다. 만약 노드 b가 (II.1.3)을 실행하면, RC_b를 시작노드에게 전달하며 e가 최초로 사라졌기 때문에 RC_b는 true로 평가된다. 그러나, e∈D이기 때문에 b는 시작노드에서 교착상태이다. 만약 b가 (II.1.2) 단계를 실행하게 된다면, 단계 (III)에 의하여 b는 시작노드에서 A 집합에 속하게 되므로 교착상태가 아닌 것으로 판단된다. 이제 e가 DST에서 non-tree 간선이라고 가정하자. e가 교착상태의 집합에서 최초로 사라진 간선이기 때문에 b는 t 시간에 a에게 REPLY 메시지를 보냈음에 틀림없다. 만약 b가 t 시각 후에 a로부터 PROBE를 받았다면, e는 (II.2.1) 단계와 (III) 단계의 replace 연산 때문에 시작노드에 존재하지 않는다. 그러므로, b는 t 시각 전에 PROBE를 받았고 그 때 이미 b는 a에게 REPLY 메시지를 보냈기 때문에 b는 교착상태가 아니다. 더욱이 시작노드에서 RC_b는 false로 판단되지 않으며 e가 REPLY 메시지를 역전송한 최초의 간선이기 때문에 e∉D이다; 만약 b가 교착상태에 속한다면 노드 a가 REPLY를 받기 전에 b가 자신의 계승자로부터 REPLY를 수신하였을 것이다. □

4. 성능 분석

4.1 Complexities

본 절에서 우선 알고리즘 실행 작업이 하나일 경우의 complexity(복잡도)를 고려한다. 임의의 두 노드간에 메시지 송신은 한 단위시간 내에 이루어진다고 가정하자. 알고리즘 수행에 참여한 WFG 간선 수는 e로 표기하고 노드 수는 n이라고 표기하자. 또한 d는 DST의 폭을 의미한다. 제안한 알고리즘은 각 간선당 하나의 PROBE를 전송한다. 응답 메시지는 진행상태 노드 또는 non-tree 간선 당 하나씩 시작노드에게 직접 전달된다. 그러므로, 필요한 전체 메시지 수는 2e를 넘지 않는다. 또한 응답 메시지는 시작노드에게로 직접 전달되기 때문에 d+2 시간을 소모한다.

교착상태의 해결 문제에 대하여 대부분의 기존 알고리즘 [10,11,13]에서는 특별한 방안을 제시하지 않았다. Chen의 알고리즘[14]은 회생노드를 중지시키는데 기인하는 새로운 프로세스간의 대기관계를 반영하기 위하여 회생노드마다 3n 메시지를 소모한다. 이에 비하여, 본 연구의 알고리즘은 교착상태의 해결을 위한 추가적인 메시지가 불필요하며 중지 명령(ABORT) 하나만이 필요할 뿐이다.

본 연구에서 제안한 알고리즘의 메시지 길이를 측정하기 위하여 다음과 같은 probing tree를 정의한다:

(i) 트리의 근노드는 알고리즘의 시작노드이다.

(ii) 임의의 노드 p가 노드 q에게서 PROBE를 수신하면, p는 트리상에서 q의 자노드가 된다.

(iii) 임의의 노드가 자신의 부노드가 보낸 PROBE에 대해 REPORT를 시작노드에게 전송하면, 그 노드는 leaf가 된다. 그렇지 않으면 non-leaf가 된다.

위의 정의에 의하여 WFG 간선 수가 probing tree의 간선 수와 같음을 알 수 있다. 그림 3은 probing tree의 예이다. 부노드-자노드 관계는 실선으로 표시하였다. n이 총 노드 수라 할 때 실선 수는 n-1임을 알 수 있다. 점선 (p, q)는 q가 p 외의 다른 노드에게서 PROBE를 수신하였으며 p에게서 PROBE를 받았을 때 시작노드에게 REPORT를 보냈음을 의미한다. Leaf 노드로 향하는 실선은 그 leaf가 진행상태임을 의미한다. 또한 probing tree의 폭은 일반적으로 DST의 폭+1임을 알 수 있다. PROBE 메시지의 길이를 알기 위하여, 메시지가 전달하는 R 집합에 초점을 둔다. 이는 다음 정리에 증명하였다.

정리 3. L_d는 probing tree의 d level에 있는 노드에게 송신된 PROBE가 전달한 R 집합의 노드 수라고 하자. 또한, d level의 노드의 계승자 수를 s_d라고 하자.

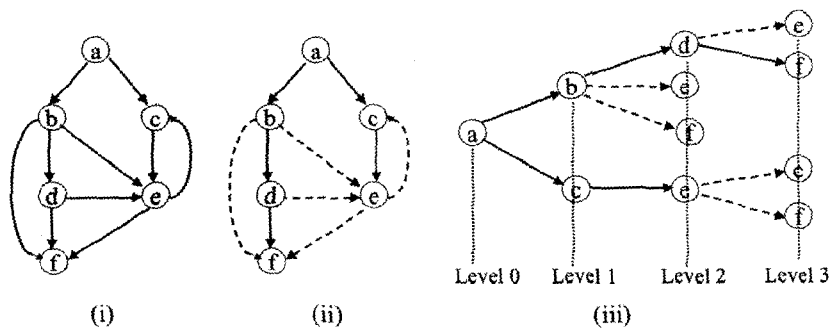

그림 3 Probing tree의 예. (i) WFG (ii) DST (iii) probing tree

$$L\_d = \frac{1}{\Pi_{i=0}^{d-1} s_i}(1+2\sum_{j=0}^{d-2}(\Pi_{i=0}^{j} s_i)) + 1, \quad d>0$$

증명. Probing tree의 근노드 r은 r→RC_r을 자신의 계승자에게 전송한다. 그러므로, |RC_r|를 RC_r의 노드 수라 할 때, r의 계승자는 평균적으로 (|RC_r| + 1)/s_0의 노드 수를 수신한다. |RC_r|=s_0이므로 L_1 = 1/s_0 + 1이다. 이제 L_2를 계산하자. n_d를 probing tree의 d level에 있는 임의의 노드라고 하자. |RC_n_1|를 n_1의 요청조건에 포함된 노드 수라고 할 때 L_1의 경우와 마찬가지로, n_2는 (|RC_n_1| + 1)/s_1의 노드수를 수신한다. 그러나, n_1은 또한 자신의 부노드로부터 전달받은 R 집합, 즉, L_1을 자신의 계승자들에게 전달한다. n_1의 각 계승자는 동일한 크기의 L_1의 일부분을 수신한다. 그러므로, L_2 = (L_1+1)/s_1 + 1 = (1+2s_0)/s_0s_1 + 1. 그러므로 귀납적으로 L_d는 정리에 기술한 것과 같다. □

두 극단의 경우에 L_d를 계산하자. 우선, 모든 d에 대해서, s_d=1인 경우에 L_d=2d이다. 그러므로, PROBE의 최대 길이는 2d+2이다. 또다른 경우, s_d=n-1을 고려하면 정리 3에 의하여 $L_d < \frac{1}{(n-1)^d} + \frac{n}{n-2}$이다. 그러므로, n이 증가함에 따라 L_d는 기하급수적으로 감소하며 1에 접근한다. REPORT 메시지의 길이는 R과 A 집합 크기에 좌우된다. 그러나, |A|≤1이므로 메시지 길이에 영향을 주지 않는다. Level d에 있는 노드가 REPORT를 통해 전달하는 R 집합의 크기는 L_d이다. 알고리즘들의 성능 비교는 표 1에 제시하였다.

단 한번의 실행을 가정하였을 때 알고리즘의 성능은 크게 차이가 없음을 알 수 있다. 그러나 분산 알고리즘의 특성에 따라, 알고리즘이 동시 실행되는 환경 하에서 전체적 성능을 평가하는 것이 옳다. 이는 다음 절에서 시뮬레이션을 통하여 평가한다.

4.2 시뮬레이션

시뮬레이션 모델은 다음과 같다. 프로세스가 시스템에 들어오면, Tpre 시간 후 리소스를 요청한다. 프로세스는 PI의 확률로 국부적인 또는 다른 사이트의 리소스를 요청한다. 프로세스가 요청하는 조건은 (r_11∧ ... ∧r_1a) ∨(r_21∧ ... ∧r_2b)이다. 이 때 r_ij는 리소스이며 a와 b는 1부터 7 중 랜덤하게 선택된다. 프로세스는 자신의 작업을 위해 또는 교착상태 발견을 위해 임의의 다른 프로세스에게 Tm 시간 내에 메시지를 보낼 수 있다.

만약 프로세스의 요청조건이 만족된다면, 허락된 각 리소스 당 Texec 시간을 소모한다. 실행이 완료되면 모든 획득한 리소스들을 해제한다. 한 프로세스가 종료되면 같은 site에 새로운 프로세스가 시작되므로, 시스템 내에는 항상 같은 수의 프로세스가 존재한다. 임의의 사이트는 교착상태 발견을 위한 메시지를 수신하거나 알고리즘을 시작함으로써, 알고리즘을 실행한다. 대기상태의 프로세스가 알고리즘을 시작하는 조건은, 교착상태 발견을 위한 메시지를 전달하지 않은 채 타임아웃 동안 대기 상태에 있는 요청이 있을 경우이다. 표 2는 시뮬레이션에서 사용한 환경 변수들이다. 수신한 메시지 종류나 알고리즘 종류에 관계 없이 메시지를 처리하기 위한 시간은 Tdlmsg이다. Tdlmsg 외의 다른 변수들은 지수분포를 가정한다.

제안한 알고리즘, Bracha와 Toueg[10], 그리고 Kshemkalyani[11]의 알고리즘의 성능을 비교 실험하였다. Chen의 알고리즘[14]은 비교 대상에서 제외하였는데, 교착상태를 오류 판단할 가능성이 있기 때문이다[14]. 기술의 편이함을 위하여 Bracha와 Toueg의 알고리즘[10]은 'BS'로, Kshemkalyani의 알고리즘[11]은 'KS'로 부르기로 한다. 또한 3.1절에서 제안한 알고리즘은 'Proposed', 그리고 3.2절에서 제안한 알고리즘은 'Extended'라고 부르기로 한다. Extended의 시뮬레이션 결과는 3.2절에서 언급한 대로 우선순위에 따라 'High-Pr'과 'LowPr'의 두 측면에서 제시한다. 즉, LowPr의 결과는 실행 도중 적어도 한번은 보다 높은 우선순위의 알고리즘을 만난 적이 있던 알고리즘의 결과이다.

교착상태의 해결을 위하여, KS는 해결 기법을 제시하진 않았으나, KS, Proposed 및 Extended는 공정한 성능 비교를 위해 3.1절의 해결 기법을 동시에 적용받도록 하였다. BR의 경우는, 교착상태에 속한 시작노드가 희생노드가 되는데, 이는 BR은 해결 방안을 제시하지 않았고 또한 사용되는 메시지는 노드 상태에 관한 정보를 전달하지 않기 때문이다.

그림 4는 다양한 프로세스와 리소스 수에 대하여 알

표 1 알고리즘의 성능 비교(e는 WFG의 간선수. d는 WFG의 폭. n은 WFG 노드 수)

	Bracha & Toueg [10]	Brzezinski [13]	Chen et.al [14]	Kshemkalyani [11]	제안된 알고리즘
메시지 수	4e	n*n/2	2n	2e	<2e
소요시간	4d	4n	2d	2d+2	d+2
메시지 크기	O(1)	O(n)	O(n)	O(e)	O(d)
교착상태 해결방안	제시 안함	제시 안함	3n 메시지[1)]	제시 안함	1 메시지[1)]

1) 희생노드 하나에 대하여

고리즘 실행 시간을 제시한다. 표 3에서 알 수 있듯이 시스템 부하가 클수록 교착상태 비율은 증가함을 알 수 있다. Proposed와 Extended는 유사한 성능을 보이나 LowPr은 HighPr 보다 더 많은 시간을 소모함을 볼 수 있다. 이는 HighPr이 LowPr보다 작은 크기의 DST를 산출함을 의미한다. 그 이유는 WFG의 크기가 작을수록 우선순위가 높은 알고리즘의 실행을 만날 확률이 더 적기 때문이다. 그림 4에서 보여준 실행시간의 차이는 표 1과 일관성이 없는 것으로 나타난다. 즉, BR이나 KS와 Extended의 차이는 표 1의 그것처럼 크지 않다. 이는 교착상태 비율에서 알 수 있듯이 대부분의 프로세스들이 진행상태이므로 WFG의 폭이 기껏해야 1이나 2가 되기 때문이다. 그러나, 교착상태 비율이 크면 실행시간의 차이도 커지는 것으로 나타난다.

표 4는 HighPr과 LowPr의 교착상태 발견 비율(DDR)을 보여 준다. 또한 각 알고리즘의 실행 결과에 따른 교착상태 지속 시간을 Extended 실행 결과에 따른 교착상태 지속 시간에 대비한 비율로 나타낸다. LowPr은 HighPr 보다 적은 수의 교착상태를 발견함을 알 수 있다. 그 이유는 다음과 같다. 우선순위가 낮은 알고리즘이 실행 도중 교착상태에 속한 노드를 만난다고 할지라도, 그의 시작노드는 높은 우선순위의 알고리즘을 만났으므로 전체 교착상태 정보를 보고받지 못할 가능성이 크다. 그러나, LowPr의 DDR은 작지 않은 수이며 따라서 표에서 볼 수 있듯이 교착상태의 신속한

표 2 시뮬레이션 시스템의 환경 변수들

Tpre	Texec	Pl	Tm	Tdlmsg
100	30	0.1	20	1.5

표 4 교착상태 지속 시간

프로세스 수	리소스 수	DDR(Extended)		교착상태 지속 시간의 비율		
		HighPr	LowPr	BR	KS	Proposed
25	50	.90	.10	6.34	3.40	1.20
25	150	.75	.25	5.10	2.96	1.88
25	250	.88	.13	2.99	2.63	2.76
25	350	.89	.11	6.79	2.46	1.61
50	100	.78	.22	3.74	2.62	1.39
50	150	.83	.17	6.26	1.72	1.39
50	250	.82	.18	4.33	1.22	1.16
50	350	.63	.38	3.33	3.06	1.93
100	250	.50	.50	10.07	1.95	1.22
100	300	.80	.20	5.70	2.17	2.49

표 3 교착상태의 비율

	리소스수 (프로세스수=25)				리소스수 (프로세스수=25)			
	50	150	250	350	100	150	250	350
비율	0.314	0.071	0.041	0.008	0.090	0.099	0.042	0.014

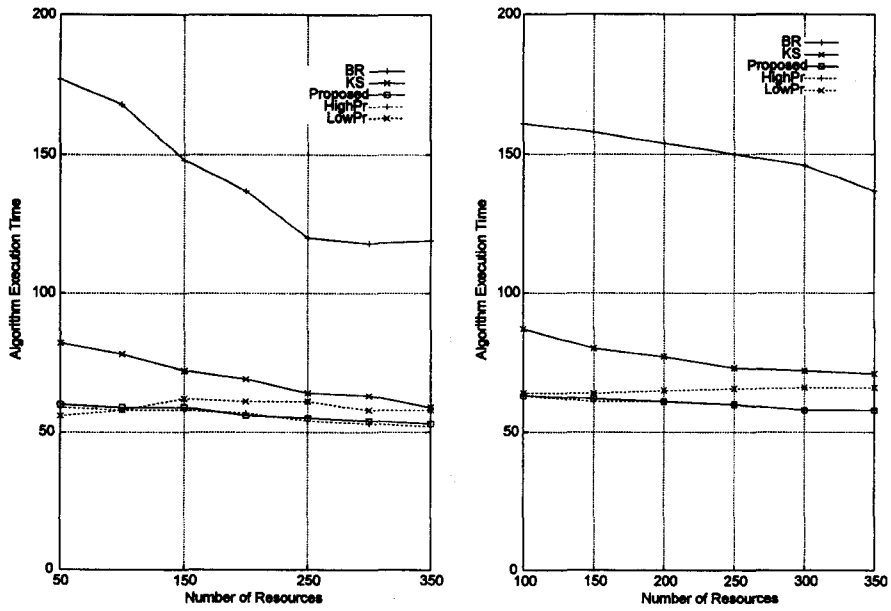

그림 4 알고리즘 실행 시간: 프로세스 수=25 (좌), 프로세스 수=50 (우)

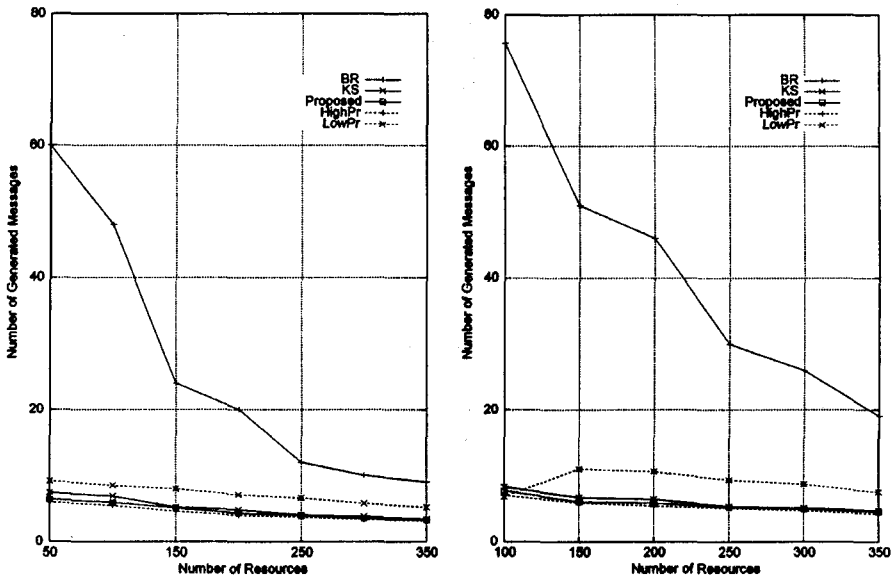

그림 5 교착상태 발견을 위한 메시지 수: 프로세스 수=25 (좌), 프로세스 수=50 (우)

발견에 공헌한다.

표 4에서 BR은 다른 알고리즘보다 훨씬 낮은 성능을 보인다. 이는 BR에서는 시작노드가 교착상태의 회생노드이기 때문에 교착상태를 해결하지 못할 가능성이 있기 때문이다. 이에 반해 다른 알고리즘들은 한번의 실행으로 시작노드에서 도달가능한 모든 교착상태를 해결한다.

교착상태 발견을 위해 사용하는 메시지 수는 그림 5에 제시되었다. 각 알고리즘은 시스템 부하가 클 때 보다 많은 수의 메시지를 발생시킴을 알 수 있다. 예상했던 바와 같이, BR은 타 알고리즘에 비해 훨씬 많은 메시지를 발생시킨다. 앞서 언급한 대로, 이는 BR 알고리즘의 특성 때문이다. 즉, 표 3에서 제시한 바와 같이 교착상태 발생 비율이 낮기 때문에, BR에서 전송하는 대부분의 메시지들은 잠재적으로 진행상태인 노드에 의해 발생되고 그에 따른 응답 메시지가 대부분을 차지하기 때문이다.

KS와 Proposed는 발생되는 메시지 수 관점에서 유사한 성능을 보임을 알 수 있는데, 이는 표 1에서 제시한 바와 일치한다. 그러나, 낮은 우선순위의 실행으로 인하여 Extended는 Proposed 보다 다소 많은 메시지를 발생시킨다. 앞에서 언급한 대로, 그 이유는 LowPr이 HighPr 보다 큰 DST를 산출하기 때문이다.

표 5는 각 알고리즘이 발생시키는 메시지의 길이를 노드 수로 측정한 결과이다. 시스템의 부하가 클수록 메시지 길이는 길어지는 것으로 나타난다. 이는 각 노드의 계승자가 더 많아지고 따라서 요청조건이 더 많은 노드를 포함하게 되기 때문이다. 단, Extended는 예외이다. 그 이유는 LowPr의 메시지 길이는 시스템 부하에 의존하지 않기 때문이다. 즉, 메시지의 최대 길이는 보다 높은 우선순위의 알고리즘을 만나는 순간 결정된다. 만약 그 시점이 빠르면 메시지 길이는 짧을 것이고, 그렇지 않으면 그 길이는 보다 길어질 것이다. 실제로 그 시점은 시스템의 부하와는 상관없는 듯이 보인다. 일반적으로, 세 알고리즘은 메시지 길이의 측면에서 유사한 성능을 보인다. 그러나, KS의 결과는 다른 두 알고리즘보다 다소 길게 나타난다.

표 5 교착상태 발견을 위한 메시지 길이

프로세스 수	리소스 수	KS	Proposed	Extended		
				Mean	HighPr	LowPr
25	50	1.56	1.49	1.36	1.43	1.29
25	150	1.46	1.39	1.39	1.29	1.48
25	250	1.33	1.18	0.99	1.14	0.84
25	350	1.25	1.05	1.18	1.02	1.33
50	100	1.68	1.55	1.20	1.48	0.93
50	150	1.65	1.48	1.49	1.45	1.54
50	250	1.53	1.46	1.54	1.44	1.64
50	350	1.47	1.38	1.05	1.29	0.81
100	250	1.71	1.56	1.17	1.51	0.83
100	300	1.64	1.54	1.47	1.48	1.45

5. 결 론

본 연구에서는 일반적 교착상태의 신속한 해결 방안

을 제시하였다. 교착상태에 필요한 정보를 수합하는데 있어서, 응답 단계에서 수합하는 것이 아니라 probe 전송 단계에서 수합함으로써 diffusing computation 기법에 근거한 방식들에 비해 교착상태 발견에 필요한 시간을 향상시켰다. 결과적으로, 현재까지 알려진 가장 우수한 성능의 알고리즘의 실행 시간을 거의 반으로 단축시키는 효과를 가져왔다. 또한, 본 연구에서는 단 한번의 알고리즘 실행 뿐만 아니라 동시 실행하는 다수의 알고리즘을 위한 방안을 제시하여 교착상태의 지속 시간을 더욱 단축시킬 수 있도록 하였다.

메시지 수, 알고리즘 실행 시간 등의 측면에서, 제안한 알고리즘의 성능은 타알고리즘 보다 뛰어난 것으로 분석되었다. 동시 실행되는 다수의 알고리즘에 대한 성능은 시뮬레이션 실험을 통하여 평가하였다. 일반적으로, 다수 실행되는 환경에서는 한번 실행되는 환경에서 볼 수 있는 성능 차이를 나타내지 않는 것으로 드러났다. 보다 구체적으로, 시뮬레이션 결과를 통하여 다음과 같은 사실들이 확인되었다: (1) 제안한 알고리즘을 한번 실행하는데 소요되는 시간은 평균적으로 기존 알고리즘보다 적은 것으로 드러났으나, 그 차이는 복잡도의 비교 평가에서 분석한 것보다는 적었다. 이는 대부분의 프로세스들이 진행상태이기 때문이다. (2) 다수의 알고리즘 실행을 위한 제안 기법은 기존보다 훨씬 신속하게 교착상태를 해결하는 것으로 나타났다. 시스템의 부하가 크면, 그 신속성은 기존에 비해 거의 세 배에 달한다. (3) 제안한 알고리즘은 발생되는 메시지 수와 메시지 길이의 측면에서 기존의 알고리즘을 다소 능가하는 것으로 확인되었다.

참 고 문 헌

[1] M. Singhal, "Deadlock detection in distributed systems," IEEE Computer, Vol. 22, pp. 37-48, 1989.

[2] S. Lee and J. L. Kim, "Performance Analysis of Distributed Deadlock Detection Algorithms," IEEE Transactions on Knowledge and Data Engineering, Vol. 13, No. 4, pp. 623-636, 2001.

[3] K. Makki and N. Pissinou, "Efficient Detection and Resolution of Deadlocks in Distributed Database Systems," Journal of Computer Communications, Vol. 22, No. 7, pp. 637-643, 1999.

[4] J. Mayo and P. Kearns, "Distributed deadlock detection and resolution based on hardware clocks," Int'l Conf. on Distributed Computing Systems, pp. 208-215, 1999.

[5] J. R. Mendivil, F. Farina, J. Garitagoitia, C. F. Alastruey, and J.M. Bernabeu-Auban, "A distributed deadlock resolution algorithm for the AND model," IEEE Trans. Parallel and Distributed Systems, Vol. 10, No. 5, pp. 433-447, 1999.

[6] H. Wu, W-N. Chin, and J. Jaffar, "An efficient distributed deadlock avoidance algorithm for the AND model," IEEE transactions on software engineering, Vol. 28, No. 1, pp. 18-29, 2002.

[7] A. Boukerche and C. Tropper, "A distributed graph algorithm for the detection of local cycles and knots," IEEE Trans. Parallel and Distributed Systems, Vol. 9, No. 8, pp. 748-757, 1998.

[8] D. Manivannan and M. Singhal, "A Distributed Algorithm for Knot Detection in a Distributed Graph," Proc. Int'l Conf. Parallel Processing, 2002.

[9] J. Villadangos, F. Farina, J. R. Mendivil, J. R. Garitagoitia, and A. Cordoba, "A safe algorithm for resolving OR deadlocks," IEEE Trans. Softw. Engr., Vol. 29, No. 7, pp. 608-622, 2003.

[10] G. Bracha and S. Toueg, "A distributed algorithm for generalized deadlock detection," Distributed Computing, Vol. 2, pp. 127-138, 1987.

[11] A. D. Kshemkalyani and M. Singhal, "A one-phase algorithm to detect distributed deadlocks in replicated databases," IEEE Transactions on Knowledge and Data Engineering, Vol. 11, No. 6, pp. 880-895, 1999.

[12] J. Wang and S. Huang and N. Chen, "A distributed algorithm for detecting generalized deadlocks," Tech. Rep., Dept. of Computer Science, National Tsing-Hua Univ., 1990.

[13] J. Brzezinski and J. -M. Helary and M. Raynal and M. Singhal, "Deadlock models and a general algorithm for distributed deadlock detection," Journal of Parallel and Distributed Computing, Vol. 31, No. 2, pp. 112-125, 1995.

[14] S. Chen and Y. Deng and P.C. Attie, "Deadlock detection and resolution in distributed systems based on locally constructed wait-for graphs," Tech. Report, School of Computer Science, Florida International University, Aug., 1995.

[15] E. W. Dijkstra and C. S. Scholten, "Termination detection for diffusing computations," Information Processing Letters, Vol. 11, No. 1, pp. 1-4, 1980.

이 수 정

1985년 이화여자대학교 수학과 졸업. 1985년~1987년 삼성생명 정보시스템실 근무 1988년~1994년 미국 Texas A&M 대학교 컴퓨터과학과 졸업 (M.S, Ph.D). 1994년~1998년 삼성전자 통신개발연구소 근무. 1998년~현재 인천교육대학교 컴퓨터교육과 조교수. 관심분야는 분산시스템, 교착상태 알고리즘, 라우팅 알고리즘 등