통신 유형 변형을 이용하여 검사점 생성 개수를 개선한 검사점 Z-Cycle 검출 기법

(New Z-Cycle Detection Algorithm Using Communication Pattern Transformation for the Minimum Number of Forced Checkpoints)

우 남 윤 † 염 헌 영 †† 박 태 순 ††
(Namyoon Woo) (Heon Young Yeom) (Taesoon Park)

요 약 통신 유도 검사점 기법(communication induced checkpointing)은 분산 프로세스들의 결함 내성을 위한 검사점 기법 중 한 가지이다. 각 프로세스가 동기화를 거치지 않고 독립적으로 생성한 지역 검사점은 일관성을 위배하는 불필요한 검사점(useless checkpoint)이 될 가능성이 있으며, 연속적인 프로세스의 롤백(rollback)을 유발시킨다. 이를 막기 위해서 통신 유도 검사점 기법은 추가로 강제적인 검사점(forced checkpoint)을 생성한다. 강제적 검사점의 개수는 전체 시스템 성능의 부하와 직결되므로 이를 줄이는 것이 중요하다. 이 논문에서는 "Z-cycle 부재" 조건을 만족하는 두 가지의 통신 기반 검사점 기법을 제안하며, 시뮬레이션 결과를 통해서 제안된 알고리즘들이 기존의 알고리즘들보다 적은 부하를 요구함을 보인다. 덧붙여, 인덱스를 사용한 기존의 통신 유도 검사점 기법은 일관적인 전역 회복점(consistent global cut)을 찾는데 비효율적임을 보인다.

키워드 : 결함 내성, 통신 유도 검사점 프로토콜, Z-cycle 검출

Abstract Communication induced checkpointing (CIC) is one of the checkpointing techniques to provide fault tolerance for distributed systems. Independent checkpoints that each distributed process produces without coordination are likely to be useless. Useless checkpoints, which cannot belong to any consistent global checkpoint sets, induce nondeterminant rollback. To prevent the useless checkpoints, CIC forces processes to take additional checkpoints at proper moment. The number of those forced checkpoints is the main source of failure-free overhead in CIC. In this paper, we present two new CIC protocols which satisfy 'No Z-Cycle (NZC)' property. The proposed protocols reduce the number of forced checkpoints compared to the existing protocols with the drawback of the increase in message delay. Our simulation results with the synthetic data show that the proposed protocols have lower failure-free overhead than the existing protocols. Additionally, we show that the classical 'index-based checkpointing' protocols are inefficient in constructing the consistent global cut in distributed executions.

Key words : fault tolerance, communication induced checkpointing, Z-cycle detection

1. 서 론

'검사점/롤백 기법'(Checkpointing and rollback-recovery)은 분산 프로세스의 결함내성을 위해서 쓰이는 대표적인 기법이다. 검사점 기법이란 실행 중인 프로세스의 상태를 디스크에 저장하는 작업을 뜻하며, 프로세스가 돌발적인 결함으로 인해 비정상으로 종료하더라도 검사점을 사용하여 프로세스 상태를 복구시킬 수 있다. 단일 프로세스의 복구와는 달리, 분산 프로세스들의 복구를 위해서는 프로세스들 상태 사이의 일관성도 만족시켜주어야 한다. 분산 프로세스들은 메시지를 전달하며 실행하기 때문에 각 프로세스의 상태는 다른 프로세스의 상태에 인과적 의존 관계(causal dependency)를 가지기 때문이다. 각 프로세스의 임의의 검사점들이 서로 의존 관계가 없을 경우, 이 검사점들의 집합을 **일관적인 전역 검사점**(consistent global checkpoint)이라고

† 비 회 원 : 서울대학교 컴퓨터공학부
nywoo@dcslab.snu.ac.kr
†† 종신회원 : 서울대학교 컴퓨터공학부 교수
yeom@snu.ac.kr
††† 종신회원 : 세종대학교 컴퓨터공학부 교수
tspark@kunja.sejong.ac.kr

논문접수 : 2003년 10월 14일
심사완료 : 2004년 9월 6일

부른다[1]. 만일 한 프로세스가 결함으로 인해 비정상 종료가 되어 복구할 경우, 다른 정상적으로 수행되던 프로세스들도 역시 일관적인 전역 검사점으로 롤백(rollback)해야 한다.

일관성이 지켜지는 복구 기법에 관한 연구는 지난 20년 간 이루어져 왔다. 기법들은 크게 '동기식 검사점 기법(coordinated checkpointing)'[2-4], '메시지 로깅 기법(message logging)'[5-7], 그리고 '통신 유도 검사점 기법(communication induced checkpointing)'[8-12]으로 분류될 수 있다. 이 중에서 통신 유도 검사점 기법은 각 프로세스들이 독립적으로 검사점을 생성하도록 하여, 확장성을 잘 보장해줄 기법으로 믿어져 왔다. 이 기법은 통신 유형을 파악하여 특정 유형이 파악되면 추가적으로 검사점을 생성시켜서, 모든 검사점이 일관성을 지킬 수 있도록 보장하는 기법이다. 통신 유도 검사점 기법은, *Lamport*의 happen-before 관계[13]를 일반화한 개념인 *Netxer*와 *Xu*의 Z-path/Z-cycle 이론에 근거하고 있다[14]. 검사점들과 메시지를 노드와 화살표로 표현하였을 경우에 한 검사점에서 시작하여 그 검사점으로 끝나는 Z-path가 존재하면, 그 Z-path를 Z-cycle이라 부른다. 만일, 한 검사점이 Z-cycle에 포함되어 있다면 이 검사점은 절대로 일관적인 전역 검사점에 속할 수가 없다는 것이 Z-cycle 이론이다. Z-cycle에 포함되어 있는 검사점은 **불필요한 검사점**이라 불리며, 무한정으로 연쇄 롤백을 일으키는 **도미노 현상**(domino effect)의 요인이 되기 때문에 치명적이다[7].

통신 유도 검사점 기법은 이러한 불필요한 검사점이 존재하지 않도록 하는 것을 목표로 하며, Z-cycle가 발견되는 지점에 추가적으로 검사점을 강제 생성하여 불필요한 검사점 생성을 막는다. 통신 유도 검사점 기법은 각 프로세스들의 독립적인 검사점 생성을 허용하기 때문에 다른 검사점 기법에 비해 확장성이 보장되리라 믿어져 왔지만, 최근 연구에 의하면 통신 유도 검사점 기법은 너무 많은 강제 검사점을 생성하여 성능 부하가 증가하게 된다[8]. 이는 실행시간 중에 Z-cycle을 정확하게 검출하는 것은 어렵기 때문에, Z-cycle이 아닌 유형에 대해서도 강제 검사점을 생성시키기 때문이다. 이 부하를 줄이기 위해서는 Z-cycle을 보다 정확하게 검출하여, 되도록 적은 수의 강제 검사점이 생성되게 하는 것이 중요하다.

대표적인 통신 유도 검사점 기법으로는 인덱스(index) 기반 방식[9,10]과 벡터 클럭(vector clock) 기반 방식[11]이 있다. 인덱스 기반 방식에서는 같은 인덱스를 가진 검사점들이 서로 일관성을 가지도록 보장하는 방식이다. 알고리즘이 간단하며 다른 기법에 비해 적은 수의 강제 검사점을 생성하는 반면에, 복구해야 하는 지점을 찾는 것이 비효율적이다. 벡터 클럭 기반 방식은 Z-cycle이 생길 가능성이 큰 통신 유형을 파악하여 미리 차단하는 방식이다. 본 연구의 경험에 의하면, 이 방법은 지나치게 많은 수의 강제 검사점을 생성시킨다.

본 연구는 보다 확장성을 보장해주며 성능 부하가 적은 두 가지 통신 유도 검사점 기법을 제안한다. 제안하는 알고리즘들은 통신 유형의 변형을 요구하며, 기존의 통신 유도 검사점 기법보다 정확하게 Z-cycle 통신 유형을 검출하여, 더 적은 수의 검사점을 생성한다.

본 논문은 다음과 같이 구성되어 있다. 2절에서는 본 연구가 가정하고 있는 시스템 모델에 대해서 설명한다. 3절에서는 제안하는 알고리즘의 설명과 올바름의 증명을 보인다. 끝으로 시뮬레이션 실험 결과 분석과 결론을 각각 4절과 5절에서 서술한다.

2. 배경

2.1 모델

본 연구가 가정한 모델에는 n개의 분산 프로세스들 $\{p_1, p_2, \ldots, p_n\}$이 있으며, 각 프로세스들은 서로 메시지를 주고받는다. 어떤 메시지 m에 대해서 메시지의 보냄과 받음에 대한 이벤트들을 각각 $send(m)$과 $deliver(m)$으로 표기한다. 프로세스 p_i의 x 번째 검사점 이벤트는 $C_{i,x}$로 표시하며, 연이은 두 검사점 $C_{i,x}$와 $C_{i,x+1}$ 사이의 구간은 $I_{i,x}$라고 한다. 각 이벤트 사이의 선행 관계는 $\xrightarrow{e}$나 $\xrightarrow{m}$로 표현된다. $\xrightarrow{e}$는 같은 프로세스에서 일어난 이벤트들 사이의 선행관계를 의미하며, $\xrightarrow{m}$는 메시지 m에 의한 이벤트 사이의 선행관계를 의미한다. 예를 들면, $x<y$일 경우, 프로세스 p_i에서 일어난 두 이벤트 $e_{i,x}$와 $e_{i,y}$는 $e_{i,x} \xrightarrow{e} e_{i,y}$를 만족한다. 메시지 m_1에 대해서 $send(m_1) \xrightarrow{m} deliver(m_1)$로 표현할 수 있다.

검사점 $C_{i,x}$에서 검사점 $C_{j,y}$로 가는 Z-path, ζ는 다음 사항을 만족할 경우, 성립된다.

1. $i=j \wedge C_{i,x} \xrightarrow{e} C_{j,y}$ 이거나,
2. 다음을 모두 만족하는 일련의 메시지 나열 $\{m_1, m_2, \ldots, m_l\}$이 존재한다.
   (a) $\forall k: 1 \le k \le l-1 \Rightarrow \exists t, (deliver(m_k) \in I_{t,p}) \wedge (send(m_{k+1}) \in I_{t,q}) \wedge (p \le q)$
   (b) $(C_{i,x} \xrightarrow{e} send(m_1)) \wedge (deliver(m_l) \xrightarrow{e} C_{j,y})$

이때, Z-path ζ의 첫 번째와 마지막 메시지를 $\zeta.first$와 $\zeta.last$라고 표기한다. 검사점 $C_{i,x}$와 두 개의 Z-path

ζ, μ로 구성이 되는 통신 유형이 아래 사항을 모두 만족하면, Z-cycle $ZC\ (C_{i,x}, \mu, \zeta, I_{j,y})$이라 한다(그림 1).

1. $C_{i,x} \xrightarrow{e} send(\mu.first)$
2. $(send(\mu.first) \in I_{i,y}) \wedge (deliver(\mu.last) \in I_{i,y})$ $\wedge\ (send(\zeta.first) \xrightarrow{e} deliver(u.last))$
3. $deliver(\zeta.last) \xrightarrow{e} C_{i,x}$

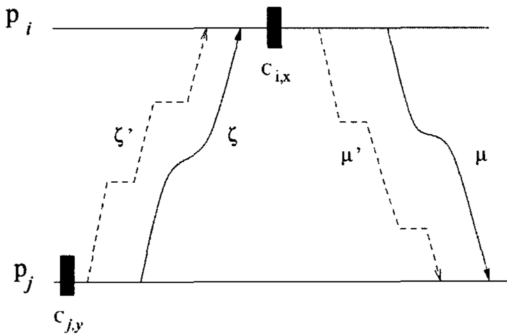


그림 1 Z-cycle의 예

2.2 배경 연구

분산 프로세스들의 통신 유형이 사슬 구조(chain), 중앙 집중(centralized) 혹은 계층적 (hierarchical) 구조로 - 이후부터는 이 세 가지 통신 유형을 통틀어 tree 통신 유형(T-유형)이라 일컫는다 - 이루어진다면, 최소 Z-cycle의 길이[1]는 2를 넘을 수가 없다는 사실이, *Park* 등에 의해서 증명된 바 있다[4]. 이를 요약하면, 다음 두 Theorem으로 정리된다.[2]

Theorem 2.1 : T-유형에서 최소 Z-cycle의 길이는 2를 넘을 수 없다.

Theorem 2.2 : T-유형에서, 길이가 2인 Z-cycle이 없음은 어떠한 Z-cycle도 존재하지 않음(Z-Cycle 부재 속성)과 필요충분조건이다.

길이가 2인 Z-cycle은 실행시간 중에 쉽게 검출될 수 있으며 프로토콜이 간단하다는 장점이 있다. Park은 T-유형에서 길이 2인 Z-cycle을 검출하여, 강제 검사점을 생성시키는 프로토콜을 제안하였으며, 이 프로토콜은 기존의 통신 유도 프로토콜보다 더 적은 수의 검사점을 생성시킴을 보였다. 그러나 임의의 통신 유형에서 최소 Z-cycle의 길이는 2 이상일 수 있으며, Park의 프로토콜을 적용하여도 Z-cycle이 존재한다.

2.3 Length Two Z-Cycle 검출 기법

길이가 2인 Z-cycle을 검출하는 방법은 다음과 같다. 임의의 프로세스, p_k는 아래의 변수들을 관리한다.

- $VC_k[]$: n크기의 정수 배열. $VC_k[k]$는 검사점이 생성될 때마다 1씩 증가된다. 메시지 m을 보낼 때 $VC_k[k]$를 덧붙여 보낸다. 이를 $m.VC$라고 한다. 만일 p_i가 보낸 m'을 받았다면, $VC_k[i]$는 $m'.VC$ 값으로 변한다.
- $last_rcvd_k$: n 크기의 정수 배열. $last_rcvd_k[i]$는 p_k의 가장 최근 검사점과 인과적 의존관계에 있는 p_i의 검사점 구간 번호를 의미한다. 검사점 생성 시 $VC_k[]$ 값이 $last_rcvd_k[]$에 복사된다. p_i에게 메시지 m을 보낼 때, $last_rcvd_k[i]$ 값이 m에 덧붙여져 보내어진다. 이를 $m.last_rcvd$ 라 한다.

프로세스 p_k가 p_i로부터 메시지를 받았을 때, $m.last_rcvd$와 $VC_k[k]$가 서로 같다면, 길이가 2인 Z-cycle이 형성되므로, m을 처리하기 전에 p_k는 검사점을 생성한다. 이 프로토콜의 이름을 Length Two Z-Cycle 검출기법 (이하 LTZC 프로토콜)이라 하며, 앞의 2절에서의 Theorem 2.1과 Theorem 2.2에 근거하여 다음과 같은 정리를 유도할 수 있다.

Corollary 2.1 : T-유형에서, LTZC 프로토콜은 Z-Cycle 부재 속성을 만족시킨다.

3. 새로운 통신 유도 검사점 프로토콜

이 절에서는 두 가지 통신 유도 검사점 프로토콜을 제안한다. 한 프로토콜은 앞 절에서 설명한 '길이 2인 Z-cycle 검출 기법'이 임의의 통신 유형을 가진 어플리케이션에서도 적용될 수 있도록, 기존의 통신 유형을 T-유형으로 변형한다. 나머지 프로토콜 역시 원래의 통신 유형을 변형하되, 길이 2 뿐만 아니라 그 이상의 길이를 가진 Z-cycle을 검출할 수 있다.

3.1 Length Two Z-Cycle 프로토콜 적용을 위한 통신 유형 변형

LTZC 프로토콜은 T-유형 이외의 통신 유형에서는 사용할 수 없다. T-유형 이외의 통신 유형에서도 LTZC 프로토콜을 적용하기 위해서, 본 연구는 통신 유형의 성형을 제안한다. 원래의 통신유형을 T-유형으로 바꾸게 되면, LTZC 검출 프로토콜을 적용할 수 있게 된다. 그림 2는 격자(mesh) 통신 유형을 변형한 사례를 보여준다. 이 변형에서 메시지를 포워드 해주는 프로세

1) Z-path(혹은 Z-cycle)의 길이는 Z-path(혹은 Z-cycle)에 포함된 메시지 개수를 뜻한다. 최소 Z-cycle이란 더 작은 Z-cycle을 포함하지 않는 Z-cycle을 뜻하며, 이는 논문 [15]에서의 Core-Z-Cycle과 같은 개념이다.

2) 위 두 Theorem의 증명은 참조 문헌 [4]를 읽기를 양해 구하며, 지면관계상 생략한다.

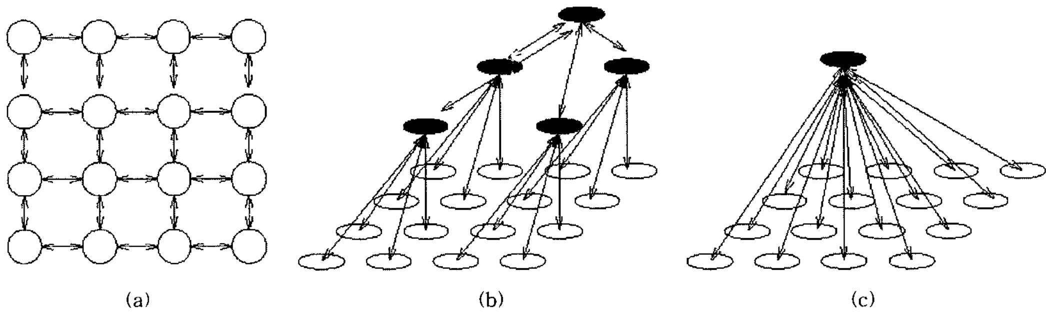


그림 2 통신 유형 변형의 예: (a) 16 프로세스의 원래 격자 통신 유형 (b) 두 단계 계층 통신 유형으로의 변형 (c) 중앙 집중 통신 유형으로의 변형

스들의 추가가 요구된다. 그림 2(b)와 (c)에서 검은색으로 표시된 프로세스가 바로 추가된 프로세스인데, 이를 Communication Monitors(CM) 프로세스라고 이름지었다.

이런 통신 유형의 변형은 평균 메시지 전송 시간을 두 배 이상으로 늘리게 되어, 성능 저하를 유발한다. 이러한 부하를 줄이기 위해서, 본 연구는 LTZC 프로토콜을 다음과 같이 수정하였다. 검사점 구간 중에서 각 프로세스에게 최초로 보내는 메시지만 변형된 통신 유형대로 보내고, 그 후의 메시지들은 직접 목적 수신 프로세스에게 보내는 것이다. 즉, p_k가 p_i에게 보낼 메시지 m이 현재 검사점 구간 중에서 p_i에게 처음 보내는 메시지라면, 변형된 통신 유형에서 정해진 이웃 프로세스인 p_j 혹은 CM에게 보낸다. 만일 현재 검사점 구간 중에서 이미 p_i에게 메시지를 보낸 적이 있다면, 바로 p_i에게 메시지를 보낸다. 임의의 검사점 구간에서 특정 프로세스에게로 처음 보내는 메시지를 **프라임 메시지**(prime message)라고 한다.[3] 프라임 메시지는 변형된 T-유형에서 정의된 대로 프로세스들에 의해 전달되어 최종 목적 수신 프로세스에게 도착된다. T-유형대로 메시지가 포워드되는 과정을 'T-유형을 따른다'라고 표현한다. 프라임 메시지 이후, 목적 수신 프로세스에게 바로 보내어지는 메시지는 숏컷 메시지 (shortcut message)라고 한다. 그림 3은 p_k에서 p_i로 가는 프라임 메시지 m과 숏컷 메시지 n을 보여준다. 메시지 l은 p_i에게 직접에게 보내어진다. 이미 m을 전달하면서 프라임 메시지를 보낸 셈이 되기 때문이다. 프라임 메시지 m이 재전송되면서 형성하게 되는 Z-path는 프라임 path 혹은 프라임 Z-path라고 하며, 숏컷 메시지 n에 대응된다. 각 프로세스들은 새로운 검사점이 생성되면, 메시지를 보내려 할 때 다시 프라임 메시지로 보내어야 한다.

3) 프라임(prime)이라는 용어는 논문 [12]에서 차용하였다. 만일 어떤 causal path, μ가 새로운 검사점의 존재를 $\mu.last$의 수신 프로세스에게 최초로 알린다면, μ는 프라임 path가 된다.

수정된 LTZC 프로토콜에서는 프라임 메시지는 항상 숏컷 메시지보다 먼저 수신되어야 한다. 하지만 그림 3과 같이 숏컷 메시지가 먼저 수신 프로세스에 도착하는 경우가 발생할 수 있다. 이를 막기 위해서는 숏컷 메시지가 먼저 도착하더라도 그 처리를 늦추어야 한다. 각 프로세스들은 변형된 T-유형에서 정해진 이웃 프로세스들에게서가 아니라 다른 프로세스에게서 받은 메시지가 새로운 검사점 이벤트를 알린다면, 이 메시지에 대응하는 프라임 메시지가 도착할 때까지 임시 버퍼에 저장시킨다. 대응되는 프라임 메시지가 도착하여 수신 처리가 되면, 그 후에 임시버퍼에 저장된 숏컷 메시지들을 수신 처리한다.

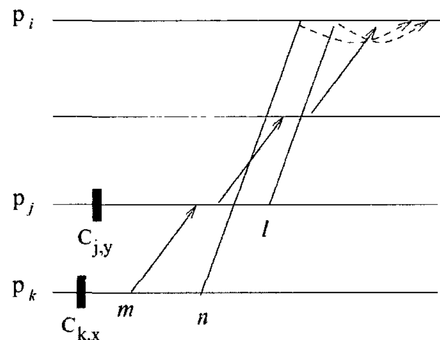


그림 3 프라임 메시지와 숏컷 메시지의 예

수정된 LTZC 프로토콜은 **LTZC' 프로토콜**이라 하며, 임의의 프로세스 p_k는 다음 자료구조를 가지고 있다.

- $VC_k[]$, $last_rcvd_k[]$: 각 배열의 크기는 n에서 N로 바뀐다. N은 원래 프로세스의 개수와 통신유형 변형과정에서 추가된 프로세스의 개수를 합한 숫자이다.

- *after_send*$_k$[] : N 크기의 boolean 배열. p_k가 현재 검사점 구간 중에 p_i에게 메시지를 보낸 적이 있으면 *after_send*$_k$[i]는 TRUE, 그렇지 않으면 FALSE 값을 가진다.
- *route_table*$_k$[] : N 크기의 정수 배열. 변형된 통신 유형에 의해 정의된 이웃 프로세스 정보를 지니고 있다. 즉, *route_table*$_k$[i]는 p_k가 p_i에게 메시지를 보내고자 할 때, 메시지를 전달해줄 CM 혹은 다른 이웃 프로세스 번호를 지니고 있다. 이 배열은 각 프로세스들이 실행을 시작할 때 정해진다.

부록에 서술된 알고리즘 1은 LTZC′ 프로토콜의 psuedo-code이다.

3.1.1 증명

숏컷 메시지를 이용하는 LTZC′ 프로토콜이 Z-cycle을 생성시키지 않음을 다음과 같이 증명하였다.

Theorem 3.1 : 숏컷 메시지를 이용하는 LTZC′ 프로토콜은 Z-Cycle 부재 조건을 만족시킨다.

Proof : (반증법에 의해) LTZC′ 프로토콜이 임의의 Z-cycle, $ZC(C_{i,x},\mu,\zeta,I_{j,y})$을 검출하지 못해서 허용한다고 가정하자(그림 1).

경우 1. 두 Z-path ζ와 μ가 T-유형을 따르는 경우. 즉, 두 Z-path 모두가 숏컷 메시지를 포함하지 않는 경우이다. Corollary 2.1에 의해서 LTZC′ 프로토콜은 T-유형에서는 Z-cycle을 생성시키지 않는다는 것을 보였다. 따라서, $send(\zeta.first)$와 $deliver(\mu.last)$는 같은 검사점 구간에 속할 수 없으며, $ZC(C_{i,x},\mu,\zeta,I_{j,y})$가 존재한다는 가정은 잘못되었다.

경우 2. Z-path ζ나 혹은 μ가 한 개 이상의 숏컷 메시지를 포함한 경우. 임의의 숏컷 메시지 m가 ζ에 속한다고 하자. 숏컷 메시지 m에 대응되는 프라임 Z-path η가 존재하며, η는 다음 사항을 모두 만족시킨다.

(a) η는 숏컷 메시지를 포함하지 않는다. 즉, η는 T-유형을 따른다.

(b) $send(\eta.first) \xrightarrow{e} send(m)$이며, $send(\eta.first)$와 $send(m)$은 같은 검사점 구간에 속한다.

(c) $deliver(\eta.last) \xrightarrow{e} deliver(m)$

ζ에 속하는 모든 숏컷 메시지들을 대응되는 프라임 Z-path로 치환한다면, 다음을 만족시키는 ζ'가 존재하게 된다.

(a) $\zeta'.first=\zeta.first$ 이거나, $send(\zeta'.first) \xrightarrow{e} send(\zeta.first)$이고 두 이벤트는 같은 검사점 구간에 속한다.

(b) $\zeta'.last=\zeta.last$ 이거나, $deliver(\zeta'.last) \xrightarrow{e} deliver(\zeta.last)$이다.

μ에 대해서도 마찬가지로 대응되는 Z-path μ'가 존재한다. 만일 Z-cycle인 $ZC(C_{i,x},\mu,\zeta,I_{j,y})$가 존재한다고 하면, 그림 1에서와 같이 $ZC(C_{i,x},\mu',\zeta',I_{j,y})$도 존재하여야 한다. 그러나, ζ'과 μ'는 T-유형을 따르므로, 경우 1에서 보였듯이 $ZC(C_{i,x},\mu',\zeta',I_{j,y})$는 존재할 수 없다. 따라서 $send(\zeta'.first)$와 $deliver(\mu'.last)$는 같은 검사점 구간에 속할 수가 없다. 결과적으로 $send(\zeta.first)$는 $deliver(\mu'.last)$나 $deliver(\mu.last)$와 같은 검사점에 속할 수가 없게 된다. 경우 2에서도 $ZC(C_{i,x},\mu,\zeta,I_{j,y})$가 존재한다는 것은 거짓이다.

결과적으로 $ZC(C_{i,x},\mu,\zeta,I_{j,y})$가 존재한다는 가정은 잘못되었으며, 숏컷 메시지를 활용한 LTZC′ 프로토콜은 Z-cycle을 생성시키지 않는다. (증명 끝)

3.2 Arbitrary Length Z-Cycle 검출 기법

두 번째로 제안하는 Arbitrary Length Z-Cycle 검출 기법(이하 ALZC 프로토콜)은 LTZC 프로토콜과 마찬가지로 메시지를 포워드하는 프로세스의 추가를 요구한다. 그러나 ALZC 프로토콜은 오직 한 개의 CM만이 존재하며, 항상 중앙 집중 통신 유형으로 변형한다. 모든 프라임 메시지는 CM에게로 보내어지며, 숏컷 메시지는 수신 프로세스에게 직접 보내어진다. 다루는 벡터 클럭의 크기는 CM을 제외한 프로세스의 개수이다. 각 프로세스는 현재의 벡터 클럭을 메시지에 덧붙여 보내게 된다. 첫 번째 프라임 메시지를 보낼 경우에는 가장 최근 검사점의 벡터 클럭도 덧붙여 보내어, 검사점의 존재를 CM에게 알린다. 임의의 길이를 지닌 Z-cycle을 검사하기 위해서 CM은 덧붙여진 정보를 이용하여 검사점 의존관계 그래프를 그린다.

그림 4(a)와 (b)는 각각 원래의 통신 유형과 ALZC 프로토콜에 의해서 변형된 통신 유형의 한 예를 보여준다. 그리고 그림 4(c)는 CM이 생성하게 되는 의존관계 그래프를 보여주고 있다. 그림 4(c)에서 각 노드들은 검사점 구간을 의미하고 화살표(directed edge)들은 의존관계를 보여주고 있다. CM은 p_i가 p_k에게 보내는 메시지 m을 받게 되면, p_k의 현재 검사점 구간에서 시작하여 p_i로 가는 path가 존재하는지, 그리고 그 path가 적어도 한 개 이상의 검사점을 포함하고 있는지를 살펴본다. 만일, 두 조건이 만족된다면 CM은 Z-cycle이 형성된다고 인식한다. 그리고 메시지 m을 p_k에게 재전송하기 전에 CM이 알고 있는 p_k의 현재 검사점 인덱스를 덧붙여 보낸다. 덧붙여진 검사점 인덱스는 *m.alzc_rcvd* 라는 곳에 저장되어 보내진다. CM은 메시지를 보낸 후

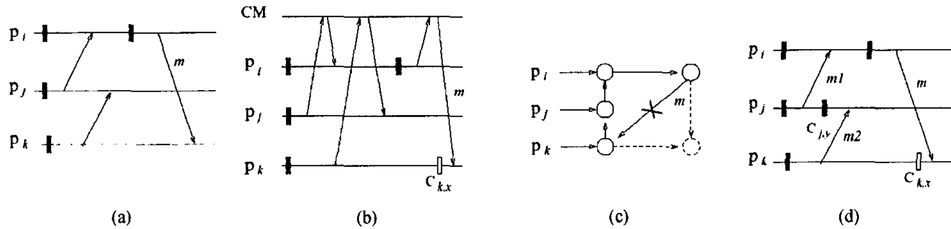


그림 4 ALZC 프로토콜의 동작 예: (a) 원래 통신 유형, (b) 변형된 통신 유형, (c) CM에서 생성하는 의존 관계 그래프, (d) ALZC 프로토콜의 오판 예

에, 의존관계 그래프에 p_k에 새로운 노드를 추가시키고 p_i에서 p_k로의 화살표를 추가시킨다. 메시지를 받은 p_k는 $m.alzc_rcvd$와 자신의 검사점 인덱스가 같다면 Z-cycle이 생성되려는 것이라 간주하고 메시지를 처리하기 전에 검사점을 생성한다. 만일 같지 않다면 검사점을 생성하지 않는다. 메시지 처리 후에는, $m.VC[]$의 값을 $VC_k[]$에 반영시킨다.

중앙에서 CM이 모든 직접적인 의존관계를 파악하고 있기 때문에, ALZC 프로토콜은 임의의 길이를 가진 Z-cycle을 감지해낼 수 있다. 하지만, ALZC에서도 그림 4 (d)에서처럼 불필요한 검사점이 생성될 수도 있다. m_1과 m_2는 Z path를 형성하지 않지만 p_j가 알려주기 전에는 CM은 검사점 $C_{j,y}$의 존재를 알지 못하기 때문에 p_k가 강제적인 검사점을 생성시켜야 한다고 결정한다. 이를 막기 위해서는 검사점 생성을 알리는 메시지를 CM에게 즉시 보내주어야 한다. 이는 추가적인 메시지 부하를 요구하며, 또 시뮬레이션으로 확인해본 바에 의하면 흔한 현상이 아니므로 본 논문에서는 검사점 알림 메시지를 구현하지 않았다.

부록에 서술된 알고리즘 2는 ALZC 프로토콜의 psuedo-code이며, LTZC′ 프로토콜과 다른 부분만을 서술하였다.

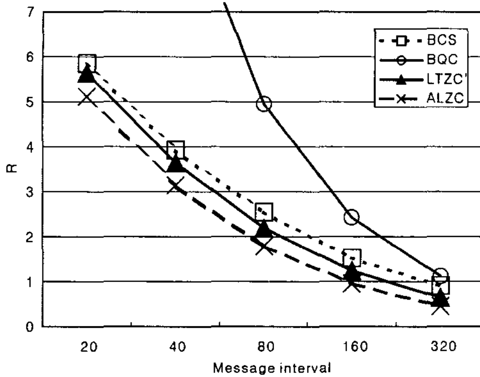
4. 실험 결과

제안한 두 가지 통신 유도 검사점 프로토콜의 평가를 위해, 본 연구에서는 가상의 통신 트레이스를 활용하여 시뮬레이션을 하였다. 16 개의 프로세스가 100,000 단위의 시간 동안 수행을 하며, 평균 1,000 단위마다 한 번씩 검사점을 생성한다. 각 검사점은 독립적으로 생성되며, 검사점 이벤트는 지수분포에 근거하여 발생한다. 최악의 경우를 가정하기 위해서, 각 프로세스의 검사점이 되도록 동시에 발생하지 않도록 설정하였다. 결함과 메시지 전송 이벤트들은 각각 평균 T_f 와 T_m 시간 단위의 지수분포에 근거하여 발생한다. '$T_f=400$ 단위'로 고정시켰으며, T_m는 20단위에서 320단위까지 변화시키면서 성능을 측정하였다. 사용한 통신 유형은 격자 구조 유형 (mesh)과 무작위 유형 (random) 두 가지를 사용하였다. LTZC′ 프로토콜의 경우, 격자 유형은 두 단계 계층적 구조로 변형이 되며 무작위 유형은 중앙 집중 유형으로 변형된다. 성능 비교를 위해서 측정한 사항들은 강제 검사점 총 개수, 평균 메시지 지연, 그리고 결함 발생 시 손실되는 계산량(rollback distance)이다. 대표적인 통신 유도 검사점 기법들인 *Briatico* 등의 프로토콜(BCS)[9]과 *Baldoni* 등의 프로토콜(BQC)[11]를 함께 구현하여 제안된 프로토콜과 성능 비교를 하였다.

4.1 검사점 개수

그림 5는 강제 검사점 개수와 원래 계획된 검사점과의 비율, R을 나타낸 그래프이며, (a)와 (b)는 각각 격자 통신 유형과 무작위 통신 유형일 때의 경우를 나타낸다. 격자유형의 경우, LTZC′/ALZC 프로토콜은 기존의 프로토콜들보다 더 적은 검사점을 생성시킨다. LTZC′ 프로토콜의 경우, 되도록 통신을 빈번히 하는 프로세스들끼리 군집을 이루도록 변형시켰기 때문에 강제 검사점의 영향이 최소화된다. 무작위 통신 유형에서는 특별히 빈번하게 통신하는 프로세스가 따로 없이, 모든 프로세스들과 통신하기 때문에 LTZC′ 프로토콜의 변형된 통신 유형은 오히려 메시지를 많이 발생시키기만 할 뿐이다. 특히 메시지를 빈번하게 보내는 경우에는 오히려 LTZC′ 프로토콜이 BCS 프로토콜보다 더 많은 검사점을 생성시킨다(그림 5(b)). 그러나 무작위 통신 유형은 최악의 상황을 고려한 시뮬레이션 환경일 뿐, 실제 어플리케이션들은 일정한 통신 유형을 가지게 되므로[16,17], 이 경우는 현실적이지 않다고 본 연구는 생각한다.

ALZC 프로토콜은 어떠한 통신 유형에서도 최소 개수의 검사점을 생성한다. 이는 ALZC 프로토콜이 보다 정확하게 Z-cycle을 검출하여 꼭 필요한 부분에서만 강제 검사점을 생성하기 때문이다. BQC 프로토콜은 Z-cycle을 검출하는 조건이 너무 약하기 때문에 불필요한



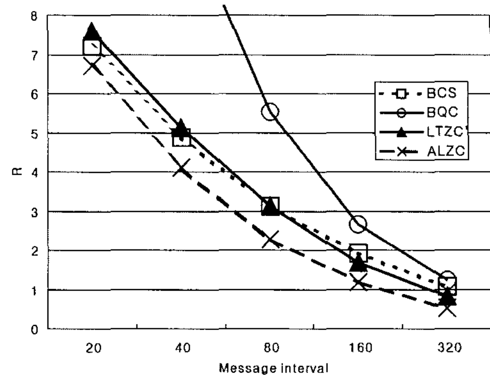


그림 5 강제 검사점 총 개수와 지역 검사점 개수와의 비율: (a) 격자 통신 유형, (b) 무작위 통신 유형

곳에서도 검사점을 생성하는 경우가 빈번하다. 이 프로토콜은 다른 프로토콜과 비교하여 너무 큰 성능 차이를 보이기 때문에, 이후의 그래프들에서는 제외를 하였다.

4.2 메시지 지연

제안된 프로토콜의 단점은 바로 메시지의 지연이 불가피하다는 점이다. 이를 완화시키기 위해서 숏컷 메시지 개념을 도입하였음을 3 절에서 설명하였다. 그림 6은 제안된 프로토콜의 평균 메시지 hop 수를 나타낸다. 예를 들면, 원래의 통신 유형에서는 메시지의 hop수는 1이다. 중앙 집중적과 두 단계 계층 통신 유형에서는 프라임 메시지의 최대 hop 수는 각각 2와 4가 된다. ALZC 프로토콜는 중앙 집중 유형이 적용되며, 최대 hop 수는 2이기 때문에 LTZC′ 프로토콜보다 낮은 평균 메시지 지연을 가진다. 무작위 통신유형에서는 두 프로토콜 모두 중앙 집중 유형으로 변형되므로, 두 프로토콜 사이의 메시지 지연 차이가 그다지 두드러지지 않음을 알 수 있다. ALZC 프로토콜이 LTZC′ 프로토콜보다 약간 더 낮은 평균 메시지 hop수를 가지는 이유는, ALZC 프로토콜이 보다 더 적은 수의 검사점을 생성하기 때문이다. 즉 검사점 구간의 수도 줄어들어 프라임 메시지도 줄어들게 된다.

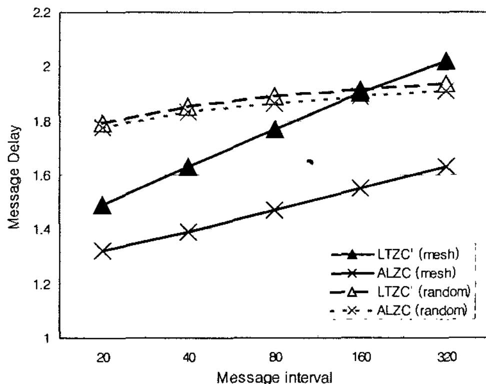


그림 6 제안된 프로토콜을 사용하였을 때, 평균 메시지 hop 수

두 프로토콜 모두 격자 통신 유형의 경우에는 평균 메시지 hop수가 최대 hop수보다 훨씬 낮은 반면, 무작위 통신 유형의 경우에는 최대 hop수에 근접한다. 이는 각 프로세스들이 자주 통신을 하게 되는 프로세스의 개수와 관계가 있다. 격자 통신 유형에서는 각 프로세스는 기껏해야 네 개의 다른 프로세스와 통신하는 반면, 무작위 유형에서는 $(n-1)$개의 프로세스와 통신을 하게 되며, 이는 한 검사점 구간에서의 프라임 메시지 개수와 직결된다.

메시지의 통신 빈도도 영향을 줌을 알 수가 있다. 메시지를 보내는 빈도가 줄어들면, 프라임 메시지 개수와 숏컷 메시지 개수의 비율이 높아지기 때문에 평균 메시지 지연이 높아진다.

요약을 하면, 메시지의 지연에 영향을 미치는 요인은, 한 프로세스 당 빈번히 통신하는 프로세스들의 개수, 생성되는 검사점 개수, 그리고 변형된 통신 유형에서의 최대 메시지 hop 수이다.

4.3 결함 부재 시 추가 부하

제안된 프로토콜은 검사점으로 인한 부하를 줄이는 반면, 메시지 지연을 유발한다. 이 두 가지가 전체 성능에 어떻게 영향을 미치는 지를 이 절에서 설명한다. 본 시뮬레이션에서는 한 개의 검사점은 3 초, 그리고 64 KBytes 크기의 메시지 전송 시간은 한 hop당 0.01초가 걸린다고 가정하였다.[4] 그림 7은 BCS 프로토콜의 부하

4) 사용한 메시지 지연 시간 값은 MPI 프로세스로 핑퐁 테스트를 100Mbps 이더넷으로 연결된 PC 클러스터에서 수행하여 얻은 값이다. 검사점에 걸린 시간도 연구 결과에 의하면 3~7 초 정도 걸린다. 본 연구는 제안된 프로토콜에 최대한 불리한 값을 선택하여 사용하였다.

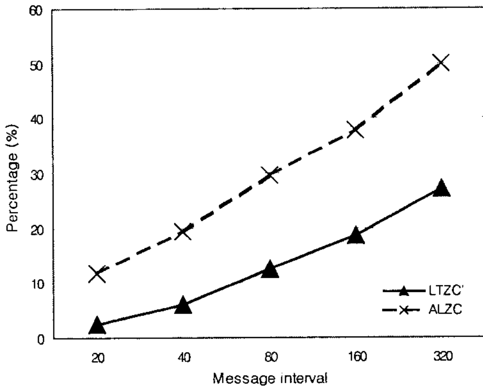


그림 7 격자 통신 유형에서 예상되는 부하 감소율

와 비교하여, 제안된 프로토콜들의 감소된 부하를 백분율로 나타낸 그래프이다.

메시지 전송 빈도가 높을 때는 메시지 지연으로 인한 부하가 최대가 되므로, 제안된 프로토콜의 효과가 최소임을 알 수 있다. 반면, 메시지 전송 빈도가 낮을 때는 메시지 지연이 미치는 영향도 낮아져서 부하의 감소량이 커진다. 어떠한 경우에서도 제안된 프로토콜은 BCS 프로토콜보다 더 적은 수행시간을 지니게 됨을 알 수 있다. 즉, 메시지 지연으로 인한 부하보다는 검사점 생성으로 인한 디스크 사용 부하를 줄이는 것이 더 이득이라는 결론을 내릴 수 있다.

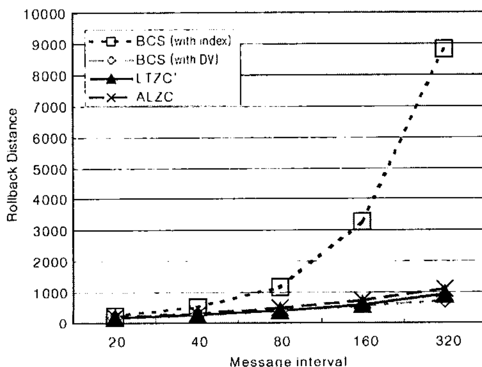
4.4 롤백 거리

롤백 거리라 함은, 현재 수행 부분부터 복구가 될 검사점까지의 거리를 시간으로 환산한 값으로, 손실되는 계산량을 의미한다. 본 연구는 실험을 통해서, BCS 프로토콜은 일관적인 전역 검사점을 찾기 위해서 지나치게 많은 검사점을 건너뛰어 롤백하는 것을 발견하였다. BCS 프로토콜은 일관적인 전역 검사점을 찾기 위해서 '전역 검사점 인덱스'라는 한 개의 정수를 사용한다. 즉 같은 인덱스를 가진 검사점들이 전역 검사점을 형성하는 방식이다. 이러한 방식에서는 같은 인덱스를 가진 검사점이 모두 프로세스 개수인 n개가 있어야 하며, 하나라도 모자라는 경우에는 이 인덱스의 나머지 검사점들은 모두 쓸 수 없게 된다. 그러므로 n개의 검사점이 모두 존재하는 인덱스를 찾아서 더 뒤로 롤백을 해야 한다. 본 시뮬레이션에서 살펴본 바로는 BCS 프로토콜을 사용할 경우, 최악의 경우에는 전체 검사점 중 오직 7 %만이 쓸모가 있고 나머지는 복구에 사용할 수 없음을 알게 되었다.

그림 8은 BCS 프로토콜과 제안된 프로토콜의 롤백 거리를 측정한 그래프이다. BCS 프로토콜의 경우는 인덱스만 사용한 경우와 벡터 클럭을 사용한 경우, 두 가지를 측정하였다. 두 가지 경우 모두, 생성하는 검사점 개수나 위치는 모두 동일하다. LTZC'와 ALZC 프로토콜은 벡터 클럭을 사용하여 일관적인 전역 검사점을 검색한다. 인덱스 기반의 BCS 프로토콜은 복구 지점을 찾는 방법이 효율적이지 않아 최악의 성능을 보였다. 반면 벡터 클럭을 사용한 나머지 기법은 기껏해야 한 검사점 구간 정도만의 계산량을 손실하는 것을 알 수 있었다. 벡터 클럭을 사용하는 BCS 프로토콜이 제안된 프로토콜보다 조금 더 적은 계산량을 손실하는데, 이는 검사점을 더 많이 생성하기 때문에 검사점과 검사점 사이의 간격이 조금 더 짧기 때문이다. 제안된 프로토콜들은 더 적은 수의 검사점을 생성하면서도 계산 손실량이 크지 않음을 알 수가 있다.

5. 결 론

본 논문은 두 개의 새로운 통신 유도 검사점 기법들을 제안하였다. 제안된 검사점 기법은 기존의 통신 유도 검사점 기법들보다 더 적은 수의 검사점을 생성하며 No



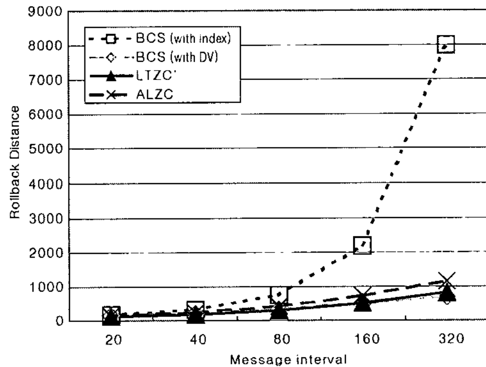


그림 8 결함 발생시 손실되는 평균 계산량(롤백거리) :
(a) 격자 통신 유형, (b) 무작위 통신 유형

Z-Cycle 속성을 만족시킨다. 제안된 프로토콜은 메시지를 재전송하는 프로세스(CM)의 추가와 함께 메시지 지연의 증가라는 부하가 요구되지만, 전체적으로 검사점 수의 감소로 인해 얻는 이득이 큼을 실험을 통해 보였다. 추가된 CM은 검사점을 생성할 필요가 없으며, 단순히 벡터 클럭만 기록을 해두면 된다. 제안된 프로토콜은 벡터 클럭을 사용하여 검사점간의 의존관계를 파악하며 메시지에 이 벡터 클럭을 덧붙여 보낸다.

제안된 프로토콜은 원래의 통신 유형을 변형시켜야 할 필요가 있기 때문에 어플리케이션의 특성을 실행 전에 미리 파악해두고 변형할 유형을 설계해야 한다. 변형된 통신 유형과 원래의 통신 유형이 유사한 정도에 제안된 프로토콜의 성능이 영향을 받기 때문이다. ALZC 프로토콜은 가장 정확하게 Z-cycle을 파악하기 때문에 제일 적은 수의 검사점을 생성하지만, Z-cycle을 검출하는 과정은 시간 복잡도가 높기 때문에 확장성에 제약이 있을 수 있다.

본 논문에서는 최악의 경우를 가정하여 실험하였기 때문에, 강제 검사점의 개수가 계획된 지역 검사점 개수의 수 배나 되었다. 하지만, 실제 상황에서는 지역 검사점이 거의 동시에 생성되며, 이 경우에서는 강제 검사점의 개수는 현저하게 줄어들게 된다.

본 연구는 앞으로, 임의의 통신 유형을 LTZC′ 프로토콜에 가장 적합한 통신유형으로 변형하는 문제와 함께 Z-cycle의 검출과정의 시간 복잡도를 줄이는 문제를 해결하고자 한다.

참 고 문 헌

[1] E. N. Elnozahy, L. Alvisi, Y. -M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, oct 1996.

[2] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computing Systems*, 3(1):63-75, AUG 1985.

[3] R. Koo and S. Toueg. Checkpointing and rollback recovery for distributed systems. *IEEE Transaction on Software Engineering*, SE-13(1):23-31, 1987.

[4] T. Park and H. Y. Yeom. Application controlled checkpointing coordination for fault tolerant distributed computing systems. *Parallel Computing*, 26(4):467-482, MAR 2000.

[5] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic and causal. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 229-236, 1995.

[6] N. Neves and W. K. Fuchs. RENEW: A tool for fast and efficient implementation of checkpoint protocols. In *Symposium on Fault-Tolerant Computing*, pages 58-67, 1998.

[7] Y. -M. Wang and W. K. Fuchs. Optimistic message logging for independent checkpointing in message-passing systems. In *Symposium on Reliable Distributed Systems*, pages 147-154, 1992.

[8] L. Alvisi, E. N. Elnozahy, S. Rao, S. A. Husain, and A. D. Mel. An analysis of communication induced checkpointing. In *Symposium on Fault-Tolerant Computing*, pages 242-249, 1999.

[9] D. Briatico, A. Ciuffoletti, and L. Simoncini. A distributed domino-effect free recovery algorithm. In *Proceedings of the IEEE International Symposium on Reliability Distributed Software wand Database*, pages 207-215, DEC 1984.

[10] J. Helary, A. Mostefaoui, R. Netzer, and M. Raynal. Preventing useless checkpoints in distributed computations. In *Proceedings of IEEE International Symposium on Reliable Distributed Systems*, pages 183-190, 1997.

[11] R. Baldoni, F. Quaglia, and B. Ciciani. A VP-accordant checkpointing protocol preventing useless checkpoints. In *Symposium on Reliable Distributed Systems*, pages 61-67, 1998.

[12] R. Baldoni, J. H´elary, and M. Raynal. Rollback-dependency trackability. Technical Report Report 1107, IRISA Research, MAY 1997.

[13] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565, 1978.

[14] R. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165-169, 1995.

[15] F. Quaglia, R. Baldoni, and B. Ciciani. On the no-z-cycle property in distributed executions. *Journal of Computer and System Sciences*, 61(3): 400-427, 2000.

[16] Y. Nah. *The Specification of Task Communication Patterns*. PhD thesis, Seoul National University, Korea, 1997.

[17] G. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1):49-90, 1991.

부 록

알고리즘 1 LTZC′ 프로토콜

```
init pk
검사점 생성;
∀i, VCk[i]: =0; VCk[k]: = 1; after_sendk[i]: =FALSE ; last_rcvdk[i]: = 0; Uk= ∅ ;
route_tablek[] 초기화 ;

procedure take_ckpt()
검사점 생성;
last_rcvdk[]: = VCk[]; VCk[k]: = VCk[K] + 1;
∀i, after_sendk[i]: =FALSE ;

procedure send( m, pj, data): // pk가 pj에게 data의 내용을 메시지 m에 실어 보냄
m.recent_sndr : =k ; m.VC[k] : = VCk[k];
if m.dest= NULL then
        m.dest : =j ; m.content : = data ;
end if
if ( pj ∈ { pk의 이웃 프로세스} ) or after_sendk[j]= TRUE then
        m.last_rcvd : =last_rcvdj[j];
        m을 pj에게 바로 보낸다; // m은 숏컷 메시지가 된다.
else
        i : =route_tablek[j] ; m.last_rcvd : =last_rcvdk[i] ;
        m을 pi에게 보낸다;
        after_sendk[j]: = TRUE ;
end if

procedure receive( m, pj) : // pk가 pj로부터 메시지 m을 받았다.
if ( pj ∈ { pk의 이웃 프로세스}) then
        if m.last_rcvd= VCk[k] then
                take_ckpt(); // 길이 2인 Z-cycle이 검출되었다.
        end if
        ∀i, VCk[i]: = max(VCk[i], m.VC[i]) ;
        if m.dest= k then
                deliver(m); deliver_umsg() ;
        else
                send( m, pm.dest, m.content);
        end if
else    // 숏컷 메시지일 경우
        if VCk[j] < m.VC[j] then
                m을 buffer Uk에 넣는다.
        else
                ∀i, VCk[i]: = max( VCk[i], m.VC[i]);
                deliver(m) ;
        end if
end if

procedure deliver_umsg()
∀m ∈ Uk , if m.VC[m.recent_sndr] ≥ VCk[m.recent_sndr] then
        ∀i, VCk[i]: = max(VCk[i], m.VC[i]);
        deliver(m) ;
endif
```

알고리즘 2. ALZC 프로토콜

```
procedure send( m, p_j, data): // p_k가 p_j에게 data의 내용을 m에 실어 보낸다.
m.content : =data ; m.dest : = j ;
m.VC[] : = VC_k[] ; m.alzc_rcvd : =−1 ;
if after_send_k[j] = TRUE then
        p_j에게 m를 보낸다;
else
        if (CM에게 아직 알리지 않은 검사점이 있다) then
                m.last_ckpt[]: =마지막 검사점의 VC[] ;
        end if
        m을 CM에게 보낸다;
        after_send_k[j] : = TRUE ;
end if

CM : procedure receive( m, p_k) : // CM이 p_k에게서 메시지 m을 받다.
if m.last_ckpt[] ≠NULL then
        의존관계 그래프에서 p_k에 새로운 노드를 추가한다;
        m.last_ckpt[]값을 새로운 노드에 할당한다;
end if
p_k의 현재 노드에서 p_m.dest의 현재 노드로 화살표를 추가한다;
m.alzc_rcvd : =−1 ;
for all ζ ( ζ는 p_m.dest의 현재 노드에서 p_k로 가는 패스) do
        if ( ζ가 한 개 이상의 검사점을 포함하고 있다.) then
                m.alzc_rcvd : = VC_{p_m.dest의 현재 노드}[m.dest] ;
        end if
end for
m을 p_m.dest에게 보낸다;

procedure receive( m) : // p_k가 p_j가 보낸 메시지 m을 받다.
if ( m이 CM에 의해서 보내어졌다) then
        if m.alzc_rcvd= VC_k[k] then
                검사점을 생성한다;
        end if
        ∀i, VC_k[i]: =max(VC_k[i], m.VC[i]) ;
        deliver(m) ; deliver_umsg() ;
else
        if VC_k[j] < m.VC[j] then
                m을 buffer U_k에 넣는다;
        else
                ∀i, VC_k[i]: =max(VC_k[i], m.VC[i]) ;
                deliver(m) ;
        end if
end if
```

우 남 윤

현재 서울대학교 컴퓨터공학부 박사과정 재학중. 1997년 KAIST 전산학과 학부졸업. 1999년 KAIST 전산학과 석사졸업 관심분야는 결함내성, 분산 시스템, 그리드

염 헌 영

현재 서울대학교 컴퓨터공학부 전임교수 재직. 1984년 서울대학교 계산통계학과 학부 졸업. 1986년 Texas A&M 대학 전산과학과 석사 졸업. 1992년 Texas A&M 대학 전산과학과 박사 졸업. 관심분야는 분산 시스템, 멀티미디어 시스템, 트랜잭션 처리

박 태 순

현재 세종대학교 컴퓨터공학부 부교수 재직. 1987년 서울대학교 계산통계학과 학부 졸업. 1989년 Texas A&M 대학 전산과학과 석사 졸업. 1994년 Texas A&M 대학 전산과학과 박사 졸업. 관심 분야는 결함 내성, 분산 시스템