TRƯỜNG ĐẠI HỌC
BÁCH KHOA HÀ NỘI

TRUNG TÂM
TÍNH TOÁN HIỆU NĂNG CAO

NGHIÊN CỨU

CÁC HỆ THỐNG TÍNH TOÁN HIỆU NĂNG CAO VÀ ỨNG DỤNG MÔ PHỎNG VẬT LIỆU VI MÔ

Trong Khuôn Khổ Hợp Tác Khoa Học Công Nghệ
Theo Nghị Định Thư Với Ấn Độ Giai Đoạn 2004 – 2005

Chủ Nhiệm Đề Tài: PGS. TS. Nguyễn Thanh Thuỷ

Đề tài nhánh

MODULE CHỐNG LỖI

(1/1)

Hà Nội, 3-2006

5957-86
257106

Lời Nói Đầu

Tài liệu này được thực hiện sau một thời gian làm việc về Tính toán hiệu năng cao của nhóm nghiên cứu High Performance Computing tại Trung tâm Tính Toán Hiệu Năng Cao trường Đại Học Bách Khoa Hà Nội.

Trong tài liệu này, chúng tôi xin trình bày, đặc tả chi tiết về vấn đề chống lỗi cho các hệ thống tính toán song song ghép cụm. Tài liệu sẽ trình bày các nghiên cứu tổng quan về chống lỗi, phân loại các phương pháp chống lỗi, khảo sát một số thư viện chống lỗi. Trên cơ sở đó sẽ thiết kế, xây dựng và triển khai module chống lỗi cho BKluster – Hệ thống tính toán song song ghép cụm tại Trung tâm Tính Toán Hiệu Năng Cao.

Tài liệu gồm 5 chương với nội dung như sau:

Chương 1: Trình bày những khái niệm chung về lỗi và chống lỗi trong các hệ thống tính toán.

Chương 2 và chương 3 trình bày chi tiết về hai phương pháp chống lỗi phổ biến nhất hiện nay được áp dụng trong các hệ thống tính toán song song ghép cụm, đó là checkpoint và message logging.

Chương 4 khảo sát 4 thư viện chống lỗi tiêu biểu, đó là BLCR, MPICH-CL, MPICH_V1 và MPICH_V2

Chương 5 trình bày việc thiết kế, xây dựng và triển khai module chống lỗi BKFT dựa trên LAM/MPI, BLCR và PBS trên hệ thống BKluster, Trung tâm Tính Toán Hiệu Năng Cao, Đại Học Bách Khoa Hà Nội.

Mục lục

Danh mục hình vẽ

CHƯƠNG 1. Tổng Quan Về Chống Lỗi

1.1. Vấn đề chống lỗi cho các hệ thống song song ghép cụm

Hệ thống tính toán song song phân cụm có đặc điểm là lỗi tiềm tàng lớn: do lỗi phần cứng, phần mềm, lỗi truyền thông trên mạng. Mỗi khi xảy ra lỗi, hệ thống phải chạy lại, và mất rất nhiều công việc đã thực hiện trước đó. Chúng ta cần phải tìm cách tối thiểu hoá các tính toán bị mất bằng cách phải xây dựng một môi trường chuyển giao các thông điệp có khả năng chống lỗi. Hiện nay, các hệ thống này đều đang sử dụng thư viện truyền thông MPI để thực hiện các ứng dụng song song của mình.

MPI là một hệ thống truyền thông điệp chuẩn. Tuy nhiên, nó không xác định được việc chống lỗi nên được thực hiện như thế nào, và ở mức nào. Do đó, các chương trình sử dụng MPI đều thiết kế không có phần chống lỗi, mà chỉ cung cấp hai trạng thái công việc: OK hoặc FAILED. Thực tế, các lỗi có thể xảy ra ở một số mức của chương trình, và thường làm cho chương trình buộc phải huỷ bỏ. Đối với các chương trình lớn, thời gian chạy tương đối dài, việc này mất nhiều phí tổn, vì các công việc thực hiện được trước khi lỗi xảy ra đều bị mất. Công việc cần cố gắng đạt được là làm sao có thể lấy lại được trạng thái của hệ thống ngay trước khi lỗi xảy ra, và vượt được qua lỗi này. Đối với các hệ thống lớn, có nhiều nút trạm, thì cần phải thực hiện sao cho chỉ các trạm liên quan mới phải thực hiện lại công việc của mình mà không ảnh hưởng đến các nút trạm khác.

Các phương pháp để khắc phục lỗi trong MPI một cách tự động và trong suốt với người dùng hay được sử dụng là phương pháp sử dụng một bộ điều phối trong việc tạo checkpoint hoặc phương pháp ghi lại thông điệp kết hợp với việc tạo checkpoint không có điều phối. Đã có rất nhiều nghiên cứu về giao thức, việc thực hiện, và phương pháp nhằm tối ưu hai cách tiếp cận này.

Phương pháp điều phối các checkpoint có ưu điểm là có chi phí rất thấp trong quá trình khôi phục lỗi trong khi việc ghi lại thông điệp phải mất thêm một số đáng kể các thông điệp khi phục hồi. Tuy nhiên phương pháp sử dụng checkpoint có chi phí bộ nhớ cao vì mỗi khi checkpoint, toàn bộ trạng thái của tiến trình lại được ghi vào bộ nhớ.

Việc sử dụng giao thức nào phụ thuộc vào tần suất xuất hiện của lỗi. Nếu hệ thống có tần suất xuất hiện lỗi càng cao, khoảng thời gian giữa hai lần checkpoint giảm, nên sử dụng phương pháp checkpoint kết hợp với ghi lại thông điệp. Ngược lại, nếu tần suất lỗi giảm, phương pháp checkpoint có sự điều phối có thể hiệu quả hơn.

1.2. Khái niệm chung về lỗi

Do lỗi gây ra tổn thất rất lớn và thường các hệ thống càng lớn thì tổn thất càng cao.Để phát hiện được lỗi, và xa hơn nữa là tránh lỗi, và sửa lỗi, trước tiên, phải có một cái nhìn tổng quát về lỗi, định nghĩa và phân loại lỗi.

Lỗi, được hiểu một cách đơn giản, là một sự kiện xảy ra làm cho hệ thống hoạt động không theo cách thức mong muốn.

Có nhiều cách phân loại lỗi. Ví dụ dựa vào tài nguyên thì có lỗi phần cứng, lỗi phần mềm, hay lỗi môi trường; dựa vào độ bền vững thì có lỗi cố định, và lỗi không cố định … Sau đây là một cách phân loại

Loại lỗi	Mô tả
Lỗi thiếu	Hệ thống lỗi khi đáp ứng yêu cầu gửi đến
+ nhận thiếu	Hệ thống lỗi khi nhận thông điệp đến
+ gửi thiếu	Hệ thống lỗi khi gửi thông điệp
Lỗi thời gian	Kết quả trả về của hệ thống nằm ngoài khoảng thời gian cho phép

Lỗi đáp ứng Lỗi giá trị Lỗi chuyển trạng thái	Đáp ứng của hệ thống không chính xác Giá trị trả về sai Hệ thống không theo luồng điều khiển
Lỗi ngắt (crash failure)	Hệ thống bị ngắt. trước đó vẫn chạy đúng
Lỗi tuỳ ý	Một hệ thống có thể sinh ra một đáp ứng tuỳ ý, tại một thời điểm tuỳ ý

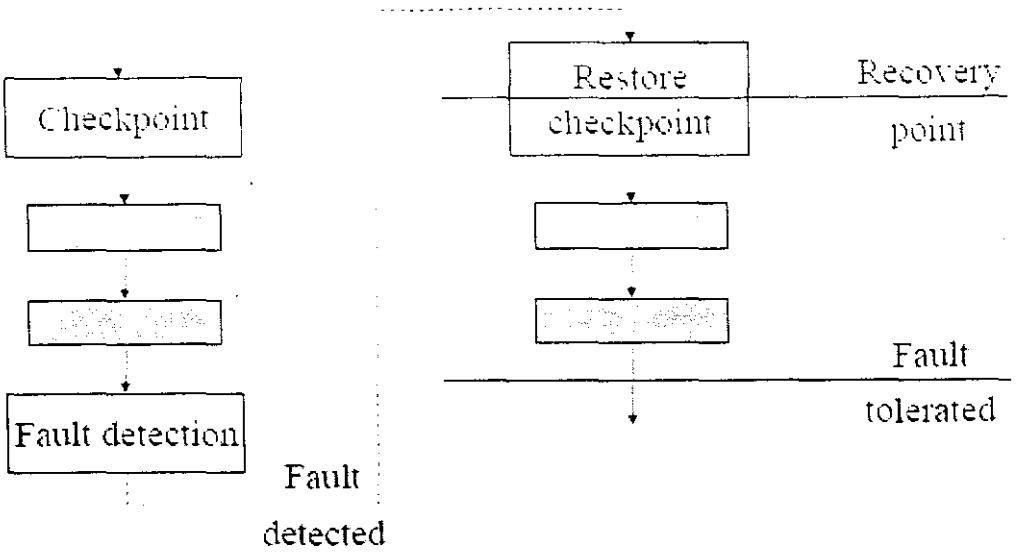
Các mô hình khôi phục lỗi

Có hai mô hình khôi phục lỗi cơ bản là

- ***Khôi phục ngược (Backward Recovery)***

  Thường được sử dụng cho phầm mềm như chương trình ứng dụng hoặc hệ điều hành.

  Trạng thái của các phần mềm được ghi lại định kỳ. Khi lỗi xảy ra, phần mềm được quay lại trạng thái đã được ghi trước đó.

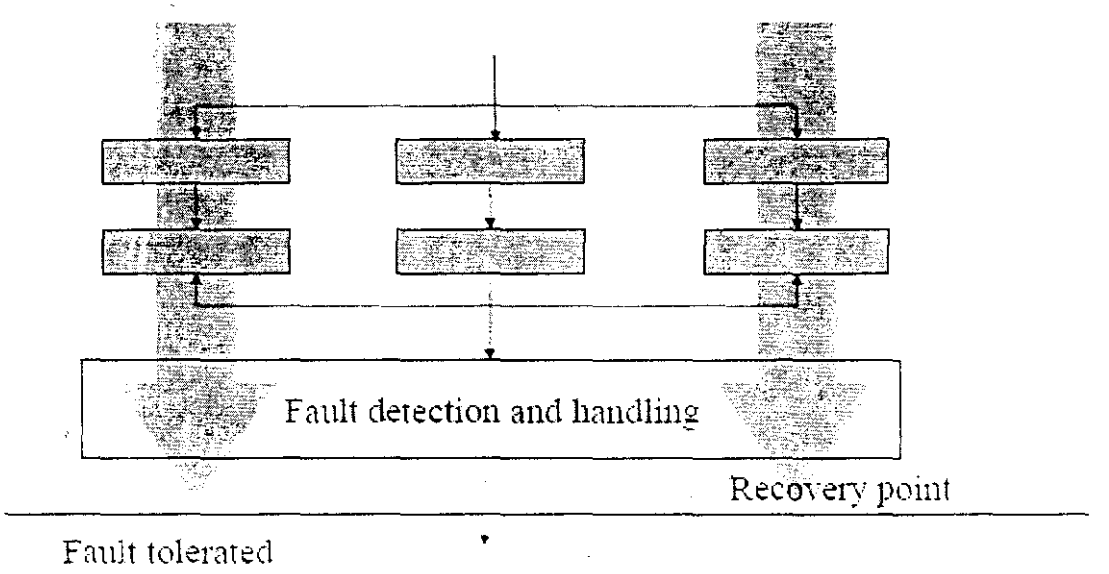


Hình 1-1 Mô hình khôi phục ngược

- ***Khôi phục tiếp (Forward Recovery)***

Thường được dùng cho khôi phục phần cứng.

Nếu hệ thống có lỗi, việc khôi phục lỗi sẽ được thực hiện dựa vào một phần mềm chuyên dụng để lấy lại trạng thái của hệ thống tại thời điểm xảy ra lỗi.



Hình 1-2 Mô hình chống lỗi khôi phục tiếp

Một hệ thống, dù hiện đại đến mấy, được thiết kế chính xác đến bao nhiêu, thì vẫn không thể tránh được lỗi xảy ra. Vì vậy, nhiều khi, nói đến lỗi, và chống lỗi, cũng

chỉ nói được một cách tương đối. Một phần mềm được coi là chống được lỗi, khi và chỉ khi:

- Chương trình có thể cho một kết quả chấp nhận được ngay cả khi nó mắc phải một lỗi logic.
- Chương trình cho kết quả chấp nhận được ngay cả khi nhận được dữ liệu làm ảnh hưởng đến hệ thống khi nó đang chạy.

Vấn đề là, như thế nào thì được gọi là "chấp nhận được"? Khái niệm này bao gồm cả một số tính chất khác như tính an toàn hay tính chính xác, và tuỳ thuộc vào từng hệ thống.

Một số hệ thống chống lỗi đã được thiết kế theo cách: quy định hành vi cho các thành phần khi có lỗi xảy ra, sao cho hệ thống vẫn chạy và cho ra kết quả chấp nhận được, hoặc họ che giấu thành phần lỗi đối với người dùng, và vẫn có thể cung cấp cho họ các dịch vụ chuẩn nhất định khi lỗi xảy ra.

Có rất nhiều hệ thống chống lỗi đã được nghiên cứu và xây dựng. Tuy nhiên, việc ứng dụng của việc chống lỗi trong các hệ thống vẫn còn chưa được ứng dụng rộng rãi.

1.3. Giới thiệu chung về hai hướng tiếp cận chống lỗi trong hệ thống tính toán

Việc dung thứ lỗi cho một tiến trình yêu cầu vài bản sao tiến trình cả về thời gian và không gian:

- Các kĩ thuật dựa trên bản sao trong không gian lỗi bằng cách nhân bản tiến trình để ít nhất một trong các bản sao vẫn có giá trị khi xảy ra lỗi, và bằng cách cùng điều phối sự thực thi các bản sao này. Ví dụ ta có thể dùng phương pháp tự nhân bản.
- Các kĩ thuật dựa trên bản sao trong thời gian, dung thứ lỗi bằng cách khôi phục các thực thi mất kiểm soát của tiến trình lỗi. Trong suốt thời gian giải

phóng lỗi. một tiến trình ghi lại các thông tin về các sự kiện nó thực hiện.Các thông tin được ghi lại này được sử dụng trong suốt quá trình khôi phục để chạy lại sinh lại các thực thi đã mất. Ví dụ ta có thể dùng sao lưu và giao thức ghi lại thông điệp _message-logging_ hay lưu ảnh tiến trình _checkpointing_.

Hai lớp kĩ thuật này đổi lại phải chi phí rất nhiều trong quá trình thực hiện.Các kĩ thuật dựa trên bản sao về không gian đưa ra sự tồn tại liên tục, nhưng gây ra chi phí cao lên hiệu năng của ứng dụng cho điều phối sự thực thi của các bản sao và yêu cầu sử dụng một số lượng lớn tài nguyên cần bổ sung. Mặt khác, các công nghệ dựa trên các bản sao theo thời gian chỉ gây ra chi phí thấp trong giải phóng lỗi và sử dụng tương đối ít tài nguyên, nhưng tốc độ khôi phục lại chậm. Việc lựa chọn theo hai kiểu này đưa ra cách dung thứ lỗi phải dựa trên yêu cầu cảu ứng dụng. Theo tiêu chí của từng ứng dụng(các phần mềm điều khiển quỹ đạo của các vệ tinh), trong các phần mềm này chỉ một lỗi cũng có thể dẫn đến tai nạn thảm khốc, việc dung thứ các lỗi một cách tuỳ biến và cung cấp thực hiện liên tục là yêu cầu cần thiết, trong trường hợp này chọn phương pháp bản sao theo không gian là hợp lí. Tuy nhiên phần lớn các ứng dụng khẩn cấp lại không theo tiêu chí này. Đối với những ứng dụng này, việc tối thiểu hoá chi phí dành cho chống lỗi được đưa lên hàng đầu.Vì vậy, các kĩ thuật dựa trên thời gian lại thích hợp.

Đối với các ứng dụng phân tán không theo tiêu chí nhiệm vụ, sử dụng mô hình yêu cầu-phục vụ _client-server_, giảm chi phí chống lỗi có thể sử dụng cách tiếp cận theo kiểu sao lưu. Tuy nhiên, để khai thác tiềm năng tính toán theo cluster, các ứng dụng có tính thời gian thường theo mô hình yêu cầu-phục vụ _client-server_ sử dụng một nhóm các tiến trình trao đổi dữ liệu theo mô hình điểm-điểm. Với kiến trúc như vậy, việc giảm chi phí sao lưu là giao thức phục hồi như lấy ảnh tiến trình _checkpointing_ và ghi lại thông điệp _message-logging_. Sau đây chúng tôi sẽ mô tả các giao thức này.

1.3.1. Chống lỗi dựa vào điều phối checkpoint

Là giao thức lưu ảnh tiến trình _checkpointing_, trong suốt quá trình giải phóng lỗi . theo định kì hệ thống tự động lưu trạng thái của mỗi tiến trình vào vùng bộ nhớ tin cậy gọi là một lần lưu ảnh tiến trình checkpoint. Mỗi khi một tiến trình lỗi. một tiến trình mới- gọi là tiến trình phục hồi - được tạo ra. trạng thái của nó được khôi phục lại từ điểm ảnh tiến trình _checkpoint_ gần nhất theo bộ đếm của tiến trình, và sự thực thi của nó được tái tạo từ trạng thái đã lưu.

1.3.2. Chống lỗi dựa vào phương pháp ghi lại thông điệp – Message Logging

Chống lỗi dựa vào phương pháp ghi lại thông điệp – Message Logging là phương pháp ghi lại ảnh CheckPoint và các thông điệp mà nó phải nhận trước khi xảy ra lỗi. Khi lỗi xảy ra trên một tiến trình thì chỉ riêng tiến trình đó khôi phục lại lỗi bằng cách khôi phục theo điểm CheckPoint gần nhất và sử dụng file ghi thông điệp của mình để yêu cầu các tiến trình khác gửi lại thông điệp cho đến thời điểm nó bị lỗi.

CHƯƠNG 2. Chống Lỗi Dựa Vào Điều Phối CheckPoint

2.1. CheckPoint và một số khái niệm liên quan

Checkpoint của tiến trình tại một thời điểm nhất định chứa thông trạng thái của tiến trình (các giá trị của thanh ghi, cờ, ô nhớ) tại thời điểm đó đủ để giúp cho việc khôi phục tiến trình về sau.

Thứ tự các sự kiện trong hệ song song

Khi các tiến trình trao đổi với các tiến trình khác thông qua việc trao đổi thông điệp, các tiến trình này sẽ bị phụ thuộc vào nhau, và thứ tự các sự kiện trở nên khó xác định.

Lamport đã chỉ ra điều này, và ông đã đề xuất một mối quan hệ, gọi là "xảy ra trước" để xác định thứ tự từng phần của của các sự kiện trong hệ thống.[2]

Quan hệ này có tính phi phản xạ (irreflexive), phản đối xứng (antinsymmetric) và bắc cầu (transitive)

Mối quan hệ "xảy ra trước"

Nếu a và b là 2 sự kiện xảy ra ở cùng một tiến trình, và nếu a xảy ra trước b, ký hiệu a (!) b.

Nếu a là một sự kiện gửi thông điệp, và b là một sự kiện nhận thông điệp đối với cùng một thông điệp thì sự kiệ a xảy ra trước sự kiện b, ký hiệu a (!) b

Sự kiện đồng thời

Hai sự kiện a và b được gọi là hai sự kiện đồng thời khi và chỉ khi a không xảy ra trước b, và b không xảy ra trước a.

Checkpoint địa phương

Checkpoint địa phương là trạng thái của tiến trình được ghi lại tại một máy trạm đang xét.

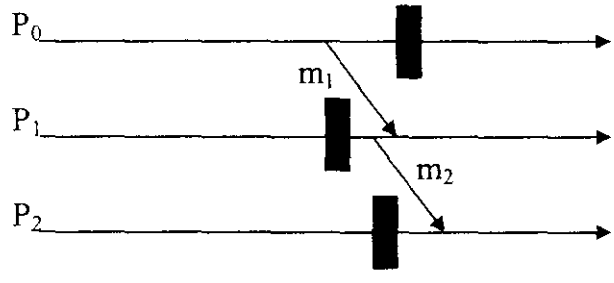
Checkpoint tổng thể

Checkpoint tổng thể là tập hợp n checkpoint địa phương của n tiến trình tương ứng (mỗi checkpoint thuộc một tiến trình).

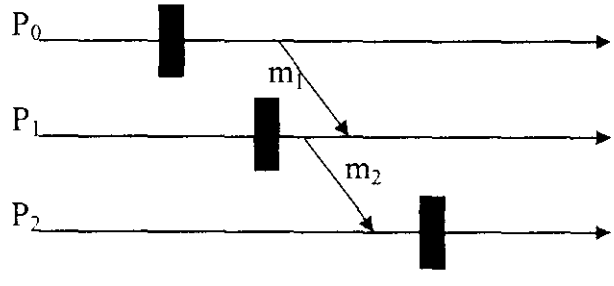
Trạng thái nhất quán tổng thể

Một trạng thái tổng thể được gọi là nhất quán, nếu nó tất cả các sự kiện của nó tạo thành một bộ đồng thời. Một trạng thái nhất quán tổng thể C_c là tập hợp n checkpoint địa phương, của n tiến trình, trong đó mỗi checkpoint địa phương là độc lập với tất cả các checkpoint của các tiến trình khác.

Về trực giác, một trạng thái nhất quán tổng thể là một trạng thái tổng thể của hệ thống, mà khi lỗi xảy ra thì nó có thể khôi phục lại hệ thống một cách đúng đắn. Nói cách khác, đây là một trạng thái mà khi có một tiến trình ghi nhận rằng nó đã nhận được một thông điệp, thì phải có một tiến trình khác, ghi nhận rằng nó đã gửi thông điệp này



Hình 2-1 Trạng thái nhất quán



Hình 2-2 Trạng thái không nhất quán

Trong ví dụ trên trạng thái đầu là trạng thái nhất quán. Vì nếu khi lỗi xảy ra, P_1 chưa nhận được m_1 thì nó có thể yêu cầu P_0 gửi lại cho mình (m_1 khi đó được gọi là tiến trình đang truyền). Trong khi đó, trạng thái thứ hai là trạng thái không nhất quán vì nếu lỗi xảy ra, tiến trình P_2 đã nhận được tin nhắn m_2, nhưng tiến trình P_1 lại không công nhận nó đã gửi tin nhắn này. Thông điệp m_2 khi đó sẽ trở thành thông điệp mồ côi, và trạng thái này được coi là trạng thái không nhất quán.

2.2. Các vấn đề liên quan đến checkpoint

Một số khía cạnh có liên quan đến checkpointing là tần suất của việc checkpoint, nội dung cần checkpoint, và các phương pháp checkpoint.

2.2.1. Tần suất checkpoint

Một thuật toán checkpoint thực hiện trong hệ thống song song thì các tính toán nằm ở mức dưới. Do đó, chi phí của thuật toán dựa chủ yếu vào việc tối thiểu chi phí của quá trình checkpoint. Khoảng thời gian giữa các lần checkpoint phải hợp lý để khi có lỗi xảy, người dùng phải cảm thấy dễ dàng, nhanh chóng trong khôi phục, và không làm mất mát dữ liệu. Số lượng checkpoint được tạo ra phải đảm bảo chi phí cho các thông tin bị mất phải nhỏ, và chi phí cho việc checkpoint phải không lớn lắm. Tần suất checkpoint thường phụ thuộc vào xác suất xảy ra lỗi của hệ thống, và tầm quan trọng của thông tin.

Ví dụ, trong một hệ thống xử lý giao dịch, nơi tất cả các giao dịch đều quan trọng, và không cho phép mất mát thông tin, thì chi phí cho việc checkpoint là khá lớn.

2.2.2. Nội dung của checkpoint

Trạng thái của tiến trình phải được ghi vào bộ nhớ để tiến trình có thể chạy lại khi có lỗi xảy ra. Trạng thái của tiến trình bao gồm mã, phân đoạn ngăn xếp, phân đoạn dữ liệu, cùng với nội dung của thanh ghi, và môi trường của tiến trình tại thời điểm đó. Môi trường này bao gồm thông tin về các tệp, và các con trỏ tệp đang được tiến trình sử dụng. Đối với hệ thống chuyển thông điệp, biến môi trường còn bao gồm các thông điệp đã gửi đi mà chưa nhận được.

2.2.3. Phương pháp checkpoint

Các phương pháp sử dụng checkpoint phụ thuộc vào kiến trúc của hệ thống. Ví dụ, trong các hệ thống đa xử lý (multiprocessor system), các phương pháp phải kết hợp với sự điều phối một cách rõ ràng, không giống như trong hệ thống liên kết xử lý (uniprocessor system). Trong hệ thống truyền thông điệp, các thông điệp phải được quản lý, và nếu cần thiết, phải được ghi vào như là một phần của trạng thái tổng thể của hệ thống. Lý do là trong hệ thống này, các thông điệp quyết định sự phụ thuộc lẫn nhau giữa các bộ xử lý. Mặt khác, hệ thống chia sẻ bộ nhớ giao tiếp thông qua các biến chung, các biến này quyết định sự phụ thuộc giữa các nút. Do đó, tại thời điểm checkpoint, bộ nhớ phải ở trạng thái nhất quán để có được một tập các trạng thái độc lập.

2.2.4. Chi phí của thuật toán checkpoint

Việc tìm hiểu chi phí của thuật toán checkpoint nói chung rất quan trọng. Vì phải biết được các chi phí này, chúng ta mới có thể tìm ra được các phương pháp cải tiến mới. Các phương pháp cải tiến đều nhằm giảm tối đa các chi phí này.

Khi chưa có lỗi xảy ra, tất cả các checkpoint tổng thể đều phải chịu chi phí cho sự điều phối và chi phí cho việc ghi trạng thái. Chúng ta định nghĩa 2 loại chi phí đó như sau:

2.2.4.1. Chi phí điều phối

Trong hệ phân tán, cần có sự điều phối thời điểm checkpoint của các tiến trình để đạt được trạng thái nhất quán tổng thể. Người ta thường sử dụng một số loại thông điệp dạng đặc biệt, hoặc các thông tin phụ đính kèm với các thông điệp thông thường để thực hiện sự điều phối này. Chi phí điều phối chính là chi phí cho các thông điệp, hay các thông tin phụ này.

2.2.4.2. Chi phí cho việc ghi lại trạng thái

Chi phí cho việc ghi trạng thái chính là thời gian dành cho việc ghi trạng thái tổng thể của hệ thống. Chi phí này tỉ lệ với sích thước của trạng thái. Nếu bộ nhớ ổn định không ở trên mỗi nút thì trạng thái của mỗi tiến trình sẽ được chuyển qua

mạng đến một nút ổn định khác. Khi đó, chi phí cho việc ghi trạng thái còn bao gồm cả việc cản trở truyền thông tin trên mạng.

2.2.5. Checkpoint trong hệ phân tán

Hệ thống với nhiều máy tính được gọi là *Hệ thống đa máy tính*. Số lượng máy tính trong hệ thống càng nhiều, thì khả năng xảy ra lỗi càng lớn. Thực tế, có đến 80% các lỗi trong các hệ thống này là các lỗi không cố hữu (tạm thời) và ngắt quãng. Trong các trường hợp này, người ta hay sử dụng phương pháp Checkpoint và khôi phục lỗi ngược (rolback recovery/backward recovery). Điều này dựa trên thực tế là các hệ thống này thường có rất nhiều dòng vận chuyển dữ liệu qua lại, trong khi lại không có đồng hồ đồng bộ của toàn hệ thống.

Khi không có đồng hồ chung của toàn hệ thống, hệ thống sẽ khó xác định được thời điểm để các tiến trình checkpoint cùng một lúc, hay các dòng khó thực hiện công việc một cách đồng bộ. Trạng thái nhất quán tổng thể được tạo nên bởi tập n checkpoint của n tiến trình. Như vậy sẽ phải có thuật toán để xác định trạng thái nhất quán tổng thể của hệ thống, từ các trạng thái tổng thể của hệ thống.

2.2.6. Yêu cầu đối với một thuật toán checkpoint

Một thuật toán sử dụng checkpoint thông thường cần phải đáp ứng các yêu cầu sau:

- Thời gian thực hiện thuật toán checkpoint phải được tối thiểu hoá. Nói cách khác, thời gian tăng lên khi có thêm việc checkpoint phải không đáng kể.
- Việc khôi phục phải nhanh khi có lỗi xảy ra. Nên có sẵn trạng thái nhất quán tổng thể trong bộ nhớ ổn định để tiến hành khôi phục dễ dàng hơn.
- Phải loại bỏ hoàn toàn hiện tượng hiệu ứng domino[1], hay hiệu ứng lan truyền ngược ra khỏi thuật toán.

[1] Là hiệu ứng lan truyền ngược làm cho hệ thống mất hết thông tin đã ghi lại, và phải thực hiện lại từ đầu

- Yêu cầu đối với tài nguyên (bộ nhớ, bộ xử lý) cho việc checkpoint phải được tối thiểu hóa.
- Việc sửa chữa trong giao thức truyền mạng đối với hệ thống truyền thông điệp phải được tối thiểu hóa.

2.3. Phân loại các phương pháp chống lỗi dựa vào checkpoint

Phương pháp chống lỗi dựa vào checkpoint là phương pháp được thực hiện đơn giản, thường được dùng cho các ứng dụng ít trao đổi thông tin với môi trường ngoài. Cơ chế khôi phục của giao thức này dựa trên trạng thái nhất quán tổng thể (dòng khôi phục) của hệ thống. Tuy nhiên, nhược điểm của giao thức này là nó có thể gặp phải lỗi đã mắc sau khi thực hiện phục hồi quay lui. Có ba phương pháp cơ bản sau:

2.3.1. Không có sự điều phối thời điểm checkpoint

Phương pháp này đề cao tính độc lập của mỗi tiến trình. Nó cho phép mỗi tiến trình được tự quyết định thời điểm tạo checkpoint. Nhờ vậy, nó có ưu điểm chính là làm tăng khả năng linh hoạt cho các tiến trình. Mỗi tiến trình có thể tạo checkpoint vào lúc thuận tiện nhất, ví dụ như khi chi phí bộ nhớ của tiến trình khi đó là nhỏ nhất. Điều đó sẽ làm giảm chi phí checkpoint của toàn hệ thống. Tuy nhiên, phương pháp này cũng có một số nhược điểm sau:

- Thứ nhất, phương pháp này dễ gây nên hiệu ứng domino, làm mất khá nhiều quá trình tính toán của hệ thống, vì các checkpoint tạo ra không đảm bảo tạo nên trạng thái nhất quán tổng thể.
- Thứ hai, các checkpoint mà một tiến trình tạo ra có thể không được sử dụng trong quá trình khôi phục (không thuộc trạng thái nhất quán tổng thể nào). Những checkpoint này có thể gây nên hiện tượng quá tải, mặc dù chúng không bao giờ được dùng đến.

- Thứ ba, trong giao thức này, mỗi tiến trình phải lưu rất nhiều checkpoint, do đó, cần phải có một thuật toán gom rác[2] hiệu quả để loại bỏ các checkpoint không cần thiết một cách định kỳ.
- Thứ tư, phương pháp này không phù hợp với các ứng dụng thường xuyên cho dữ liệu ra vì nó phải thường xuyên đồng bộ hệ thống để tính toán đường phục hồi.

Như vậy, có thể thấy, nhược điểm của phương pháp náy quá nhiều. Nếu cài đặt phương pháp này, phải có một thuật toán tìm ra trạng thái nhất quán tổng thể của hệ thống để tiện đường khôi phục, và phải có một thuật toán gom rác để loại bỏ đi các checkpoint không sử dụng đến nữa. Chi phí để khắc phục nhược điểm của nó có thể cao hơn nhiều so với các ưu điểm mà nó có. Vì vậy, phương pháp này không được cài đặt riêng lẻ trong thực tế. Nó thường được sử dụng trong phương pháp kèm với việc ghi lại thông điệp. Như vậy sẽ tránh được hiệu ứng domino xảy ra. Tuy nhiên ta sẽ không đề cập đến các phương pháp này ở đây.

2.3.2. Có sự điều phối checkpoint

Đây là một phương pháp làm cho việc phục hồi trở nên đơn giản. Tư tưởng của phương pháp này là các tiến trình chỉ được phép tạo checkpoint khi có sự điều phối từ bên ngoài. Như vậy, các tiến trình phải thu xếp với nhau để tạo các checkpoint làm thành trạng thái nhất quán tổng thể. Khi đó, nếu lỗi xảy ra, tiến trình chỉ cần trở về điểm checkpoint gần nhất. Do đó, phương pháp này tránh được hiện tượng domino, và mỗi tiến trình chỉ cần lưu trạng thái của một checkpoint. Nhờ vậy, phương pháp này giảm được chi phí về bộ nhớ, và không cần đến thuật toán gom rác.

Tuy nhiên, nhược điểm chính của phương pháp này là lỗi tiềm tàng lớn, liên quan đến việc đưa dữ liệu ra, vì trước khi đưa dữ liệu ra, cần tạo một checkpoint tổng thể, nếu không cẩn thận có thể gây ra lỗi. Hơn nữa, việc điều phối thời điểm

[2] là thuật toán được thực hiện một cách định kỳ để xoá đi các checkpoint không còn sử dụng nữa

checkpoint giữa các tiến trình sao cho đạt chi phí thấp vẫn là một vấn đề khó và vẫn đang tiếp tục được nhiều người quan tâm nghiên cứu.

Chương sau sẽ đi sâu vào phương pháp này, phân tích ưu, nhược của phương pháp, phân tích các biện pháp cách cải tiến, và đưa ra một ứng dụng của phương pháp checkpoint này để tích hợp vào hệ thống MPI.

2.3.3. Checkpoint dựa vào việc giao tiếp

Phương pháp này nhằm kết hợp ưu điểm của hai phương pháp trên. Ngoài các checkpoint độc lập của mình, mỗi tiến trình còn phải tạo các checkpoint cưỡng bức (forced checkpoint) để đảm bảo hệ thống có thể khôi phục lại được. Các checkpoint do mỗi tiến trình độc lập tạo ra gọi là các checkpoint địa phương (local checkpoint), còn các checkpoint mà các tiến trình buộc phải tạo ra gọi là các checkpoint cưỡng bức (forced checkpoint). Communication-induced checkpointing (checkpoint dựa vào giao tiếp) gắn các thông tin về mối liên hệ giữa các giao thức vào mỗi thông điệp ứng dụng (application message). Một tiến trình khi nhận các thông điệp này, sẽ dựa vào thông tin kèm theo để xác định xem liệu nó có phải tạo checkpoint cưỡng bức hay không. Checkpoint cưỡng bức phải được tạo ra trước khi thông điệp ứng dụng được xử lý, điều này có thể dẫn đến lỗi tiềm ẩn, và làm tăng chi phí thực hiện. Khác với phương pháp điều phối việc checkpoint, giao thức này không phải trao đổi bất cứ một thông điệp điều phối nào (coordinated messages) nào.

Phương pháp này được chia làm hai loại nhỏ: thực hiện checkpoint dựa trên mô hình và thực hiện checkpoint dựa trên chỉ số.

Thực hiện checkpoint dựa trên mô hình: xuất phát từ ý tưởng làm cho các checkpoint đang tồn tại không được tạo thành trạng thái không nhất quán, và hệ thống hiện tại luôn nhất quán. Một mô hình được tạo ra để tìm ra các khả năng tồn tại của các mẫu như vậy, theo một số chiến lược heuristic. Các checkpoint thường được tạo ra để tránh các mẫu đó. Tất cả các thông tin cần thiết để thực hiện giao

thức này được mang theo ở đầu mỗi thông điệp ứng dụng. Quyết định để tạo checkpoint được thực hiện một cách cục bộ dựa trên các thông tin có sẵn. Do đó, trong giao thức này, có thể hai tiến trình cùng thấy nếu không tạo checkpoint sẽ gây nên hiện tượng không nhất quán, và cùng tạo checkpoint một cách độc lập, trong khi thực tế chỉ cần tạo một checkpoint. Như vậy, giao thức này có thể tạo ra số checkpoint nhiều hơn số lượng cần thiết, vì ngoài bộ điều phối, không tiến trình nào có được toàn bộ thông tin về trạng thái của toàn bộ hệ thống.

Thực hiện checkoint dựa trên chỉ số: thực hiện bằng cách gắn các chỉ số tăng dần vào các checkpoint để các checkpoint có cùng chỉ số của các tiến trình khác nhau sẽ tạo thành trạng thái nhất quán tổng thể. Các chỉ số được mang theo thông điệp ứng dụng để giúp cho bên nhận biết khi nào chúng phải tạo checkpoint. Ví dụ như trong mô hình của Briatico và đồng sự [10], một tiến trình sẽ tạo checkpoint khi nhận được chỉ số kèm theo thông điệp gửi đến lớn hơn chỉ số hiện tại của mình

2.4. Thực hiện checkpoint

Các nghiên cứu hiện nay đều chỉ ra rằng việc ghi trạng thái của một tiến trình vào bộ nhớ chiếm chi phí nhiều nhất trong tổng các chi phí. Cách đơn giản nhất để ghi trạng thái của tiến trình là ngừng các hoạt động khác lại, ghi không gian địa chỉ của tiến trình vào trong bộ nhớ, và lại thực hiện tiếp các công việc. Cách làm này có thể tăng chi phí cho chương trình, nếu không gian địa chỉ lớn, và nhất là khi bộ nhớ được thực hiện sử dụng đĩa từ. Một số kỹ thuật đã được tìm hiểu để giảm chi phí này.

Checkpoint trong khi thực hiện tiến trình

Các nghiên cứu hiện nay chỉ ra rằng việc thực hiện checkpoint ngay trong lúc thực hiện chương trình giảm đáng kể chi phí của việc ghi lại trạng thái của một tiến trình. Phương pháp thực hiện này dựa trên các khả năng bảo vệ phần cứng trong kiến trúc máy tính hiện đại, nhờ đó các tiến trình vẫn có thể tiếp tục thực hiện trong khi checkpoint được ghi vào bộ nhớ. Không gian địa chỉ được bảo vệ không bị chỉnh sửa ngay khi bắt đầu checkpoint và trang nhớ được ghi vào đĩa trong lúc

tiến trình vẫn tiếp tục được thực hiện. Nếu chương trình cố gắng sửa một trang nhớ, nó sẽ bị vi phạm việc bảo vệ này. Hệ thống checkpoint copy trang nhớ này vào bộ đệm riêng, và trang nhớ ban đầu được bỏ nhãn bảo vệ, chương trình ứng dụng được tiếp tục thực hiện. Kỹ thuật này còn được gọi là *copy-on-write*. Nếu không sử dụng kỹ thuật này, hệ thống phải chịu chi phí thực hiện cao, ngoại trừ khoảng cách giữa hai lần checkpoint lớn.

Thực hiện checkpoint lại khi bị thay đổi

Kỹ thuật thứ hai nhằm giảm chi phí khi thực hiện checkpoint là chỉ thực hiện checkpoint lại đối với các checkpoint bị thay đổi kể từ lần checkpoint gần nhất. Như vậy chỉ những trang nhớ mà không gian địa chỉ của nó đã bị thay đổi từ lần checkpoint trước mới phải ghi lại vào bộ nhớ ổn định. Tập các trang nhớ này được xác định nhờ bit dirty được duy trì bởi phần cứng quản lý bộ nhớ hoặc một dirty-bit mô phỏng bởi phần mềm. Một số kỹ thuật khác cũng được đề xuất dựa trên ý tưởng này. Hệ thống có thể sử dụng tính toán chẵn lẻ để xác định trang nhớ đã bị thay đổi kể từ lần checkpoint trước. Kỹ thuật này tương tự như kỹ thuật tính chẵn lẻ trong hệ thống đĩa RAID. Những trang chẵn lẻ có thể được ghi vào bộ nhớ tạm thời (ở máy trạm), và sẽ giảm được sự truy nhập vào bộ nhớ ổ định. Một kỹ thuật khác đề nghị so sánh trạng thái hiện tại của tiến trình với trạng thái của lần checkpoint khác, và những khác nhau này được lưu vào một checkpoint mới. Tuy nhiên việc so sánh này cũng làm mất khá nhiều chi phí.

Sau đây chúng ta sẽ tìm hiểu chi tiết về các phương pháp này:

2.4.1. Phương pháp dừng thực hiện tiến trình khi thực hiện checkpoint

Đây là một cách tiếp cận đơn giản nhất đối với phương pháp điều phối checkpoint. Trong phương pháp này, khi một tiến trình thực hiện tạo checkpoint, nó sẽ ngừng giao tiếp với tất cả các tiến trình khác. Như vậy, tất cả các tiến trình nào muốn giao tiếp với tiến trình này đều phải đợi cho đến khi nó thực hiện checkpoint xong.

Một checkpoint, như ta đã biết, ghi trạng thái của toàn bộ tiến trình. Khi thực hiện checkpoint, nó sao chép vùng nhớ của tiến trình này vào một vùng nhớ có tính chất ổn định để khôi phục về sau. Vì vậy, khối lượng của checkpoint thường khá lớn (từ hàng chục đến hàng trăm MG). Do đó, các tiến trình thường phải đợi trong thời gian khá dài. Phương pháp này tỏ ra kém hiệu quả, mặc dù cách thực hiện của nó rất đơn giản. Một cách thực hiện phương pháp này được áp dụng trong hầu hết các ứng dụng, đó là sử dụng thuật toán hai pha, thuật toán được trình bày như sau:

Thuật toán hai pha:

Pha 1:

Bộ điều phối gửi yêu cầu đến tất cả các tiến trình yêu cầu checkpoint

Các tiến trình, khi nhận được yêu cầu checkpoint, thực hiện các công việc sau:

- Dừng hoạt động của tiến trình lại (không trao đổi thông điệp với tiến trình khác nữa)
- Tạo ra checkpoint tạm thời (tentitive checkpoint)
- gửi thông điệp xác nhận 1 đến bộ điều phối

Bộ điều phối sau khi nhận được tất cả các thông điệp từ tất cả các tiến trình thì gửi một thông điệp xác nhận 2 đến tất cả các tiến trình

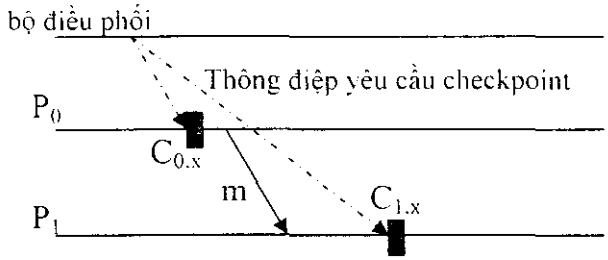
Pha 2:

Các tiến trình khi nhận được thông điệp xác nhận 2 checkpoint từ bộ điều phối thì biến checkpoint tạm thời thành checkpoint cố định (permenant checkpoint)

2.4.2. Phương pháp thực hiện checkpoint trong khi vẫn thực hiện tiến trình

Vấn đề cơ bản trong phương pháp chống lỗi dựa vào checkpoint là phải đảm bảo các tiến trình tránh nhận được các thông điệp ứng dụng (application messages) mà

có thể tạo ra các checkpoint không nhất quán. Nếu sau khi tạo checkpoint thăm dò, các tiến trình tiếp tục hoạt động ngay thì trạng thái không nhất quán rất dễ xảy ra.



Hình 2-3 Hiện tượng không nhất quán

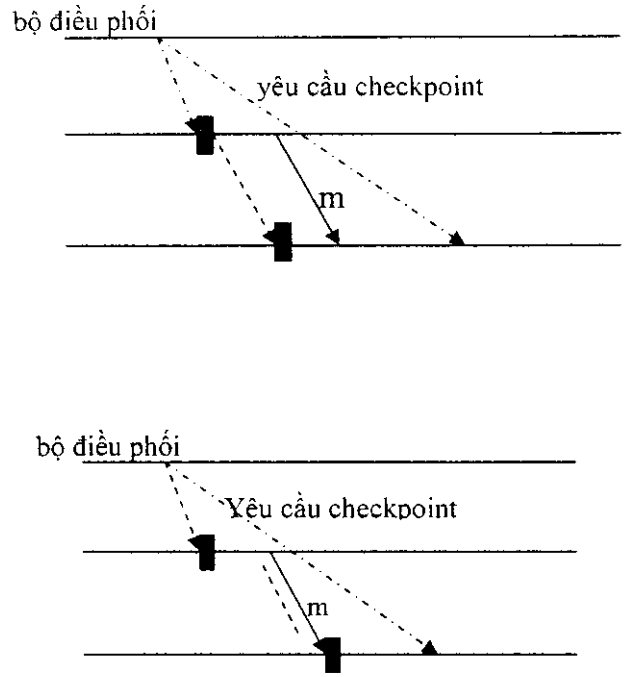
Theo dõi ví dụ hình trên, với m được gửi từ P_0 sau khi đã nhận được tin nhắn yêu cầu từ bộ điều phối. Giả sử m đến được P_1 trước khi P_1 nhận được tin nhắn yêu cầu. Trường hợp này dẫn đến một checkpoint không nhất quán, khi checkpoint $C_{1,x}$ chỉ ra rằng nó nhận tin nhắn từ P_0, trong khi checkpoint $C_{0,x}$ không công nhận nó được gửi bởi P_0.

Để tránh hiện tượng này, người ta thường sử dụng các biện pháp sau:

Một tiến trình sau khi tạo checkpoint tạm thời, nó sẽ gửi thông điệp yêu cầu checkpoint tới tất cả các tiến trình khác trước khi gửi bất cứ thông điệp nào khác. Một tiến trình có thể nhận được yêu cầu tạo checkpoint này trước khi nhận được yêu cầu tạo checkpoint từ bộ điều phối. Khi nhận được yêu cầu tạo checkpoint này, nó sẽ tạo checkpoint tạm thời ngay mà không cần chờ thông điệp yêu cầu từ bộ điều phối.

Các tiến trình sau khi tạo checkpoint tạm thời, muốn gửi tin nhắn cho tiến trình khác, nó sẽ đính kèm thông tin checkpoint vào đầu tin nhắn. Như vậy, khi một tiến trình nhận được tin nhắn này, mà chưa nhận được yêu cầu từ bộ điều phối, nó cũng sẽ hiểu, và tạo checkpoint tạm thời trước khi xử lý tin nhắn.

Việc xử lý các tình huống khi không dừng tiến trình lại (non-bloking) rất phức tạp. Nếu xử lý không khéo sẽ dẫn đến hiệu ứng thác nước[3] (avalanche effect) có thể gây tràn bộ nhớ. Một trong những thuật toán đơn giản và được ưa dùng hiện nay là thuật toán Chandy-Lamport, do hai nhà khoa học Chandy và Lamport đề nghị.



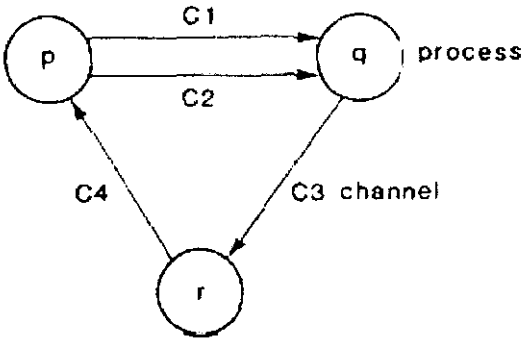
Hình 2-4 Khắc phục hiện tượng không nhất quán

Thuật toán Chandy-Lamport

Mô hình hệ thống:

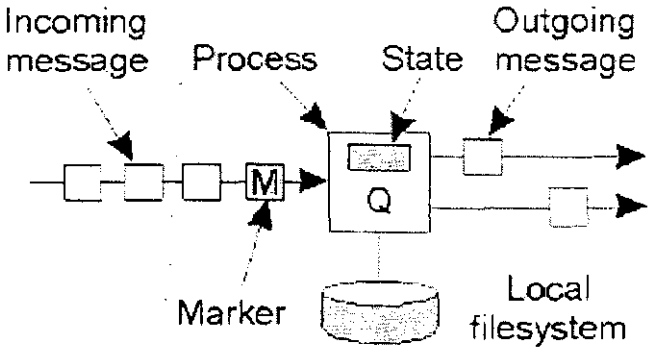
Hệ thống có dạng một đồ thị có hướng: Các đỉnh là các tiến trình, các cạnh là kênh giao tiếp giữa các tiến trình này. Các kênh giao tiếp lưu giữ và chuyển các thông điệp theo cơ chế FIFO.

[3] là trường hợp hệ thống phải tạo quá nhiều checkpoint bởi mối quan hệ phức tạp giữa các tiến trình, gây tràn bộ nhớ



Hình 2-5 Mô hình hệ thống trao đổi thông điệp

Các tiến trình thông báo việc tạo checkpoint cho nhau thông qua các thông điệp đánh dấu (marker message) (Ký hiệu là M)



Hình 2-6 Thực hiện checkpoint trong thuật toán Chandy-Lamport

Thuật toán chung:

Bước 1:

Trong một tiến trình khởi đầu checkpoint (coordinator) diễn ra như sau:

- Chuyển trạng thái thành đã tạo checkpoint
- Ghi trạng thái của mình vào
- Gửi thông điệp đánh dấu theo tất cả các kênh giao tiếp với nó

Bước 2:

Tất cả các tiến trình khác, ngay khi nhận được thông điệp đánh dấu lần đầu tiên sẽ thực hiện các công việc sau (các công việc được thực hiện một cách nguyên tố):

- Chuyển thành đã tạo checkpoint
- Ghi trạng thái của mình vào
- Gửi các thông điệp đánh dấu dọc theo các kênh giao tiếp với nó

Thuật toán kết thúc khi tất cả các tiến trình đều tạo xong checkpoint, và tất cả các tiến trình đều đã nhận được thông điệp đánh dấu qua mỗi kênh đến.

Việc ghi lại trạng thái của kênh giao tiếp:

Khi một tiến trình nhận thông điệp đánh dấu

Giả sử P_i nhận thông điệp đánh dấu thông qua kênh c:

Nếu (*trạng thái của P_i chưa được ghi*)

Thì nó sẽ ghi trạng thái của nó lại

Ghi trạng thái của kênh c là rỗng

tiếp tục ghi lại các tin nhắn đến từ các kênh khác;

Nếu không thì:

P_i ghi trạng thái của c là một tập các thông điệp mà nó nhận từ c từ lúc trạng thái của nó được ghi lại.

Hết nếu.

Khi một tiến trình gửi một thông điệp đánh dấu thông qua kênh c

Sau khi P_i đã ghi lại trạng thái của nó, thì với mỗi kênh ra c:

P_i gửi một thông điệp đánh dấu qua c

(trước khi nó gửi bất cứ một thông điệp nào khác).

Việc ra quyết định checkpoint có thể là tập trung hoặc phân tán.

Trong mô hình ra quyết định tập trung, hệ thống chỉ có một bộ điều phối. Phương pháp này có thuật toán đơn giản nhưng mất chi phí về thời gian. Mô hình ra quyết định phân tán có nhiều bộ điều phối, thuật toán trở nên phức tạp, cài đặt khó hơn. Ta chỉ quan tâm đến mô hình sử dụng một bộ điều phối.

2.4.3. Thực hiện checkpoint dựa vào đồng hồ đồng bộ

Một trong các vấn đề của hệ phân tán là không thể có một đồng hồ chung cho toàn bộ hệ thống. Do vậy, một số nghiên cứu tập trung vào việc tìm cách đồng bộ hóa thời gian của toàn bộ hệ thống.

Trong phương pháp này, mỗi tiến trình có một đồng hồ. Các đồng hồ này đồng bộ một cách tương đối với nhau, và đồng bộ với đồng hồ của hệ thống. Sau mỗi khoảng thời gian nhất định, các tiến trình sẽ tạo checkpoint gần như cùng một lúc mà không cần trao đổi bất kỳ một tin nhắn điều phối nào. Như vậy, chi phí về các tin nhắn điều phối đã được loại bỏ.

Có rất nhiều người đã tập trung nghiên cứu về phương pháp này, bởi nó giảm được rất nhiều chi phí so với phương pháp hai pha.

Z.Tong và nhóm của ông [15] sử dụng phương pháp tạo checkpoint một cách định kỳ, khi đồng hồ của các tiến trình đạt đến thời gian checkpoint. Giao thức này sử dụng tín hiệu ack để phát hiện các thông điệp bị mất. Tiến trình gửi sẽ giữ một bản copy của mỗi thông điệp nó gửi đi cho đến khi nó nhận được tín hiệu ack từ bên nhận. Checkpoint của một tiến trình bao gồm tất cả các tin nhắn mà chưa có tín hiệu ack. Các thông điệp trên đường truyền (in-transist messages) được phát hiện bằng cách thêm chỉ số checkpoint vào tất cả các tin nhắn và ack của chúng. Các tiến trình sẽ ghi các thông điệp này lại ngay khi nhận được chúng.

Cristian và Jahanian [16] cũng sử dụng thời gian để tạo checkpoint. Giao thức này yêu cầu rất chặt chẽ về mặt đồng bộ giữa các tiến trình. Tuy nhiên, nó chỉ cần một lần truy nhập vào bộ nhớ ổn định để ghi lại các *thông điệp đang trong đường*

truyền. Trong giao thức này, mỗi thông điệp được gắn thêm vào số checkpoint hiện tại và thời gian cục bộ của tiến trình. Số checkpoint được sử dụng để nhận ra các thông điệp đang trong đường truyền, còn thời gian cục bộ dùng để phát hiện sự vi phạm là khi thời gian truyền vượt quá thời gian cho phép.

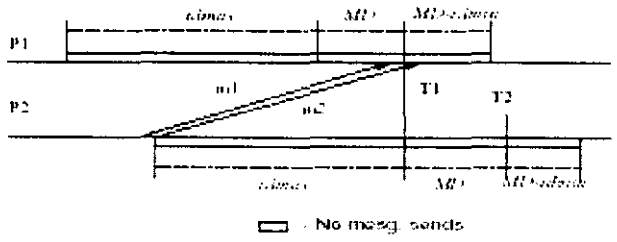
Nuno Neves và W.Kent Fuchs [17] không cần phải có đồng hồ đồng bộ tại các tiến trình. Giao thức này sử dụng một thủ tục khởi tạo để đặt thời gian cho các tiến trình. Các tiến trình dựa vào thời gian này để tạo checkpoint. Thủ tục này được sử dụng khi bắt đầu chạy tiến trình, khi sinh một tiến trình mới, hoặc sau một khoảng thời gian, khi độ lệch thời gian giữa các tiến trình đủ lớn. Phương pháp này tránh các thông điệp trong đường truyền bằng cách ngăn không cho phép các tiến trình gửi tin nhắn trước khi tạo checkpoint một khoảng thời gian là MD+ tdmax

MD: khoảng thời gian chênh lệch lớn nhất giữa các timer của các tiến trình

tdmax: thời gian lớn nhất để gửi một thông điệp.

Ngoài ra, để tránh các checkpoint không nhất quán, giao thức này cũng buộc các tiến trình phải không được gửi tin nhắn sau khi checkpoint trong khoảng thời gian MD-tdmin

Tdmin: thời gian nhỏ nhất để gửi một thông điệp.



Hình 2-7 Sử dụng đồng hồ đồng bộ cho toàn hệ thống

Giao thức này mất thời gian chờ đợi trước và sau khi tạo checkpoint. Tuy nhiên, nó giảm được chi phí của các thông điệp điều phối, và cũng không cần thêm thông tin phụ vào thông điệp.

2.4.4. Tối thiểu hóa số tiến trình tham gia tạo checkpoint

Phương pháp sử dụng checkpoint có điều phối luôn tạo ra được trạng thái nhất quán của hệ thống. Tuy nhiên, chi phí cho việc tạo checkpoint vẫn là khá lớn. Các phương pháp nhằm tối thiểu hoá chi phí này là tối thiểu hoá sự đồng bộ của các thông điệp (non-blocking); và một cách tiếp cận khác là tối thiểu hoá các checkpoint cần phải tạo. Chỉ những tiến trình nào tham gia việc trao đổi thông điệp kể từ lần checkpoint trước mới cần phải checkpoint lại.

Người ta đã tìm cách kết hợp hai phương pháp này lại với nhau. Như đã nói ở trên, chúng ta chỉ cần phải tạo các checkpoint cho các tiến trình có tham gia giao tiếp, kể từ checkpoint gần nhất. Tuy nhiên, Guohong Cao và Mukesh Singhal [7] đã tìm ra được hai vấn đề trong giải thuật này. Thứ nhất, giải thuật của họ có thể dẫn đến trạng thái không nhất quán (inconsistent) của hệ thống. Thứ hai, giải thuật có thể dẫn đến hệ thống tạo checkpoint sai thời điểm, làm cho xuất hiện tin nhắn mồ côi.

Guohong Cao và Mukesh Singhal đã chứng minh được định lý quan trọng: Không tồn tại giải thuật nào nonblocking, và chỉ cần một số tối thiểu tiến trình tạo checkpoint. Từ đó, họ đưa ra thuật toán không dừng tiến trình khi checkpoint, và hạn chế số lượng tiến trình tham gia tạo checkpoint.

Trong giải thuật của đó, họ đề ra một số khái niệm mới, đó là mối quan hệ phụ thuộc z- *z-depends*, và *transitively z-depends*. Hai khái niệm này gần giống với khái niệm *z-path* thường dùng để phát hiện ra trạng thái không nhất quán trong phương pháp không có sự điều phối checkpoint. Ngoài checkpoint tạm thời và checkpoint cố định như trong giao thức hai pha, họ còn sử dụng checkpoint cưỡng bức. Đây là checkpoint được tạo ra khi nó nhận được yêu cầu tạo checkpoint từ một tiến trình khác với tiến trình khiến nó tạo một checkpoint tạm thời phía trước.

Checkpoint cưỡng bức không cần phải ghi vào bộ nhớ ổn định, mà chỉ cần ghi vào bộ nhớ địa phương. Checkpoint cưỡng bức sau đó có thể biến thành checkpoint tạm thời, hoặc bị xoá đi, tuỳ theo các thông điệp nó nhận tiếp theo. Nhờ vậy, phương pháp này có thể tránh được hiệu ứng thác nước, và giảm đáng kể số checkpoint cần phải tạo.

Tuy nhiên, các phương pháp cải tiến nêu trên vẫn còn hạn chế khi đưa vào thực hiện trong thực tiễn, và vẫn chỉ mang tính chất nghiên cứu là chính. Ví dụ như với một hệ thống tương đối lớn, thì việc đồng bộ của các đồng hố của các tiến trình chỉ mang tính tương đối. Sau một thời gian, việc sai khác có thể lớn, làm cho bài toán sai đi. Một số phương pháp khác, thuật toán cài đặt khó và khá phức tạp, nên cũng chưa được áp dụng nhiều trong các hệ thống.

CHƯƠNG 3. Chống Lỗi Dựa Vào Phương Pháp Ghi Lại Thông Điệp – Message Logging

3.1. Một số khái niệm cơ bản

3.1.1. Khái niệm về chống lỗi dựa vào phương pháp ghi lại thông điệp

Giao thức phục hồi log-based còn có tên gọi là: "message logging protocols", được xây dựng trên cơ sở ý tưởng của phục hồi checkpoint-based. Ngoài việc ghi lại trạng thái cho việc phục hồi khi có lỗi thì hệ thống còn xác định tất cả các sự kiện không xác định (non-deterministic event) ở mỗi tiến trình, ghi thêm thông tin về các sự kiện không xác định ví dụ như nhận các thông điệp xảy ra giữa các checkpoint liên tiếp. Sau khi có lỗi, hệ thống sử dụng checkpoint để phục hồi trạng thái không có lỗi gần nhất và phát lại các sự kiện được ghi. Giao thức ghi thông điệp (log-based) ít bị hiệu ứng domino.

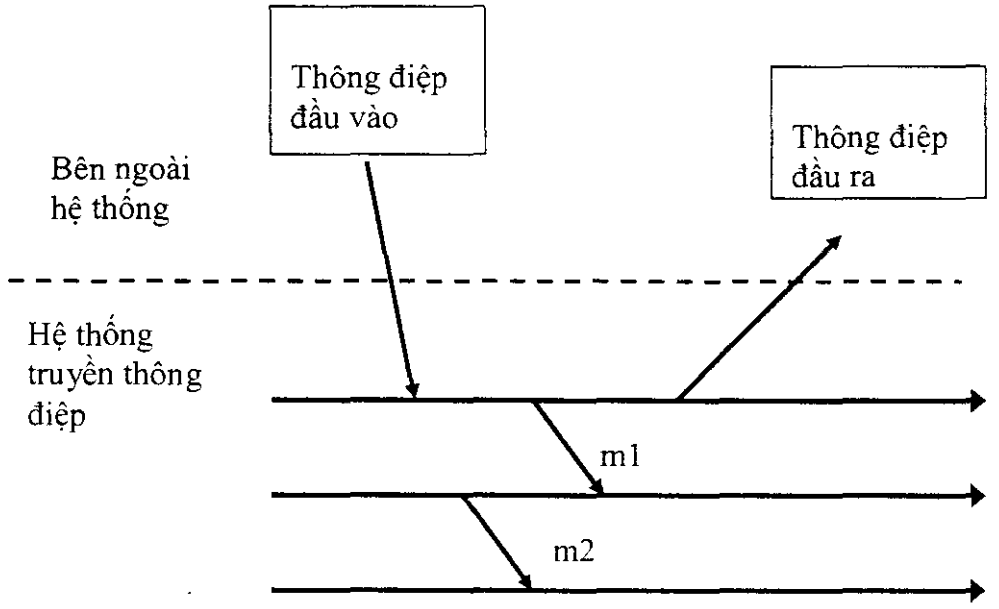
Giao thức phục hồi Log-based rollback-recovery đảm bảo phục hồi tất cả các tiến trình lỗi, hệ thống không chứa tiến trình mồ côi (ví dụ như một tiến trình có trạng thái phụ thuộc sự kiện không xác định không thể mô phỏng lại trong khi phục hồi). Trong một hệ thống sử dụng ghi thông điệp và checkpointing để cung cấp khả năng chịu lỗi, mỗi thông điệp nhận bởi tiến trình sẽ được ghi trong file, và trạng thái của mỗi tiến trình thường xuyên được lưu lại như một *checkpoint*. Mỗi tiến trình được checkpoint riêng biệt, và không có điều phối giữa checkpointing của các tiến trình khác nhau. Thông điệp được ghi và checkpoint được lưu trong bộ chứa ổn định trên đĩa. Một tiến trình lỗi được xây dựng lại, sử dụng các điểm checkpoint trước của tiến trình và bản ghi các thông điệp nhận được. Đầu tiên, trạng thái của tiến trình lỗi sẽ được nạp lại từ điểm checkpoint trên một vài bộ xử lý có thể. Tiến trình cho phép thực hiện, và nhận trình tự các thông điệp được ghi bởi tiến trình sau khi điểm checkpoint này được phát lại đến tiến trình từ log. Thông điệp được phát lại phải được tiến trình nhận theo thứ tự mà nó nhận trước

khi lỗi. Trong quá trình thực hiện. tiến trình sẽ gửi lại thông điệp theo cách nó gửi trước khi lỗi. Các tiến trình phục hồi sau đó thực hiện lại từ trạng thái checkpoint này, theo thứ tự các thông điệp để biết được trình tự thông điệp đến. Khi thực hiện lại. tiến trình sẽ gửi lại thông điệp mà nó đã gửi khi thực hiện không có lỗi.

Giao thức sử dụng cho ghi thông điệp có thể chia làm 2 nhóm, gọi là ghi thông điệp Pessimistic (pessimistic message logging) và ghi thông điệp Optimistic(optimistic message logging). Phương pháp ghi thông điệp Pessimisticlà phương pháp ghi thông điệp đồng bộ, có nghĩa là các thông điệp phải được ghi trước khi được phát ra. Còn phương pháp ghi thông điệp Optimistic thì là phương ghi thông điệp không đồng bộ, các thông điệp cứ phát ra, rồi sẽ ghi vào các thời điểm sau đó.

3.1.2. Một số thuật ngữ cơ bản

3.1.2.1. Mô hình truyền thông điệp



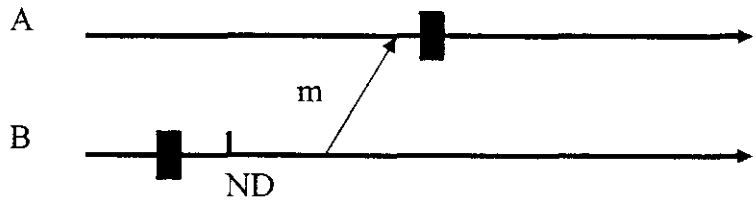
Hình 3-1 Mô hình truyền thông điệp

Hình trên là một ví dụ về hệ truyền thông điệp với 3 tiến trình. Hệ thống nhận thông điệp từ bên ngoài vào, sau đó sẽ truyền thông điệp đến các tiến trình, để các tiến trình thực hiện công việc. Việc thực hiện của các tiến trình bắt đầu bằng một sự kiện không xác định. Sau khi kết thúc công việc, thì hệ thống sẽ đưa một thông điệp ra ngoài.

3.1.2.2. Sự kiện không xác định (non-deterministic event)

Phương pháp truyền thông điệp luôn tồn tại một khái niệm sự kiện không xác định. Đây là sự kiện nhận được một thông điệp từ một tiến trình khác hoặc là một sự kiện bên trong của một tiến trình.

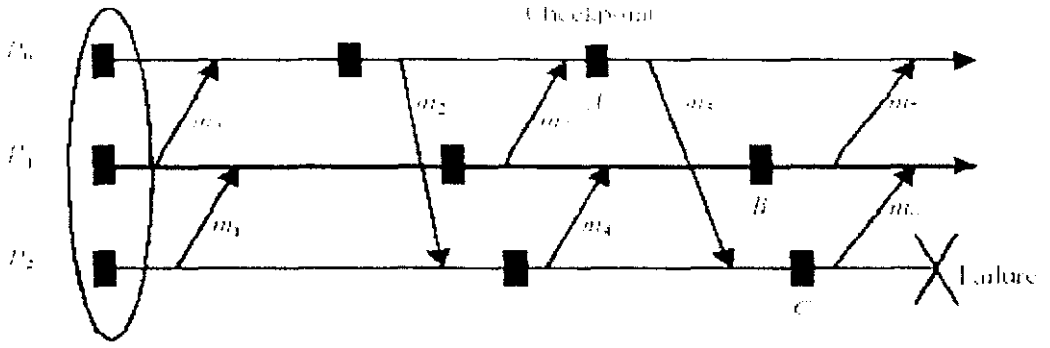
Ta xét một ví dụ đơn giản:



Hình 3-2 Sự kiện không xác định 1

Hình dưới có 2 tiến trình A và B. Các hình khối đen là điểm đã được checkpoint. Ta có : thông điệp m được gửi từ tiến trình B tới tiến trình A. Điểm đánh dấu ND chính là sự kiện không xác định. Có nghĩa là, tiến trình chưa ghi lại sự kiện này vào.

Ta xét một ví dụ khác:

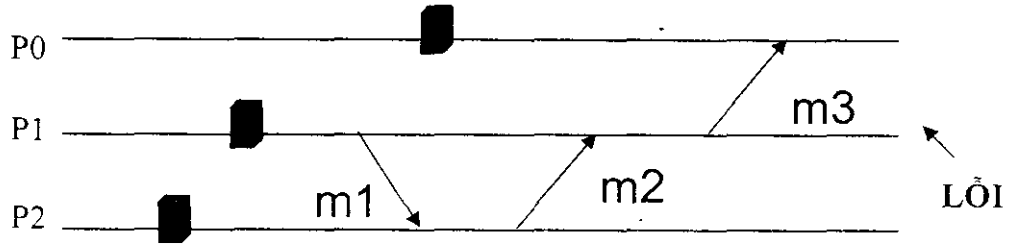


Hình 3-3 Sự kiện không xác định 2

Có 3 tiến trình P0,P1,P2. Tiến trình P_0 thực hiện có thể chia thành bốn khoảng. Khoảng đầu tiên, tiến trình bắt đầu khởi tạo, 3 khoảng còn lại sẽ bắt đầu bằng việc nhận các thông điệp m_0, m_3 và m_7 tương ứng trên hình. Ta thấy rằng thông điệp m2 theo định nghĩa là thông điệp không xác định, tuy nhiên, trong thực tế, thông điệp m2 vẫn có thể xác định được. Thông điệp m2 được gửi đi sẽ được xác định bởi trạng thái đầu tiên của tiến trình P_0 và sự kiện nhận thông điệp m_0. , do đó sự kiện gửi thông điệp m_2 có thể được xác định. Giao thức phục hồi ghi thông điệp coi như các sự kiện không xác định đều có thể xác định được và nội dung của các thông điệp sẽ được ghi vào trong bộ chứa.

3.1.2.3. Tiến trình mồ côi (orphan process)

Xét hình sau :



Hình 3-4 Tiến trình mồ côi

Có 3 tiến trình P0, P1,P2. Các tiến trình đang thực hiện như hình trên.

Ta thấy m3 phụ thuộc m2 vì m3 phải đợi m2 đến tiến trình P1, xử lý một số tháo tác, sau đó, m3 mới được gửi đi.

Khi tiến trình P1 bị lỗi, khi đó, sẽ không sinh ra được thông điệp m3 gửi tới P0. Khi phát hiện tiến trình P1 bị lỗi, ta sẽ lấy từ trong bộ chứa điểm checkpoint và phục hồi từ điểm checkpoint gần nhất, sau đó phát lại các thông điệp đến P1. Nếu như m2 không lưu vào bộ chứa ổn định, thì thông điệp m2 ko được phát lại → trạng thái P0 sẽ không phù hợp vì: thông điệp đã nhận nhưng không bao giờ được gửi → tiến trình đó gọi là tiến trình mồ côi (**orphan process)**

3.1.2.4. Ghi đồng bộ

Ghi đồng bộ (Synchronous logging) ghi lại sự kiện không xác định trước khi sự kiện đó được thực hiện và ảnh hưởng đến hệ thống. Ví dụ: một thông điệp không được phát đi nếu như nó không được ghi lại trước đó. Điều này rất quan trọng trong phương pháp chống lỗi ghi thông điệp pessimistic. Các thông điệp đều được lưu vào bộ chứa, sẵn sàng cho việc phục hồi.

3.1.2.5. Bộ chứa ổn định (stable storage)

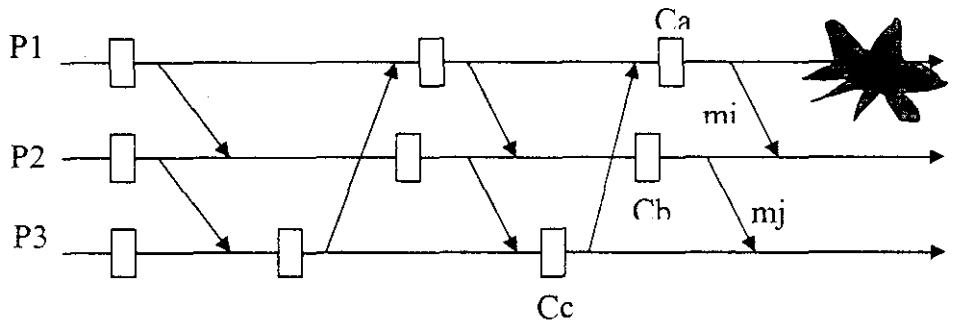
Bộ chứa ổn định trong quá trình thực hiện, sẽ có thời điểm hệ thống gặp lỗi. Khi gặp lỗi, hệ thống sẽ quay lui trở lại, truy cập vào bộ chứa ổn định để lấy ra các điểm đánh dấu trạng thái, các thông tin liên quan khác dùng cho quá trình phục hồi. Chú ý rằng bộ chứa ổn định hay bộ chứa linh động ở đây chỉ là một khái niệm trừu tượng, đó có thể đó là một không gian đĩa của một ổ đĩa cứng. Bộ chứa ổn định phải đảm bảo rằng dữ liệu sẽ còn nguyên vẹn khi hệ thống gặp lỗi. Bộ chứa ổn định có thể lưu dữ liệu, checkpoint, các sự kiện ghi,... Khi thiết kế, bộ chứa ổn định cần phải được để trên một nút ít bị ảnh hưởng bởi lỗi nhất. Điều này hết sức cần thiết, và quan trọng vì bộ chứa ổn định lưu trữ toàn bộ thông tin đã và đang hoạt động, khi hệ thống bị lỗi, mà bộ chứa ổn định cũng có lỗi, thì hệ thống sẽ bị mất mát thông tin trong thời điểm đó.

3.1.2.6. Bộ chứa linh động (volatile memory)

Bộ chứa linh động là bộ nhớ không ổn định, thường để chứa tạm thời các thông tin, các thông tin này sau khi được xử lý sẽ không cần thiết đối với hệ thống, ví dụ về bộ nhớ không ổn định: RAM...Trong hệ thống, có những lỗi nhỏ, không ảnh hưởng lớn tới hệ thống, thì những lỗi này sẽ được phân về lưu trữ trên bộ chứa linh động, để làm giảm bớt gánh nặng cho bộ chứa ổn định

3.1.2.7. Hiệu ứng dây chuyền (domino effect)

Hiệu ứng Domino (Domino effect) sẽ được miêu tả dưới đây



Hình 3-5 Hiệu ứng dây chuyền

Có 3 tiến trình P1,P2,P3. Các điểm checkpoint Ca,Cb,Cc.

Trong quá trình thực thi, ta giả sử P1 lỗi, ta sẽ phục hồi tại điểm checkpoint Ca. Lúc này thông điệp mi đã đượcgửi. P2 nhận thấy nó nhận được thông điệp mi-thông điệp không bao giờ được gửi, sẽ phục hồi về Cb. P3 nhận thấy nó nhận được thông điệp mj-thông điệp không bao giờ được gửi, sẽ phục hồi về Cc. Quá trình này tiếp tục lặp đi lặp lại, không có điểm dừng. Việc thực hiện này gọi là hiệu ứng Domino

3.1.2.8. Thu gom rác (garbage collection)

Việc lưu các checkpoint và các sự kiện sẽ mỗi lúc một nhiều sẽ làm các tài nguyên lưu trữ trở nên quá tải, nhiều thông tin được lưu trữ đã không còn cần thiết cho

việc phục hồi. Do đó, ta cần phải xoá đi các thông tin không cần thiết đó, chỉ giữ lại các thông tin cần thiết

Có một cách loại bỏ các thông tin rác rất phổ biến đó là xác định cách mà thông tin đã được thực hiện, thì khi phục hồi nó sẽ phục hồi theo cách đó để từ đó loại bỏ những thông tin liên quan đến đường phục hồi. Ví dụ: các tiến trình sẽ luôn khởi tạo lại hệ thống ở checkpoint gần nhất ở mỗi tiến trình và vì vậy tất cả các checkpoint trước đó có thể loại bỏ và chỉ lưu lại điểm ảnh trạng thái gần nhất.

Trong thực tế, có nhiều thuật toán để loại bỏ rác và người ta cũng đã sử dụng các thuật toán để loại bỏ rác rất phức tạp nhằm đảm bảo việc đồng nhất các thông tin lưu trong bộ chứa.

3.2. Phân loại các phương pháp chống lỗi ghi lại thông điệp

Chống lỗi ghi lại thông điệp là phương pháp được đánh giá rất cao về hiệu năng của nó. Hiện nay người ta sử dụng 3 phương pháp cơ bản về cách chống lỗi này:

- Chống lỗi pessimistic logging
- Chống lỗi optimistic logging
- Chống lỗi causal logging

3.2.1. Chống lỗi Pessimistic logging

Phần lớn các ứng dụng công nghiệp chọn pessimistic logging vì nó cho phép phục hồi nhanh chóng, đơn giản và được khoanh vùng để phục hồi. Tuy nhiên, thì chi phí cho việc áp dụng chống lỗi pessimistic logging là khá cao.

Về ý tưởng cơ bản, phương pháp pessimistic message logging sẽ thực hiện việc ghi lại checkpoint của từng tiến trình, ngoài ra còn ghi lại các thông điệp mà tiến trình đã nhận vào bộ nhớ ổn định, để phục vụ cho mục đích phục hồi trạng thái của tiến trình và phát lại các thông điệp theo đúng thứ tự mà thông điệp đã phát ra (sender-based message logging).

3.2.1.1. Tổng quan

Điều kiện nhất quán về trường hợp không mồ côi (No-Orphans)

Như chương 2 đã định nghĩa về tiến trình mồ côi:là tiến trình mà trạng thái của nó phụ thuộc vào một sự kiện không xác định. trong khi sự kiện này không thể phục hồi lại được trong qúa trình phục hồi.

Nếu e là một sự kiện không xác định xuất hiện ở tiến trình p, ta định nghĩa:

Phụ thuộc(e): tập các tiến trình ảnh hưởng tới sự kiện không xác định e, đó là tiến trình p, và một số tiến trình phụ thuộc sự kiện e

Ghi(e): tập các tiến trình ghi lại bản sao của sự kiện không xác định e trong bộ nhớ linh động, không ổn định (RAM)

Bộ chứa(e): xác nhận là đúng nếu sự kiện không xác định e được ghi vào bộ chứa ổn định

Bây giờ giả sử tập các tiến trình P bị ngắt. Tiến trình p trong P trở thành tiến trình mồ côi khi bản thân p không bị lỗi, và trạng thái của p phụ thuộc vào việc thực hiện sự kiện không xác định e, mà sự kiên e không thể phục hồi từ bộ chứa ổn định hoặc từ bộ nhớ RAM của tiến trình sống sót.

$$\forall e : \neg stable\ (e) \Rightarrow Depend\ \ (e) \subseteq \log(\ e)$$

Chúng ta gọi thuộc tính này là điều kiện không xảy ra mồ côi. Nó quy định nếu tiến trình sống sót phụ thuộc sự kiện e, thì hoặc sự kiện được ghi vào bộ chứa ổn định, hoặc tiến trình có bản sao việc checkpoint và ghi sự kiện không xác định e.

Giao thức ghi thông điệp Pessimistic ghi thông điệp đồng bộ. Giao thức này đảm bảo các tiến trình bị lỗi có thể phục hồi riêng lẻ mà không ảnh hưởng đến trạng thái của các tiến trình không bị lỗi, ngăn cản tiến trình thực hiện cho đến khi ghi lại thông điệp. Giao thức này được gọi là "pessimistic" vì giả thiết là lỗi có thể xảy ra ở bất cứ thời gian nào, có thể là trước khi việc cần ghi được hoàn thành.

Giao thức pessimistic thực hiện theo thuộc tính:

$$\forall e : \neg stable\ (e) \Rightarrow |Depend\ (e)| = 0$$

Thuộc tính này nói lên rằng nếu như một sự kiện không được ghi vào trong bộ chứa ổn định thì không tiến trình nào phụ thuộc vào nó

Giao thức ghi đồng bộ pessimistic có thể đạt được đảm bảo này bằng việc dừng tiến trình khi nó nhận thông điệp, cho đến khi thông điệp được ghi. Điều này đảm bảo nếu một tiến trình bị lỗi, tất cả các thông điệp được nhận sẽ được phục hồi lại từ điểm checkpoint cuối cùng, không quan tâm thời điểm nó gặp lỗi.

Phục hồi lỗi trong một hệ thống sử dụng ghi thông điệp pessimistic là không phức tạp. Một tiến trình lỗi luôn luôn khởi động lại từ checkpoint mới xảy ra của nó, và tất cả các thông điệp nhận bởi tiến trình đó sau khi checkpoint sẽ được phát lại đến tiến trình từ log, theo cái thứ tự mà họ nhận trước khi có lỗi. Trên cơ sở các thông điệp đó, tiến trình thực hiện lại từ trạng thái được xây dựng lại từ điểm checkpoint đến trạng thái mà nó bị lỗi. Ngoài ra thì số tiến trình bị ngắt sẽ phải bằng số tiến trình sau khi phục hồi. Điều này đảm bảo cho việc phục hồi về trạng thái tương thích của các tiến trình trên toàn bộ hệ thống.

Ví dụ : khi ghi lại hoạt động khi không có lỗi của tiến trình P0;P1;P2 chứa yếu tố cần thiết để để phát lại :

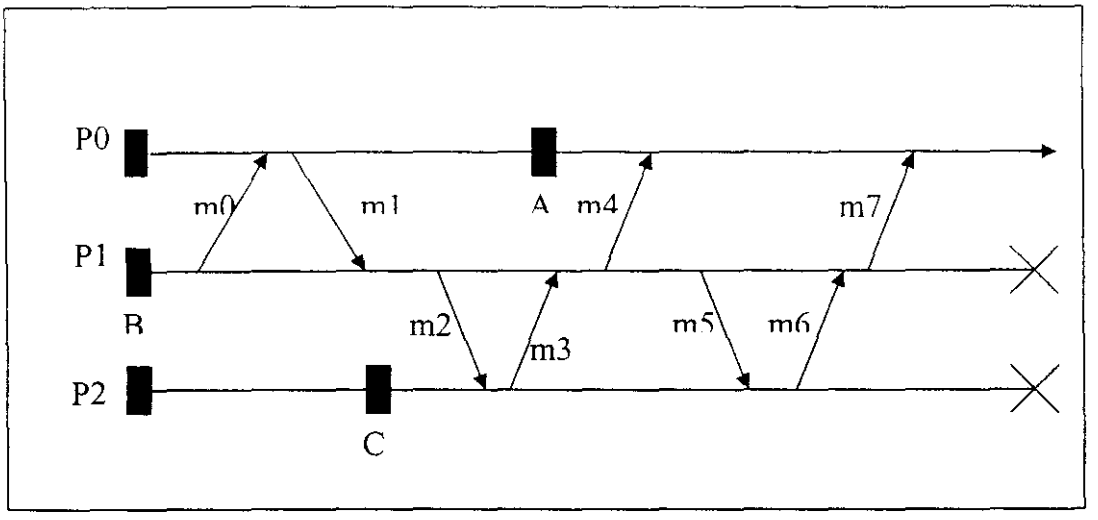
{m0,m4,m7} tương ứng với P0

{m1,m3,m6} tương ứng với P1

{m2,m5} tương ứng với P2.

Giả sử tiến trình P1,P2 bị lỗi, khởi động lại từ checkpoint B và C, xác định bản ghi để phát lại các thông điệp theo trình tự như trước khi bị lỗi. Điều này đảm bảo P1,P2 sẽ lặp lại chính xác hoạt động trước khi lỗi, và gửi lại các thông điệp. Kể từ

đó, phục hồi được hoàn thành, cả 2 tiến trình sẽ tương ứng với trạng thái của P0 gồm cả việc nhận thông điệp m7 từ P1.



Hình 3-6 Chống lỗi ghi thông điệp Pessimistic

Ví dụ: P2 bị lỗi, sẽ quay trở lại điểm checkpoint C, sau đó phát lại thông điệp m2 và m5 lấy ra từ file log.

3.2.1.2. Ưu điểm

Tiến trình khởi động lại từ điểm checkpoint gần nhất, vì vậy sẽ giới hạn quy mô của việc thực hiện phát lại thông điệp, tránh hao tổn các chi phí. Tần suất xuất hiện của checkpoint có thể xác định bằng sự cân bằng giữa yêu cầu thời gian chạy với yêu cầu bảo vệ của công việc đang xảy ra. Nếu như yêu cầu về tính an toàn trong hệ thống là cao: thì ta có thể lưu checkpoint trong những khoảng rất ngắn, ví dụ như sau 3', sau 5', hay sau 10' thì sẽ checkpoint 1 lần. Điều này sẽ làm giảm việc mất mát thông tin khi có lỗi, tuy nhiên hệ thống sẽ chạy chậm đi. Nếu như yêu cầu hệ thống cần chạy nhanh, cần yếu tố thời gian, và do hệ thống đã chạy rất ổn định, thì ta có thể thời gian checkpoint kéo dai hơn, lâu hơn.

Thông tin phục hồi không cần thiết sẽ dễ dàng bị loại bỏ. Các điểm checkpoint cũ hơn thì sẽ không cần thiết cho việc phục hồi. do đó có thể loại bỏ ngay khi tiến trình có checkpoint mới hơn. Điều này tuỳ theo yêu cầu của người dùng đưa ra. Có

thể lưu giữ vài điểm checkpoint, có thể lưu giữ chỉ 1 điểm checkpoint tuỳ thuộc yêu cầu

Công việc phục hồi đơn giản vì có thể phục hồi lại hệ thống sau khi có lỗi mà không ảnh hưởng trạng thái của bất cứ tiến trình nào không bị lỗi. Các tiến trình sẽ vẫn tiếp tục hoạt động, mà không lo sẽ xảy ra tiến trình mồ côi vì một tiến trình luôn phục hồi tới trạng thái mà nó vừa tương tác với các tiến trình khác hoặc với bên ngoài

3.2.1.3. Nhược điểm

Nhược điểm chính của giao thức ghi thông điệp Pessimistic là làm giảm hiệu năng vì sự đồng bộ hoá trong giao thức ghi thông điệp. Giao thức ghi pessimistic message [Powell83,Borg83, Borg89] đã thử giảm tổng phí bằng cách sử dụng phần cứng với những mục đích đặc biệt (special-purpose) để giúp cho việc ghi có hiệu quả hơn. Để hiệu năng hoạt động cao thì có thể sử dụng phần cứng đặc biệt. Ví dụ như bộ nhớ bán dẫn được sử dụng để thực hiện bộ chứa ổn định, thiết bị đĩa từ...Ngoài ra có một số phương pháp không dựa vào phần cứng, ví dụ như Sender-Based Message Logging (SBML), đây là phương pháp sử dụng việc ghi thông điệp ở phía người gửi, nhằm giảm chi phí ghi đồng bộ cho bộ chứa ổn định, và để phía người gửi chịu một phần chi phí, giúp cho bộ chứa ổn định không cùng 1 lúc nhận ghi các thông tin của các tiến trình, ít xảy ra trường hợp quá tải, hoặc nghẽn.

3.2.1.4. Đánh giá hiệu năng

Phần lớn các lý thuyết về ghi thông điệp và checkpointing trong nhiều năm trước đều dựa trên cơ sở phương pháp ghi thông điệp *Optimistic*, quan tâm về vấn đề chi phí khi chống lỗi hơn là khả năng và hiệu quả chống lỗi. Hiện nay, phần lớn các hệ thống đã nghiên cứu sử dụng phương pháp *Pessimistic* vì mục đích chính của ghi thông điệp và checkpoint là hoàn thành việc phục hồi một cách nhanh chóng và cục bộ. Chi phí giải phóng lỗi của Pessimistic có thể được giảm xuống hợp lý tuỳ theo yêu cầu của ứng dụng cụ thể.

Có 3 yếu tố chính cần xem xét để cân bằng trong khi thiết kế phương pháp ghi thông điệp:

- Chi phí chống lỗi
- Số tiến trình còn sống phải rollback
- Thời gian phục hồi

Xét phương pháp Optimistic, phương pháp này nhìn vấn đề lỗi ở góc độ lỗi ít khi xảy ra, do đó việc thực hiện chống lỗi quan trọng hơn nhiều so với việc thực hiện một phục hồi tốt. Ngược lại, phương pháp Pessimistic luôn luôn chuẩn bị sẵn sàng chống lỗi. Do đó, sẽ có một chi phí chống lỗi cao hơn để phục hồi nhanh chóng khi có lỗi xảy ra. Vì vậy khi xét về mặt hiệu năng thực hiện và sự ổn định, tuỳ thuộc vào hệ thống chúng ta thực hiện, mà chúng ta sẽ có những chọn lựa phù hợp. Về mặt hiệu năng, ta chỉ có thể can thiệp, thay đổi được 2 yếu tố: chi phí chống lỗi và thời gian phục hồi. Tuy nhiên, phương pháp Pessimistic là phương pháp tối ưu hơn, tuy mất chi phi cho việc ghi lại các thông tin, nhưng đảm bảo được sự an toàn dữ liệu, đảm bảo hệ thống luôn luôn tồn tại thông suốt.

Hiện tại phương pháp ghi thông điệp Pessimistic được đánh giá là một phương pháp chạy ổn định và có hiệu năng cao so với các phương pháp ghi thông điệp hiện có.

3.2.2. Phương pháp Optimistic Logging

3.2.2.1. Tổng quan

Giao thức ghi thông điệp Optimistic Logging là giao thức ghi thông điệp không đồng bộ. Khác với Pessimistic, giao thức này không đảm bảo chỉ duy nhất một tiến trình phải phục hồi lại lỗi mà còn có thể kéo theo các tiến trình khác cũng phải phục hồi lại.

Trong phương pháp này, khi một tiến trình nhận được một thông điệp, nó ghi thông điệp đó vào bộ nhớ không ổn định (volitile storage) trước (khác với

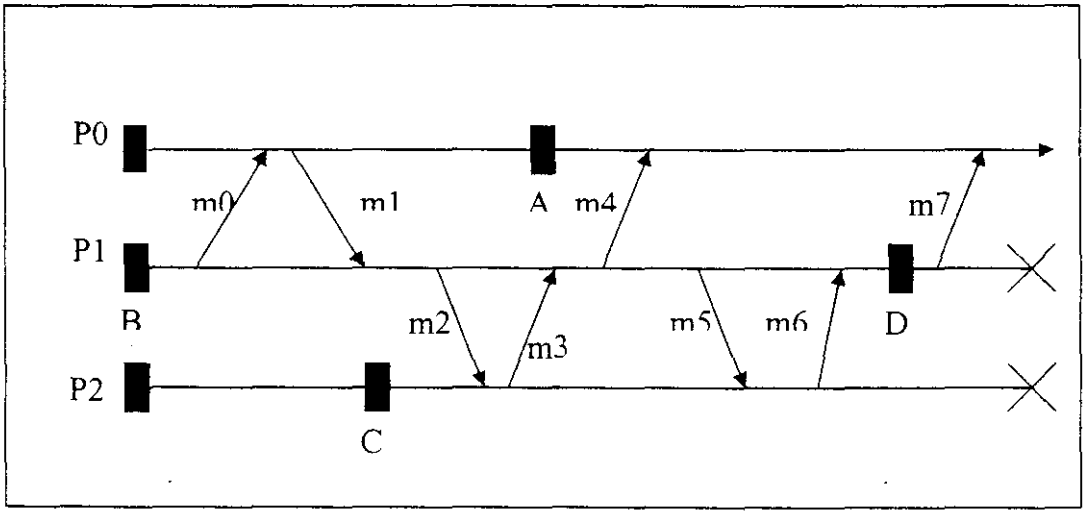
optimisstic là ghi luôn vào bộ nhớ ổn định). Sau đó đợi khi nào tiến trình rỗi. nó sẽ chuyển sang bộ nhớ ổn định.

Optimistic Logging không đảm bảo trạng thái "không mồ côi", vì vậy cho phép tạo những tiến trình mồ côi tạm thời.Quá trình khôi phục phải quay lui các tiến trình mồ côi cho đến khi trạng thái của nó và bất cứ thông điệp nào cũng được xác định.

Quá trình khôi phục lỗi của Optimistic Logging cũng giống như Pessimistic Logging nếu tất cả các tiến trình được ghi trong bộ nhớ không ổn định được ghi lại trong bộ nhớ ổn định (stable storage). Lúc đó, chỉ một tiến trình khôi phục dữ liệu lấy trong bộ nhớ ổn định mà thôi.

Nhưng nếu khi một tiến trình lỗi mà sự kiện trong bộ nhớ không ổn định bị mất (chưa kịp lưu lại) thì trạng thái trước khi xảy ra lỗi sẽ không được phục hồi. Mặt khác, nếu một tiến trình lỗi lại gửi một thông điệp trong quá trình này thì thông điệp đó sẽ trở thành mồ côi. Quá trình khôi phục sẽ quay ngược lại các điểm Checkpoint của các tiến trình khác để tránh trường hợp mồ côi này.

Sau đây là một ví dụ :



Hình 3-7 Optimistic logging

Giả sử khi lỗi xảy ra, thông điệp m5 chưa được ghi vào bộ nhớ ổn định, tiến trình P1 trở thành mồ côi. Và thông điệp m6 trở thành thông điệp mồ côi. Do đó đáng nhẽ P1 sẽ khôi phục tại điểm Checkpoint gần nhất là D thì nó lại khôi phục tại điểm B để tránh trường hợp mồ côi.

3.2.2.2. Ưu điểm

Phương pháp này có một số ưu điểm sau:

- Hạn chế thời gian sao chép thông điệp vào bộ nhớ ổn định vì thông điệp chỉ được ghi vào bộ nhớ tạm thời và khi nào tiến trình rỗi, nó sẽ ghi vào bộ nhớ ổn định. Đối với một chương trình không có nhiều thông điệp thì đây là sự lựa chọn đúng đắn, đảm bảo thời gian thực hiện chương trình
- Hiệu năng hệ thống cao.

3.2.2.3. Nhược điểm

Phương pháp này mặc dù vậy vẫn còn nhiều nhược điểm:

- Tốn nhiều không gian lưu trữ. Do bộ nhớ tạm thời có thể mất bất cứ lúc nào nên các trạng thái của các thời điểm Checkpoint của các tiến trình cũng đều phải được ghi lại. Phương pháp này cần có hệ thống gom rác hiệu quả.
- Quá trình khôi phục hệ thống khá phức tạp, vì hệ thống không đảm bảo "không mồ côi". Trong trường hợp xấu không chỉ một tiến trình phải khôi phục lại mà các tiến trình khác cũng phải khôi phục lại.
- Optimistic chỉ áp dụng cho trường hợp lỗi xảy ra trên một tiến trình. Khi lỗi xảy ra trên nhiều tiến trình, việc khôi phục tỏ ra rất phức tạp.

3.2.3. Phương pháp Causal Logging

3.2.3.1. Tổng quan

Phương pháp Causal Logging kết hợp ưu thế của cả 2 phương pháp trên. Nó vừa tận dụng việc ghi thông điệp vào bộ nhớ tạm thời mà không ghi đồng bộ vào bộ

nhớ ổn định. Vừa kết hợp với việc các tiến trình khác nhau thì ghi một cách độc lập, kết hợp với việc đưa thông tin ra ngoài.

Causal Logging hạn chế số lượng phục hồi, hệ thống luôn phục hồi tại điểm Checkpoint gần nhất. Nó giảm dung lượng lưu trữ và khối lượng rủi ro.

Phương pháp này đảm bảo hệ thống luôn ở trạng thái "no orphan" bằng cách đảm bảo chắc chắn rằng các sự kiện không xác định luôn ở trong bộ nhớ ổn định hoặc đã nằm trong chính tiến trình đó.

3.2.3.2. Ưu điểm

- Phục hồi lỗi đơn giản, do chỉ cần phục hồi tại điểm Checkpoint gần nhất.
- Dung lượng bộ nhớ ổn định cần lưu trữ là thấp
- Hiệu năng cao

3.2.3.3. Nhược điểm

Phương pháp này rất khó cài đặt

CHƯƠNG 4. Một Số Môi Trường Truyền Thông Điệp Chống Lỗi

4.1. Môi trường LAM/MPI kết hợp BLCR

4.1.1. Giới thiệu phần mềm chống lỗi BKCR

Phần mềm BLCR là kết quả nghiên cứu của trường Đại học Berkerley, Hoa Kỳ với chức năng chính là lấy checkpoint và restart các tiến trình bất kỳ trong hệ điều hành Linux. Các tiến trình này có thể là tiến trình đơn, là một tập các tiến trình của các ứng dụng đa luồng hoặc song song. Khả năng trên cho phép tạo ra môi trường truyền thông điệp chống lỗi bằng phương pháp checkpoint khi sử dụng BLCR với LAM/MPI.

BLCR cung cấp cho người sử dụng các câu lệnh để có thể ghi lại checkpoint của một chương trình song song bất kỳ cũng như khởi động lại chương trình song song từ những checkpoint đó.

BLCR bao gồm hai thành phần kernel, các thư viện mức người dùng, và các lệnh thực thi. BLCR có đã được kiểm tra và hoạt động hiệu quả trên rất nhiều loại nhân linux (Linux kernel, Vanilla Linux kernels, Experimental Linux 2.6) và các nhà phân phối nhân linux khác nhau (RedHat, SuSE, CentOS, ...). Tuy nhiên, BLCR sử dụng mã máy để lưu giữ trạng thái của chương trình, do đó các thành phần kernel của BLCR không hoạt động với tất cả các kiến trúc máy tính. Hiện tại chỉ hoạt động được với các hệ thống X86.

Các yêu cầu khi cài đặt:

- Mã nguồn của nhân Linux để biên dịch cùng với BLCR
- File `linux/version.h` (có trong mã nguồn của nhân Linux)
- File `System.map` hoặc `vmlinux`

- Mã nguồn BLCR

Để BLCR hoạt động được với LAM/MPI trong việc checkpoint/restart các chương trình theo chuẩn MPI. có một số chú ý khi cài đặt LAM/MPI

- LAM/MPI phải được cài đặt sau khi cài đặt thành công BLCR.
- BLCR chỉ có thể được sử dụng khi RPI (Request Progression Interface - một nhóm giao diện SSI được LAM/MPI sử dụng trong truyền thông điểm điểm) đang chạy là crtcp. Do vậy khi cấu hình cài đặt LAM/MPI cần có tham số `--with-rpi=crtcp` để cho RPI mặc định là crtcp.
- Khi cấu hình LAM/MPI cũng phải có tham số cho biết thư mục cài đặt BLCR để LAM/MPI tìm được các file tiêu đề và thư viện để biên dịch LAM : `--with-blcr=PAHT_TO_INSTALL_BLCR` (như trong trường hợp mặc định là `/usr/local`).

Nạp các thành phần Kernel vào hệ thống

Trước khi có thể checkpoint/restart chương trình, các thành phần modun kernel cần phải được nạp vào trong nhân của hệ điều hành.Các thành phần kernel này được đặt ở trong thư mục con `lib/blcr` của thư mục cài đặt BLCR. Ví dụ, nếu thư mục cài đặt BLCR là `/usr/local` và phiên bản nhân của hệ điều hành là `2.4.22-1.2115.nptl` thì thư mục của các thành phần kernel là `/usr/local/lib/blcr/2.4.22-1.2115.nptl/`. Sẽ có hai thành phần kernel trong thư mục này, và chúng phải được nạp vào trong hệ thống một cách có thứ tự :

```
# /sbin/insmod /usr/local/lib/blcr/2.4.22-1.2115.nptl/vmadump_blcr.o
# /sbin/insmod /usr/local/lib/blcr/2.4.22-1.2115.nptl/blcr.o
```

Cấu hình các biến môi trường

Các biến môi trường `$PATH`, `$LD_LIBRARY_PATH` , và `$MANPATH` cần được thay đổi để người dùng sử dụng các chức năng của BLCR cung cấp. Người dùng có thể thay

đổi trực tiếp từ dòng lệnh hoặc thay đổi trong file /etc/profile , /etc/cshrc files.For Bourne-style shells:

```
$ PATH=$PATH:PREFIX/bin
$ MANPATH=$MANPATH:PREFIX/man
$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:PREFIX/lib
$ export PATH MANPATH LD_LIBRARY_PATH
```

For csh-style shells:

```
% setenv PATH ${PATH}:PREFIX/bin
% setenv MANPATH ${MANPATH}:PREFIX/man
% setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:PREFIX/lib
```

Ở đây, PREFIX là đường dẫn đến thư mục cài đặt BLCR.

4.1.2. Checkpointing sử dụng lệnh của BLCR

Trước khi có thể checkpoint, cần phải đảm bảo rằng BLCR đã được nạp vào hệ thống. Lệnh để kiểm tra xem các module kernel của BLCR đã được nạp vào hệ thống hay chưa ở đây là 'lsmod' :

```
% /sbin/lsmod
Module                  Size  Used by    Not tainted
blcr                   46936   0
vmadump_blcr           16544   0 [blcr]
iptable_filter          2412   0 (autoclean) (unused)
ip_tables              15864   1 [iptable_filter]
```

Mặt khác, người quản trị hệ thống cũng cần phải đảm bảo rằng các biến môi trường đã được thiết lập. Biến PATH cần chỉ đến thư mục chứa các lệnh của BLCR, biếnLD_LIBRARY_PATH cần có thêm thư mục chứa thư viện 'libcr.so', và biến MANPATH cần chứa thư mục chứa manual của BLCR.

Các loại chương trình có thể checkpoint/restart

- Chương trình chỉ có một luồng đơn

- Các chương trình đa luồng sử dụng các luồng theo chuẩn Linux.
- Các chương trình sử dụng các tín hiệu.

Một loại chương trình không được BLCR hỗ trợ để checkpoint/restart là các chương trình có mở socket TCP/UDP vào thời điểm checkpoint. Các chương trình song song theo chuẩn MPI lại sử dụng giao tiếp giữa các tiến trình thông qua socket. BLCR không hỗ trợ để đóng các socket khi checkpoint và mở lại các socket khi restart. Tuy nhiên LAM/MPI lại làm việc với BLCR theo cách này khi cài đặt LAM/MPI theo hướng dẫn ở trên.

Làm cho chương trình có thể checkpoint được

Để có thể checkpoint được bằng BLCR, chương trình phải bao gồm mã thư viện mà BLCR cung cấp. Có một vài cách sau.

1. Chạy chương trình bằng lệnh '`cr_run`' :

   ```
   % cr_run your_executable [arguments ]
   ```

   '`cr_run`' sẽ nạp thư viện BLCR vào trong chương trình lúc bắt đầu chạy, do đó không cần phải thay đổi chương trình ban đầu.

2. Liên kết chương trình với thư viện '`libcr`'.

   ```
   % gcc -o hello hello.c -LPREFIX/lib -lcr
   ```

3. Liên kết chương trình với thư viện mà có sử dụng BLCR như là thư viện MPI nếu nó được cài đặt để làm việc với BLCR.

Checkpoint chương trình

Với PID là định danh của tiến trình đang chạy, để checkpoint chương trình, đơn giản thực hiện lệnh:

```
% cr_checkpoint PID
```

'`cr_checkpoint`' sẽ lưu lại một điểm checkpoint, và sau đó tiếp tục cho chương trình chạy. file chứa điểm checkpoint được gọi là *context files* và chúng được lưu với tên theo dạng '`context.PID`', với PID là định danh của tiến trình đã được

checkpoint. File này được lưu trong thư mục làm việc mà lệnh 'cr_checkpoint' được chạy.

Restart chương trình

Có một số điều kiện để có thể restart lại chương trình từ context file mà checkpoint tạo ra:

- Hệ thống không có tiến trình nào đang chạy mà có định danh trùng với định danh của tiến trình trong context file.
- File thực thi vẫn còn và không bị thay đổi.
- Tất cả các thư viện chia sẻ mà chương trình sử dụng vẫn còn và không bị thay đổi.

Chương trình có thể được restart trên một máy khác nhưng tất cả các điều kiện trên đều phải được thoả mãn.

Lệnh để thực hiện restart đơn giản như sau:

```
% cr_restart context.PID
```

4.1.3. Quy trình checkpoint/restart của LAM/MPI

Gói phần mềm LAM/MPI cũng cung cấp khả năng tạo checkpoint. Khả năng này được thiết kế độc lập, không làm thay đổi chức năng cũng như các biến đầu vào, đầu ra của các hàm có sẵn theo chuẩn MPI. Mặt khác, hệ thống tạo checkpoint của LAM/MPI cũng cho phép có thể sử dụng kết hợp với các bộ tạo checkpoint khác, có thể hoạt động tốt trên nhiều platform.

Quá trình tạo checkpoint đối với một ứng dụng viết theo chuẩn MPI được bắt đầu bằng việc gửi yêu cầu đến hàm mpirun, mpirun sẽ quảng bá yêu cầu đến các tiến trình của ứng dụng. LAM/MPI sử dụng phương pháp checkpoint có điều phối. Sau khi nhận được yêu cầu checkpoint, các tiến trình sẽ trao đổi với nhau thông qua kênh thông tin nội bộ (TCP communication sub system) để xác định một trạng thái

nhất quán toàn cục. Sau đó là quá trình xóa sạch các kênh truyền thông để sẵn sàng cho việc lấy checkpoint.

Khi có yêu cầu restart, tất cả các tiến trình sẽ được khởi động lại dựa trên các checkpoint đã lưu với những kênh truyền thông đã được thiết lập lại.

Quá trình lấy checkpoint của LAM được thực hiện qua các bước sau đây, trong đó mpirun đóng vai trò là người điều phối giữa các tiến trình song song của ứng dụng:

1. **mpirun:** nhận yêu cầu checkpoint từ người sử dụng hay hoặc từ hệ thống lập lịch và phân tải.
2. **mpirun:** quảng bá yêu cầu checkpoint cho các process
3. **mpirun:** kiểm tra xem trạng thái toàn cục đã sẵn sàng cho việc lấy checkpoint chưa
4. **Với mỗi tiến trình:** giao tiếp với các tiến trình khác để thiết lập trạng thái nhất quán toàn cục.
5. **Với mỗi tiến trình:** ixác nhận đã đạt đến trạng thái nhất quán toàn cục
6. **Bộ tạo checkpoint (checkpointer):** lưu toàn bộ checkpoint của các tiến trình vào bộ nhớ ngoài.
7. **Với mỗi tiến trình:** tiếp tục hoạt động bình thường sau khi lấy checkpoint

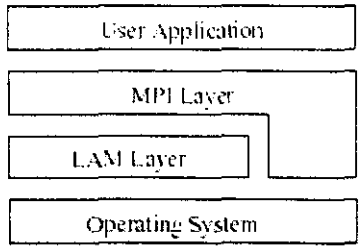
Quá trình restart sẽ được tiến hành như sau:

1. **mpirun:** khởi động lại tất cả các tiến trình từ những checkpoint
2. **Với mỗi tiến trình**:gửi thông tin trạng thái đến mpirun
3. **mpirun:** cập nhất lại thông tin tổng thể về toàn bộ ứng dụng song song và gửi đến các tiến trình.
4. **Với mỗi tiến trình:** nhận thông tin tổng thể từ mpirun

5. **Với mỗi tiến trình:** thiết lập lại các kênh truyền thông với tất cả các tiến trình khác.

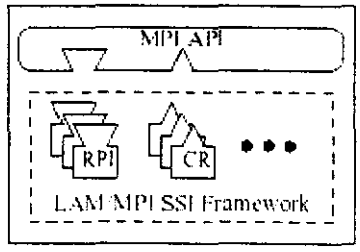
6. **Với mỗi tiến trình:** Khôi phục lại hoạt động, tiếp tục thực hiện ứng dụng song song.

4.1.4. Môi trường tính toán song song chống lỗi LAM/MPI - BLCR



Hình 4-1 Kiến trúc phân tầng LAM/MPI

LAM/MPI bao gồm hai tầng chính, tầng LAM và tầng MPI. Tầng LAM cung cấp khung chương trình và môi trường song song, làm nền cho tầng MPI. Tầng LAM cung cấp những dịch vụ như: truyền thông điệp, điều khiển tiến trình, truy cập file từ xa, định hướng vào/ra. Tầng MPI cung cấp giao diện truyền thông điệp, ... LAM cung cấp cho hệ thống môi trường song song (Run-time environment – RTE) với các tiến trình ngầm (daemon).



Hình 4-2 Cấu hình BLCR hoạt động dưới vai trò dịch vụ CR của LAM/MPI

Các dịch vụ của LAM/MPI được thiết kế sao cho có thể tùy chọn trong quá trình hoạt động của toàn bộ môi trường chống lỗi. Như đã trình bày trong phần quy trình checkpoint/restart của LAM/MPI, việc lấy checkpoint sẽ là một dịch vụ được

thực hiện bởi một bộ tạo checkpoint bất kỳ. BLCR có thể được cấu hình như một tùy chọn của dịch vụ checkpoint/restart (CR) của LAM/MPI.

Đối với các các chương trình MPI, để có thể checkpoint/restart với BLCR thì thư viện MPI phải được biên dịch lại để sử dụng BLCR. Hiện tại chỉ có LAM/MPI cho phép như vậy. Sau khi cài đặt đúng đắn, thư viện MPI sẽ được liên kết với BLCR để các chương trình MPI có thể checkpoint/restart được. Các công việc của người sử dụng là:

Chạy chương trình MPI bằng lệnh 'mpirun' :

```
% mpirun C hello_mpi
```

Checkpoint chương trình MPI:

```
% cr_checkpoint PID
```

Với 'PID' là định danh của tiến trình 'mpirun' chứ không phải là định danh của tiến trình của chương trình MPI đang chạy. Khi mà 'mpirun' được checkpoint, nó sẽ tự động checkpoint tất cả các tiến trình trong chương trình MPI.

Sử dụng lệnh 'cr_restart' để restart lại chương trình MPI và các tiến trình của nó tại thời điểm đã được checkpoint:

```
% cr_restart context.PID
```

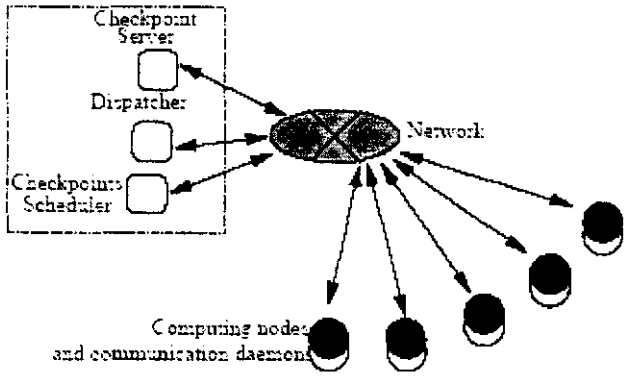
4.2. Thư viện MPICH-CL

4.2.1. Kiến trúc thư viện MPICH-CL

MPICH-CL sử dụng hai thành phần: Generic Device và giao tiếp CL Deamon. Generic device bao gồm hai hàm giao tiếp là PIbsend và PIbrecv. Hai hàm này ngừng các thao tác khi gửi và nhận một khối dữ liệu. Nó bao gồm 4 hàm khác là PInprobe để kiểm tra xem một thông điệp đã được xử lý chưa (có phải *pending messages* không); PIfrom để lấy đặc điểm của bên gửi tin nhắn cuối cùng; PIiInit để khởi tạo kênh, và PIiFinish để kết thúc.

Cũng giống như việc thực hiện của kênh P4, các tiến trình MPI không kết nối trực tiếp với tất cả các nút tính toán khác, mà thông qua tiến trình ngầm *deamon*. Như vậy, các giao tiếp *daemon* nối với các tiến trình MPI và giải quyết vấn đề không đồng bộ của mạng. Khi *daemon* chạy, nó sẽ thiết lập một UNIX socket và sinh ra các tiến trình MPI. Hai loại tin nhắn được trao đổi sử dụng *socket* này là tin nhắn điều khiển (control message) (để khởi tạo, thăm dò và kết thúc) và tin nhắn giao thức (protocol message) (ví dụ như bsend, breceive).

MPICH-CL thực hiện giao thức checkpoint sử dụng thuật toán của Chandy-Lamport. Kiến trúc của nó gồm một bộ điều khiển *dispatcher*, một bộ lập lịch checkpoint (checkpoint scheduler - SC), các trạm checkpoint (checkpoint server-CS), các nút tính toán (Computing Node-CN) cùng với các daemon giao tiếp của chúng. MPICH-CL chỉ khác MPICH-V2 là nó không có các bộ ghi lại tin nhắn (event logger.)



Hình 4-3 Kiến trúc của phân hệ MPICH-CL

Trong hệ thống của ta, các server (Dispatcher, CS, SC) được đặt trên một máy chủ ổ định, không có lỗi. Mỗi tiến trình và *daemon* của nó được chạy trên một nút tính toán.

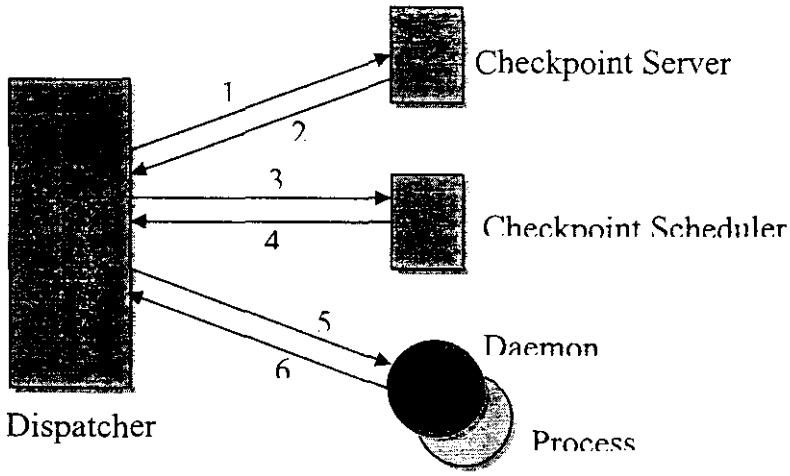
Sau đây, ta sẽ đi chi tiết vào từng thành phần của phân hệ.

4.2.1.1. Dispatcher

Nhiệm vụ của Dispatcher là khởi tạo các Computing Node (CN), Checkpoint Server, Checkpoint Scheduler vào các nút sẵn có của hệ thống khi một *job* bắt đầu chạy, theo dõi các nút này để phát hiện lỗi xảy ra. Khi phát hiện được một lỗi trên CN, nó sẽ khởi động lại *daemon* trên nút này, với các tham số thích hợp. Cuối chương trình, mỗi CN khi kết thúc sẽ gửi tín hiệu về cho Dispacher.

Thông thường thì các Server (SC và SC) được đặt trên các nút ổn định, do đó, ta coi như lỗi không xảy ra trên các nút này.

Ngay khi khởi tạo các phần xong, mỗi CN, SC, CS sẽ gửi pid và auxID về cho Dispatcher quản lý.



Hình 4-4 Giao tiếp giữa Dispatcher với các phần khác

1: Dispatcher khởi tạo Checkpoint Server, hoặc huỷ CS khi có tín hiệu kết thúc

2: Checkpoint Server trả pid và auxID lại cho Dispatcher quản lý, hoặc gửi tín hiệu kết thúc đến Dispatcher khi công việc kết thúc

3: Dispatcher khởi tạo SC

4: SC gửi pid và auxID cho Dispatcher, hoặc gửi tín hiệu kết thúc

5: Dispatcher khởi tạo Daemon. hoặc huỷ Daemon khi kết thúc tiến trình, hoặc chạy lại Daemon khi có lỗi xảy ra

6: Daemon gửi cho Dispacher pid. auxID. hoặc gửi thông điệp kết thúc tiến trình.

Việc khởi tạo danh sách các nút : được thực hiện qua 2 hàm sau :

void parseProgramFile(JS *);

void parseCommandsFile(char *, JS *);

Việc phát hiện lỗi:

Dispatcher liên kết với các CN (thực tế là các *daemon*) thông qua các *socket*, và lắng nghe tín hiệu thông qua câu lệnh ***select***. Do đó, khi một tiến trình xảy ra lỗi, *daemon* sẽ bị mất kết nối với *Dispatcher*, nhờ đó, *Dispatcher* phát hiện ra nút lỗi này để chạy lại.

Khi công việc kết thúc, các nút gửi tín hiệu kết thúc (FINALIZE_MSG) cho Dispatcher, Dispatcher dựa vào pid và auxID của nút này để xóa nút này đi.

Dispatcher quản lý các nút theo cấu trúc sau:

```
typedef struct computingNode {
  int     rank;
  char    hostName[64];
  char    ipAddress[15];
  char    fastIpAddress[15]; /* Dành cho mạng hiệu năng cao */
  int     communicationPort;
  char    eventLogger[15];
  int     eventLoggerPort;
  char    checkpointServer[15];
  int     checkpointServerPort;
  char    checkpointScheduler[15];
```

```
  int     checkpointSchedulerPort;
  pid_t   pid; /* để phục vụ cho việc kill tiến trình */
  struct computingNode * next;
} CN;  /* Computing Node */


typedef struct checkpointServer {
  char    ipAddress[15];
  char    fastIpAddress[15];
  int     port;
  char    tmp[128];
  pid_t   pid;
  struct checkpointServer * next;
} CS; /* Checkpoint server */


typedef struct checkpointScheduler {
  char    ipAddress[15];
  char    fastIpAddress[15];
  int     port;
  pid_t   pid;
  struct checkpointScheduler * next;
} SC; /* Checkpoint Scheduler */
```

4.2.1.2. CheckPoint Server

Checkpoint Server (CS) lưu ảnh checkpoint của các tiến trình, phục vụ cho quá trình khôi phục. Thông thường, CS được nằm trên một nút ổn định, ít xảy ra lỗi. Mỗi ảnh của một tiến trình được lưu vào một file trong thư mục đường dẫn (dưới dạng tham số đưa vào).

File của tiến trình *rank*, thuộc nhóm *group* có dạng: *ckptimg-group-rank.*

Đầu vào của CS :

```
theJob.rshCmd, Thư mục rsh

auxCS->ipAddress, địa chỉ lập lịch

theJob.csCmd, đường dẫn đến CS

theJob.jobId,    ID của job

auxCS->port, cổng của CS

theJob.debugCommand, thư viện gỡ lỗi

auxCS->tmp, tên của thư viện đặt checkpoint

nbAuxTotal, số CS

theJob.dispatcherIP, địa chỉ của Dispacher

theJob.dispatcherPort
```

Checkpoint Server lắng nghe các tiến trình mà mình quản lý. Nếu có tín hiệu PUT, hay GET checkpoint, nó sẽ sử dụng hàm fork, sinh ra một tiến trình mới để thực hiện công việc này. Tiến trình cha vẫn tiếp tục làm việc để lắng nghe từ các tiến trình khác.

Khi ghi ảnh của tiến trình, CS ghi vào file *oldname* có dạng *ckptimg-group-rank.bak.* Sau khi ghi thành công, nó xác nhận file đã checkpoint được bằng cách xóa file *newname* (*ckptimg-group-rank)* đi, và đổi tên *oldname* thành *newname.* Như thế sẽ đảm bảo được an toàn nếu quá trình ghi checkpoint bị lỗi.

Khi đọc ảnh checkpoint của tiến trình, CS dựa vào *group* và *rank* mà nó đọc được qua *socket* với *daemon* yêu cầu, CS sẽ mở file checkoint tương ứng, và gửi file này về qua *socket* đó.

Các hàm chính của CS:

static int get_checkpoint(int sproto, int sdata);

mở tệp, lấy ảnh checkoint.

static int put_checkpoint(int sproto, int sdata);

tạo ra tệp mới trong CS, ghi ảnh của tiến trình vào tệp này

static int recv_type_message(int s);

Trả về GET (=1) nếu tín hiệu đọc từ socket s là "G"

PUT (=2) nếu là "P"

ERROR_SYNTAX(=-2) nếu khác

Ngoài ra có một số hàm để thao tác với tệp:

```
int getopenCheckpointFile(int group, int rank, int seq)
```

mở tệp để lấy ảnh checkpoint. Tệp có dạng ckptimg-g-r

```
int putopenCheckpointFile(int group, int rank, int seq)
```

mở tệp mới để ghi ảnh checkpoint vào

```
int discardCheckpointFile(int fd, int group, int rank, int seq)
```

Xóa tệp

```
int confirmCheckpointFile(int fd, int group, int rank, int seq)
```

xóa tệp *newname*, đổi tên *newname* thành *oldname*

```
int protowriteCheckpointFile(int fd, int protosize, int where,
void *buff, int size)

int datawriteCheckpointFile(int fd, int protosize, int where,
void *buff, int size)

int datasizewriteCheckpointFile(int fd, int protosize, int
datasize)
```

4.2.1.3. Checkpoint Scheduler

Checkpoint Scheduler có nhiệm vụ báo cho các tiến trình thời điểm thích hợp để checkpoint. Theo lý thuyết, SC phải nắm được toàn bộ trạng thái của các tiến trình, để quyết định thời điểm checkpoint cho thích hợp, nhằm giảm chi phí bộ nhớ cho việc tạo checkpoint. Tuy nhiên, trong hệ thống này, chúng ta chỉ dừng lại

ở mức: SC gửi tín hiệu yêu cầu checkpoint cho các tiến trình sau một khoảng thời gian t nhất định. Khoảng thời gian này gọi là khoảng thời gian *timeout* t. Đồng hồ trên SC sẽ đếm lùi, cứ hết khoảng thời gian t. nó lại gửi tín hiệu yêu cầu checkpoint cho tất cả các tiến trình.

Đầu vào của SC được định dạng như sau:

Nếu tham số đầu là –t thì tiếp theo sẽ là thời gian giữa 2 lần checkpoint.

Nếu tham số đầu là –f, số thứ 2 khác 0, thì số này là thời gian giữa hai lần checkpoint (***timeout***), còn thời gian cảnh báo lỗi được tính bằng ***faulttime = 60 + timeout/2***

Nếu tham số đầu không phải –t hay –f thì có nghĩa không có yêu cầu checkpoint. (Thường dùng cho *uncoordinated checkpoint* trong *v2d*)

Các tham số tiếp theo lần lượt là:

```
theJob.rshCmd,   Thư mục rsh
auxSC->ipAddress, địa chỉ lập lịch
theJob.scCmd, đường dẫn đến SC
auxSC->port,  cổng của SC
theJob.nprocs,số tiến trình
theJob.jobId, ID của job
nbAuxTotal, số CS
theJob.dispatcherIP,  theJob.dispatcherPort,  địa  chỉ  của
Dispacher
theJob.debugCommand;  thư viện gỡ lỗi
```

Đến thời gian cảnh báo lỗi,(sau khoảng thời gian faulttime), SC gửi tín hiệu cảnh báo lỗi (CSCHED_REQ_URGENTFAULT) đến *daemon bất kỳ* trong các tiến trình.

Trong khi đó, cứ hết thời gian *timeout*, SC lại gửi tín hiệu yêu cầu checkpoint (CSCHED_REQ_CHECKPOINT) đến *tất cả các daemon*.

Các tiến trình, khi checkpoint xong sẽ gửi lại tín hiệu cho SC.

Nếu SC nhận được tất cả các tín hiệu từ các tiến trình, công việc checkpoint coi như đã hoàn thành.

SC sử dụng cơ chế *select* để lắng nghe tín hiệu của các *daemon* của tiến trình. Ngoài ra, nó cũng kết nối với Dispatcher, để chuyển pid và auxID đến cho Dispatcher khi vừa khởi tạo xong.

Cấu trúc của các tiến trình được SC quản lý thông qua một danh sách các tiến trình được tổ chức như sau:

```
struct proc_info {
  unsigned char status;  trạng thái của tiến trình
PROC_DOWN  (0x01 )hoặc
PROC_UP_NOTHING (0x02)
  int fd;
  unsigned long *infos;
  time_t last_successfull_cp; thời điểm checkpoint thành công gần
nhất
  char *cur_message;
  int   cur_alloc;
  int   cur_len;
  int   cur_read;
};
```

Các hàm, thủ tục được sử dụng trong SC:

static void on_quit(int s)

đóng tất cả các tiến trình lại để kết thúc

static int isAnybodyOutThere( )

Kiểm tra xem có tiến trình nào chết (status = PROC_DOWN) không

Nếu có, trả về 0, ngược lại, trả về 1

static int atomic_send(int fd, void *buf, int size)

gửi size byte vào fd, (gửi buf byte một)

Nếu lỗi, trả về -1, nếu không, trả về size

static int new_connection(int fd)

Thiết lập liên kết đến fd để đọc dữ liệu

Thành công, trả về 0, ngược lại, trả về -1

static void disconnect(int rank)

bỏ kết nối với tiến trình *rank*

void send_urgent_checkpoint_order(int rank)

gửi tín hiệu báo lỗi đến tiến trình *rank*

void urgent_checkpoint()

gửi tín hiệu checkpoint đến tất cả các tiến trình

void urgent_fault()

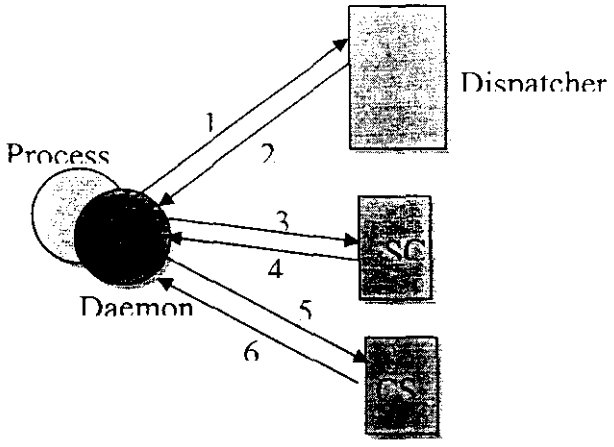
gửi tín hiệu cảnh báo lỗi đến một tiến trình bất kỳ

4.2.1.4. Daemon của MPICH-CL

Daemon làm nhiệm vụ trung chuyển thông điệp giữa các tiến trình với nhau, nhận tín hiệu checkpoint từ SC, thực hiện checkpoint bằng cách gửi ảnh checkpoint của tiến trình, cùng với *pending messages* đến CS

Daemon của MPICH-CL sử dụng 2 socket: 1 để kết nối với các Server, còn 1 để giao tiếp với các nút tính toán khác.

Để giao tiếp với tiến trình của mình (trên cùng một máy), *daemon* sử dụng phương thức AF_UNIX.



Hình 4-5 Giao tiếp giữa với các thành phần khác

Daemon giao tiếp với Dispatcher như phần Dispatcher đã trình bày, chúng ta không nhắc lại nữa. Phần giao tiếp giữa Daemon và các phần khác như sau:

3: Daemon gửi tín hiệu chuẩn bị checkpoint đến cho SC, hoặc tín hiệu khi đã hoàn thành checkpoint đến cho SC.

4: SC theo định kỳ (cứ sau khoảng thời gian t) lại gửi yêu cầu checkpoint đến cho Daemon

5:Daemon gửi tín hiệu GET, hoặc PUT đến cho SC.

Daemon cũng gửi ảnh của tiến trình và các *pending message* đến cho CS.(nếu tín hiệu gửi trước là PUT)

6: Daemon lấy ảnh tiến trình từ CS về.

Các tin nhắn được trao đổi theo cách sau:

Gửi đi:

Đầu tiên, phần đầu *header* của tin nhắn được gửi đi. *Header* mô tả các thông tin về tin nhắn mà nó sẽ gửi tiếp. Cấu trúc của *header* được tổ chức như sau:

```
typedef struct pkt_header {
  int iTag;
  int iSrc;  //thông điệp được gửi từ đâu
  int iDst;       //thông điệp được gửi đến đâu
  int iSize; //kích thước thông điệp
  int iType; // kiểu thông điệp
} pkt_header;
```

Trong đó, giá trị của *iType* được xác định như sau:

```
Unknown                 không thuộc [1..8]
ANY_MSG_AVAIL           1  //tin nhắn thăm dò
RECV_ANY_CONTROL        2  //yêu cầu nhận tin nhắn điều khiển
SEND_CONTROL            3  //yêu cầu gửi tin nhắn điều khiển
RECV_FROM_CHANNEL       4  //yêu cầu nhận tin nhắn
SEND_CHANNEL            5  //yêu cầu gửi tin nhắn
CP_START                6  //tín hiệu checkpoint
REPLAY_START            7  // tín hiệu restart ( chưa sử dụng đến)
CLIENT_EXIT;            8  //tín hiệu kết thúc
```

iTag giữ kết quả thăm dò, bằng 1 nếu có message đang chờ để gửi cho nó

Trạng thái của *daemon* gồm các thông tin sau:

- Trạng thái của tiến trình: NO_CKPT / IN_CKPT
- Một danh sách các tin nhắn chưa được xử lý (pending message)
- Tập các tín hiệu *marker* đã nhận từ các tiến trình khác

Cấu trúc của daemon như sau:

```
typedef struct clvar
{
  unsigned short state;          /* NO_CKPT / IN_CKPT */
  LinkedList_t pm;    /* danh sách các message chưa được xử lý
  unsigned short *markers; /* bảng các markers từ các tiến
/* trình khác gửi đến
                          * markers[myrank] = marker từ csched
                          */
  int markersSize;        /* số phần tử của markers */
  int myRank;             /* rank của tiến trình mpi này */
  CkptInfo *cin;      /* struct CkptInfo */
}
clvar;
```

CkptInfor chứa ảnh tiến trình, được ghi lại bởi thư viện *Condor* trong *pipe* và một số thông tin có liên quan khác.

Cấu trúc của Checkpoint Infor:

```
typedef struct CkptInfo
{
  int seq;            /* sequence number of the checkpoint image
*/
  int pipe;      /* pipe where to read checkpoint image generated
by condor */
  CkptSock sock;      /*    socket connected to the checkpoint
server */
  long h;             /* H date of the begining of the checkpoint
*/
```

```
  int np;                 /* number of MPI processes */
//   int i; /* counter indicating the destination of the message
we are writting */
//   int j; /* counter indicating current message to i in this
queue */
//   sElement_t *segment; /* pointer to the message we are sending
*/
//   int offset; /* where we are in this message */
  int psize;              /* size of the pending message to send
*/
  int datasize;           /* size of the image sent */
  char state;             /* state may be CKPTSTATE_SBSEND |
CKPTSTATE_IMGSEND | CKPTSTATE_FINISHED */
  int block_size;         /* iff state == CKPTSTATE_IMGSEND,
hold the number of bytes to send */
}
```

Toàn bộ thông tin của clvar sẽ được gửi đến CS khi có tín hiệu yêu cầu cạo checkpoint từ SC.

Giao tiếp với tiến trình:

Daemon giao tiếp với tiến trình của mình thông qua giao thức AF_UNIX nhằm trao đổi tín hiệu checkpoint, hoặc các tin nhắn từ các tiến trình khác.

Tiến trình có thể gửi cho daemon các loại tin nhắn sau:

ANY_MSG_AVAIL: để thăm dò xem có tin nhắn nào gửi cho nó mà chưa được xử lý không. (**_v2probe()**)

RECV_ANY_CONTROL hoặc RECV_FROM_CHANNEL: yêu cầu nhận tin (**_v2brecev()**)

SEND_CONTROL hoặc SEND_CHANNEL: yêu cầu gửi tin (**_v2bsend()**)

CLIENT_EXIT: Khi kết thúc tiến trình.

Bất cứ khi nào nhận được tin nhắn từ tiến trình, Daemon cũng có tin nhắn gửi trả lời. Nếu có tín hiệu checkpoint, thì tin nhắn mà tiến trình nhận được là CP_START. Nếu không thì tin nhắn trả về là *Unkown*.

Khi nhận được tin nhắn kiểu ANY_MSG_AVAIL, daemon sẽ gọi hàm **on_probe()** để xử lý. Hàm này sẽ tìm trong pending messages xem có tin nhắn nào gửi cho tiến trình không. Nếu có, tin nhắn nó trả về cho tiến trình có ***Itag=1***. Đồng thời, nếu trạng thái tiến trình có thể checkpoint được, tin nhắn trả về có kiểu CP_START.

Khi nhận được RECV_ANY_CONTROL hoặc RECV_FROM_CHANNEL, daemon gọi hàm **on_read()**. Hàm này tìm trong *pending messages* xem có tin nhắn nào gửi cho nó không hoặc nếu có tin từ CN tương ứng, nó sẽ gửi tin nhắn này cho tiến trình thông qua hàm **on_writeback().** Nếu có tín hiệu checkpoint, nó cũng sẽ gửi tin nhắn CP_START vềc cho tiến trình.

Khi nhận được SEND_CONTROL hoặc SEND_CHANNEL, daemon gọi hàm **on_write().** Hàm này nhận tin nhắn từ daemon, và thêm thông điệp này vào ***sb[iDst]*** tương ứng để gửi cho ***iDst*** khi có tín hiệu.

Khi nhận được tin nhắn CLIENT_EXIT, daemon gửi tín hiệu FINALIZE_MSG đến cho Dispatcher, đóng kết nối với Dispatcher, và kết thúc.

Giao tiếp với CS:

Thông qua giao thức AF_INET, để gửi và lấy các ảnh checkpoint.

Gửi ảnh lên checkpoint cần chủ yếu hai thủ tục sau:

on_sending_ckptimage (cl.cin);

gửi ảnh tiến trình lên CS

on_sending_ckptpending (cl.cin, &(cl.pm));

gửi các *pending messages* lên CS

Việc nhận lại checkpoint được thực hiện bởi 2 hàm:

unserialize_image (s, mygroup, myrank)

unserialize_pendings (s, &pnd_msgs)

Hai hàm này thực hiện đọc dữ liệu từ CS. ghi vào pipe có tên ***wid.rankid.restart.pipe***, và tách các thông điệp ra. cho vào **pnd_msgs**.

Giao tiếp với SC:

Để trao đổi tín hiệu checkpoint

Khi nhận được tín hiệu của SC, *daemon* sẽ thay đổi trạng thái theo mô hình như sau: (trong hàm **handle_csched_request (int fd)**)

CSCHED_REQ_CHECKPOINT:

status = STATUS_CP_REQUESTED;

CSCHED_REQ_URGENTCKPT:

status = STATUS_UCP_REQUESTED;

CSCHED_REQ_URGENTFAULT:

/*ko làm gì

CSCHED_REQ_SENDINFO:

Gửi tín hiệu CSCHED_REP_INFOS và thông tin của các tiến trình đi

Khi gửi tín hiệu cho SC, sử dụng hàm **handle_csched_write().**

Giao tiếp với các tiến trình khác:

Trao đổi tin nhắn, gửi hoặc nhận *marker* (là các tin nhắn kiểu CP_START)

Nhận tin nhắn từ tiến trình khác, sử dụng hàm **on_receive()**

Gửi tin nhắn đến tiến trình khác, sử dụng hàm **on_send()**

Thực hiện checkpoint

Trước khi thực hiện công việc checkpoint. daemon phải thực hiện các công việc sau:

- Thiết lập kết nối với CS

```
connectCheckpointServer (addr);
```

- Khởi tạo giao thức

```
putCheckpointProto (s, mygroup, myrank, seqnumber, 3 * sizeof (int)
```

- Ghi ctab
- Khởi tạo cấu trúc của checkpoint:

```
ckptinfo->seq = seqnumber;
ckptinfo->pipe = pipe;
ckptinfo->sock = s;
ckptinfo->np = np;
ckptinfo->psize = 0;
ckptinfo->datasize = 0;
ckptinfo->state = CKPTSTATE_IMGSEND;
ckptinfo->block_size = 0;
```

- Đặt kiểu truyền NONBLOCK

```
fcntl (ckptinfo->sock.data, F_SETFL, O_NONBLOCK)
```

Tất cả các công việc này được thực hiện trong hàm:

```
CkptInfo * on_ckpt_signal (struct sockaddr_in * addr, int mygroup,int myrank, int seqnumber, int np)
```

Các hàm được sử dụng cho các công việc trên như sau:

(Chỉ nêu các hàm chính)

```
changeState (clvar * var)
```

đổi trạng thái:

```
raz (clvar * var, LinkedList_t * pendings)
```

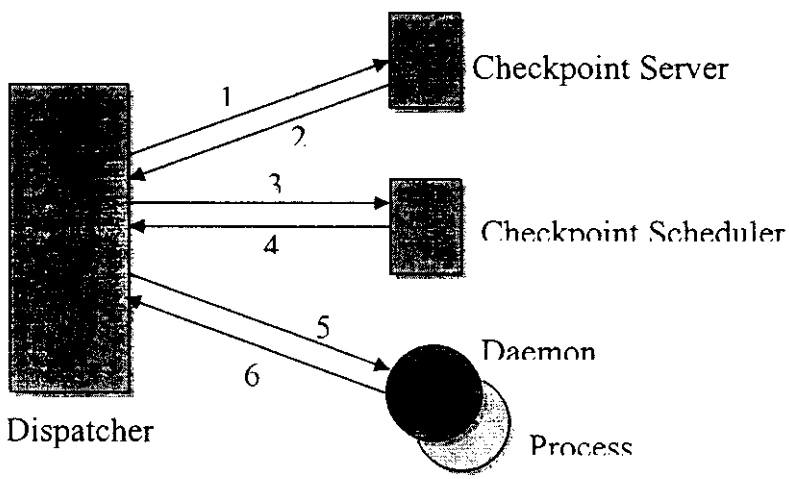
giải phóng các thành phần của daemon

```
launchCheckpoint (clvar * lastruct, struct sockaddr_in *addr, int
mygroup,
            int myrank, int seqnumber, int np, listeMessages *
sb,
            LinkedList_t * pndmsgs)
```

Tạo checkpoint

4.2.2. Mô hình truyền – nhận thông điệp trong hệ MPICH-CL

4.2.2.1. Quá trình khởi tạo



Hình 4-6 Quá trình khởi tạo

1, 3,5: Dispatcher khởi tạo CS, SC và các daemon

2,4,6: Các nút sau khi khởi tạo, gửi pid, auxID lại cho Dispatcher

Bắt đầu chạy, file batch nhận đầu vào là file lập lịch PBS_NODEFILE

Trong quá trình chạy, nó sinh ra 2 file:

v2pgfile.$id

File này xác định địa chỉ IP cho từng node, với mỗi CN, nó xác định cả CS, SC nào quản lý nó.

v2pgfile.command

File này chứa các dòng lệnh để khởi tạo SC, CS và các daemon.

Hai file này sau đó được chuyển xuống cho Dispatcher. Dispatcher (file mpirun) thực hiện khởi tạo các CN, CS và SC.

Sau đó *Dispatcher* nhận kết nối từ các CN này tới nó, lấy *rank*, *pid* và *acceptSocket* bổ sung vào danh sách *ConnectedNode*,và trả về danh sách *nodeListArray* cho từng CN để nó dùng nhận biết các *Daemon* truyền thông khác.

Khi CN nhận *nodeListArray* từ Dispatcher, ngay lập tức nó khởi tạo mảng *mpipeer[]* lưu trữ các thông tin *port* và *IP* của các CN khác.

mỗi phần tử của *mpipeer[]* là một phần tử kiểu *SockInfo*, lưu các thuộc tính của một CN khác, có cấu trúc như sau:

```
typedef struct SockInfo {
  int cofd;
  int costate;
  int hstate;
  int fd;
  struct sockaddr_in addr;
  char * buf;
  unsigned int size;
```

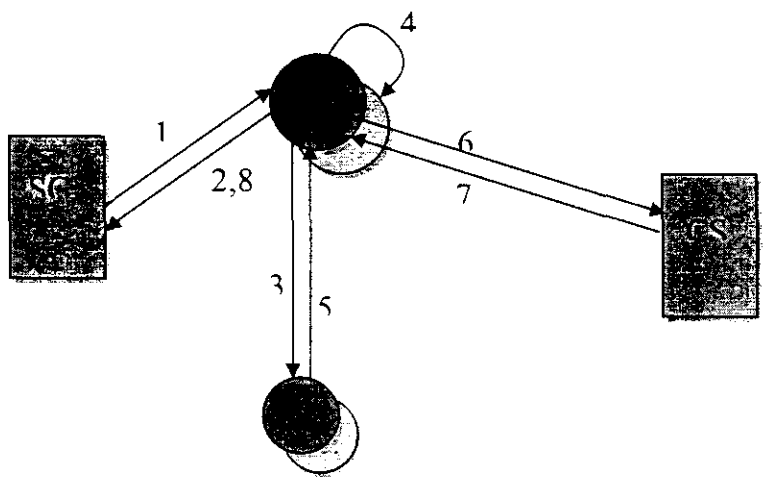
```
  unsigned int recvd;
  long int wait_for_ack;
} SockInfo;
```

Sau khi kết thúc giao tiếp với *Dispatcher*, CN sẽ kết nối tới các CN khác

Sở dĩ hiện nay chưa thể chạy trên nút truyền thông khác được là vì khi được khởi tạo, các CN đã nhận sẵn danh sách *nodeListArray* sau đó khởi tạo cứng luôn mảng mpipeer, vì vậy khi 1 CN chết danh sách này không được cập nhật lại địa chỉ IP mới, nên ta vẫn phải dùng cơ chế khôi phục trên nút cũ. Giải pháp đề xuất là khi chạy lại CN mới trên máy có địa chỉ IP mới, bên phía Dispatcher ta cập nhật lại danh sách *nodeListArray* rồi gửi cho CN mới danh sách này , còn với các CN cũ đang chạy, ta gửi địa chỉ IP mới này về cho nó để nó cập nhật lại địa chỉ này cho *mpipeer[i]* trước khi nút CN mới chạy lại. Như thế ta mới có thể chống lỗi trên nút có địa chỉ IP mới.

Sau khi được khởi tạo, CN (thực chất là daemon) mới thực sự chạy tiến trình của mình bằng cách gọi hàm **`execvp (path, argv)`**

4.2.2.2. Quá trình checkpoint



Hình 4-7 Quá trình checkpoint với MPICH-CL

1: SC gửi yêu cầu checkpoint cho daemon

2: Daemon gửi lại ack cho SC

3: Daemon gửi thông báo checkpoint đến cho các daemon khác

4: Daemon giao tiếp với tiến trình của mình để lấy ảnh checkpoint

5: Daemon nhận thông điệp từ các daemon khác để ghi vào danh sách các thông điệp chưa được xử lý

6: Daemon gửi ảnh tiến trình, cùng danh sách thông điệp chưa được xử lý lên CS

7: CS gửi tín hiệu đã checkpoint xong về cho Daemon.

8: Daemon gửi lại tín hiệu hoàn thành checkpoint cho SC

Việc checkpoint được thực hiện khi daemon nhận được tín hiệu yêu cầu checkpoint từ SC, hoặc tin nhắn CP_START từ tiến trình khác. (đây chính là các marker được nói đến trong phần lý thuyết).

Khi SC gửi tín hiệu checkpoint (CSCHED_REQ_CHECKPOINT) hoặc (CSCHED_REQ_URGENTCKPT) cho daemon, daemon đổi trạng thái của mình tương ứng thành STATUS_CP_REQUESTED

hoặc STATUS_UCP_REQUESTED. Lúc đó, hàm **csched_can_cp_now** () có giá trị bằng 1

Nếu nhận được tin nhắn CP_START từ tiến trình khác, biến *cp_from_marker* được gán bằng 1.

Bất cứ khi nào tiến trình giao tiếp với daemon (nhận hoặc gửi tin nhắn), nó đều nhận được một tín hiệu trả về. Tín hiệu này là một tin nhắn có dạng CP_START hoặc là Unkown. Nếu tín hiệu này là CP_START, nghĩa là **csched_can_cp_now() = 1**, hoặc ***cp_from_marker* = *1***(nghĩa là nhận được marker từ tiến trình khác). Khi đó, nó liền đóng kết nối với daemon, và gửi ảnh của mình vào pipe nhờ hàm ckpt_and_exit của thư viện condor.

Trước khi checkpoint, daemon gửi cho SC tín hiệu

CSCHED_REP_BEGIN_CHECKPOINT

Sau đó, daemon thực hiện việc checkpoint như sau:

- Đổi trạng thái thành IN_CKP
- gửi marker cho tất cả các tiến trình khác
- Tiếp tục nhận các tin nhắn từ các tiến trình khác, và lưu vào danh sách

(Như vậy, tin nhắn cuối cùng mà daemon nhận được từ tiến trình khác trong lần checkpoint này là tin nhắn kiểu CP_START)

- Cho đến khi nhận được các marker(tin nhắn kiểu CP_START) từ tất cả các tiến trình.
- Sau đó, nó tiến hành checkpoint:

  Đọc thông tin từ pipe ra để lấy ảnh của tiến trình

  Gửi ảnh tiến trình này, cùng các pending message đến CS

Việc ghi checkpoint vào CS được thực hiện như sau:

Khi CS nhận được tín hiệu PUT (=2) từ daemon, CS đọc thông tin từ cổng của daemon này, và ghi vào file *oldname* có dạng *ckptimg-group-rank.bak*. Sau khi ghi thành công, nó xác nhận file đã checkpoint được bằng cách xóa file *newname* (*ckptimg-group-rank)* đi, và đổi tên *oldname* thành *newname*. Như thế sẽ đảm bảo được an toàn nếu quá trình ghi checkpoint bị lỗi. Như vậy, tại một thời điểm, mỗi tiến trình chỉ có một checkpoint chính thức (permenant checkpoint) là file *newname* tương ứng. Tệp này được sử dụng để khôi phục lỗi nếu xảy ra. Tập các tệp *oldfile* tương ứng của tất cả các tiến trình tạo thành một lát cắt (recover line) của toàn hệ thống.

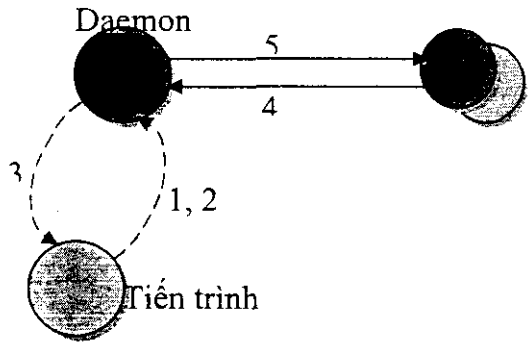
Khi hoàn thành việc checkpoint, daemon sẽ giải phóng danh sách *pending message*, trở về trạng thái bình thường (NO_CKP).

Hoàn thành việc checkpoint, daemon còn gửi cho SC tín hiệu

CSCHED_REP_END_CHECKPOINT

Khi SC nhận được tất cả các tín hiệu này từ daemon, chứng tỏ công việc checkpoint cho toàn hệ thống đã hoàn tất.

4.2.2.3. Quá trình truyền – nhận các thông tin thông thường



Hình 4-8 Quá trình gửi thông điệp

1: Tiến trình gửi header của thông điệp đến Daemon

2: Tiến trình gửi thông điệp đến cho Daemon

3: Daemon gửi cho tiến trình tín hiệu xác nhận đã nhận được tin nhắn

4: Daemon của tiến trình đích gửi yêu cầu nhận đến Daemon

5: Daemon gửi thông điệp đến cho daemon của tiến trình đích

Khi một tiến trình muốn gửi một tin nhắn đến cho tiến trình khác, đầu tiên nó gửi *header* kiểu SEND_CONTROL hoặc SEND_CHANNEL đến cho daemon, tuỳ thuộc tin nhắn này là tin nhắn kiểu điều khiển, hay tin nhắn thông thường.

Tiếp theo, nó gửi *iSize* byte, là kích thước của tin nhắn mà đã được lưu trong *header* vừ gửi, xuống cho daemon.

Daemon dựa vào header sẽ biết được các thông số về thông điệp sẽ nhận được tiếp theo, và địa chỉ của tiến trình cần gửi thông điệp đến (trong trường *iDst*).

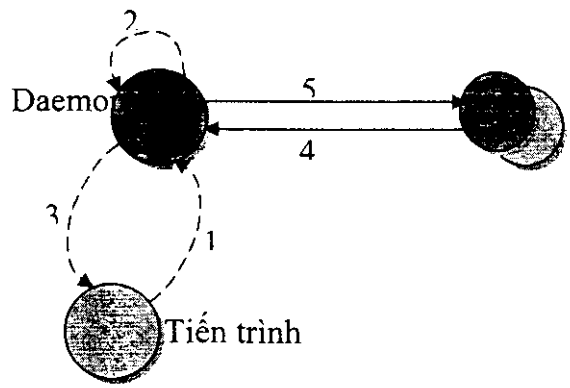
Daemon nhận tin nhắn. liền gắn thêm vào phần thông tin sẽ gửi cho *iDst*. Nếu có tín hiệu checkpoint thì daemon sẽ gửi lên cho tiến trình tin nhắn CP_START, nếu không, gửi tin nhắn xác nhận đã nhận được tin.

Khi daemon nhận được yêu cầu đọc tin từ *iDst* này, nó sẽ chuyển *header* và tin nhắn ở trên cho *iDst* đó.

Việc gửi tin nhắn từ tiến trình đến daemon thông qua hàm ***_v2bsend***

Daemon nhận được tin nhắn kiểu `SEND_CONTROL` hoặc `SEND_CHANNEL` sẽ gọi hàm on

Quá trình nhận tin nhắn



Hình 4-9 Quá trình nhận thông điệp

1: Tiến trình gửi header đến daemon, yêu cầu đọc tin nhắn

2: Daemon lục tìm trong danh sách các tin nhắn chưa được xử lý xem có tin nhắn nào phù hợp không.

3: Daemon gửi thông điệp phù hợp cho tiến trình.

4: Daemon của tiến trình gửi gửi yêu gửi tin cho daemon

5: Daemon đọc thông điệp của tiến trình gửi để chuyển cho tiến trình của mình.

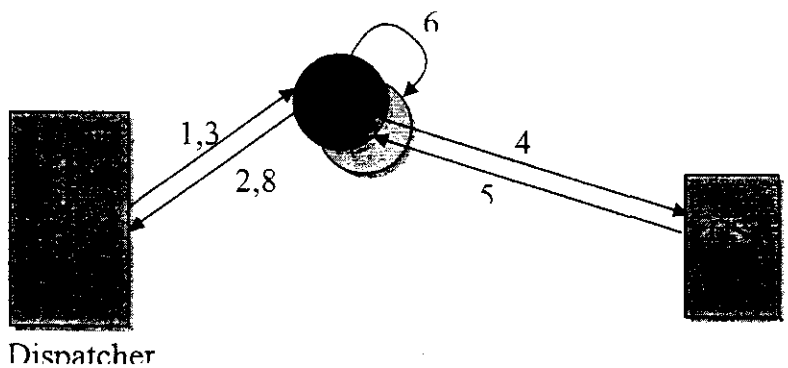
Tiến trình muốn nhận tin nhắn, đầu tiên sẽ gửi header có kiểu

RECV_ANY_CONTROL hoặc RECV_FROM_CHANNEL cho daemon.

Đầu tiên, daemon lục tìm trong pending message xem có tin nhắn nào phù hợp gửi cho tiến trình không. Nếu có, nó sẽ chuyển tin nhắn này lên cho tiến trình, và xoá tin nhắn này trong pending messages đi.

Nếu daemon nhận được tín hiệu gửi tin từ daemon tương ứng, nó sẽ xem kiểu tin nhắn này có phải CP_START (marker) không. Nếu không phải, nó xem xét trạng thái của mình. Nếu trạng thái của daemon lúc này là NO_CKPT, tin nhắn này sẽ được đọc và gửi cho tiến trình. Ngược lại, nếu trạng thái của daemon là IN_CKPT, tin nhắn này sẽ được lưu vào trong pending messages

4.2.2.4. Quá trình khôi phục lỗi



Hình 4-10 Quá trình khôi phục lỗi

1: Dispatcher nghe ngóng tín hiệu từ các Daemon

2: Daemon chết tạo ra tín hiệu đánh thức Dispatcher, Dispatcher phát hiện ra nút lỗi

3: Dispatcher chạy lại Daemon, với thông số *-restart*

4: Daemon gửi tín hiệu GET đến CS yêu cầu lấy ảnh checkpoint

5: Daemon đọc ảnh checkpoint từ CS

6: Daemon tách ảnh tiến trình và các *pending messages*, chạy lại tiến trình với tham số *-condor*

Dispatcher phát hiện ra lỗi khi socket với một tiến trình nào đó bị đánh thức vì bị ngắt kết nối.

Khi xác định được tiến trình lỗi, Dispacher khởi tạo lại daemon, với một tham số thêm vào là ***–restart***. Khi đó, daemon sẽ đặt biến **ckpt_restart = 1.**

Khi khởi tạo xong, nếu **ckpt_restart = 1,** daemon sẽ tiến hành lấy ảnh checkpoint trên CS về.

Daemon lấy ghi ảnh tiến trình vào pipe có tên ***wid.rankid.restart.pipe*** bằng thủ tục **unserialize_image (s, mygroup, myrank),** và tách các pending messages, cho vào **pnd_msgs** bằng thủ tục

unserialize_pendings (s, &pnd_msgs).

Deamon chạy lại tiến trình kèm thêm tham số **-condor**. Khi gặp tham số này, thư viện condor sẽ lấy ảnh trên pipe có tên **wid.rankid.ckpt.pipe.tmp,** bung tiến trình ra. Như vậy, quá trình restart đã hoàn thành.

4.3. Thư viện MPICH_V1

4.3.1. Giải pháp Pessimistic logging trong MPICH_V1

Về ý tưởng cơ bản, phương pháp pessimistic message logging sẽ thực hiện việc ghi lại checkpoint của từng tiến trình, ngoài ra còn ghi lại các thông điệp mà tiến trình đã phát ra, để phục vụ cho mục đích phục hồi trạng thái của tiến trình và phát lại các thông điệp theo đúng thứ tự mà thông điệp đã phát ra

4.3.1.1. Phạm vi

Để thiết kế một hệ thống áp dụng chống lỗi ghi thông điệp Pessimistic chúng ta cần xác định rằng:

- Lỗi có thể xảy ra tại thời điểm bất kỳ trong hệ thống, bất cứ tình huống nào ta cũng sẽ phải đảm bảo rằng dữ liệu và hệ thống sẽ không bị mất, không bị ảnh hưởng đến công việc chạy và chúng ta cần thiết kế để việc phục hồi sẽ cố gắng ảnh hưởng càng ít đến hệ thống càng tốt. Chú trọng đến chống lỗi, phục hồi khi có lỗi, đến sự an toàn của toàn bộ hệ thống

- Hoạt động phục hồi sẽ giới hạn với thành phần bị lỗi, mà không bị ảnh hưởng đến các thành phần khác. Khi có một tiến trình bị lỗi xảy ra thì các tiến trình khác vẫn hoạt động bình thường, và không biết tiến trình này bị lỗi.

Việc truy cập vào bộ chứa ổn định sẽ được xem là nguyên nhân ảnh hưởng đến hiệu năng của hệ thống, với chi phí khá cao, do đó nên cân bằng giữa các yêu cầu đê xác định phương án phù hợp. Phải xác định rằng có khả năng tạo ra các tiến trình mồ côi (Orphan processes), do đó sẽ tăng thêm chi phí cho hệ thống phục hồi.

Ngoài ra, mô hình này nhằm giúp ta có thể hiểu rằng mô hình chống lỗi cần phải có những hoạt động, chức năng gì cần thiết để chống lỗi. Điều này sẽ giúp ta có thể hiểu dễ dàng hệ thống trong thực tế hơn.

4.3.1.2. Giải pháp

Để lưu các sự kiện không xác định, ta tạo ra một file để ghi chứa đủ thông tin về các sự kiện, lưu chúng lại một chỗ không bị ảnh hưởng khi có lỗi. Ở trong thiết kế này gọi là thực thể *Bộ chứa ổn định* , dùng để lưu các sự kiện không xác định, đảm bảo việc ghi các sự kiện không xác định trước khi thực hiện

Thực thể *Ghi sự kiện* được thiết kế có trách nhiệm cho việc ghi các hoạt động của thực thể *tiến trình có thể phục hồi*. *Ghi sự kiện* tạo các bản ghi tương ứng cho các sự kiện không xác định và chuyển chúng vào *bộ chứa ổn định*

Nhiệm vụ của *Quản lý phục hồi* : điều khiển toàn bộ việc phục hồi, khi lỗi được báo cáo xảy ra ở thực thể *tiến trình có thể phục hồi*, Thực thể *quản lý phục hồi* sẽ chỉ thị các thành phần từ bỏ trạng thái có lỗi và nạp trạng thái không có lỗi (đã biết) từ bộ chứa ổn định . Sau đó quản lý phục hồi phát lại cục bộ đến các thành phần phục hồi tất cả các sự kiện không xác định được ghi vào *bộ chứa ổn định*

Quét lỗi (error detector) chịu trách nhiệm về việc quét lỗi có thể xảy ra với *tiến trình có thể phục hồi*, khi lỗi được tìm thấy, sẽ thông báo cho *quản lý phục hồi* biết về chúng

Thực thể *Tiến trình có thể phục hồi* là thành phần ghi các sự kiện không xác định khi thực hiện không có lỗi để có thể phát lại chúng khi hệ thống phục hồi

4.3.1.3. Thiết kế

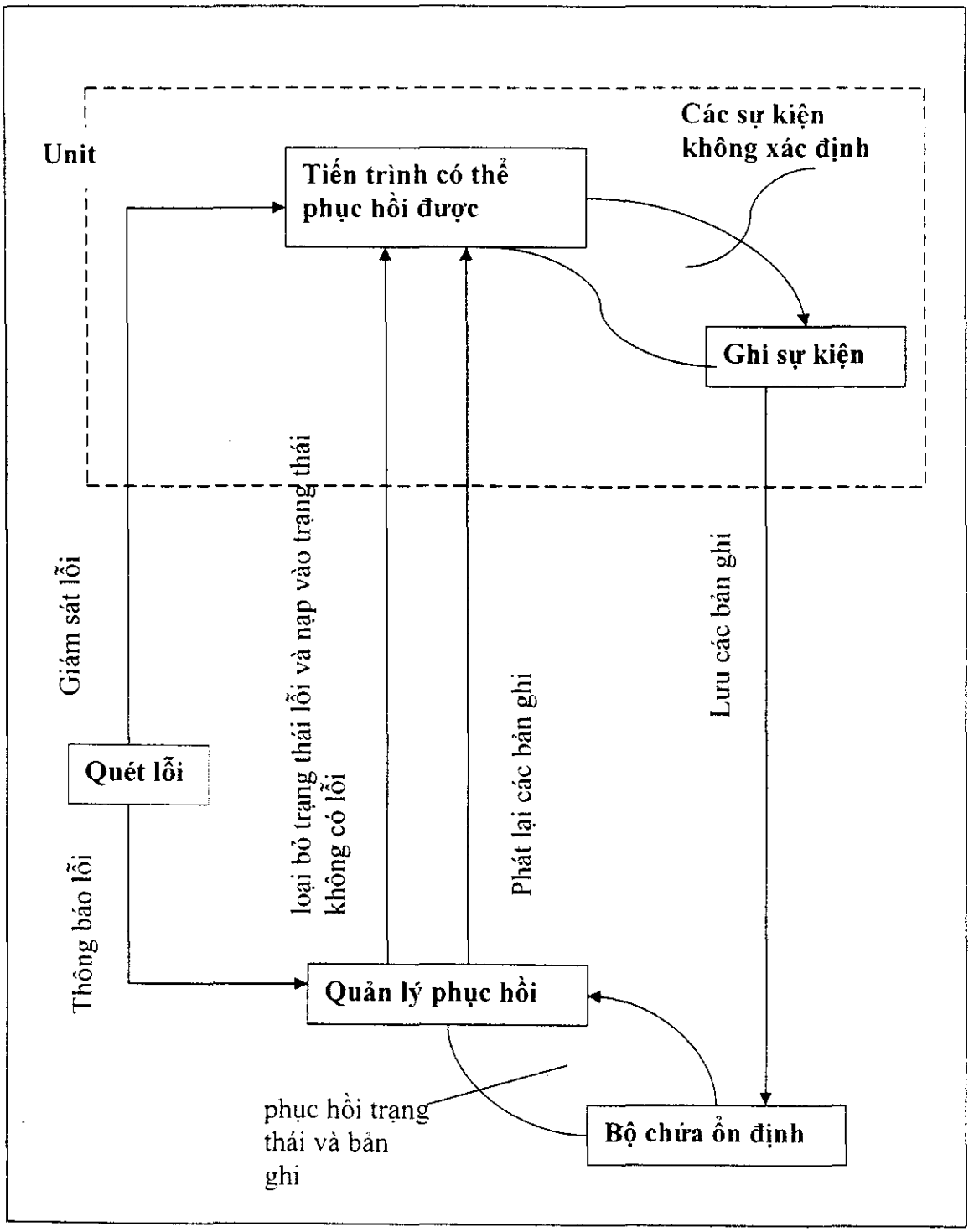
Khi thiết kế mô hình ghi thông điệp Pessimistic trong hệ thống, cần phải có nhiều *tiến trình có thể phục hồi* như là số các thành phần phải phục hồi. Hơn nữa, *Ghi sự kiện* được gắn liền với *tiến trình có thể phục hồi* thành 1 khối, điều này cần thiết để đảm bảo khi 2 thực thể đó lỗi, hoạt động phục hồi sẽ được bắt đầu. *Quét lỗi, bộ chứa ổn định, tiến trình có thể phục hồi* không gắn cùng với một khối *tiến trình có thể phục hồi*, và không ảnh hưởng tới việc xảy ra lỗi.

Trong thiết kế này thì có thể thấy *quản lý phục hồi* và *bộ chứa ổn định* là 2 thực thể duy nhất trong hệ thống, trong khi đó có thể có nhiều *Ghi sự kiện, quét lỗi, tiến trình có thể phục hồi.*

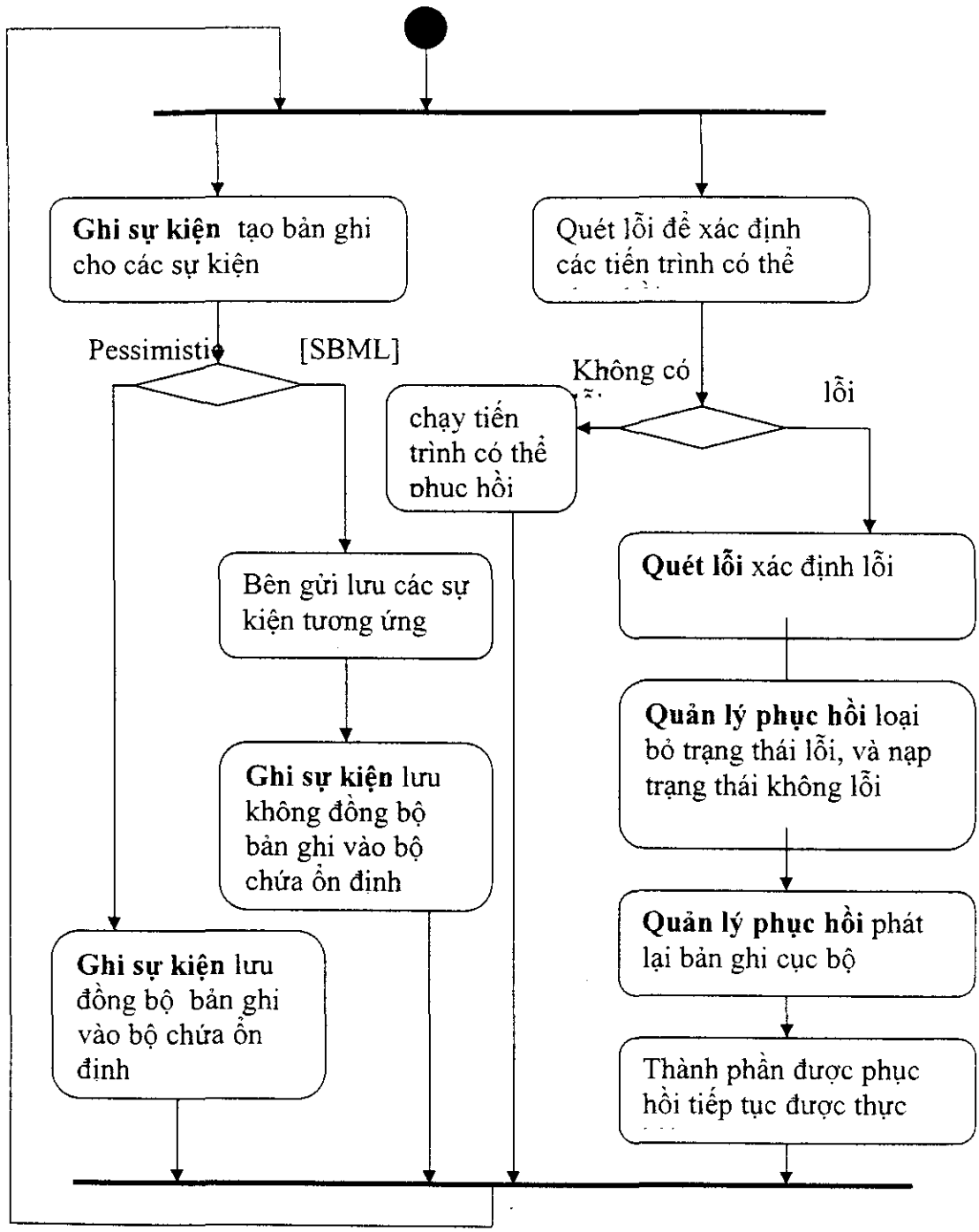
Các thông điệp đã gửi để phục hồi các thành phần phải được thiết kế trễ hoặc vào vùng đệm cho đến khi phục hồi hoàn thành, và các thành phần có thể nhận và xử lý các thông điệp.

Khi một thông điệp được gửi đi, thực thể tiến trình có thể phục hồi được sẽ ghi các sự kiện không xác định, *Ghi sự kiện* sẽ ghi các hoạt động của tiến trình sau đó các thông tin được ghi sẽ được gửi đến bộ chứa ổn định để lưu trữ. Bộ chứa ổn định này luôn đảm bảo được sự an toàn cao, để khi xảy ra lỗi thì có thể khôi phục dữ liệu chứa trong đó. Nếu như không xảy ra lỗi, thì các thông điệp tiếp tục được gửi đi, tiến trình có thể phục hồi tiếp tục ghi các sự kiện, *Ghi sự kiện* tiếp tục ghi lại hoạt động của các tiến trình. Ở đây, trong mô hình lý thuyết, ta chưa đặt ra điều kiện khi nào thì loại bỏ các thông tin thừa trong bộ chứa ổn định, còn trong thực tế trình bầy ở phần sau, ở trong chương trình ứng dụng MPICH-V1 thì ta có thể loại bỏ thông tin thừa, ngay sau khi tạo lưu trữ thành công trạng thái của tiến trình tại điểm ảnh gần nhất. Trong trường hợp xảy ra lỗi, thực thể quét lỗi sẽ phát hiện ra lỗi, khi đó thực thể quét lỗi sẽ thông báo ngay cho thực thể quản lý phục hồi, để thực thể này bỏ trạng thái có lỗi, và nạp vào trạng thái không có lỗi (được lấy ra từ

điểm ảnh lưu trữ trong bộ chứa ổn định). Đồng thời thực thể quản lý phục hồi lấy các thông điệp được ghi trong bộ chứa ổn định, phát lại đến tiến trình có thể phục hồi được để thực thể này có thể thực hiện lại các hoạt động như khi chưa gặp lỗi.



Hình 4-11 Cấu trúc của ghi thông điệp Pessimistic



Hình 4-12 Biểu đồ hoạt động của ghi thông điệp Pessimistic

4.3.1.4. Kết quả

Với mô hình thiết kế trên ta thấy rằng mô hình đưa ra các thực thể để có thể ghi lại các thông điệp gửi, nhận, có cơ chế quét lỗi phát hiện ra lỗi, có cơ chế quản lý tổng thể, để quản lý việc phục hồi lỗi, có thể đáp ứng được việc phục hồi lỗi khi xảy ra lỗi trong hệ thống.

Khi lỗi xảy ra, việc phục hồi sẽ tiến hành ở tiến trình bị lỗi, chứ không phục hồi toàn bộ hệ thống. Thực thể *Quản lý phục hồi* sẽ làm nhiệm vụ phục hồi tiến trình lỗi này. Điều này khác hẳn với checkpoint-based, là tất cả các thành phần trong hệ thống sẽ liên quan đến hoạt động phục hồi, và sẽ kiểm tra toàn bộ các checkpoint của các thành phần trong hệ thống để đảm bảo cho trạng thái của hệ thống được thích hợp.

Tuy nhiên, ta cũng sẽ gặp một số vấn đề

Khi thiết kế mô hình này, có một vấn đề đặt ra đó là: do cơ chế ghi đồng bộ trong pessimistic gắn liền với bộ chứa ổn định , do vậy khi có nhiều tiến trình hoạt động, cùng ghi vào bộ chứa ổn định, sẽ gây ra hiện tượng quá tải nghẽn cổ chai (bottle-neck)tại bộ chứa ổn định. Đây là vấn đề rất đáng quan tâm, vì một chương trình trong thực tế luôn đòi hỏi chạy ổn định với hiệu năng cao.

Hiệu năng bị ảnh hưởng trực tiếp khi tiến hành ghi vì nó đòi hỏi sẽ có 2 giai đoạn commit do việc ghi 2 lần: tại phía gửi và tại phía nhận sự kiện truyền thông. *Ghi vào thực thể*: *Ghi sự kiện*, và ghi vào thực thể *bộ chứa ổn định*.

4.3.2. Kiến trúc thư viện MPICH_V1

4.3.2.1. Tổng quan

Ở phần trước ta đã đưa ra một mô hình dựa trên lý thuyết của phương pháp ghi thông điệp Pessimistic. Ở phần này, ta sẽ xét đến một gói mã nguồn mở MPICH-V1, nghiên cứu kiến trúc và cơ chế chống lỗi theo phương pháp ghi thông điệp pessimistic

Các hệ cluster và hệ TeraGRID trong tương lai với hàng ngàn nút cho việc các ứng dụng khoa học tính toán song song. Ở đây, việc nút bị lỗi hoặc bị ngắt kết nối là sự kiện xảy ra thường xuyên. Yêu cầu đặt ra rằng chúng ta phải có một môi trường truyền thông ổn định, và một cơ chế chống lỗi thích hợp để đảm bảo rằng các hệ thống luôn luôn hoạt động tốt. Hiện nay, đang có một hướng nghiên cứu áp dụng môi trường truyền thông theo chuẩn MPICH vào trong thực tế. Phiên bản phát triển MPICH-V1, được đưa ra, giúp mọi người trên toàn thế giới cùng nhau nghiên cứu, với mục đích đưa đến một phiên bản hoàn thiện nhằm đảm bảo được về mặt truyền thông, và có khả năng chống lỗi.

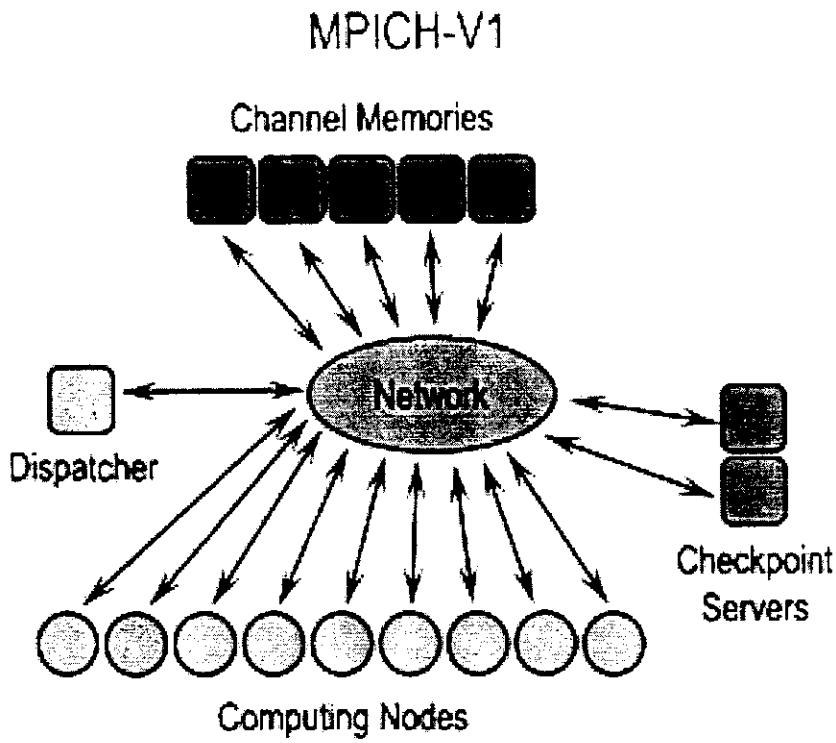
Môi trường MPICH-V1 bao gồm các thư viện truyền thông trên cơ sở MPICH và một môi trường runtime. Thư viện MPICH-V1 có thể liên kết với các chương trình MPI đã có như thư viện MPI bình thường. Điều này sẽ giúp ích rất nhiều cho chúng ta khi muốn nâng cấp lên một phiên bản mới với những cải tiến mới.

Thư viện thực hiện truyền thông tất cả các phần tử con được cung cấp bởi MPICH1. Nó được thiết kế theo kiến trúc tầng (layer), các điều khiển truyền thông phía dưới riêng biệt sẽ được gói gọn trong một gói phần mềm gọi là *device*. Từ device tất cả các hàm MPI được tự động xây dựng lại bởi hệ biên dịch MPICH1. Thư viện MPICH1 được xây dựng ở device mức cao đảm bảo MPICH-V1,2,3 đầy đủ, thực hiện hàm truyền thông cấp độ Chameleon-level (luôn thích ứng với mọi trường hợp). Tầng truyền thông phía dưới sẽ nhờ TCP để đảm bảo toàn vẹn thông điệp. MPICH-V1 runtime là môi trường phức tạp có rất nhiều các thực thể: Dispatcher (Bộ điều phối), Channel Memories (Kênh truyền thông), Checkpoint Servers, các nút tính toán (như hình)

Trong phiên bản phát triển MPICH-V1 này đã có sự thay đổi theo mô hình thiết kế ban đầu. Ở đây Bộ điều phối đã làm nhiệm vụ của thực thể Quét lỗi và quản lý phục hồi. Channel Memories đảm nhiệm việc ghi các thông điệp được gửi từ các nút- của thực thể tiến trình có thể phục hồi được và ghi sự kiện. Còn Checkpoint

Server là kho chứa chứa dữ liệu. Chúng ta sẽ đi sâu hơn vào kiến trúc của MPICH-V1 trong phần dưới đây.

Hình dưới đây mô tả tổng hợp kiến trúc của MPICH-V1



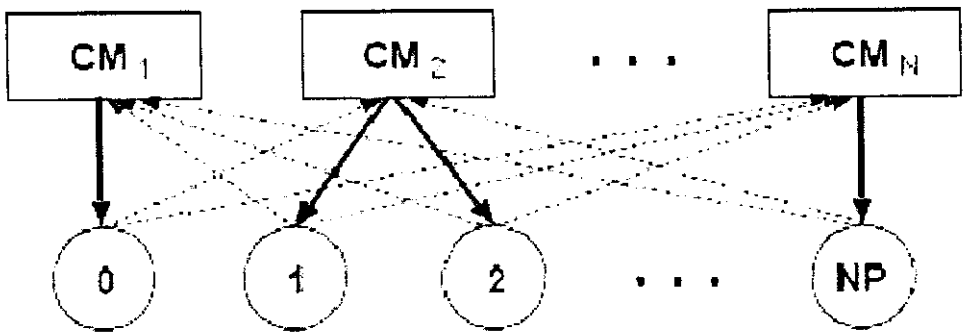
Hình 4-13 Kiến trúc MPICH-V1

Kiến trúc này đưa ra yêu cầu kết nối ứng dụng với libmpichv thay vì libmpich chuẩn

MPICH_V1 dựa chủ yếu vào khái niệm Kênh truyền thông để đảm bảo chống lỗi. Kênh truyền thông cung cấp dành cho các nút dịch vụ của thông điệp ngầm và kho chứa. Chống lỗi được thực hiện với sự phân quyền cao, lưu giữ các tính toán và phụ thuộc ngữ cảnh truyền thông. Với mỗi nút, ngữ cảnh thực hiện sẽ được lưu (một cách định kỳ hoặc ngay sau khi nhận) trên một Checkpoint Server ở xa. Ngữ cảnh truyền thông được chứa trong suốt quá trình thực hiện bằng cách lưu trữ các

in-transit message trong Kênh truyền thông. Vì vậy toàn bộ ngữ cảnh thực hiện parallel được lưu và chứa theo những cách khác nhau

MPICH_V1 runtime phân công Kênh truyền thông đến nút bằng cách kết hợp với mỗi Kênh truyền thông với tập nút khác nhau. Theo cách này, một nút luôn có một *home Kênh truyền thông* với tập các nút khác nhau. nhận các thông điệp đó từ Kênh truyền thông này. Khi một nút gửi một thông điệp, nó sẽ đưa thông điệp đến Kênh truyền thông home nhận. Mỗi một nút có một Kênh truyền thông riêng



Hình 4-14 Kênh truyền thông riêng

Kiến trúc này sẽ giúp giới hạn số thông điệp thực hiện không rõ ràng (nhận từ bất cứ đâu) đến các nút.

Bộ điều phối làm nhiệm vụ quản lý toàn bộ các hoạt động trong hệ thống, Bộ điều phối có nhiệm vụ khởi tạo Channel Memory, Checkpoint Server, thiết lập kết nối từ các nút đến Bộ điều phối, đưa các yêu cầu đến Checkpoint Server, Channel Memory để các thành phần này thực thi nhiệm vụ. Bộ điều phối luôn theo dõi các tín hiệu báo về về tình trạng của các nút còn sống hay đã chết, và có trách nhiệm khởi tạo một nút mới một cách tự động, có nghĩa là sẽ sinh ra một nút khác, là bản sao của nút chết, chạy đến điểm bị lỗi, và tiếp tục hoạt động tiếp trong khi các nút khác không phát hiện ra việc có lỗi tại nút này. Cuối cùng, nếu như nút chết tiếp tục hoạt động, ta sẽ có 2 nút cùng chạy 1 hoạt động, gồm nút chết và nút vừa sinh

ra để chống lỗi. Do đó Bộ điều phối có nhiệm vụ huỷ đi một nút, đảm bảo chỉ có một nút chạy 1 hoạt động trong cùng một thời gian.

Cấu trúc của các nút tính toán:

```
typedef struct computingNode {
  char    hostName[64];
  char    ipAddress[15];
  char    channelMemory[15];
  int     channelMemoryPort;
  char    checkpointServer[15];
  int     checkpointServerPort;
} CN;  /* as in 'Computing Node */
```

Cấu trúc của Channel Memory:

```
typedef struct channelMemory {
  char    ipAddress[15];
  int     port;
  int     dispatcherPort;
  int     threads;
} CM; /* Channel Memory */
```

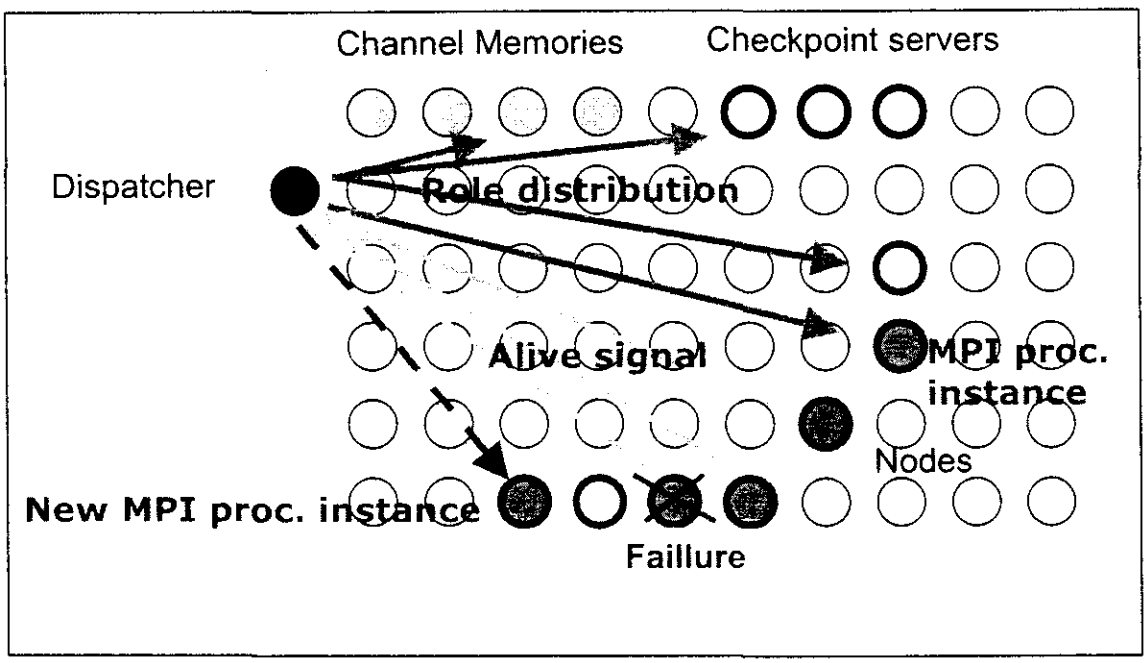
Cấu trúc của Checkpoint Server :

```
typedef struct checkpointServer {
  char    ipAddress[15];
  int     port;
  char    tmp[128];
} CS; /* Checkpoint server */
```

4.3.2.2. Bộ điều phối

Bộ điều phối có nhiệm vụ quản lý tài nguyên thực hiện các chương trình song song, quản lý các instance của các tiến trình MPI truyền thống. Trong suốt quá trình khởi tạo, Bộ điều phối khởi tạo một tập các nhiệm vụ của các nút tham gia, mỗi nhiệm vụ khởi tạo một tiến trình MPI. Nhiệm vụ chính của Bộ điều phối là điều phối nguồn tài nguyên cần thiết để thực hiện một chương trình song song

Bao gồm: Khởi động Channel Memory, Check point Server, cung cấp nút cho việc thực hiện dịch vụ (Kênh truyền thông và Checkpoint Server) và khởi tạo tiến trình MPI trên nút, quản lý trạng thái các nút, bằng cách theo dõi tín hiệu time-out, có vai trò lập lịch cho các nút hiện có để thay các nút bị chết...



Hình 4-15 Bộ điều phối

Kết nối những dịch vụ đang chạy, tất cả những dịch vụ được lập lịch như nhiệm vụ bình thường

Khi lập lịch, một dịch vụ được ghi trong Stable Service Registry, quản lý một cách tập trung.

1) Khi một nút chạy MPI process . nó phải tra cứu Registry để tìm ra địa chỉ CS và CM liên quan

2) Tra cứu này dựa vào rank trong nhóm tiến trình MPI liên quan

3) Nói chung các CM và CS được gán cho các nút vào thời điểm bắt đầu thực thi chương trình song song

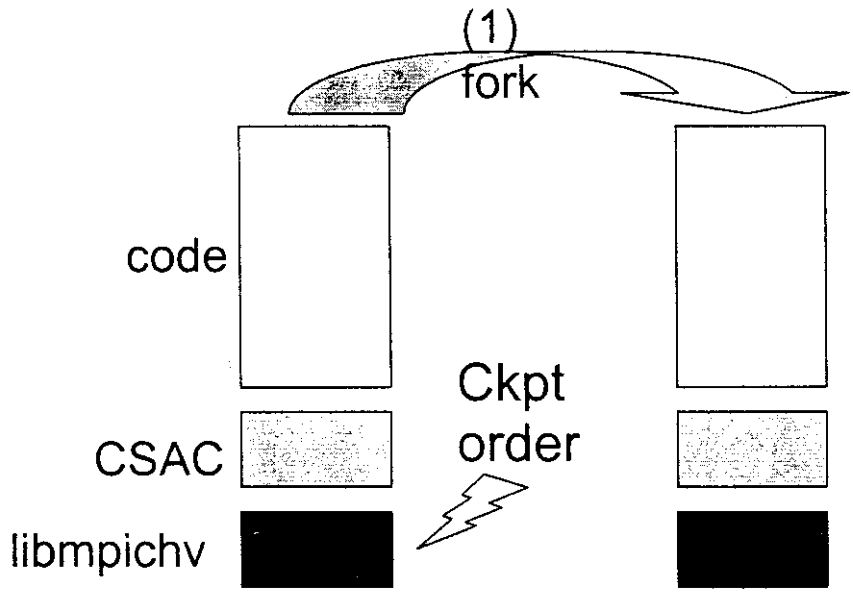
Trong bản ghi này, mỗi dịch vụ có một tập các host cung cấp dịch vụ này và thông tin giao thức được sử dụng để kết nối người yêu cầu dịch vụ đến host dịch vụ. Khi một nút thực hiện một ứng dụng MPI, đầu tiên nó liên lạc với Service Registry để xác định địa chỉ của Checkpoint Server và mối quan hệ với Channel Memory tuỳ theo các nút rank trong nhóm truyền thông MPI.

Trong suốt quá trình thực hiện, mọi nút tham gia gửi một cách định kỳ tín hiệu "alive" đến Bộ điều phối. Tín hiệu "alive" này chỉ chứa số MPI rank. Theo trình tự được đặt 1 phút, sẽ điều chỉnh sự cân bằng giữa việc tác động Bộ điều phối và sự tắc nghẽn truyền thông. Bộ điều phối giám sát các nút tham gia bằng cách theo dõi tín hiệu alive. Theo dõi các lỗi tiềm năng khi mà tín hiệu alive bị time out. Bộ điều phối không quan tâm đây là do nút lỗi hay do mạng lỗi. Sau đó nó sẽ khởi tạo task khác (một instance của tiến trình tương tự). Task này khởi động lại công việc thực hiện, đi đến điểm xảy lỗi và tiếp tục thực hiện từ điểm này. Mọi nút khác không nhận biết được lỗi. Nếu một nút lỗi kết nối trở lại hệ thống, 2 instance của cùng một tiến trình MPI (2 clone – 2 cái giống nhau) không thể chạy tại một thời điểm trong hệ thống. Kênh truyền thông quản lý trường hợp này chỉ cho phép một kết nối trên một MPI rank. Khi một nút rời hệ thống (ngắt kết nối hoặc lỗi), nó sẽ dừng kết nối thường xuyên của nó với Kênh truyền thông. Khi một nút cố gắng kết nối với Kênh truyền thông, Kênh truyền thông kiểm tra MPI rank, và trả lại mã lỗi cho việc thăm dò lỗi. nó sẽ dừng job hiện tại và liện hệ lại với Bộ điều phối cho việc tìm nạp một job mới. Vì vậy chỉ nút đầu tiên trong clone sẽ có thể mở kết nối với Kênh truyền thông

4.3.2.3. Nút tính toán

Khi một tiến trình gặp lỗi, sẽ có cơ chế sinh ra một tiến trình mới, để thực hiện việc phục hồi. Vấn đề checkpoint do thư viện Condor đảm trách.

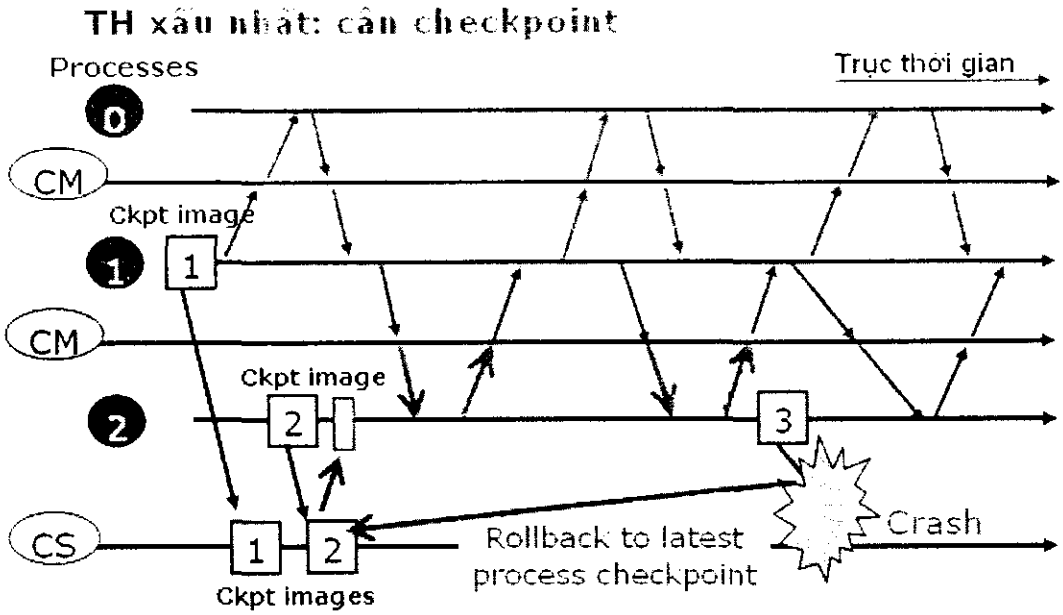
Ảnh checkpoint phải được gửi tới Checkpoint Server, chứ không lưu trữ trên máy cục bộ. Ngoài ra, ta cần chú ý rằng việc checkpoint là do bản thân các nút



Hình 4-16 Cơ chế fork

tự checkpoint chứ không phải do Bộ điều phối thực hiện.

Hình dưới đã mô tả việc checkpoint của các tiến trình. Các tiến trình khi có lỗi, thì sẽ thực hiện việc checkpoint, gửi ảnh về Checkpoint Server. Khi tiến trình 2 gặp lỗi, thì tiến trình 2 sẽ quay lại checkpoint gần nhất, lấy điểm ảnh đó trên Checkpoint Server để phục vụ cho việc phục hồi.

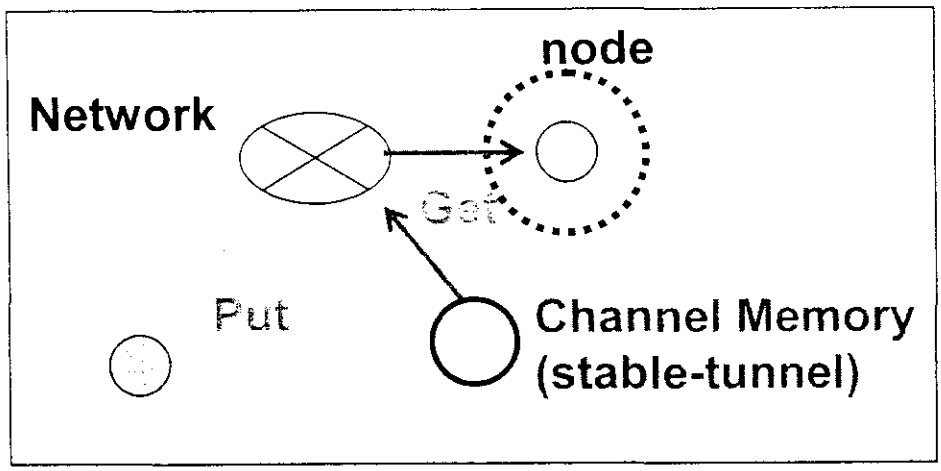


Hình 4-17 Quá trình phục hồi

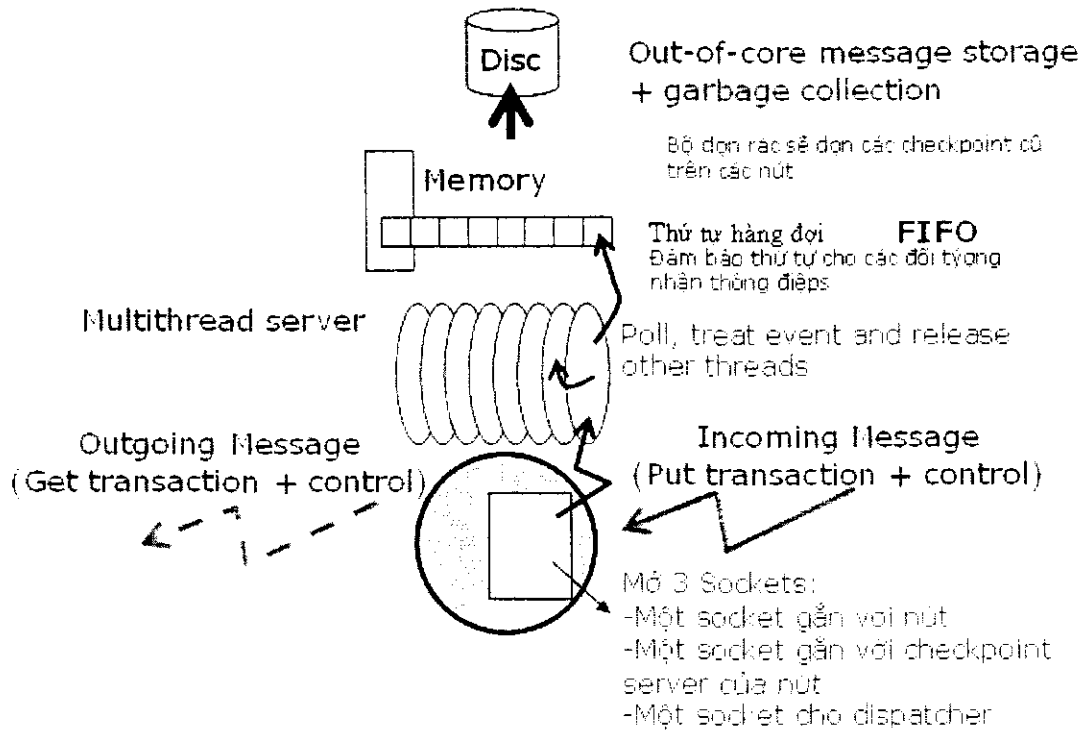
4.3.2.4. Kiến trúc kênh truyền thông (Channel Memory)

Kênh truyền thông là kho chứa ổn định, vững chắc cho truyền thông thông điệp giữa các nút Mục đích chính của nó là để lưu trữ truyền thông trong khi thực hiện để thực hiện các nhiệm vụ. Mọi nút liên hệ với Kênh truyền thông server kết hợp với ứng dụng MPI theo cách mà Service Registry cung cấp bởi Bộ điều phối. Để gửi (nhận) một thông điệp đến (từ) một nút khác, các nút truyền thông gửi, hoặc nhận thông điệp MPI thông qua Kênh truyền thông transaction. Mọi truyền thông giữa các nút và Kênh truyền thông sẽ được bắt đầu bằng nút.

Các nút có độ tin cậy cao sẽ được dùng làm "Kênh truyền thông".Nó sẽ ghi lại tất cả các thông điệp. Các sự trao đổi giữa các node chỉ cần thực hiện qua 2 lệnh PUT và GET tới CM. PUT và GET là các giao dịch. Khi tiến trình khởi động lại , nó sẽ thực hiện lại các trao đổi thông điệp trước đó bằng cách sử dụng CM. Hiện nay CM lưu và phân phối các thông điệp theo FIFO



Hình 4-18 Giao tiếp kênh truyền thông với nút



Hình 4-19 Thực hiện kênh truyền thông

Kênh truyền thông là multi-thread server với một fixed pool of thread, cả 2 cùng nghe kết nối, và xử lý các yêu cầu từ nút, Checkpoint Server và Bộ điều phối. Để đảm bảo thứ tự của thông điệp được nhận, Kênh truyền thông quản lý tất cả các thông điệp như tập các hàng đợi FIFO.

Theo Kênh truyền thông, mỗi bên nhận là sự kết hợp với hàng đợi chuyên dụng. Để thực hiện yêu cầu với hàm MPICH cấp độ cao, Kênh truyền thông quản lý 2 loại queue: một queue phổ biến cho tất cả các thông điệp điều khiển, và một queue cho một sender cho mỗi thông điệp dữ liệu. Mỗi kiến trúc của kho chứa dữ liệu cho phép được tối ưu hoá thời gian để nhận các thông điệp

Như đã trình bầy ở trên thì mỗi nút được gắn với một home CM, mỗi nút chỉ nhận thông điệp từ chính Channel Memory home của mình, nếu như các nút muốn gửi thông điệp tới các nút khác thì phải gửi tới home Channel Memory của nút nhận, chứ không gửi trực tiếp đến nút đó. Các nút muốn liên lạc với Channel Memory thì phải thông qua SSR – Stable Service Registry)

Channel Memories Device

MPICH_V Channel Memories device thực hiện thư viện MPI trên cơ sở MPICH cho các nút tham gia. Device, gọi là ch_cm, được nhúng vào trong các phân phối MPICH chuẩn và được thực hiện trong giao thức TCP

Chức năng của các device là hỗ trợ các hàm việc khởi tạo, truyền thông và kết thúc của tầng cao hơn tuỳ theo đặc điểm của MPICH-V. Tại thời điểm khởi tạo, khi một ứng dụng song song đã được gửi đi, device cung cấp kết nối của ứng dụng với Kênh truyền thông server và Checkpoint Server. Việc thiết lập kết nối với Kênh truyền thông server khi TCP socket được mở kéo dài cho đến khi ứng dụng kết thúc tất cả các truyền thông.

Việc thực hiện của tất cả các chức năng truyền thông bao gồm gửi và nhận các thông điệp đặc biệt của hệ thống, được sử dụng cho sự chuẩn bị phần truyền thông chính và xác định loại truyền thông cho cả nút và cho phần Kênh truyền thông server.

Việc kết thúc ứng dụng bao gồm thông báo ngắt kết nối đến tất cả các Kênh truyền thông

4.3.2.5. Checkpoint Server

Checkpoint Server có nhiệm vụ chứa và cung cấp các ảnh của các tiến trình trên các nút yêu cầu. MPICH giả định rằng

1) Checkpoint Server là kho chứa và không được bảo vệ bằng fire-wall.

2) truyền thông giữa nút và Checkpoint Server là transaction

3) Mọi truyền thông giữa nút và Checkpoint Server tại lúc bắt đầu của nút.

Khi thực hiện, mọi nút thực hiện checkpoint và gửi chúng đến một Checkpoint Server. Khi một nút khởi động lại việc thực hiện của một nhiệm vụ, Bộ điều phối cho biết về:

1) Nhiệm vụ được khởi động lại

2) Checkpoint Server liên hệ để lấy checkpoint cuối cùng. Các nút đưa ra yêu cầu đến Checkpoint Server thích hợp sử dụng MPI rank và id ứng dụng. Checkpoint Server trả lời yêu cầu bằng cách gửi trở lại checkpoint cuối cùng tương ứng với MPI rank.

Checkpoint có thể thực hiện định kỳ hoặc vào lúc nhận tín hiệu bên ngoài. Thực tế hiện nay, việc quyết định khởi tạo một checkpoint là lấy định kỳ trên nút cục bộ, mà không cần điều phối trung tâm.

Hiện nay chúng ta sử dụng Condor Stand-alone Checkpointing Library (CSCL), với checkpoint là một phần của một tiến trình, và cung cấp tuỳ chọn nén ảnh tiến trình. CSCL không cung cấp các cơ chế để điều khiển truyền thông giữa các nút. Thực tế, socket được sử dụng bởi một tiến trình checkpoint không mở lại sau khi khởi động lại. Để làm việc với giới hạn này thì khi một tiến trình khởi động lại , MPICH-V phục hồi kết nối với Kênh truyền thông trước khi tiếp tục thực hiện

Để cho phép chồng lên giữa việc tính toán checkpoint và mạng truyền ảnh các tiến trình, ảnh checkpoint không bao giờ được chứa trên nút cục bộ. Việc sinh ra, và việc nén lại các ảnh checkpoint, và truyền đến checkpoint server được thực

hiện trong 2 tiến trình tách riêng. Kích thước của ảnh không tính toán trước khi gửi dữ liệu. vì vậy chúng ta bắt đầu truyền ngay khi thư viện CSCL bắt đầu sinh ra dữ liệu checkpoint. Giao thức truyền tin cậy vào mạng thực hiện TCP/IP để có thể quét lỗi.

Khi khởi động lại một tiến trình từ ảnh checkpoint của nó. thông điệp không phát lại từ điểm bắt đầu. vì vậy Checkpoint Server và Kênh truyền thông kết hợp với một nhiệm vụ phải được điều phối để đảm bảo khởi động lại một cách chặt chẽ .

Tiến trình checkpoint là một chương trình đòi hỏi được dừng việc thực hiện của nó. Tuy nhiên, sinh, nén và gửi ảnh checkpoint có thể cần một thời gian dài. Để giảm thời gian chết, MPICH_V thực tế checkpoint một bản sao của nhiệm vụ đang chạy. Tiến trình sao này được tạo ra sử dụng fork chuẩn. Hệ thống này gọi bản sao tất cả cấu trúc bộ nhớ và các cờ tiến trình, các file trạng thái mô tả, được thừa kế thêm socket đã mở. Khi tiến trình có cấu trúc dữ liệu tương tự và trạng thái tương tự như tiến trình gốc, chúng ta sử dụng nó để thực hiện checkpoint khi tiến trình gốc có thể tiếp tục thực hiện

Trong modul Checkpoint có các file chính:

- XWCheckpointServer.c
- XWCheckpointServerProtocol.h

Modul checkpoint có nhiệm vụ sinh ra tiến trình con để thực hiện các nhiệm vụ GET_CHECKPOINT , PUT_CHECKPOINT

Khi thực hiện Get_checkpoint: nếu có tín hiệu yêu cầu từ client, Checkpoint Server sẽ thực hiện các nhiệm vụ: mở file checkpoint ra, đọc nội dung trong file checkpoint đưa ra buff của socket đang kết nối, để gửi lại ảnh checkpoint cho client, phục vụ cho phục hồi

Khi thực hiện Put_checkpoint: Công việc ghi checkpoint thực hiện thông qua Channel Memory, Channel Memory kết nối với Server, gửi jobID, rankID, gửi tín hiệu ACK để báo cho Server biết là đang thực hiện công việc. Server sẽ ghi lại checkpoint ra một file tạm thời oldfile, sau khi ghi xong sẽ đổi tên file đó thành tên một file mới, khi đó sẽ tránh trường hợp đang lúc ghi checkpoint vào file oldfile thì bị lỗi, khi đó vẫn còn 1 checkpoint để đảm bảo có thể phục hồi được. Sau khi ghi thành công, nó sẽ xoá file checkpoint oldfile đi, khi đó chỉ tồn tại có một file checkpoint của client.

Chi tiết:

Đầu tiên, CS gọi hàm

```
void parseArgs(int argc,char *argv[])
{
jobId = atoi(argv[1]);
xwtmp = argv[2];
if(argc >= 4) hport = atoi(argv[3]);
if(argc >= 5) chkptpath = argv[4];
if(argc > 5) {printf("Usage : %s jobId xwtmp [port
[chkptfilespath]]]\n",
argv[0]);exit(-1);}
 return;
}
```

để lấy jobID, lấy số hiệu cổng hport , xwtmp, thư mục lưu checkpoint chkptpath trong đó, mặc định để PORT = 4001 ; CHKPTPATH = /tmp.

Sau đó ghi ra file với tiến trình có thể thực hiện PID.

Mở Socket theo AF_INET sử dụng địa chỉ IP để xác định kết nối

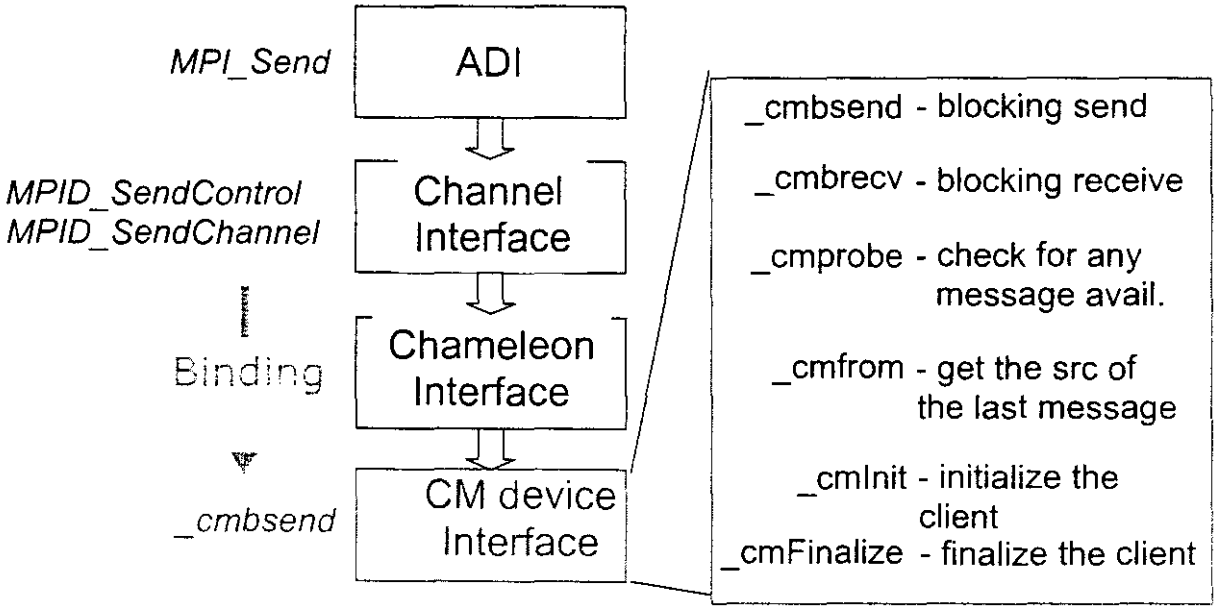
Đặt tên cho socket, gọi hàm bind với tên chính là số hiệu cổng (sin_port) trong cấu trúc sockaddr_in .

Socket sẽ tạo hàng đợi lưu các kết nối chưa được xử lý bằng hàm listen (se,5) với số kết nối tối đa được phép đưa vào hàng đợi là 5

Lúc này đã xong các thủ tục. Server đã sẵn sàng, đang chờ cổng kết nối tới.Nếu như có kết nối, nó sẽ tiếp tục thực hiện công việc với hàm traiter_message. Tuỳ theo yêu cầu của Client gửi tới là checkpoint hay Get point mà Server sẽ gọi hàm thích hợp.

4.3.2.6. Thư viện

Thư viện dùng trong MPICH-V1 dựa trên chuẩn MPICH, sử dụng device "ch_xw". Tất cả các hàm trong device "ch_xw" đều truyền thông trên cơ sở TCP



Hình 4-20 Thư viện MPI

Giao thức MPICH-V:

Thuật toán của Checkpoint Server và Worker W

Nhận checkpoint từ W → lưu checkpoint

Gửi checkpoint cuối cùng đên CM

Nhận tín hiệu khởi động lại → gửi tín hiệu checkpoint thành công cuối cùng đến W

Thuật toán của Channel Memory và Worker W

Nhận các sự kiện không xác định từ W→

Nếu (không có sự kiện nào truyền đi)

replay ← False

Nếu (replay)

Tìm các sự kiện kế tiếp

Gửi sự kiện kế tiếp đến W

Ngược lại

Chọn sự kiện có thể

Thêm sự kiện vào hàng đợi của sự kiện truyền

Gửi sự kiện đến W

Nhận thông điệp m đến W từ W' →

Thêm m vào các sự kiện có thể

Tăng lên 1 (số thông điệp nhận [W'])

Nhận tín hiệu bắt đầu checkpoint từ W →

Đánh dấu sự kiện cuối cùng vừa kết thúc như một checkpoint

Nhận tín hiệu checkpoint từ CS →

giải phóng các sự kiện đã gửi cho đến khi checkpoint ghi lại các sự kiện đó

Nhận tín hiệu khởi động từ W` ≠ W →

Gửi thông điệp [W`] đến W`

Nhận tín hiệu khởi tạo từ W →

Gửi thông điệp nhận [W] to W

Replay ← true

Thuật toán của Worker

Nhận tín hiệu/thăm dò →

gửi tín hiệu nhận/thăm dò yêu cầu CM

nhận tín hiệu trả lời của CM

phân phát tín hiệu trả lời

Gửi thông điệp m đến W' →

Nếu (nbmessage >limit)

gửi m đến Channel Memory của W'

nbmessage ← nbmessage +1

Bắt đầu Checkpoint →

Gửi tín hiệu bắt đầu checkpoint đến CM

Checkpoint và gửi checkpoint đến CS

Khởi động lại →

gửi khởi động lại đến CS

nhận Checkpoint và phục hồi

Với mỗi Channel Memory C

gửi tín hiệu khởi động đến C

nhận nbmessage

limit ← nbmessage + limit

Nhận xét

Ta thấy rằng giao thức MPICH-V1 là một giao thức chống lỗi ghi thông điệp pessimistic. Ở đây, MPICH-V1 có 2 thuộc tính của giao thức pessimistic đó là thuộc tính ghi và thuộc tính thực hiện lại trong suốt

MPICH-V1 có thuộc tính ghi: ghi lại các trạng thái của các sự kiện, và thứ tự của các sự kiện của các tiến trình. Các sự kiện này được ghi vào Kênh truyền thông cùng với thứ tự của các sự kiện. Các sự kiện này sẽ được lưu lại cho đến khi có một điểm ảnh kế tiếp được lưu lại thành công. Do đó, MPICH-V1 có thuộc tính ghi.

MPICH-V1 có thuộc tính thực hiện lại trong suốt: Channel Memory quản lý tất cả các thông điệp gửi đến các tiến trình. Khi khởi động lại tiến trình sẽ liên hệ với tất cả các Channel Memory trong hệ thống, và phục hồi số các thông điệp vừa được gửi đến bộ nhớ này. Vì vậy nó sẽ biết được số các thông điệp phát ra mà không làm ảnh hưởng đến các tiến trình không bị lỗi, cũng như các thông điệp sẽ được phát lại đến các tiến trình lỗi. Trong quá trình phát lại này, thì các hoạt động của các tiến trình vẫn tiếp tục thực hiện mà không hề biết đến tiến trình bị lỗi.Vì vậy MPICH-V1 thực hiện lại một cách trong suốt

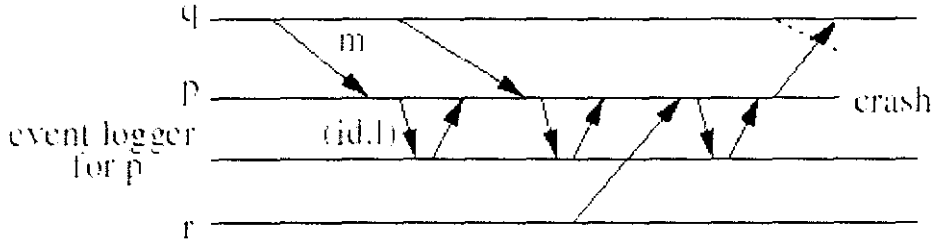
Như vậy, ta có thể thấy rằng MPICH-V1 thoả mãn các thuộc tính của giao thức chống lỗi ghi thông điệp pessimistic. Do đó, MPICH-V1 là giao thức truyền thông có áp dụng chống lỗi ghi thông điệp.

4.4. Thư viện MPICH-V2

4.4.1. Sender Based Message Logging trong MPICH-V2

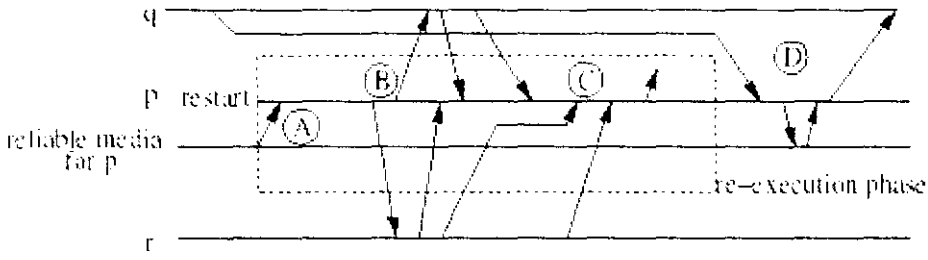
4.4.1.1. Chương trình điều khiển sự kiện theo giao thức ghi lại thông điệp

Trong giao thức này, khi một thông điệp được gửi đi thì một bản sao của nó được lưu lại trong bộ nhớ của phía bên gửi (potentially volatile), và các thông tin liên quan được ghi lại trong một phương tiện truyền thông đáng tin cậy (reliable media)



Hình 4-1 Tiến trình p trong pha thực thi bình thường

Khi một tiến trình gửi hoặc nhận một thông điệp nó tự tăng bộ đếm logic cục bộ của nó lên (chính bộ đếm này lưu giá trị SSN và RSN cho các tiến trình). Mỗi thông điệp được gửi từ tiến trình q cho tiến trình p đều có một id riêng (nhìn hình vẽ 8). Bản copy của thông điệp đó được lưu giữ trong bộ nhớ của q. Khi p nhận được thông điệp m, nó chuyển thông điệp đó cho tiến trình MPI để xử lý, sau đó nó ghi bộ thông tin (id,l) vào reliable media của nó, trong đó, l là giá trị hiện thời của bộ đếm logic cục bộ của tiến trình p (chính là giá trị RSN hiện thời của nó). Giao thức trên phải đảm bảo rằng tiến trình p không được gửi bất kỳ một thông điệp nào cho các tiến trình khác trước khi thông điệp được ghi lại hoàn toàn. Chú ý các thông điệp được gửi đi đều được đóng gói kèm theo các thông tin như đã trình bày ở trên như: SSN của tiến trình gửi, id của tiến trình gửi...



Hình 4-2 Tiến trình p trong pha khôi phục

Khi tiến trình p bị lỗi, nó được thực hiện lại từ trạng thái phía trước (lưu trong checkpoint) (nhìn hình vẽ 9). Sau đó, tập các bộ đôi (id,l) của các thông điệp mà p đã nhận sẽ được reliable media gửi lại cho p (phase A). Sau đó tiến trình p gửi yêu cầu đến tất cả các tiến trình khác yêu cầu các tiến trình này gửi lại các thông điệp cho nó kể từ thông điệp có id nhỏ nhất (phase B). Chú ý rằng các tiến trình khác không phải rolled back lại mà chúng chỉ gửi lại các bản sao thông điệp mà chúng đã ghi trong bộ nhớ cho tiến trình p như là chúng đang thực hiện bình thường vậy. Bộ đôi (id, l) và bộ đếm logic cục bộ (local logic clock) được sử dụng để xác định chính xác thông điệp nào được gửi và nhận lại theo đúng thứ tự của nó. Nếu không đúng thứ tự thì thông điệp đó sẽ bị loại bỏ và phát lại từ đầu (phase C).Sau khi nhận hết lại các thông điệp thì tiến trình p sẽ tiến tới điểm ngay trước khi bị chết và nó lại tiếp tục hoạt động bình thường . Vì lí do hiệu năng , việc lưu bản sao của mọi thông điệp trong toàn bộ quá trình thực thi là không hợp lí trong việc sử dụng lưu trữ . Ngay sau khi tiến trình được lấy checkpoint trạng thái hiện tại của nó , nó không cần các tiến trình khác gửi các thông điệp đã nhận trước khi checkpoint lần tiếp theo . Để làm điều này , giao thức cần sử dụng thêm một thành phần dọn rác để giải phóng cho các thiết bị lưu trữ các bản sao thông điệp này .

Khi một tiến trình chết , tiến trình đó sẽ được khởi động lại từ điểm checkpoint cuối cùng . Nếu lại có tiến trình khác cũng bị chết và nó khởi động lại tại cái thời điểm trước tiến trình kia , như vậy tiến trình hai này sẽ yêu cầu các thông điệp cũ từ tiến trình một .Như vậy giải pháp đặt ra là ta phải quay tiến trình đầu về trạng

thái trước khi nó phát ra các thông điệp được yêu cầu này . Tuy nhiên , việc này sẽ dẫn tới hiệu ứng lan truyền domino effect . Để tránh tình huống này , tiến trình thứ nhất phải chạy lại kèm với bản sao của các thông điệp cũ , ví vậy các thông điệp phải được gắn kèm cùng với các điểm checkpoint .

4.4.1.2. Lựa chọn thiết kế trong giao thức phục hồi

Trong tài liệu "Optimistic recovery in distributed systems (page 204-266)" có đề cập tới các đặc tính chống lỗi cần thiết cho các giao thức pessimistic message logging.Trong phần này , ta sẽ chứng minh giao thức truyền thông điệp của MPICH-V2 cũng thoả đáng các đặc tính đó. Để chứng minh giao thức của nó làm việc chính xác , ta sẽ định nghĩa một số các kí pháp sử dụng các định nghĩa cổ điển trước đây.

- Hệ thống là một tập các tiến trình , các tiến trình này liên hệ với nhau qua các liên kết truyền thông.
- Mỗi tiến trình sẽ thực hiện một giải thuật ; chúng có thể trao đổi các thông tin với các tiến trình bằng cách gửi và nhận thông điệp (hay nói đúng hơn là các dòng dữ liệu) thông qua một môi trường truyền thông .
- Ta định nghĩa tập hợp các giải thuật cho mọi tiến trình là một giao thức truyền thông . Một thao tác nguyên tố là ứng dụng của một quy tắc của giải thuật .
- Các lỗi xảy ra khi hệ thống tái tạo lại một tiến trình chết về trạng thái gần nhất có thể . Với mỗi trạng thái này , một tiến trình sẽ kết hợp với một clock logic duy nhất .
- Trạng thái tổng thể của mọi thành phần trong hệ thống gọi là cấu hình của hệ thống .
- Giao dịch là ứng dụng mô phỏng của một thao tác nguyên tố cho một tiến trình .

- Một thực thi là một chuỗi liên tiếp các giao dịch và trải qua các cấu hình khác nhau của hệ thống trong quá trình thực hiện .

Định nghĩa 1 *(Depend(m)): Gọi P là một giao thức truyền thông , và E là một thực thi trên giao thức truyền thông P . Gọi C là cấu hình trạng thái của thực thi E tại một thời điểm và m là một thông điệp của E trao đổi tại thời điểm đó . ta định nghĩa $Depend_C(m)$ là tập các tiến trình phụ thuộc vào sự nhận thông điệp m khi thực thi E đang trong trạng thái cấu hình C*

Định nghĩa *2 (Re-Executable message) : Gọi m là một thông điệp được nhận bởi tiến trình q trong thao tác nguyên tố a_m , hệ thống sẽ chuyển sang trạng thái s . Gọi c là clock logic được gắn với trạng thái s . Ta định nghĩa m là thông điệp có khả năng gửi lại cho q nếu c được ghi lại trên các thiết bị có độ tin cậy cao , và nếu thông điệp m có thể nhận được từ tiến trình gửi (sender) .*

Định nghĩa 3 *(Pessimistic Logging Protocol) : Gọi P là một giao thức truyền thông , và E là thực thi trên giao thức P với f là khả năng lỗi xảy ra đồng thời lớn nhất tại một thời điểm . Gọi M_C là tập các thông điệp trao đổi giữa các tiến trình đưa hệ thống thực thi E từ trạng thái cấu hình ban đầu tới cấu hình trạng thái C . Ta định nghĩa P là một giao thức pessimistic message logging khi và chỉ khi*

$$\forall C \in E, \forall m \in M_C,$$

$$(|Depend_C(m)| > 1) \Rightarrow Re\text{-}Executable(m)$$

Điều này có nghĩa là : nếu số tiến trình có quan hệ với thông điệp m lớn hơn 1 thì giao thức phải đảm bảo cho việc có thể gửi lại thông điệp m hơn một lần nữa.

Định lí 1 : *Gọi P là một giao thức truyền thông , và E là một thực thi của giao thức P , nếu P là pessimistic logging protocol , thì thực thi E có thể giải phóng lỗi trong quá trình thực hiện*

Định lí này đã được chứng minh ở tài liệu kể trên . Chúng ta chỉ sử dụng kết quả của nó như là nền tảng cho giao thức chống lỗi của kiến trúc MPICH-V

Định lí 2 : Giao thức cài đặt cho MPICH-V2 là pessimistic message logging protocol

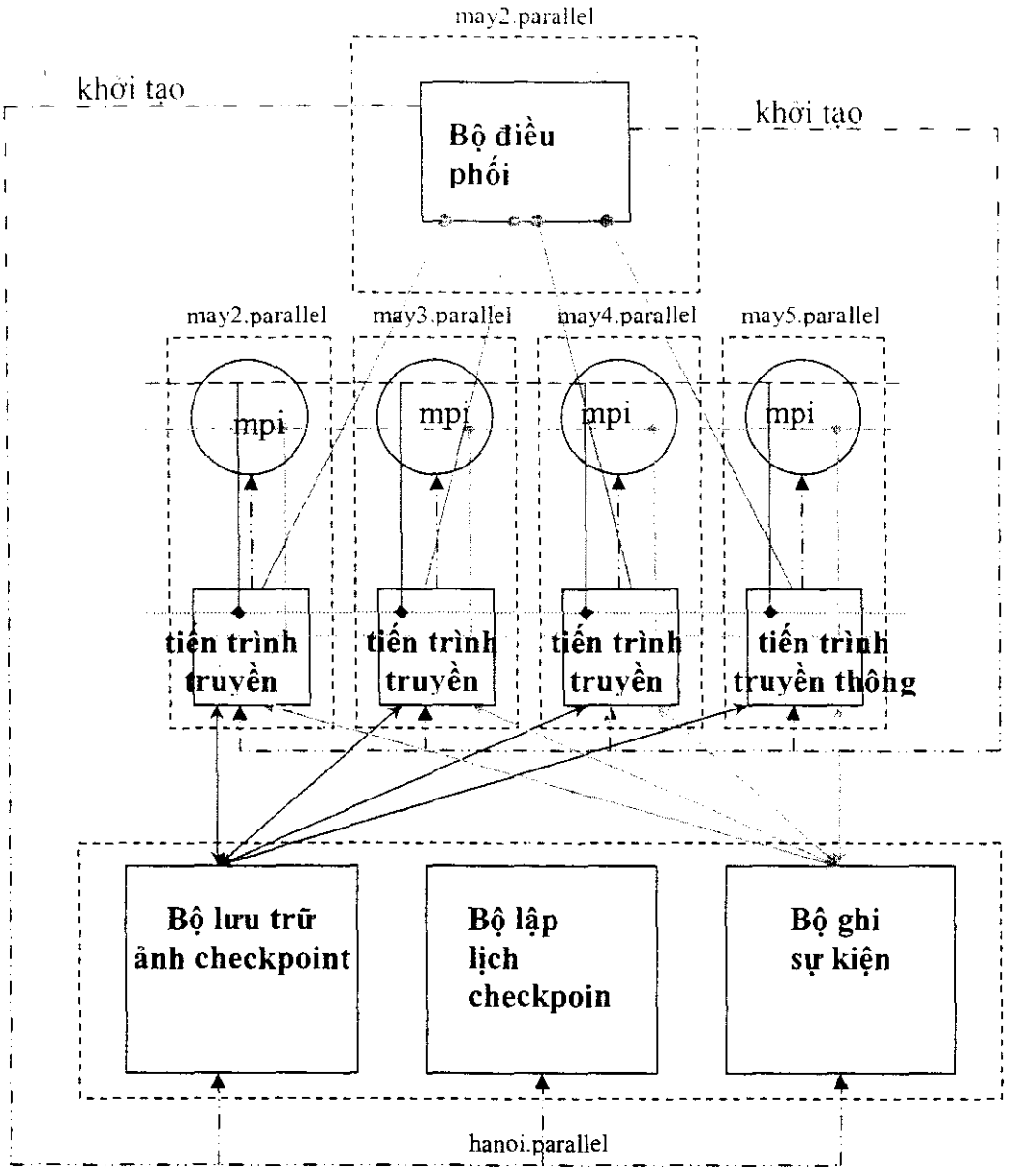
4.4.2. Kiến trúc thư viện MPICH-V2

4.4.2.1. Giới thiệu

MPICH-V2 là kiến trúc truyền thông điệp dựa trên giao thức ghi lại thông điệp theo kiểu pessimistic (pessimistic sender-base message logging protocol) . Sau đây là phần mô tả về các thành phần và cài đặt của hệ chống lỗi trên.

MPICH-V2 là hệ thống chống lỗi cài đặt trên giao thức pessimistic sender-based protocol dùng cho MPICH 1.2.5 , hệ thống truyền thông điệp có chống lỗi này sử dụng một thành phần gọi là "dispatcher " , một checkpointscheduler , vài bộ event loggers , checkpointserver , các computing nodes và các Daemond truyền thông trên các nút tính toán . Hình 1 biểu diễn phân bố cơ bản của hệ thống khi chạy MPICH-V2 , trong đó dispatcher , event loggers và checkpointscheduler được đặt trên cùng một máy là máy chạy tiến trình đầu tiên của job .

Một điểm lưu ý là , nếu checkpointscheduler và checkpointserver mà nằm trên các trạm có độ tin cậy thấp (dễ lỗi) thì khi xảy ra lỗi trên các thành phần này , lúc đó nếu các nút tính toán yêu cầu các ảnh checkpoint thì sẽ không nhận được vì các thành phần checkpointserver đã chết . Vì vậy mà MPICH-V2 đã cho phép đặt checkpointscheduler trên cùng nút với dispatcher và event logger , chính vì vậy các nút đặt các thành phần này phải có độ tin cậy cao .



Chú thích

Tiến trình MPI — gửi thông điệp

Tiến trình truyền thông — nhận thông điệp

Hình 4-3 Hệ thống chống lỗi MPICH-V2

4.4.2.2. Các phương thức giao diện và các module

Thư viện truyền thông dựa trên chuẩn MPI _libdev_

Hệ thống MPICH là tầng cài đặt của MPI . MPI API (giao diện lập trình ứng dụng MPI) được cài đặt bởi tầng giao diện Abstract Device Interface (ADI) dùng phổ biến trong các môi trường tính toán song song . Giao diện ADI lại được cài đặt trên các tầng khác : tầng giao thức như short , eager , rendez-vous protocols . Cuối cùng , các tầng giao thức này lại cài đặt trên tầng nguyên thuỷ : channel interface (giao diện kênh vào ra) . Hệ thống chống lỗi của ta xây dựng nằm trên tầng nguyên thuỷ này .

Đối với MPICH-V2 , nó được cung cấp như là một thiết bị (device) hay là kênh truyền thông nguyên thuỷ dành cho MPICH : nó cài đặt 6 giao thức nguyên thuỷ và được dùng như là tầng giao thức . Các kênh truyền thông này gồm 2 hàm truyền thông (protocol interface PI) PIbsend và PIbrecv , hai hàm này ngừng các thao tác để dành riêng cho việc gửi và nhận các khối dữ liệu . Đồng thời nó còn 4 hàm truyền thông khác : PInprobe để kiểm tra xem liệu có thông điệp nào đang treo mà chưa được nhận ; PIfrom để lấy định danh của đối tượng gửi thông điệp cuối cùng ; PIiInit để khởi tạo kênh truyền thông và PIiFinish để kết thúc toàn bộ công việc thực thi .

Cũng giống như kênh truyền thông P4 , (truyền thông chuẩn của MPICH dùng qua giao thức TCP/IP) , các tiến trình MPI sẽ không kết nối trực tiếp với nhau trên các nút mạng khác nhau . Việc liên hệ giữa các tiến trình trên các nút tính toán là nhiệm vụ các Daemon truyền thông chạy trên cùng trạm làm việc có tiến trình MPI thực thi và các Daemon truyền thông này nó kết nối với các tiến trình MPI địa phương , đồng thời nó điều khiển sự không đồng bộ trong mạng . Daemon truyền thông này thiết lập một UNIX socket để đón nhận phục vụ cho tiến trình MPI địa phương của nó và sau đó sẽ sinh ra (spawn) tiến trình MPI địa phương do Daemon này quản lí. Trong UNIX socket này , có hai loại thông điệp được trao đổi : thông điệp điều khiển (control messages) dùng cho khởi tạo , kết thúc và

thăm dò và các thông điệp giao thức (protocol message) dùng cho việc gửi và nhận thông điệp (bsend .breceive) .

Cài đặt các Daemond này dựa trên một vòng lặp dùng select : nó điều khiển một socket cho mọi nút tính toán và một socket dành cho các dịch vụ khác như (phục vụ lưu sự kiện _event logger_ ,phục vụ lưu trữ ảnh tiến trình _checkpoint server _, và bộ lập lịch lấy ảnh tiến trình _checkpointscheduler_) . Các socket này là dòng truyền thông TCP/IP và mọi thao tác gửi hay nhận thông điệp là không đồng bộ . Vì vậy , một sự truyền thông sẽ không bị ngưng lại bởi một truyền thông khác chẳng may bị chậm hơn , điều nay sẽ tốt cho hiệu năng của hệ thống . Mặt khác , sự truyền thông dùng UNIX socket tới các tiến trình MPI địa phương thì lại là đồng bộ và tính chất này áp dụng cho toàn giao thức về thông điệp.

Giao diện trừu tượng ADI (Abtract Device Interface)

Hàm PIbsend

Hàm PIbrecv

Hàm PInprobe

Hàm PIiInit

Hàm PIiFinish

Các hàm truyền thông cài đặt qua giao diện ADI

Hàm _v2bsend

Hàm _v2brecv

Hàm _v2probe

Hàm V2_Init

Hàm V2_Finalize

Môi trường truyền thông chống lỗi theo chuẩn MPI

Thành phần bộ điều phối

Thành phần phục vụ lưu trữ ảnh tiến trình

Hệ thống Checkpoint được chia làm 2 phần : Một thành phần là server dùng để lưu trữ các Checkpoint images và thành phần còn lại là bộ lập lịch điều phối cho sự kiện checkpoint cho toàn bộ các tiến trình trong hệ thống chạy job .

Thành phần này được gọi là " Checkpoint Server " , server này được đặt trên các trạm tin cậy dùng để lưu trữ các checkpoint images của các tiến trình MPI và của cả các Daemond truyền thống . Việc checkpoint các tiến trình MPI sử dụng bộ thư viện chuẩn Checkpoint Condor . Khi tiến trình MPI nhận một yêu cầu checkpoint từ Daemond truyền thông của nó , tiến trình này sẽ dùng cơ chế fork để sinh ra làm 2 phần . Phần thứ nhất sẽ gửi ảnh tiến trình của nó tới Daemond truyền thông . Đồng thời , phần thứ 2 được fork ra (gọi là tiến trình con) tiếp tục thực thi ứng dụng MPI để việc chuyển giao ảnh checkpoint không làm tăng chi phí thời gian tính toán . Khi sự kiện checkpoint được kích hoạt bởi Daemond truyền thông , nó đảm bảo rằng sẽ không có các hoạt động truyền thông vào thời điểm fork thêm tiến trình .

Theo cách nhìn trên đối với các tiến trình MPI , thì không có sự khác biệt giữa các tiến trình thực thi và các tiến trình phải khôi phục . Chương trình được gọi bởi Daemond truyền thông với một tham số cụ thể được điều khiển bởi thư viện Condor . Cái ảnh tiến trình được gửi đi bởi Daemond trong một pipe cụ thể và tiến trình nhảy tới checkpoint cuối cùng và tiến trình lại tiếp tục thực thi . Tất cả sự phức tạp trong khi chạy lại truyền thông được điều khiển bởi Daemond .

Riêng việc checkpoint cho Daemond truyền thông lại không được điều khiển bởi thư viện Condor nhưng lại được điều khiển ở mức người dùng , tuần tự hoá tất cả thông tin thông điệp .Khi một sự kiện Checkpoint được kích hoạt , Daemond sẽ

gửi yêu cầu checkpoint cho tiến trình MPI và lấy lại ảnh của tiến trình . Sau đó daemond sẽ tuần tự hoá tất cả các dữ liệu thông điệp của nó và gửi chúng tới Checkpoint Server . Trong suốt pha này , sự truyền thông với các nút tính toán khác sẽ không bị tạm thời ngưng lại để đảm bảo rằng chuyển giao checkpoint không ảnh hưởng tới tính toán của các tiến trình MPI .

Khi một nút tính toán hoàn thành việc checkpoint . nút này sẽ gửi cho tất cả các Daemond truyền thông khác giá trị logical clock của điểm checkpoint này . Khi một điểm checkpoint được hoàn thành tại một thời điểm logical clock , tất cả các thông điệp đã nhận trước đó sẽ không bao giờ yêu cầu lại nữa . Vì vậy , tất cả các thông điệp này có thể bỏ đi trên các sender tương ứng gửi các thông điệp này (bộ dọn rác) .

Thành phần phục vụ lưu trữ các chuỗi sự kiện

Giao thức sender-base pessimistic message logging của MPICH-V2 giả thiết việc ghi các thông điệp được chia làm hai phần như ở trên đã trình bày . Một phần (phần này nằm trên các Daemond)thì dùng phương thức sender-base logging lưu các thông điệp payload trên các trạm độ tin cậy không cao . Phần còn lại (là event logger) được dùng để lưu các thông tin phụ thuộc liên kết với các thông điệp và event logger này phải đặt trên các trạm ổn định cao .

Theo mô tả kiến trúc MPICH-V2 , mỗi tiến trình sẽ tăng logical clock trên trạm địa phương khi tiến trình này gửi hay nhận một thông điệp . Hệ thống ghi lại thông điệp payload được tích hợp vào Daemond truyền thông đặt tại các Computing Node . Mỗi khi một thông điệp được gửi đi tới một nút tính toán khác , thông điệp này được lưu lại ngay tại nút gửi vào trong một danh sách cho mục đích sử dụng sau này luca gặp lỗi (sender-base) . Hơn nữa giá trị của logical clock phía bên gửi cũng được lưu lại gắn kèm với bản sao của thông điệp được gửi lưu lại.

Do các nút tính toán của ta vốn không ổn định , nên các thông tin này sẽ bị mất trong trường hợp có sụp đổ nút , nhưng các thông tin này sẽ được tái tạo lại nếu cần thiết trong quá trình khôi phục lỗi .

Thành phần event logger là một kho lưu trữ chạy trên trạm tin cậy cao trong hệ thống . Thành phần này lưu trữ và phân phối các thông tin phụ thuộc về các thông điệp trao đổi giữa các nút tính toán trong mạng . Thông tin phụ thuộc được soạn ra gồm 4 trường kết hợp với mọi thông điệp đã nhận : (sender's identity , sender's logical clock tại lúc phát ra thông điệp , receiver's logical clock tại lúc nhận thông điệp , number of probe since last delivery ' số lần phát tín hiệu thăm dò kể từ lần nhận cuối cùng ') .

Khi một tiến trình nhận một thông điệp khi mà thực thi job của ta chưa có lỗi , tiến trình này sẽ kết tập sender's identity và sender logical clock , tìm ra lõi của thông điệp , với bản thân logical clock của nó lên gói phân phối và số lần thăm dò không thành công từ lần nhận cái cuối cùng . Trong kiến trúc MPICH , tầng phía trên có thể thăm dò sự tồn tại của thông điệp để được nhận , để cài đặt các thao tác non-blocking lên trên các thủ tục truyền thông có blocking . Giả thiết rằng một sự nhận luôn chỉ theo sau một sự kiện thăm dò thành công . Số lần đã thăm dò kể từ khi sự nhận cuối cùng cho tới sự nhận tiếp theo , vì bên nhận sẽ đếm con số này để thêm nó vào các thông tin phụ thuộc , mục đích là để đảm bảo chạy lại tiến trình bị chết một cách chính xác .

Thông tin này được tập hợp lại trong suốt quá trình nhận thông điệp và gửi không đồng bộ tới Event Logger . Tuy nhiên , thông tin này phải được gửi và được nhận biết bởi Event Logger trước khi nút nhận này có thể làm chuyển trạng thái của các tiến trình MPI bằng cách thực hiện hành động gửi thông điệp . Để cài đặt được điều này , Daemond truyền thông phải không được gửi các thông điệp trước khi Event Logger đã nhận biết được sự nhận của các sự kiện nhận trước đó .

Khi xảy ra lỗi , thành phần dispatcher sẽ dò ra được tiến trình bị lỗi , và nó sẽ sinh ra tiến trình mới để vẫn có khả năng kết thúc tính toán đang dở dang . Các nút phải

chạy lại sẽ kết nối với Event Logger tương ứng của nó và lấy thông tin phụ thuộc của tất cả các sự nhận của nó trước đây . Sau đó , các nút này thực thi chương trình MPI và yêu cầu các thông điệp cũ đã được ghi lại lên trên các nút sender tương ứng của chúng . Các thông điệp yêu cầu phải gửi lại được tìm ra nhờ bộ thông tin (sender's identity ; sender's logical clock) của thông tin phụ thuộc nằm trên Event Logger , các thông tin đó sẽ là một phần trong cái thông điệp yêu cầu phát lại (re-emitted message) . Nếu như vài nút trong số phải gửi lại cũng là nút chết , các thông điệp bị mất cuối cùng cũng được gửi trong suốt quá trình thực thi lại của bản thân các sender đó .

Theo nâng cấp , ta sẽ dùng một số các thành phần Event Logger trong hệ thống , nhưng mọi Daemond truyền thông phải kết nối chính xác tới một Event Logger . Khi đó , sẽ không có thông tin phụ thuộc bị quản lí giữa các nút phải chạy lại , các Event Logger sẽ không phải không phải trao đổi thông tin với nhau .

Thành phần điều khiển lấy ảnh của các tiến trình

Bộ lập lịch dành cho Checkpoint được xây dựng với 2 lí do : 1) giao thức sender-base pessimistic logging khiến bộ nhớ phải dùng trong các Daemond truyền là rất lớn và cả ở trong các Checkpoint Server (bởi vì các thông điệp sau khi lưu vào vùng nhớ trên các Daemond sender thì sau khi lấy ảnh Checkpoint của tiến trình MPI , các thông điệp này cũng được gửi đi và lưu vào thiết bị lưu trữ do Checkpoint Server quản lí) . Việc kích hoạt Checkpoint không chỉ cần thiết phải tránh ảnh hưởng tới thời gian tính toán mà còn phải đảm bảo tối ưu không gian lưu trữ các thông điệp được ghi lại . 2) Lấy Checkpoint cho Daemond truyền thông gây ra giao vận trên mạng tương ứng bằng kích cỡ các thông điệp được phát đi . Giao vận này sẽ cạnh tranh với giao vận truyền thông ứng dụng trên băng thông mạng , vì vậy càng giảm được Checkpoint thì càng tốt .

Vai trò của thành phần Checkpoint Scheduler là đánh giá lợi nhuận và chi phí của một checkpoint , tại bất cứ một thời điểm xác định , và yêu cầu checkpoint sao cho hợp lí . Một cách định kì , Checkpoint Scheduler yêu cầu các Daemond truyền

thông gửi tình trạng của chúng (giới hạn số lượng các thông điệp đã ghi) , và đánh giá lợi ích của một checkpoint . Chú ý là khi giao thức này không cần điều phối checkpoint , bộ lập lịch cũng không cần phải hợp lí . Hiện nay thành phần này đang thực hiện 2 chính sách lấy Checkpoint : round robin và adaptive one .

Ưu điểm vượt trội của giải thuật round-robin là giảm truyền thông giữa bộ lập lịch và các nút . Nhược điểm chính là sự bất đối xứng trong vài lược đồ truyền thông . Nếu như các truyền thông trong chương trình MPI rất đối xứng (ví dụ lược đồ all-to-all) , dung lượng dữ liệu lưu trữ trên tất cả các nút tương đối giống nhau thì lúc đó giải thuật round-robin là hợp lí . Nếu bài toán truyền thông của ta không cân xứng thì sẽ giải thuật sẽ yêu cầu một số nút cần lấy checkpoint nhiều hơn các nút khác .

Đối với các lược đồ truyền thông này , hệ thống đã phát triển thêm một giải thuật thích nghi , giải thuật này đánh giá tỷ số "số lượng các thông điệp đã nhận " / "số lượng các thông điệp đã gửi " trên mỗi nút tính toán . Giải thuật này dựa trên thứ tự giảm dần của tỉ số này rồi tính ra lịch lấy Checkpoint . So sánh 2 chiến lược trên với các lược đồ truyền thông điển hình (point to point , đồng bộ all-to-all , broadcasts and reduces) , kết quả cho thấy giải thuật này thường đưa ra chiến lược lấy Checkpoint tốt hơn (thậm chí dưới n lần trong đó n là số nút tính toán cho asynchronous broadcast) . Hiện nay đang chưa biết là hệ thống này dùng giải thuật gì ?

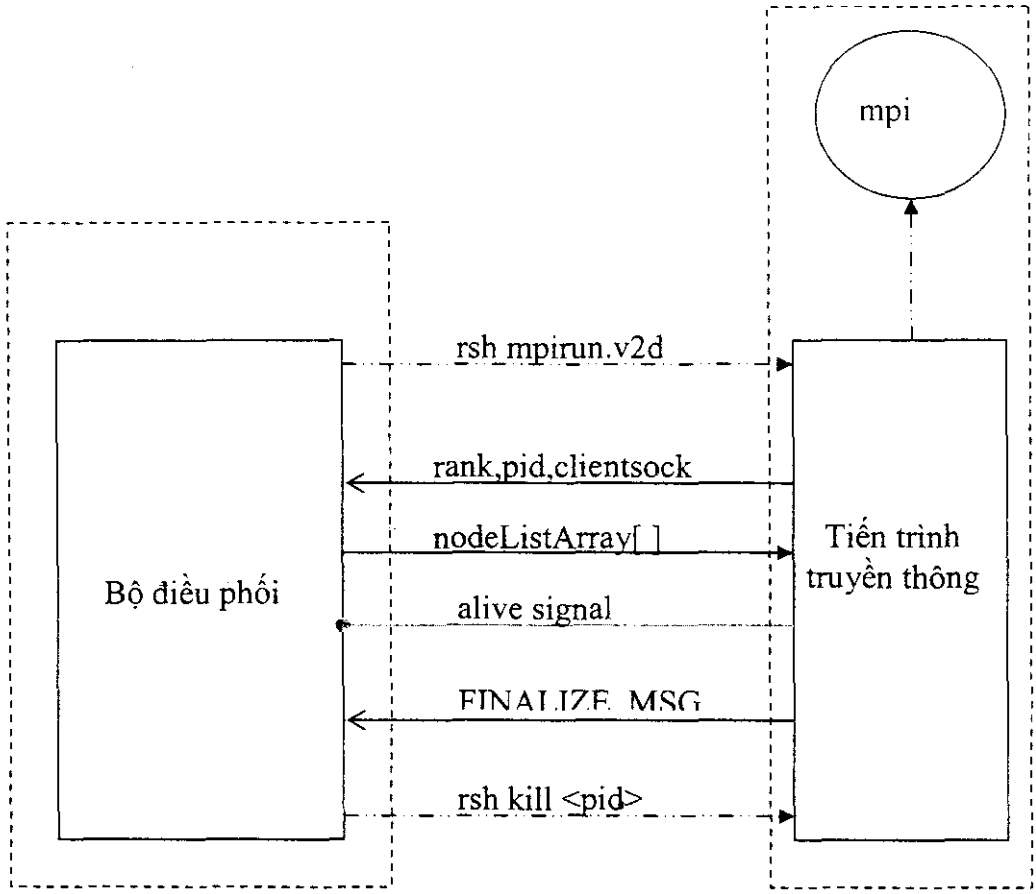
Thành phần Daemon truyền thông giữa các tiến trình

4.4.2.3. Đồng bộ các giao thức qua các kết cấu thành phần

(phần này sẽ giải thích rõ cách hoat động của các giao thức)

Đồng bộ các giao thức truyền thông

Truyền thông giữa bộ điều phối và tiến trình truyền thông



Chú thích

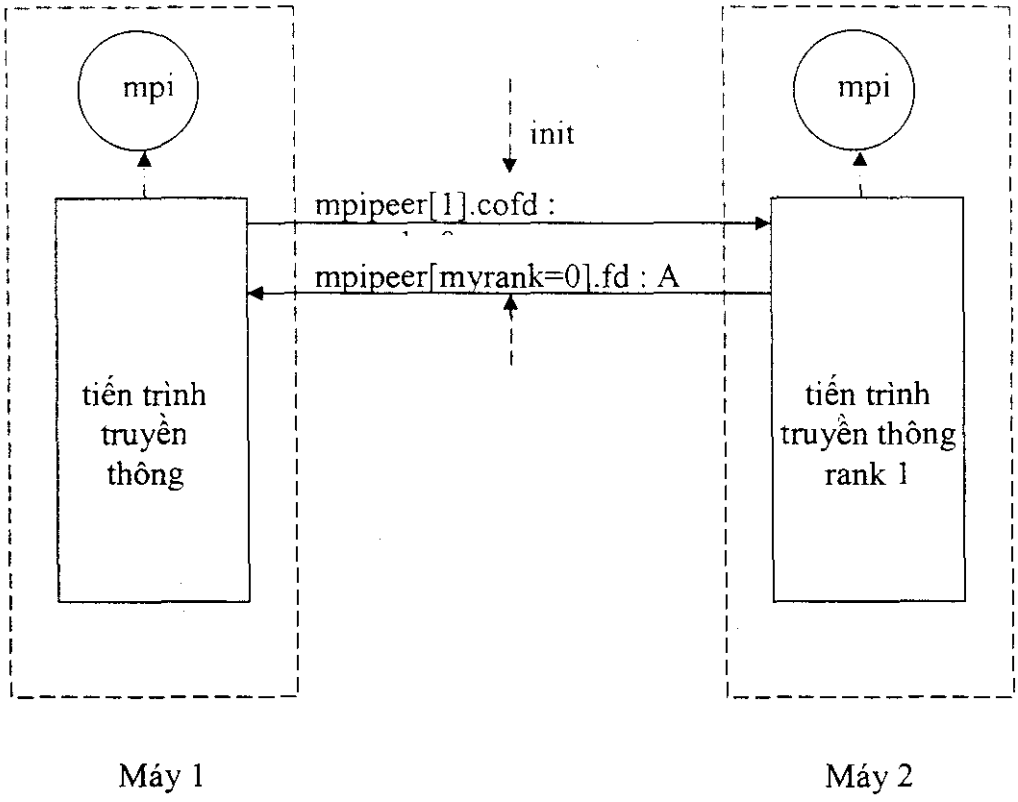
lệnh rsh

dòng truyền thông

tín hiệu báo tiến trình còn sống

Hình 4-4Giao thức khởi tạo giữa Bộ điều phối và tiến trình truyền thông cho mỗi nút MPI

Truyền thông giữa các tiến trình truyền thông



Hình 4-5 Mô phỏng khởi tạo giữa 2 tiến trình rank 0 và rank 1

Truyền thông bằng socket giữa các tiến trình truyền thông dựa trên danh sách quản lí các socket truyền thông có cấu trúc như sau :

```
struct SockInfo {
    int cofd; /* lưu socket kết nối tới các tiến trình truyền thông khác*/
    int costate; /*mô tả trạng thái của kết nối tới tiến trình truyền thông khác*/
    int hstate; /*trạng thái gửi giá trị đồng hồ của rank kết nối tới nó*/
```

```
    int fd; /*luu socket ma tiến trình truyền thông khác kết
nối tới nó*/
    struct sockaddr_in addr;/*luu địa chỉ IP và cổng tiến trình
truyền thông mà nó kết nối tới*/
}
```

Trong đó

costate gồm 4 trạng thái

CO_NOCONNECT

CO_COWAITING

CO_COINPROGRESS

CO_CONNECT

hstate gồm 6 trạng thái

CO_CO_HSEND

CO_AC_HSEND

CO_CO_HRECV

CO_AC_HRECV

CO_HDONE

CO_HNOTDONE

Mỗi tiến trình truyền thông sử dụng các biến toàn cục và hàm kết nối sau :

1.Biến toàn cục

```
static SockInfo * mpipeer ;/*mảng lưu trữ các thông tin cho về
                            socket cho tất cảcác tiến trình
                            truyền thông kết nối tới nó , được
                            định danh bởi rank, số phần tử mảng
                            này bằng tổng số rank chạy  */
```

2.Hàm kết nối

a.Các hàm yêu cầu kết nối

```
/*Mô tả : hàm này sử dụng biến toàn cục mpipeer[<rank>] để yêu
cầu kết nối với <rank> được truyền vào cho hàm

   Tham số : rank _ định danh của tiến trình truyền thông mà nó
cần kết nối

   Chú thích : kết nối theo kiểu NONBLOCK

*/

static void init_connect_to_one_MPI(int rank)


/*Mô tả : hàm này sử dụng hàm trên để kết nối với tất cả các tiến
trình truyền thông khác trừ bản thân nó. Vì bản thân tiến trình
truyền thông này lại kết nối với tiến trình MPI địa phương qua
socket UNIX mà ta sẽ mô tả trong phần giao thức giữa tiến trình
truyền thông với tiến trình MPI địa phương của nó

Tham số : không có

Chú thích : sử dụng vòng lặp qua các rank để yêu cầu kết nối

*/

static void init_connect_to_MPI(void);
```

b. Các hàm đón nhận kết nối

```
/*Mô tả : Khi cổng đợi kết nối của chính tiến trình truyền thông
được kích hoạt sau lệnh select() thì cổng đợi này được truyền vào
làm tham số cho hàm này trả về socket kết nối đến từ các socket
cofd từ các tiến trình truyền thông khác

 Tham số : scon _ socket đợi kết nối

 Chú thích : hàm này chỉ nhận kết nối rồi trả về socket kết nối,
ngoài ra không làm gì khác

*/
```

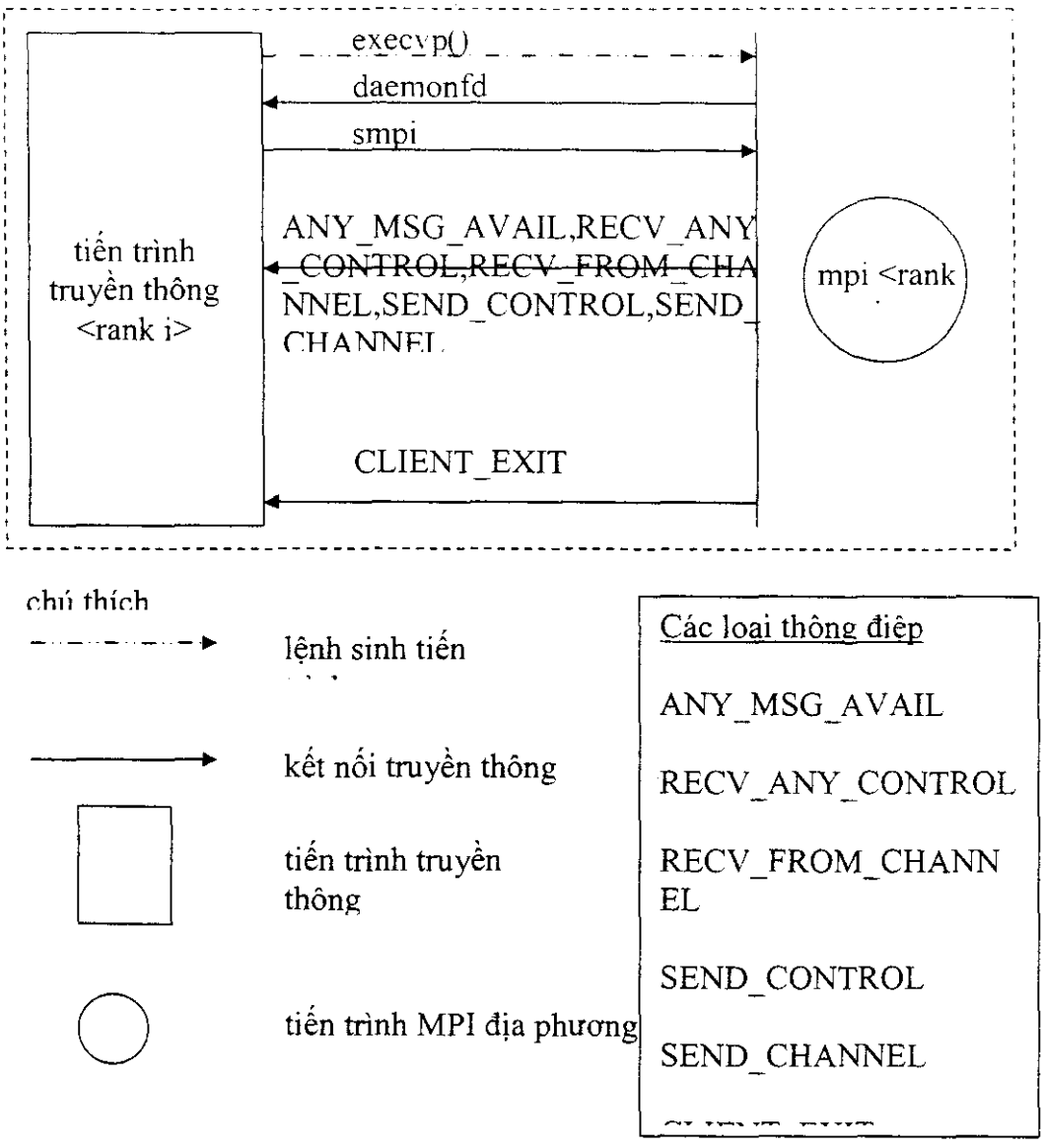
```
int on_accept(int scon)


/*Mô tả : hàm này đọc dữ liệu là rank kết nối tới tiến trình này,
kiểm tra một số điều kiện, nếu chấp nhận kết nối được thì nó sẽ
cập nhật giá trị mpipeer[<rank>].fd bằng socket trả về từ hàm
trên

 Tham số : s _ socket kết nối tới tiến trình này

           mpipeer[]   _ mảng chứa thông tin về các tiến trình
truyền thông khác

           int myrank _ sử dụng để kiểm tra điều kiện

 Chú thích : mpipeer[<rank>].hstate = CO_AC_HSEND _sẵn sàng gửi
giá trị đồng hồ

*/

int on_accepted(int s ,SockInfo mpipeer[], int myrank)
```

Truyền thông giữa tiến trình MPI với tiến trình truyền thông

Truyền thông giữa tiến trình truyền thông với tiến trình MPI địa phương do nó quản lí sử dụng kết nối qua socket PF_UNIX .Khi mới khởi tạo, tiến trình truyền thông sẽ mở một cổng lắng nghe sử dụng socket PF_UNIX dựa trên số hiệu <rank> và <group> của tiến trình, socket này có tên là "<group>:<rank>.v2d.sock" . Tiến trình MPI địa phương sau khi được tiến trình truyền thông khởi tạo với lệnh execvp như trên hình vẽ với tham số truyền vào là <group> , <rank> sẽ nhận biết thấy định danh của socket lắng nghe của tiến trình truyền thông của nó. Từ đó nó kết nối với tiến trình truyền thông quản lí nó.

Kết nối từ tiến trình MPI tới tiến trình truyền thông qua socket daemonfd

Kết nối trả về từ tiến trình truyền thông tới tiến trình MPI là socket smpi như hình vẽ



Hình 4-6 Mô phỏng truyền thông khởi tạo giữa tiến trình truyền thông với tiến trình MPI địa phương

Có 6 loại thông điệp trao đổi giữa tiến trình MPI với tiến trình truyền thông

```
ANY_MSG_AVAIL

RECV_ANY_CONTROL
```

```
int listenLocalMPI(int group, int rank);


/*Mô tả : Hàm này chấp nhận kết nối từ tiến trình MPI địa phương
do tiến trình truyền thông này quản lí
Tham số : listenfd _ socket trả về từ hàm listenLocalMPI
Chú thích : Hàm trả về socket đã được kết nối với tiến trình MPI
địa phương
*/
     int acceptLocalMPI(int listenfd);
```

b.Bên phía tiến trình MPI địa phương

```
/*Mô tả : Hàm này sử dụng biến toàn cục daemonfd để kết nối với
cổng đợi của tiến trình truyền thông của nó. Socket cấp phát theo
kiểu PF_UNIX. Hàm này được triệu gọi trong hàm V2_Init( ) được
định nghĩa lại thông qua giao diện truyền thông ADI[5]
   Tham số : Không có
   Chú thích : hàm này  được gọi khi tiến trình MPI vừa khởi tạo
*/
void connect_my_daemon( );
```

Đồng bộ các giao thức chống lỗi

Giao thức phát hiện lỗi tiến trình MPI

Mô tả cơ chế dò lỗi tiến trình MPI :

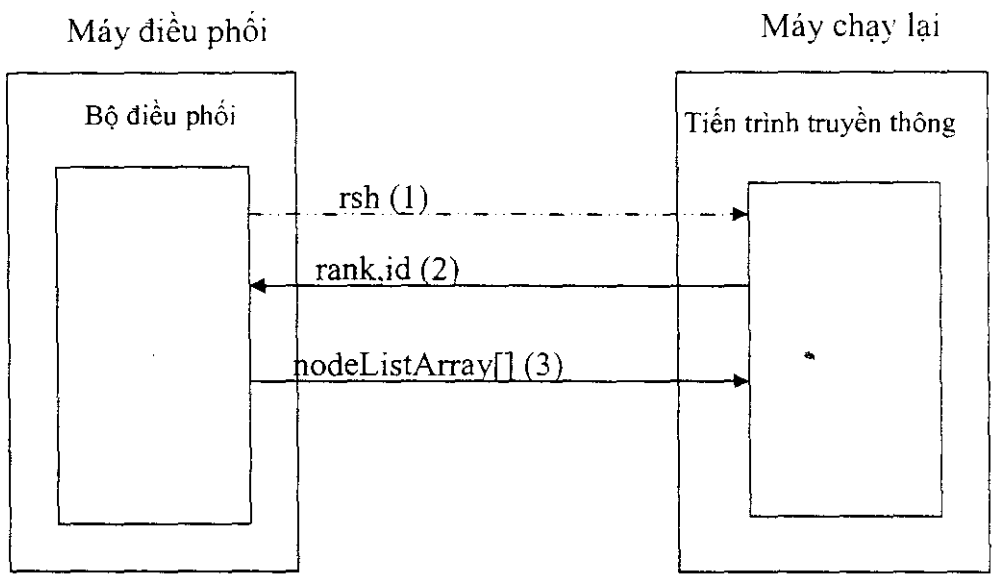
(1).Khi tiến trình MPI bị đổ vỡ (_crash_), nó sẽ gửi tín hiệu SIGCHLD về cho tiến trình cha sinh ra nó là tiến trình truyền thông.Tiến trình truyền thông sẽ chấm

[5] Xem phần Thư viện truyền thông dựa trên chuẩn MPI _libdev

Ko có 123

```
/*Mô tả : hàm này đóng tất cả các kết nối khi gặp tín hiệu
SIGCHLD
*/
void getSignal(int numSignal)
```

Giao thức chạy lại tiến trình MPI lỗi



Chú thích

sử dụng rsh để chạy tiến trình từ

sử dụng socket để truyền thông

Hình 4-8Cơ chế chạy lại tiến trình bị lỗi

Mô tả cơ chế chạy lại tiến trình lỗi :

(1).Khi chạy lại tiến trình lỗi , Bộ điều phối sẽ sử dụng rsh để chạy lại tiến trình truyền thông trong chế độ khôi phục. Tham số đầu vào lúc này có

"restart <group>.<rank>.restart.pipe" dùng để làm tham số đầu vào cho thư viện condor phục hồi tiến trình MPI.[6]

(2).Tiếp theo. Bộ điều phối nhận id và rank gửi về từ tiến trình truyền thông phục hồi

(3).Dựa vào các thông tin trên, Bộ điều phối gửi danh sách nodeListArray tới cho tiến trình truyền thông. Danh sách này bao gồm địa chỉ IP và cổng lắng nghe của các tiến trình truyền thông khác trong cùng một group.

Các biến và các hàm sử dụng trong giao thức phục hồi này

1.Bên phía Bộ điều phối

a.Các biến sử dụng

```
struct jobSpecifications {
      int jodId ; /*định danh nhóm tiến trình*/
      int nprocs ; /*số tiến trình MPI*/
      int dispatcherPort ; /*cổng lắng nghe của Bộ điều phối*/
      CN * nodeList ; /*danh sách các nút tính toán*/
      EL * elList ; /*danh sách các Bộ lưu sự kiện*/
      CS * csList ; /*danh sách các Bộ lưu ảnh tiến trình*/
      SC * scList ; /*danh sách các bộ lập lịch lấy ảnh tiến
trình*/
}JS;
JS * theJob /*biến này lưu trữ toàn bộ đặc tả của nhóm tiến
trình, khởi tạo khi bắt đầu Bộ điều phối*/
```

[6] Xem phần tiến trình MPI khôi phục

b.Các hàm sử dụng

```
/*Mô tả : hàm này dựa vào số hiệu rank bị sụp đổ và đặc tả của
nhóm tiến trình để chạy lại tiến trình truyền thông theo chế độ
phục hồi

 Tham số : int rank : số hiệu rank bị lỗi

           JS * js    : Đặc tả của nhóm tiến trình. Nó được khởi
tạo ngay khi Bộ điều phối

                      hoạt động thông qua tham số đầu vào là các
file lập lịch cho nhóm

                      tiến trình.

 Chú thích : hiện nay bộ công cụ chống lỗi này mới chỉ cung cấp
cho ta phục hồi lỗi trên chính máy mà rank lỗi chạy lúc trước.
Điều này có nghĩa là chưa sử dụng nguyên lí di trú tiến trình áp
dụng vào cho chống lỗi.

*/

void relaunchByRank(int rank, JS * js)
```

2.Bên phía tiến trình truyền thông[7]

Giao thức tiến trình MPI lấy ảnh tiến trình

Mô tả quá trình tiến trình MPI thực hiện đổ ảnh tiến trình và tiến trình truyền thông gửi ảnh lên Bộ lưu trữ ảnh tiến trình :

(1).Đầu tiên khi nhận được tín hiệu CP_START từ tiến trình truyền thông (tín hiệu này thực ra xuất phát từ Bộ lập lịch lấy ảnh tiến trình theo thuật toán roundRobin _giải thuật xét lần lượt các tiến trình đều được quan tâm để lấy ảnh tiến trình_) thì tiến trình sẽ sinh ra một tiến trình bản sao bằng fork [8]

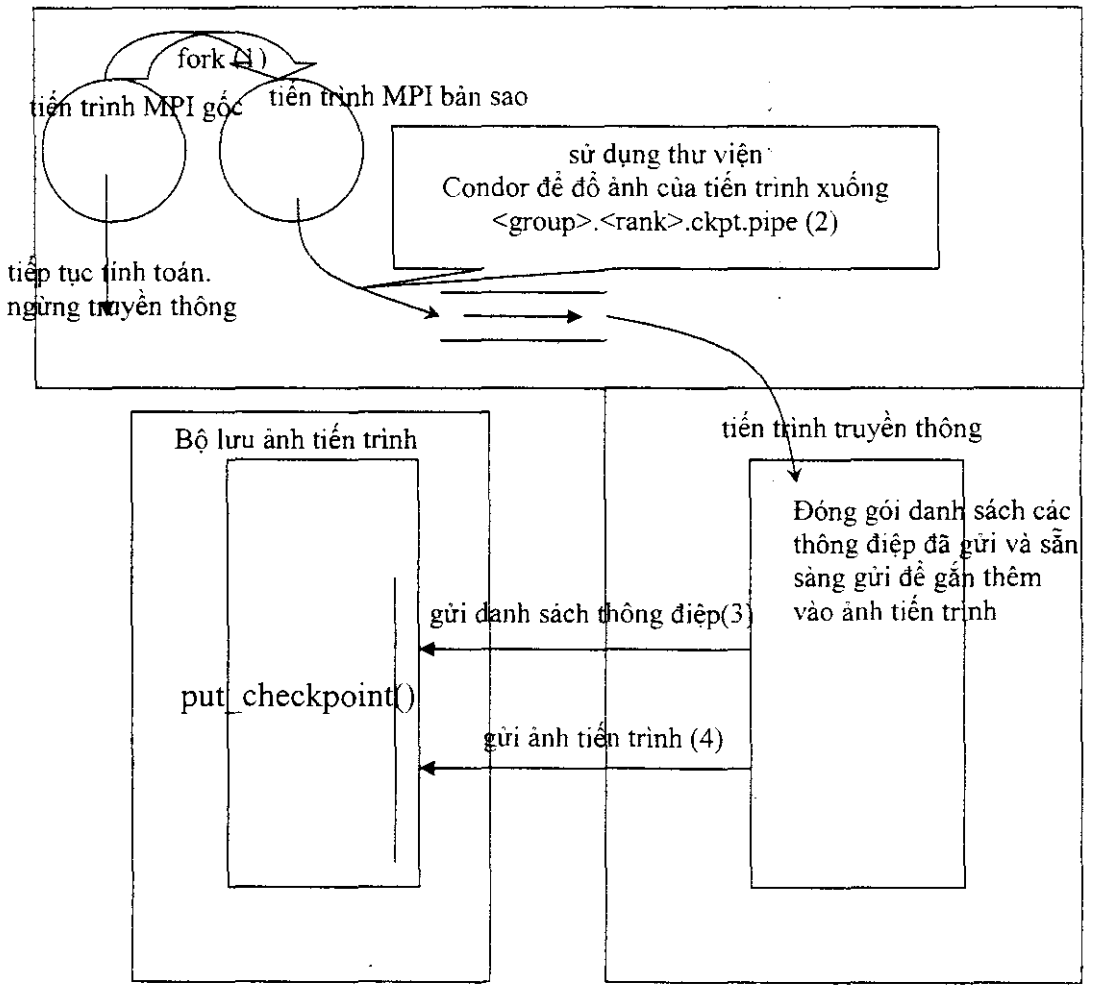
(2).Tiến trình bản sao này đóng kết nối truyền thông tới tiến trình truyền thông quản lí nó rồi sử dụng thư viện condor để đổ ảnh tiến trình xuống pipe

[7] Xem phần khởi tạo nhóm tiến trình ban đầu
[8] Xem phần thư viện truyền thông theo chuẩn MPI _libdev_

<group>.<rank>.ckpt.pipe. Lúc đó tiến trình gốc vẫn tiếp tục thực hiện các chức năng tính toán, còn các chức năng truyền thông thì tạm ngưng lại.

(3).Tiến trình truyền thông lúc này rơi vào trạng thái lấy ảnh tiến trình, nên nó thực hiện đóng gói gửi bảng danh sách các thông điệp mà nó lưu lại bên gửi trước khi gửi đi cho bên nhận sau đó gửi lên Bộ lưu ảnh tiến trình để nó gắn vào file lưu trữ ảnh[9]

(4).Sau khi gửi xong các thông điệp lưu, nó sử dụng pipe mà tiến trình MPI đổ ảnh xuống và gửi lên cho Bộ lưu trữ ảnh tiến trình



Hình 4-9Tiến trình MPI thực hiện đổ ảnh tiến trình và lưu

[9] Xem phần Bộ lưu trữ ảnh tiến trình

Các biến và hàm sử dụng trong giao thức lưu ảnh tiến trình

1.Bên phía tiến trình truyền thông

a.Các biến sử dụng

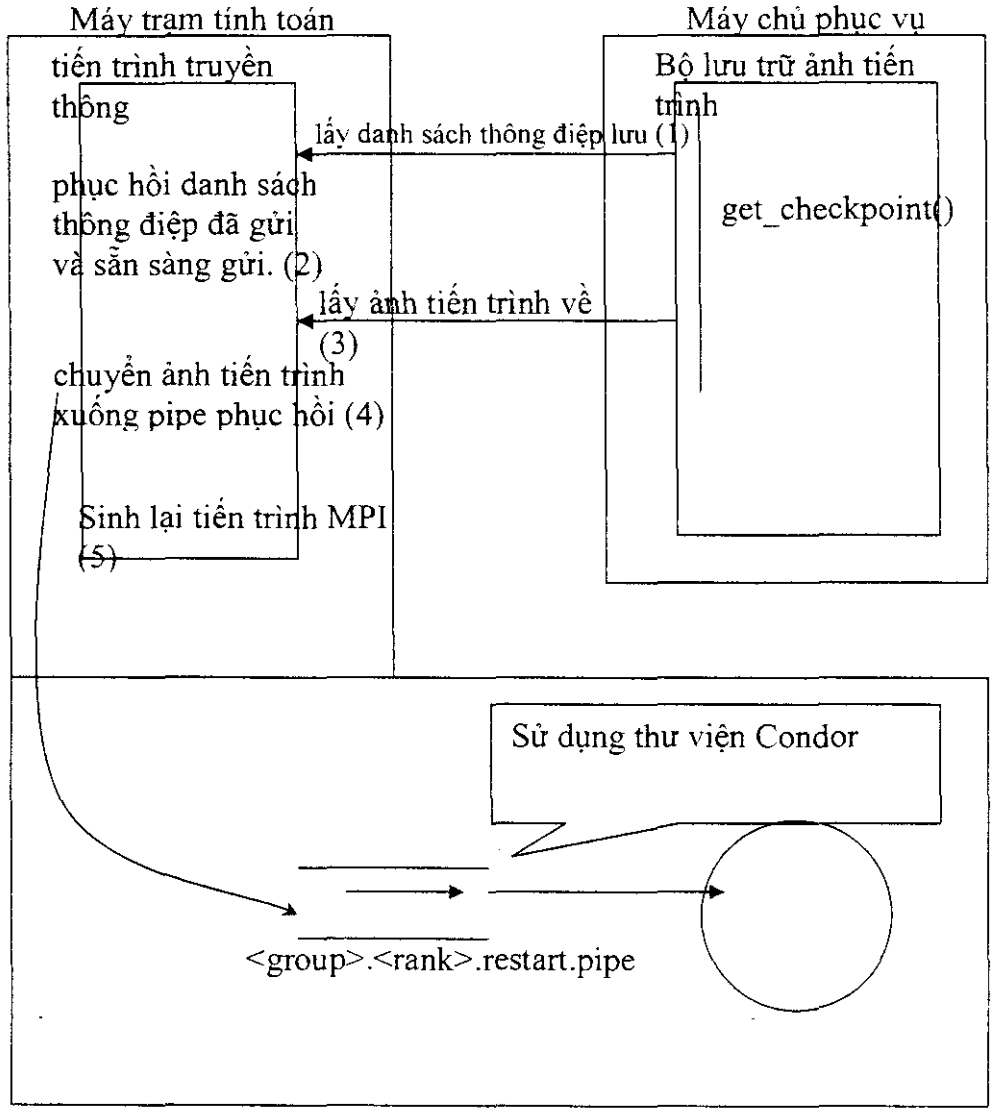
```
struct CkptInfo
{
    int pipe ; /* đường ống mà condor sử dụng đổ ảnh tiến
trình*/
    CkptSock sock ; /* kết nối tới Bộ lưu trữ ảnh tiến trình*/
    long h ; /*giá trị đồng hồ của tiến trình bắt đầu từ thời
điểm lưu ảnh*/
    char state ; /*rơi vào một trong 3 trạng thái
                        CKPTSTATE_SBSEND                    |
CKPTSTATE_IMGSEND |
                        CKPTSTATE_FINISHED*/
}
static CkptInfo cin ; /*biến lưu đặc tả trong quá trình lưu ảnh*/
listemessages sb ; /* bảng danh sách thông điệp ghi lại bên gửi*/
```

b.Các hàm sử dụng

```
/*Mô tả : hàm này được gọi nhiều lần để lần lượt gửi toàn bộ bảng
danh sách các thông điệp ghi lại bên tiến trình gửi cho tới khi
bảng này trống
Tham số : CkptInfo cin : cấu trúc này được tạo ra khi có tín hiệu
bắt đầu đổ ảnh do tiến
                        trình MPI gửi đến : hàm gọi là
on_ckpt_signal(CkptInfo * cin)
    listemessages sb[] : Bảng danh sách các thông điệp lưu
 */
```

```
int on_sending_ckptsb(CkptInfo *cin, listemessages sb[] )

/*Mô tả : hàm này gọi nhiều lần cho tới khi gửi hết ảnh tiến
trình tới Bộ lưu trữ ảnh tiến trình

Tham số : CkptInfo cin : như đã giải thích ở trên

*/

int on_sending_ckptimage(CkptInfo * cin)
```

Giao thức tiến trình MPI khôi phục



Hình 4-10Tiến trình MPI khôi phục từ ảnh tiến trình qua pipe

Mô tả cơ chế phục hồi tiến trình MPI khi xảy ra lỗi :

(1).Đầu tiên, tiến trình truyền thông sinh ra do Bộ điều phối nhận lại danh sách các thông điệp đã gửi và các thông điệp sẵn sàng gửi từ lúc trước khi đổ ảnh tiến trình về vào một danh sách gọi là sb. Với sự phục hồi danh sách này, nếu tiến trình khác lại bị lỗi và yêu cầu tiến trình vừa phục hồi này gửi lại thông điệp thì nó đã sẵn sàng gửi lại thông điệp. Đây chính là cách mà hệ thống truyền thông này có thể phục hồi lỗi đa tiến trình.

(2).Tiếp theo, các thông điệp được gắn liền với ảnh tiến trình nhận về này được gói vào một bảng danh sách, đây chính là bảng danh sách ghi thông điệp trước khi gửi của tiến trình này tới các tiến trình khác theo đúng lý thuyết về "ghi lại thông điệp từ phía bên tiến trình gửi thông điệp".

(3).Tiếp theo nữa, tiến trình truyền thông này tiếp tục nhận ảnh tiến trình MPI đã lưu lúc trước trên máy chạy Bộ phục vụ lưu trữ ảnh tiến trình về.

(4).Quá trình nhận ảnh tiến trình diễn ra đồng thời với quá trình đổ ảnh tiến trình lại vào <group>.<rank>.restart.pipe để thư viện Condor sử dụng để khôi phục tiến trình MPI.

(5)Tiến trình truyền thông sử dụng hàm fork() rồi sinh tiến trình MPI địa phương với tham số đầu vào là " restart <group>.<rank>.restart.pipe". Tiến trình MPI sử dụng thư viện Condor dùng pipe này để phục hồi tiến trình.

Bên phía Bộ lưu ảnh tiến trình, khi nhận yêu cầu kết nối phục vụ cho việc lấy ảnh tiến trình, nó gọi hàm get_checkpoint() để gửi cả danh sách thông điệp lưu lẫn ảnh tiến trình về dựa vào bộ số <group> và <rank>[10]

Các biến và hàm sử dụng trong giao thức này :

1.Bên phía tiến trình truyền thông :

a.Các biến toàn cục sử dụng :

[10] Xem phần Bộ lưu ảnh tiến trình

```
struct CkptSock
{
      int proto;
      int data;
}
static listemessages *sb /*bảng danh sách các thông điệp ghi bên phía gửi*/
```

b.Các hàm sử dụng để phục hồi

```
/*Mô tả : hàm này trả về cho 2 kết nối tới Bộ lưu ảnh tiến trình thông nằm trong struct CkptSock
   Tham số : không có
   Chú thích : thực tế hàm này gọi đến hàm connectCheckpointServer yêu cầu phục vụ
*/

CkptSock v2dinitrestart(void)
/*Mô tả : Hàm này thực hiện 5 nhiệm vụ khi nhận dữ liệu từ bộ lưu trữ ảnh tiến trình
      1.Lấy số tiến trình gọi :
    Hàm unserialize_np(CkptSock s, int* np)
      2.Lấy số lần thăm dò thông điệp của tiến trình trước khi đổ ảnh :
   Hàm unserialize_nbprobe(CkptSock s,int * nbprobe )
3.Lấy các giá trị đồng hồ của các tiến trình khác tại thời điểm đổ ảnh tiến trình :
   Hàm unserialize_ctab(CkptSock s, long * ctab, int sizeofmpi)
4.Lấy ảnh tiến trình đổ vào pipe <group>.<rank>.restart.pipe :
   Hàm unserialize_image(CkptSock s, int mygroup, int myrank)
```

```
5.Lấy bảng danh sách thông điệp lưu bên phía gửi :
 Hàm unserialize_sb(CkptSock s, int sizeofmpi, listemessages *sb)
*/
void v2dfinishrestart( CkptSock s)
```

Giao thức gửi ảnh tiến trình từ tiến trình truyền thông lên bộ lưu ảnh tiến trình

1.Cơ chế phục vụ của Checkpoint Server :

a.Mô tả cơ chế : CS dùng một vòng lặp while đợi 1 kết nối từ phía daemon là sproto , sau đó nó tự fork ra thêm một tiến trình .

Trong tiến trình cha : nó đợi tới khi tiến trình con kết thúc mới tiếp tục trở lại vòng lặp để đón nhận phục vụ.

Trong tiến trình con : nó đợi thêm kết nối sdata từ daemon tới rồi gọi hàm traiter_message(sproto, sdata) để xử lí yêu cầu phục vụ.Hàm traiter_message được viết như sau:

```
/**********************/
int traiter_message(int sproto, int sdata){
   switch(recv_type_message(sproto))
   {
         Case GET :
           SHOW_TIME(get_checkpoint(sproto,sdata));
                    Return 0;
         Case PUT :
                   SHOW_TIME(put_checkpoint(sproto,sdata));
           Return 0;
        Default :
           Return -1;
   }
```

}

/**********************/

b.Định dạng file lưu trữ ảnh checkpoint

file lưu trữ ảnh lưu tên theo định dạng : ckptimg-g<group>-r<rank>

header	sproto_dat	sdata_data

+Header: 20 byte : 5 giá trị kiểu int

Int group, rank,sequence,protosize,datasize.

Trong đó sequence : chỉ ra thứ tự lần checkpoint

+Sproto_data : protosize byte

+Sdata_data : datasize byte

c.Các loại phục vụ : có 2 loại phục vụ GETCHECKPOINT và PUTCHECKPOINT

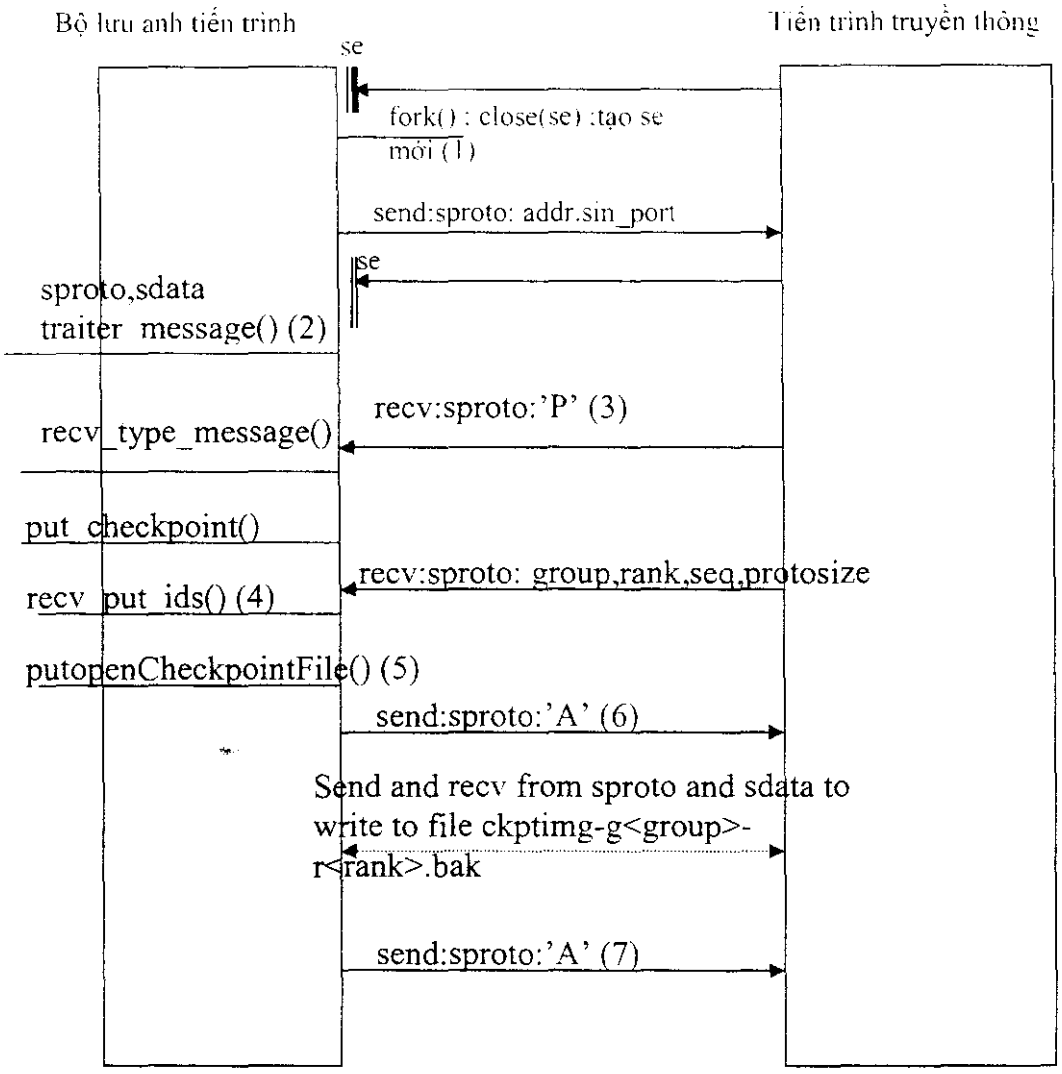
1.PUTCHECKPOINT:

Hàm thực hiện giao thức này là put_checkpoint(int sproto, int sdata) :

(1).tiến trình con đóng server kết nối se , tạo ra server kết nối mới se với cổng tự chọn : addr.sin_port=htons(0); sau đó sproto sẽ gửi số hiệu cổng này về cho daemon để daemon kết nối đường dữ liệu sdata sau này

(2).tiến trình con sau khi nhận được 2 kết nối đảm bảo từ daemon là sproto và sdata thì nó gọi hàm traiter_message(int sproto, int sdata) để xử lí yêu cầu.

(3).Trong trường hợp nhận ảnh tiến trình thì kí tự đọc về là P.Hàm recv_type_message() trả về PUT=2



Hình 4-11So đồ giao thức gửi ảnh tiến trình tới Checkpoint Server từ Daemon

(4).Hàm này trả về các giá trị group,rank,seq,protosize lấy từ socket sproto.

(5).Hàm này truyền 3 tham số group,rank,seq để tạo file với tên ckptimg-g<group>-r<rank> và ghi 3 dữ liệu trên vào header của file.Khi mở file này tên tam thời của nó có phần mở rộng là .bak.Khi nào việc lưu ảnh kết thúc thì tên nó được cắt phần mở rộng đi.

(6).gửi tín hiệu 'A'ccept về cho daemon đang kết nối gửi ảnh.Nếu có lỗi xảy ra thì gọi hàm discardCheckpointFile(), để huỷ file mới tạo ra.

(7).Quá trình nhận dữ liệu ảnh được thực hiện theo cơ chế như trong hàm put_checkpoint().khi kết thúc công việc , sproto gửi về tín hiệu 'A' báo hoàn tất quá trình ghi file.Sau đó nó gọi hàm confirmCheckpointFile() để thay thế file.bak cho file gốc.

2.GETCHECKPOINT:

Hàm thực hiện giao thức này là get_checkpoint(int sproto, int sdata) :

(1).tiến trình con đóng server kết nối se , tạo ra server kết nối mới se với cổng tự chọn : addr.sin_port=htons(0); sau đó sproto sẽ gửi số hiệu cổng này về cho daemon để daemon kết nối đường dữ liệu sdata sau này

(2).tiến trình con sau khi nhận được 2 kết nối đảm bảo từ daemon là sproto và sdata thì nó gọi hàm traiter_message(int sproto, int sdata) để xử lí yêu cầu.

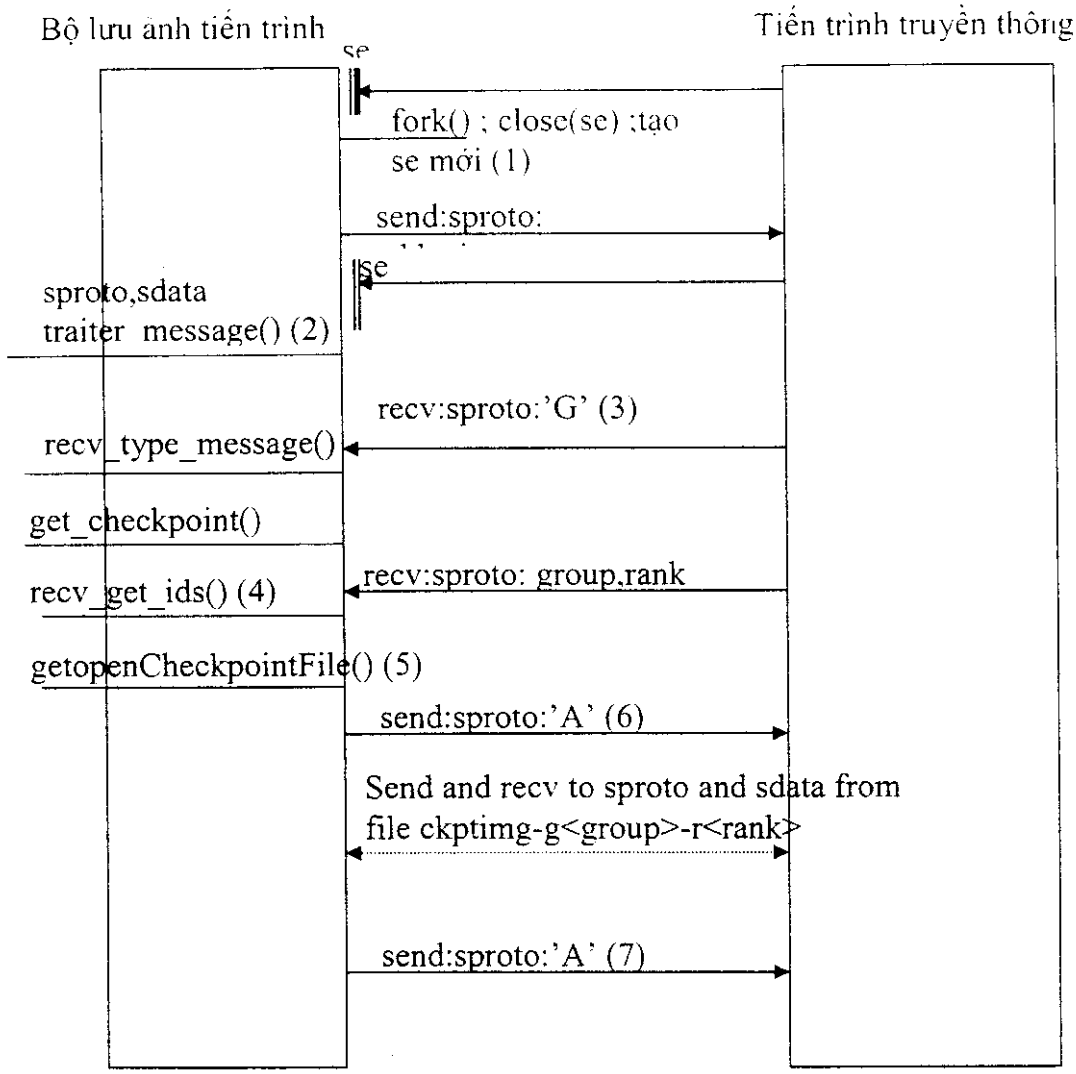
(3).Trong trường hợp nhận ảnh tiến trình thì kí tự đọc về là P.Hàm recv_type_message() trả về GET=1

(4).Hàm này trả về các giá trị group,rank lấy từ socket sproto.

(5).Hàm này truyền 3 tham số group,rank,seq để mở file với tên ckptimg-g<group>-r<rank> và đọc3 dữ liệu trên từ header của file.

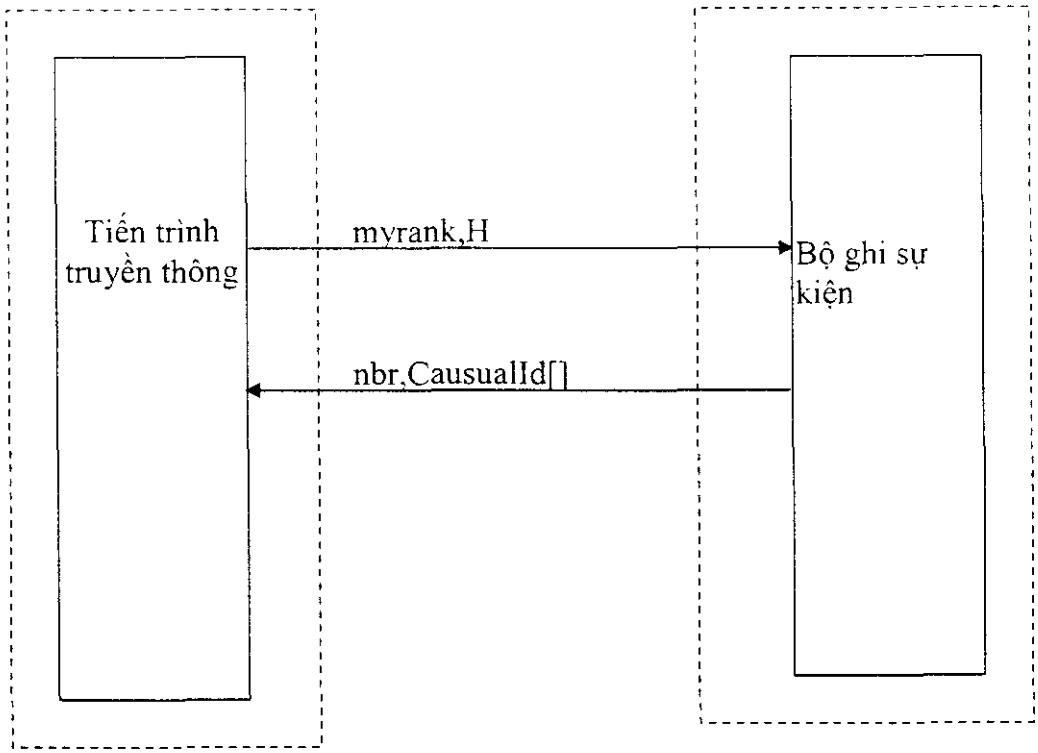
(6).gửi tín hiệu 'A'ccept về cho tiến trình truyền thông đang kết nối nhận ảnh.

(7).Quá trình gửi dữ liệu ảnh và danh sách các thông điệp đã gửi và sẵn sàng gửi được thực hiện theo cơ chế như trong hàm get_checkpoint().khi kết thúc công việc , sproto gửi về tín hiệu 'A' báo hoàn tất quá trình gửi file ảnh. Sau đó nó đóng các kết nối phục vụ tới tiến trình truyền thông.



Hình 4-12 Sơ đồ giao thức gửi ảnh tiến trình tới Checkpoint Server từ Daemon

Giao thức giữa tiến trình truyền thông và bộ lưu trữ các sự kiện



Hình 4-13 Giao thức truyền thông khởi tạo phục hồi giữa tiến trình truyền thông và bộ lưu trữ sự kiện

Trình bày về giao thức khởi tạo :

1. Tiến trình truyền thông gửi rank và giá trị đồng hồ của tiến trình lên cho Bộ ghi sự kiện qua socketEL trên nó.
2. Sau đó tiến trình truyền thông nhận danh sách các CausualId[ ] từ Bộ ghi sự kiện dựa trên giá trị đồng hồ h và rank của nó và trả về số các sự kiện tính từ giá trị h. Các giá trị này được ghi vào danh sách LinkedList_t.
3. Nếu danh sách LinkedList_t này rỗng , đồng nghĩa với việc tiến trình truyền thông là tiến trình mới khởi tạo, không phải là tiến trình chạy lại.

Các hàm truyền thông :

1. Từ Tiến trình truyền thông → Bộ ghi sự kiện :

```
el_get_reex(int s, long h, LinkedList_t *reex, int rank, int
*nbprobe_already_done )
```

2 thành phần quan trọng gửi về là danh sách reex và nbprobe_already_done

dựa vào 2 thành phần này , nếu khác trống có nghĩa là tiến trình ở chế độ chạy lại , 2 tham số này sẽ được sử dụng để tách danh sách các thông điệp ghi lại được gắn vào ảnh tiến trình lấy từ Bộ lưu trữ ảnh tiến trình về vào trong bảng ghi các sự kiện cần gửi lại cho tiến trình địa phương sau khi tiến trình này được chạy lại từ ảnh tiến trình bằng thư viện condor.Sau đó thì tiến trình MPI sẽ thăm dò nếu có thông điệp trong hàng đợi của tiến trình truyền thông thì nó sẽ nhận lại các thông điệp này. Nếu bằng trống thì tiến trình là hoàn toàn mới , không cần phải tách danh sách thông điệp với ảnh tiến trình.

2. Từ Bộ ghi sự kiện → Tiến trình truyền thông :

Biến dùng trong lưu trữ :

```
struct tabElement
{
    long int ident ; /*lưu trữ số hiệu rank*/
    int socketfd; /*lưu socket kết nối từ tiến trình truyền
thông tới Bộ ghi sự                                    kiện*/
    char* bufferBegin;/* dùng cho việc nhận thông điệp*/
    long int restant; /*để lưu số sự kiện sẽ gửi lại cho tiến
trình khi phục lỗi*/
    struct listEventLog * first; /*thành phần đầu tiên trong
danh sách log*/
    struct listEventLog * last; /*thành phần cuối trong danh
sách log*/
}
Struct listEventLog
```

```
{
      struct listEventLog * prec;
      struct eventLog * theEvent;
      struct listEventLog * next;
}
Struct eventLog
{
      long int identP; /*lưu rank của tiến trình truyền thông gửi*/
      long int senderClock;/*lưu giá trị đồng hồ bên gửi lúc gửi
thông điệp*/
      long int receptClock;/*lưu giá trị đồng hồ bên nhận lúc
nhận thông điệp*/
      long int nbProbe;/*số lần thăm dò kể từ lần nhận thông điệp
cuối cùng của bên                                          nhận*/
}
static struct tabElement ** conexion ;
struct waitingID ** tabOfWait ;

eventSending(struct tabElement * element, int filedesc)
```

Hàm này nằm trong luồng đón nhận kết nối của Bộ ghi sự kiện.

trong đó filedesc được lấy từ socket s do tiến trình truyền thông kết nối tới Bộ ghi sự kiện

Bộ ghi sự kiện hoạt động theo cơ chế đa luồng. nó sử dụng hai luồng phục vụ cho hai mục đích. Một là đón nhận các kết nối từ các tiến trình truyền thông lúc khởi tạo và khi chạy lại : luồng thực hiện hàm void * listenFct(). Hai là phục vụ việc ghi lại các sự kiện được gửi tới từ các tiến trình truyền thông trong quá trình trao đổi trao thông điệp : luồng này thực hiện hàm void * loggingFct().

CHƯƠNG 5. Module Chống Lỗi BKFT Cho Hệ Thống Tính Toán Song Song Ghép Cụm BKluster

BKluster là một hệ thống tính toán song song ghép cụm được xây dựng tại Trung tâm Tính Toán Hiệu Năng Cao trường Đại Học Bách Khoa Hà Nội. LAM/MPI đã được sử dụng để thiết lập môi trường song song có chống lỗi cho toàn bộ hệ thống. Nội dung chương này cũng trình bày về quá trình phân tích, phát triển module chống lỗi BKFT để có thể liên kết hoạt động của bộ quản lý tài nguyên và phân tải PBS với môi trường song song có chống lỗi.

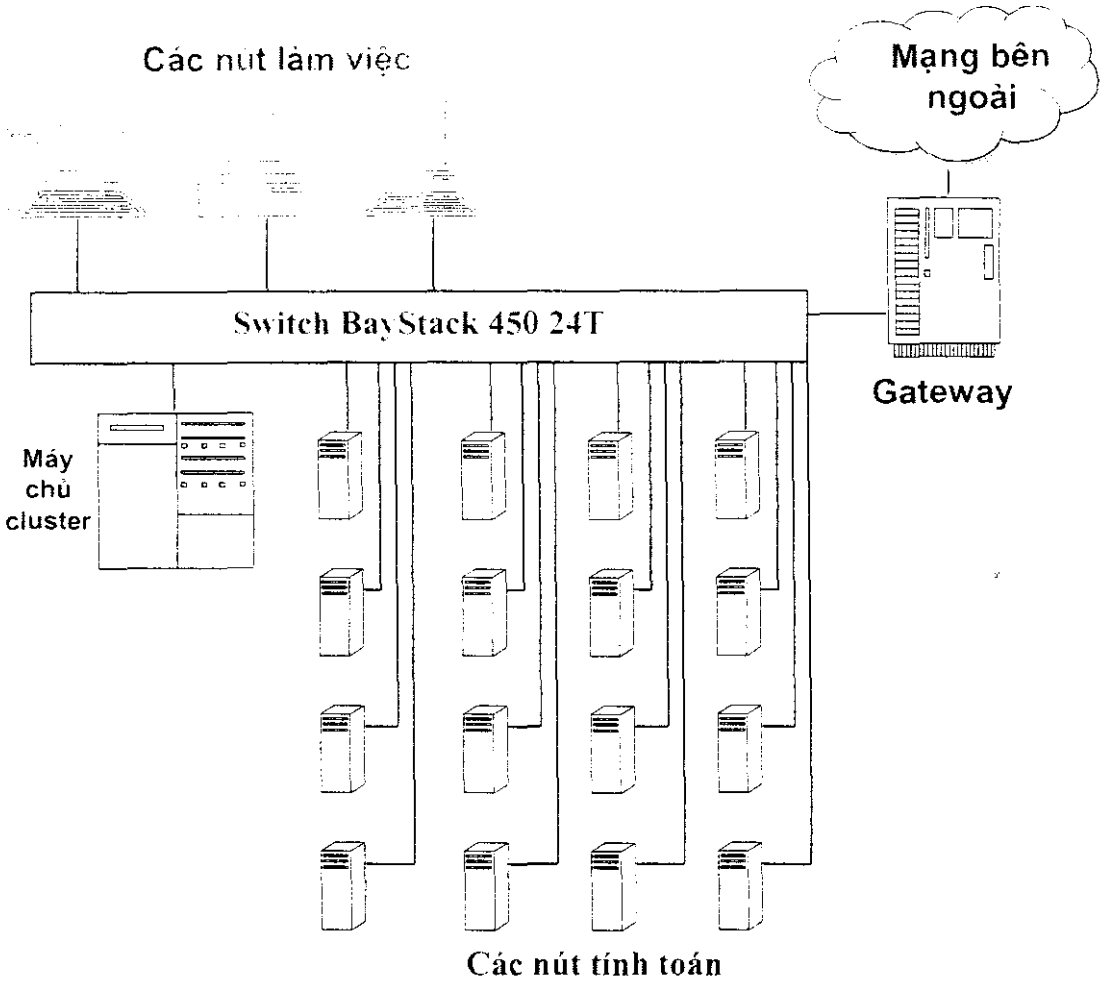
5.1. Thiết lập môi trường tính toán song song chống lỗi cho hệ thống BKluster

5.1.1. Hệ thống tính toán song song ghép cụm BKluster

BKluster là một hệ thống tính toán song song ghép cụm xây dựng theo mô hình hệ Beowulf từ những máy tính PC hoặc HP NetServer, công nghệ mạng Fast Ethernet. BKluster được thiết kế với những mục đích chính sau:

- Cho phép người sử dụng soạn thảo, quản lý và biên dịch mã nguồn của các ứng dụng song song theo chuẩn MPI.
- Cho phép người sử dụng đệ trình chương trình song song dưới dạng các công việc (các job), theo dõi và quản lý kết quả trả về của quá trình thực hiện job.
- Cung cấp cho người quản trị hệ thống các công cụ trực quan để quản lý cấu hình, quản lý người sử dụng và đánh giá hiệu năng của hệ thống.

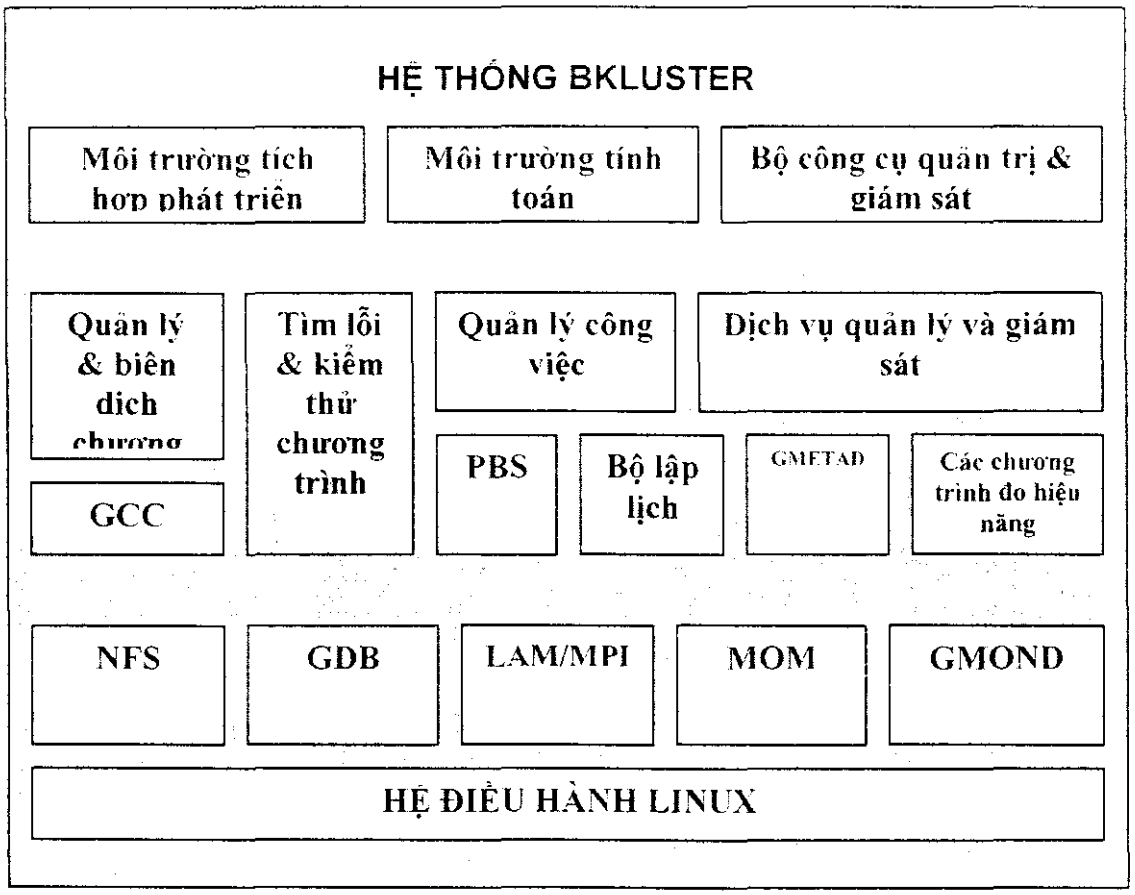
Toàn bộ hệ thống được kết nối theo kiến trúc mạng như sau:



Hình 5-1Kiến trúc mạng ghép nối của hệ thống BKluster

Để BKluster có thể hoạt động như một máy tính song song duy nhất, hỗ trợ nhiều mức người sử dụng, Trung tâm Tính Toán Hiệu Năng Cao đã phát triển bộ phần mềm BKlusware. BKlusware là một sự kết hợp giữa các gói phần mềm mã nguồn mở có sẵn với những module được phát triển bởi các nhóm nghiên cứu của trung tâm nhằm tạo ra một giải pháp phần mềm cho các hệ thống tính toán song song phân cụm. Với 3 chức năng chính: Phát triển tích hợp, Thực thi chương trình, Giám sát quản trị, BKlusware là một gói phần mềm lớn, nhiều thành phần được triển khai trên các nút tính toán và máy chủ cluster của hệ thống. Tuy nhiên, môi trường chống lỗi chỉ ảnh hưởng trực tiếp đến **PBS - hệ thống quản lý tài nguyên**

và phân tải của hệ thống. Nội dung các phần sau sẽ trình bày môi trường tính toán song song có chống lỗi, về kiến trúc và hoạt động của PBS, và quá trình phát triển module chống lỗi BKFT.



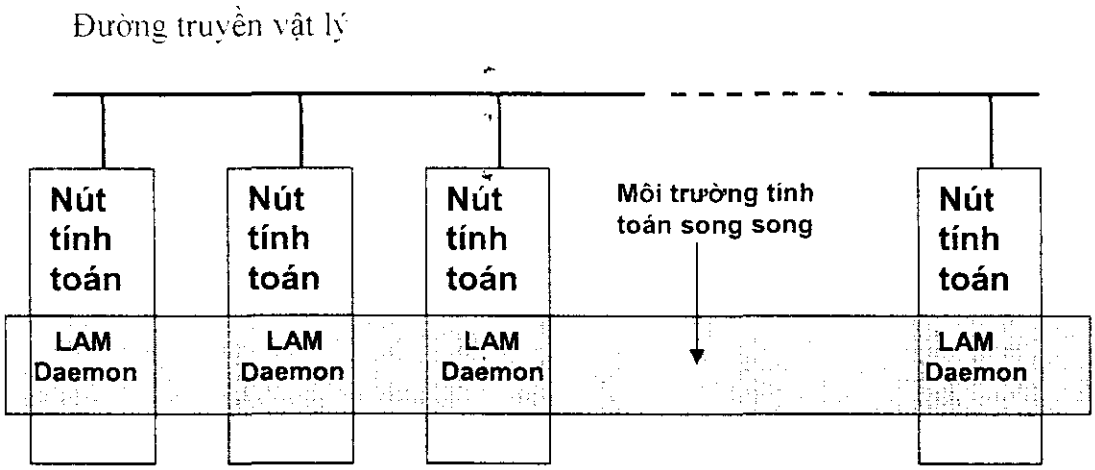
Hình 5-2 Kiến trúc phân tầng hệ thống BKluster

5.1.2. Môi trường tính toán song song có chống lỗi

Một chương trình song song viết theo mô hình truyền thông điệp (Message Passing) bao gồm nhiều tiến trình chạy trên các nút tính toán, trao đổi thông tin với nhau trong quá trình hoạt động. Hệ thống BKluster sử dụng gói phần mềm LAM/MPI để thực hiện việc thiết lập môi trường tính toán, biên dịch và chạy các ứng dụng song song. Sau khi cài đặt và cấu hình, LAM/MPI cung cấp cho người sử dụng các module sau:

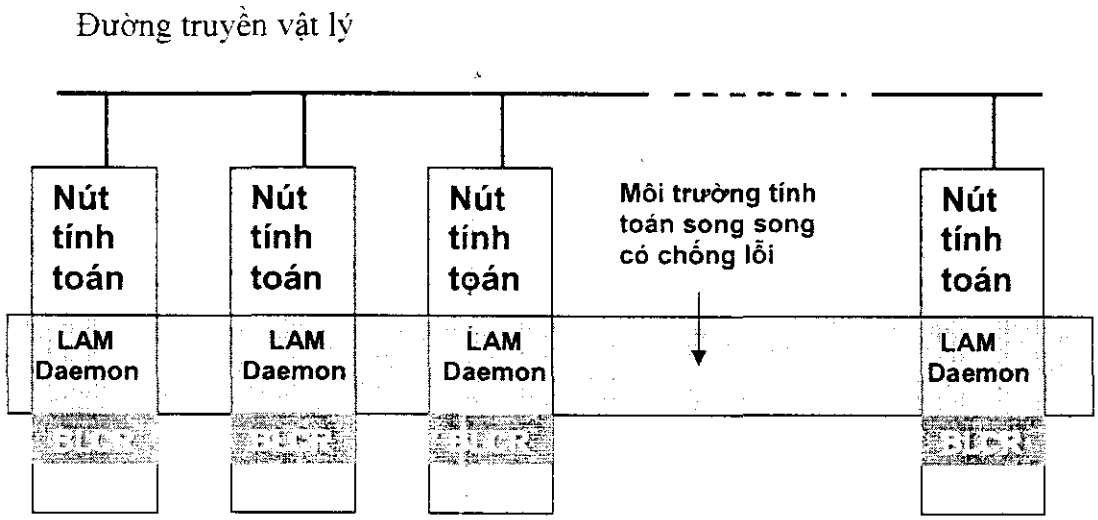
- Các script hỗ trợ việc biên dịch chương trình viết bằng các ngôn ngữ bậc cao như: C, C++, Fortran theo chuẩn MPI.

- Các tiến trình ngầm (daemon) đảm nhiệm việc quản lý và truyền thông điệp giữa các tiến trình song song. Các thông điệp được truyền thông qua giao thức TCP/IP



Hình 5-3 Môi trường tính toán song song trong hệ thống BKluster

Môi trường truyền thông có chống lỗi được triển khai dựa trên việc cấu hình BLCR hoạt động dưới dạng dịch vụ CR của LAM/MPI.



Hình 5-4 Môi trường tính toán song song có chống lỗi trong hệ thống BKluster

5.1.3. Hệ thống quản lý tài nguyên và phân tải PBS

Một vấn đề quan trọng đặt ra với các hệ thống đóng vai trò dịch vụ tính toán đó là phân bổ các tiến trình tính toán trên tài nguyên của hệ thống sao cho hiệu qua hoạt động của toàn bộ hệ thống là tối ưu. Để làm được việc này, trong hệ thống cần cài đặt các phần mềm có chức năng quản lý tài nguyên và lập lịch. Hai loại phần mềm này thường được tích hợp chung vào một gói phần mềm gọi là "**hệ quản lý tài nguyên và phân tải**". Trong hệ thống BKluster, hệ quản lý tài nguyên và phân tải được sử dụng là PBS – Portable Batch System. PBS là phần mềm mã nguồn mở của NASA. Hiện nay, PBS đang được ứng dụng và phát triển trong nhiều hệ thống tính toán song song ghép cụm trên Thế Giới.

PBS là một gói phần mềm mã nguồn mở được cung cấp kèm theo đầy đủ tài liệu đặc tả, hướng dẫn sử dụng. Những đặc điểm trên giúp cho việc cài đặt và cấu hình PBS có thể tiến hành một cách tương đối dễ dàng, đồng thời cũng tạo điều kiện cho việc cải tiến, tích hợp thêm các chức năng chuyên biệt cho phù hợp với mục đích sử dụng cụ thể.

Hệ thống quản lý tài nguyên và phân tải PBS bao gồm 3 module chính :

+ **pbs_server**: nhận và điều phối các yêu cầu tính toán.
+ **pbs_scheduler**: lập lịch cho các yêu cầu tính toán.
+ **pbs_mom**: quản lý tài nguyên và thực thi công việc trên các nút tính toán.

Việc quản lý tài nguyên và thực thi công việc là hai quá trình độc lập với nhau.

Khi tiến hành cài đặt, thông thường hai môđun pbs_server và pbs_scheduler được cài đặt trên máy chủ, pbs_mom được cài đặt trên các nút tính toán. Cả 3 môđun trên đều hoạt động dưới dạng các tiến trình ngầm – daemon.

Quá trình hoạt động của toàn bộ hệ thống như sau:

1. Người sử dụng đệ trình 1 công việc lên pbs_server.
2. pbs_server yêu cầu pbs_scheduler lập lịch.

3. pbs_scheduler yêu cầu pbs_server và pbs_mom các thông tin về trạng thái và tài nguyên.

4. pbs_server và pbs_mom chuyển các thông tin về trạng thái và tài nguyên cho pbs_scheduler.

5. pbs_scheduler tiến hành lập lịch và chuyển kết quả về cho pbs_server.

   Kết quả lập lịch có thể là 1 trong 2 trường hợp sau :

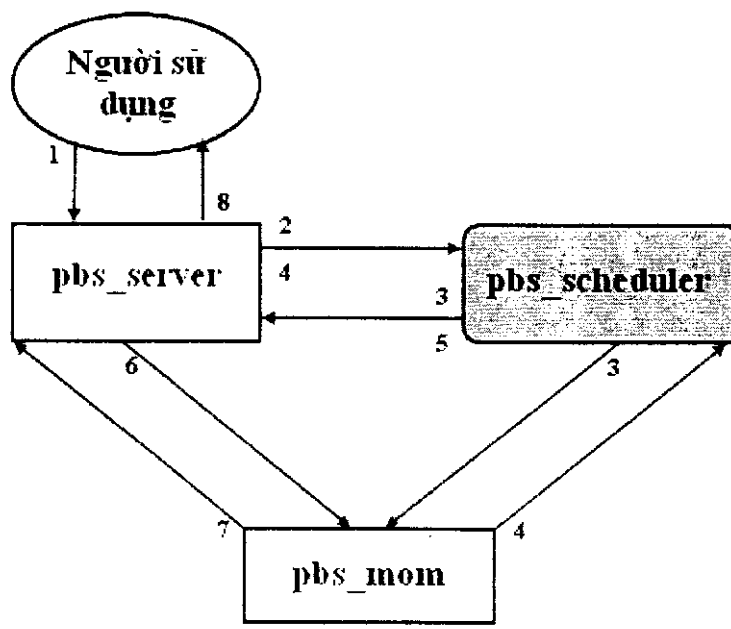
   + Xác định rõ tên công việc được chạy và danh sách các nút tính toán tương ứng

   + Không có công việc nào được chạy, trong trường hợp này công việc mới sẽ được đưa và hàng đợi.

6. Trong trường hợp có công việc được chạy, pbs_server sẽ phát tín hiệu yêu cầu các pbs_mom thực hiện công việc trên các nút tính toán.

7. Khi công việc kết thúc, pbs_mom chuyển kết quả cho pbs_server.

8. pbs_server trả kết quả về cho người sử dụng.



Hình 5-5 Hoạt động của hệ thống quản lý tài nguyên và phân tải PBS

Trong gói phần mềm PBS còn có các thư viện lập trình và một tập các câu lệnh cho phép người quản trị và người sử dụng dễ dàng tương tác với hệ thống ở chế độ dòng lệnh hoặc qua các chương trình viết bằng ngôn ngữ bậc cao. Với những đặc điểm trên, PBS đã đáp ứng được những yêu cầu cơ bản của người quản trị và người sử dụng hệ thống tính toán song song ghép cụm.

5.2. Nhược điểm của hệ LAM/MPI – BLCR khi sử dụng với PBS

Khi hoạt động kết hợp với LAM/MPI, BLCR đã thiết lập được một môi trường tính toán song song có khả năng checkpoint/restart các ứng dụng.

Tuy nhiên, khi triển khai trong mô hình chung của hệ thống BKluster, môi trường song song chống lỗi LAM/MPI – BLCR đã có những hạn chế sau:

- Không tương tác được với hệ thống quản lý tài nguyên và phân tải PBS để có thể lấy được danh sách các ứng dụng song song đang chạy trên hệ thống.
- Khi một chương trình song song được khởi động lại bằng BLCR, PBS không biết và không quản lý được công việc này

5.3. Module chống lỗi BKFT

Để khắc phục những nhược điểm trên, nhóm nghiên cứu đã phát triển môđun chống lỗi BKFT với những chức năng sau:

- Tương tác với PBS để lấy được danh sách các công việc đang thực hiện trong hệ thống.
- Gọi các lệnh của BLCR nhằm lấy checkpoint của một hoặc một số công việc bất kỳ do PBS quản lý
- Quản lý các checkpoint trong 1 cơ sở dữ liệu
- Đệ trình lại công việc song song từ các checkpoint với PBS

Khi thực hiện checkpoint một chương trình MPI bao gồm n tiến trình đang chạy. BLCR sẽ tạo ra n+1 context file để lưu giữ ảnh của chương trình được checkpoint:

- Một context file của chính tiến trình mpirun được thực hiện lệnh checkpoint. File này được lưu trữ trong thư mục tại thời điểm mà người dùng thực hiện lệnh cr_checkpoint.
- n context file còn lại chính là ảnh của từng tiến trình được tạo ra khi thực hiện chương trình song song. Các file này được ghi trong thư mục $HOME của người dùng.

Do đó cần phải quản lý các file ảnh của tiến trình này. Người dùng có thể chỉ định thư mục lưu trữ các file này bằng một trong hai cách sau:

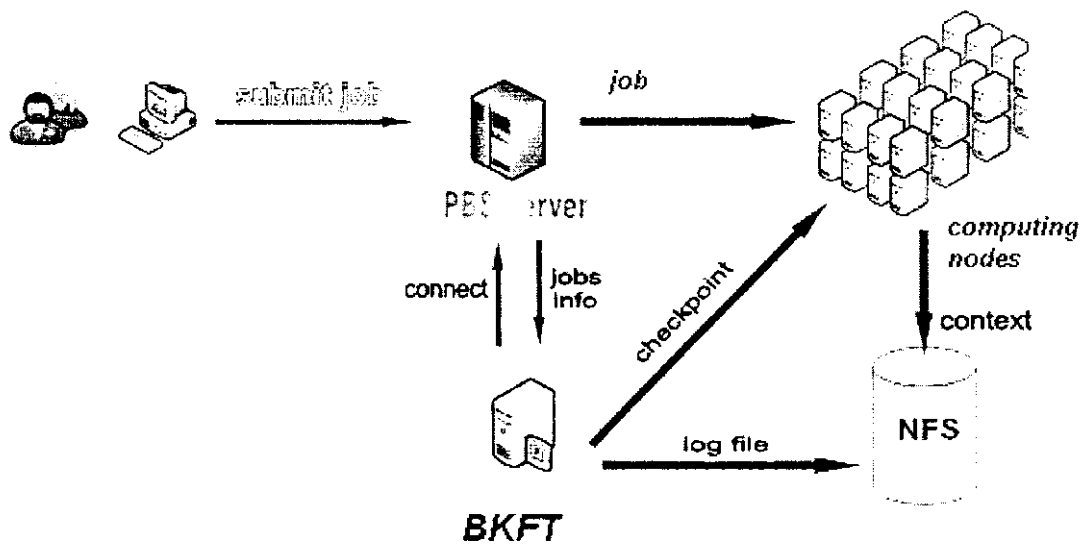
- Thêm tham số vào khi thực hiện lệnh mpirun: mpirun -ssi cr_blcr_base_dir PATH_TO_SAVE_FILE
- Thiết lập biến môi trường LAM_MPI_SSI_cr_base_dir

Khi thực hiện lệnh restart chương trình đã được checkpoint, các tiến trình của chương trình song song tiếp tục được thực hiện tại thời điểm checkpoint. Như vậy môi trường truyền thông giữa các tiến trình không được thiết lập (các tiến trình trong chương trình song song khi gặp lện MPI_Init sẽ thiết lập môi trường truyền thông, trao đổi thông tin cho nhau để nhận biết nhau). Do đó khi các tiến trình đã lưu ảnh tiếp tục được thực hiện thì không thể trao đổi thông tin với nhau được.

Việc thiết lập môi trường truyền thông giữa các tiến trình có thể được thực hiện bằng cách cho chạy một chương trình song song (có hàm MPI_Init) trên tất cả các máy trước khi restart lại chương trình đã được checkpoint. Do đây chỉ là bước thiết lập môi trường truyền thông, chương trình song song này nên đơn giản, thậm chí có thể không thực hiện lệnh gì ngoài lệnh MPI_Init. Bằng cách này các máy trong môi trường đã được nhận biết nhau và các tiến trình được phục hồi tại thời điểm restart có thể trao đổi thông điệp với nhau. Chương trình song song tiếp tục được thực thi mà không phải chạy lại từ đầu khi xảy ra lỗi.

Trong hệ thống BKluster, việc thực hiện khởi tạo môi trường LAM (lamboot) được thông qua PBS. Với mỗi lần đệ trình công việc, lệnh lamboot sẽ tạo ra các session của LAM được ghi trên các thư mục khác nhau do PBS tạo ra. Khi restart lại chương trình cũng cần phải có công việc khởi tạo môi trường LAM và session của LAM lúc này sẽ khác với lúc thực hiện checkpoint. Vấn đề xảy ra là khi checkpoint, thư mục chứa sesion của LAM được ghi cố định trong file ảnh của tiến trình. Do đó các tiến trình sau khi restart sẽ không tham chiếu session của LAM hiện thời mà tham chiếu đến session của LAM lúc thực hiện checkpoint. Các tiến trình lại không thể truyền thông điệp cho nhau.

Để giải quyết vấn đề, là phải làm cho session của LAM khi thực hiện restart giống như sesion của LAM khi thực hiện checkpoint. Trong LAM cung cấp các biến LAM_MPI_SESSION_PREFIX, LAM_MPI_SESSION_SUFFIX, TMPDIR để người dùng có thể chỉ định thư mục sesion. Như thế, công việc trước khi thực hiện lệnh lamboot để sau đó restart lại chương trình là xác định lại thư mục chứa session của LAM bằng cách thay đổi các biến môi trường kể trên.



Hình 5-6 BKluster với môi trường tính toán song song chống lỗi

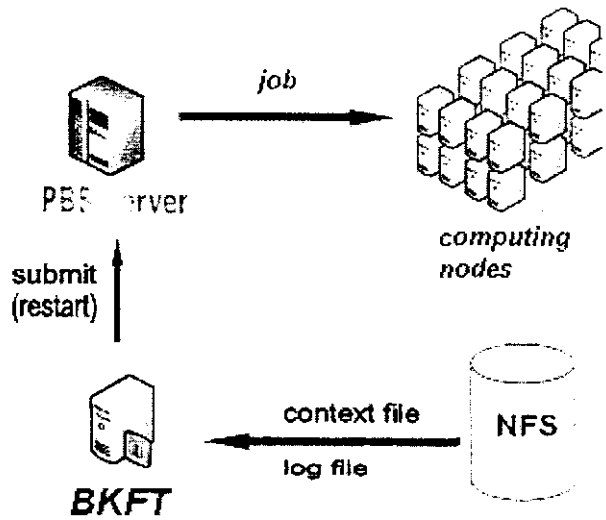
Kịch bản thực hiện công việc của hệ thống BKluster với khả năng chống lỗi như sau:

- Công việc song song được đệ trình đến PBS server qua Hệ thống thực thi
- BKFT định kỳ truy vấn đến PBS server để cập nhật danh sách công việc

- BKFT định kỳ lấy checkpoint các công việc đang thực hiện trên hệ thống. BKFT cũng cho phép người quản trị lấy checkpoint của một công việc vào một thời điểm bất kỳ
- Các checkpoint được cập nhật vào cơ sở dữ liệu trên hệ thống file mạng NFS

Khi một hoặc một số tiến trình gặp lỗi, nó sẽ được BKFT khởi động lại như sau:

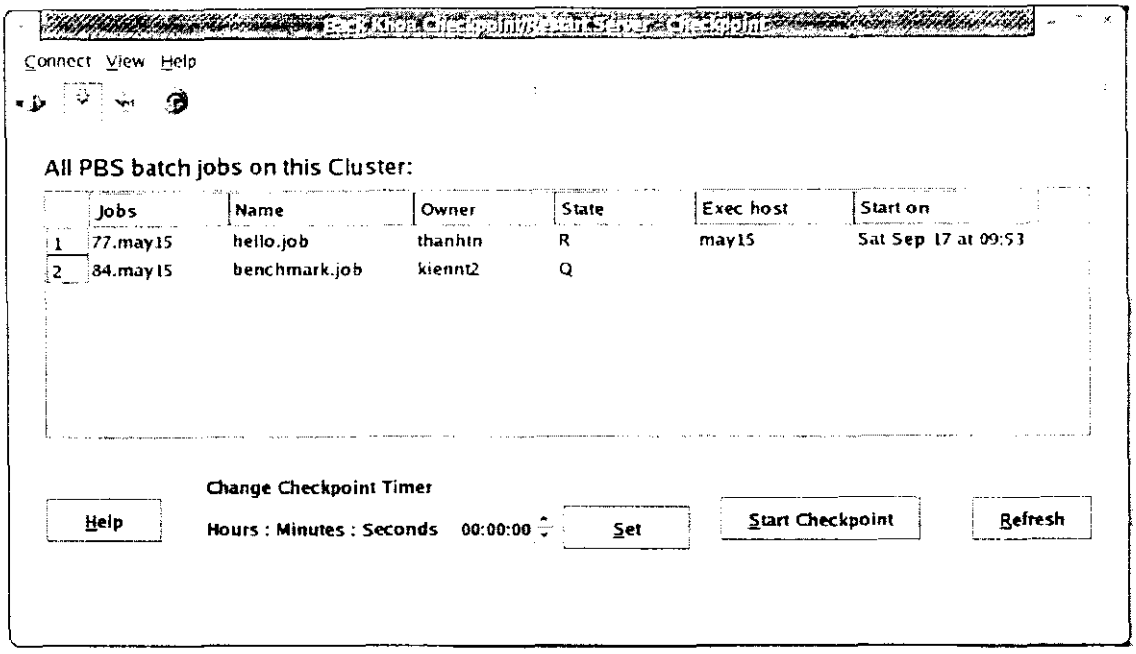
- Đọc lại checkpoint từ cơ sở dữ liệu
- Tự động sinh ra script, khởi động lại công việc từ thời điểm lấy checkpoint dưới dạng một công việc được quản lý bởi PBS



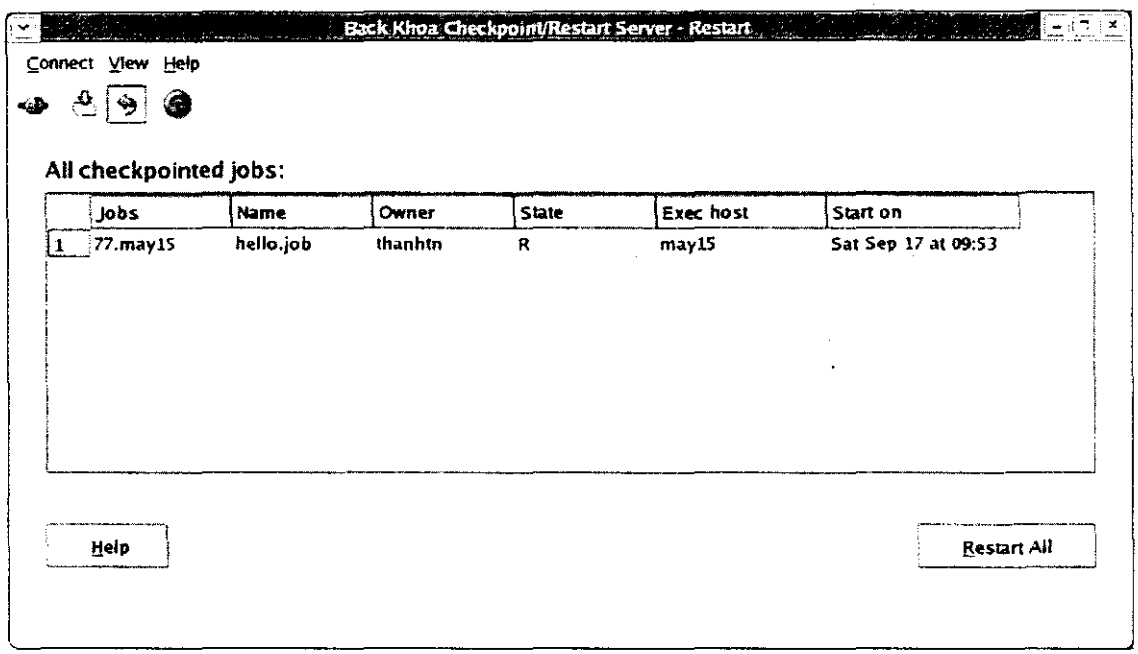
Hình 5-7 Khởi động lại công việc với BKFT

5.4. Các kết quả đạt được và hướng phát triển

Module BKFT đã được hoàn thành và tích hợp hoạt động tốt cùng với gói phần mềm BKlusware. Với khả năng chống lỗi, BKluster có thể phục hồi lại toàn bộ công việc từ thời điểm gần nhất trước khi xảy ra lỗi hệ thống.



Hình 5-8 Giao diện chương trình BKFT - chức năng lấy checkpoint



Hình 5-9 Giao diện chương trình BKFT - chức năng restart

Tài liệu tham khảo

[1]. Mootaz, Lorenzo, Yi-min Wang, David B.Johnson, "A Survey of Rollback Recovery Protocols in Message Passing Systems", *Technical Report CMU-CS-96-181, Dept. of Computer Science, Carnegie Mellon University*, 1996

[2]. K.Many Chandy and Leslie Lamport, "Distributed Snapshots: Determining

Global States of Distributed Systems," *ACM Trans. Computer Systems*, Feb. 1985.

[3]. R. Koo and S. Toueg. "Checkpointing and rollbackrecovery for distributed systems", *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[4]. S. Kalaiselvi and V. Rajaraman, "A survey of checkpointing algorithms for parallel and distributed computers", *Sadhana, Vol. 25, Part 5*, October 2000, pp. 489±510.

[5]. Jinhua Guo, "Fault Tolerant Computing", *Ph.D*, Winter 2004

[6].Flaviu Cristian, "Understanding Fault-Torerant Distributed Systems", May 25, 1993

[7]. Guohong Cao, and Mukesh Singhal, "On Coordinated Checkpointing in Distributed Systems", 1998

[8] Aur´elien Bouteiller, Franck Cappello, Thomas H´erault, G´eraud Krawezik, Pierre Lemarinier, Fr´ed´eric Magniette. MPICH-V : A Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Base Message Logging. *LRI, Universit´e de Paris Sud, Orsay, France INRIA Futurs, Saclay, France* URL: http://www.lri.fr/ ~gk/MPICH-V

[9] L. Alvisi and K. Marzullo. Message logging : Pessimistic, optimistic, and causal. In *Proceedings of the 15th International Conference on Distributed ComputingSystems (ICDCS 1995)*, pages 229–236. IEEE CS Press, May-June 1995.

[10] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS Parallel Benchmarks 2.0. Report NAS-

95-020, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, 1995.

[11] George Bosilca, Aur´elien Bouteiller, Franck Cappello, Samir Djilali, Gilles F´edak, C´ecile Germain, Thomas H´erault, Pierre Lemarinier, Oleg zodygensky, Fr´ed´eric Magniette, Vincent N´eri, and Anton Selikhov. Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes. In *SC2002: High Performance*

Networking and Computing (SC2002), Baltimore USA, Novembre 2002. IEEE/ACM.

[12] M. 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, October 1996.

[13] Robert E.Strom, David F. Bacon, and Shaula A. Yemini. Volatile logging in n-fault-tolerant distributed systems. In *18th Annual International Symposium on Fault-Tolerant Computing (FTCS-18)*, pages 44–49. IEEE CS Press, June 1988.

[14] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of unix processes in the condor distributed processing system. Technical Report Technical Report 1346, University of Wisconsin-Madison, 1997.

[15] Sriram Rao, Lorenzo Alvisi, and Harrick M. Vin. Egida: An extensible toolkit for low-overhead faulttolerance. In *29th Symposium on Fault-Tolerant Computing (FTCS'99)*, pages 48–55. IEEE CS Press, 1999.

[16] Georg Stellner. Cocheck: Checkpointing and process migration for mpi. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, April 1996. IEEE CS Press.

[17] Yuqun Chen, Kai Li, and James S. Planck. Clip: A checkpointing tool for message-passing parallel programs. In *SC97: High Performance Networking and Computing (SC97)*. IEEE/ACM, November 1997.

[18] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, Eric Roman. "The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing", *Open Systems Laboratory, Indiana University and Lawrence Berkeley National Laboratory*

[19] Jason Duell, "The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart", *Lawrence Berkeley National Laboratory*