DIỄN VIÊN CHO INTERNET OF THINGS

qua

Arjun Shukla

Một luận án

nộp một phần để hoàn thành

của các yêu cầu về mức độ

Thạc sĩ Khoa học Máy tính

Đại học Boise State

Tháng 8 năm 2021

TRƯỜNG CAO ĐẲNG ĐẠI HỌC BANG BOISE

CHẤP NHẬN ĐỀ XUẤT

của luận án được nộp bởi

Arjun Shukla

Tiêu đề luận án: Các tác nhân cho Internet vạn vật

Ngày thi vấn đáp cuối kỳ: 25 tháng 6 năm 2021

Những cá nhân sau đây đã đọc và thảo luận về luận án do sinh viên Arjun Shukla nộp, và họ đã đánh giá bài thuyết trình và trả lời các câu hỏi trong kỳ thi vấn đáp cuối cùng. Họ thấy rằng sinh viên đã vượt qua kỳ thi vấn đáp cuối cùng.

Tiến sĩ Amit Jain	Chủ tịch, Ủy ban giám sát
Tiến sĩ Catherine Olschanowsky	Thành viên, Ủy ban giám sát
Casey Kennington, Tiến sĩ	Thành viên, Ủy ban giám sát

Sự chấp thuận đọc cuối cùng của luận án đã được Amit Jain, Tiến sĩ, Chủ tịch Ủy ban giám sát chấp thuận. Luận án đã được chấp thuận bởi Cao đẳng sau đại học.

LỜI CẢM Ơ N

Tác giả xin gửi lời cảm ơ n đến cha mẹ Laura và Ashish, và gửi đến dì Arti, vì tình yêu thư ơ ng và sự ủng hộ của họ. Anh ấy biết ơ n Tiến sĩ Amit Jain vì sự hư ớng dẫn của ông và Khoa Khoa học Máy tính của Đại học Boise State đã tài trợ cho ông nghiên cứu.

TÓM TẮT

Mô hình diễn viên là một mô hình tính toán đồng thời, tập trung vào tin nhắn truyền giữa các thực thể trong một hệ thống. Nó rất phù hợp cho lập trình phân tán, do ngữ nghĩa của nó bao gồm rất ít đảm bảo hoặc giả định về độ tin cậy. Việc triển khai mô hình diễn viên đã trở nên phổ biến hơ n ở nhiều ngôn ngữ.

Thư viện Akka (viết bằng Scala) là một trong những thư viện diễn viên phổ biến nhất. Tuy nhiên, Akka còn thiếu một số tính năng chính. Mục tiêu của chúng tôi là tạo ra diễn viên của riêng mình thư viện có tên là Aurum, không chỉ có những tính năng này mà còn có hiệu suất cao hơ n mance. Các tính năng mới bao gồm các cách dễ dàng để tạo tham chiếu, cấu hình và khởi chạy cụm, dịch loại tin nhắn và khả năng đư a tin nhắn bị loại bỏ và chậm trễ vào mọi phần của ứng dụng. Aurum sẽ đư ợc triển khai trong Rust, một chư ơ ng trình ngôn ngữ đư ợc thiết kế cho hiệu suất cao, không đồng bộ và mức độ trừu tư ợng cao rất phù hợp với các thiết bị IoT.

Kết quả của chúng tôi cho thấy Aurum hoạt động tốt hơ n Akka. Trong các chuẩn mực của chúng tôi, một máy chủ chạy Aurum cung cấp thông lư ợng gấp ba lần so với máy chủ Akka tư ơ ng đư ơ ng, trong khi vẫn duy trì khả năng lập trình tốt và có các tính năng hữu ích cho IoT.

MỤC LỤC

DANH SÁCH CÁC BẢNG

DANH SÁCH CÁC HÌNH ẢNH

CHƯƠNG 1

GIỚI THIỆU

Mô hình diễn viên của tính toán đồng thời [1] gần đây đã nhận được sự chú ý lớn hơn
vì tiềm năng tận dụng nhiều bộ xử lý trên một máy duy nhất và
sự đơn giản của việc dịch các thuật toán sang một môi trường phân tán. Trong mô hình này,
các diễn viên là các luồng chỉ có quyền truy cập duy nhất vào không gian bộ nhớ của họ. Họ có thể sửa đổi một
trạng thái riêng tư , nhưng chỉ có thể tương tác với các tác nhân khác thông qua việc truyền không đồng bộ
của dữ liệu không thay đổi. Các diễn viên xử lý các thông điệp theo trình tự, được lưu trữ trong hàng đợi (hoặc
"hộp thư") cho đến lúc đó. Tương tự như ngữ nghĩa giao tiếp mạng, tin nhắn của diễn viên
hoạt động với giao hàng nhiều nhất một lần. Không có đảm bảo tin nhắn được nhận trong
cùng thứ tự chúng được gửi, thậm chí từ cùng một tác nhân nguồn. Khả năng chịu lỗi đến
tự nhiên đối với các diễn viên, những người có thể bắt đầu, chấm dứt và giám sát các diễn viên khác. Các diễn viên
gửi tin nhắn thông qua tham chiếu diễn viên, được biết từ các tin nhắn khác hoặc sinh sản
diễn viên đó. Tham chiếu đến diễn viên là vị trí minh bạch: ngữ nghĩa chuyển phát tin nhắn
không thay đổi theo vị trí của các diễn viên mục tiêu. Khả năng mở rộng hiện rất đơn giản, vì
việc triển khai tới nhiều nút không cần phải thay đổi bất kỳ mã hiện có nào.

Mô hình diễn viên có thể lý tưởng cho việc triển khai các ứng dụng trong Internet
của Sự vật [2]. Để đạt được mục đích này và để dễ dàng phát triển, một chương trình cấp hệ thống
việc triển khai ngôn ngữ của mô hình diễn viên là cần thiết, do đó các diễn viên có thể thuận tiện
tương tác với các thiết bị vật lý được kết nối với máy mà chúng được đặt trên đó. IoT

các thiết bị có thể đang gửi một lượng lớn dữ liệu thô đến các máy chủ phụ trách chúng. Để đối phó với tải trọng cao, việc triển khai của chúng tôi phải có hiệu suất cao, với thời gian chạy tối thiểu chi phí chung như thu gom rác, biên dịch đúng lúc, xử lý ngoại lệ hoặc sắp xếp giữa các không gian bộ nhớ của các tác nhân cục bộ. Môi trường IoT yêu cầu nhanh chóng thời gian khởi động, vì sự cố có thể xảy ra thường xuyên.

Lý tưởng nhất là mô hình diễn viên sẽ an toàn về kiểu. Điều này có nghĩa là liệu một diễn viên có thể việc có nhận được một loại tin nhắn nhất định hay không được quyết định tại thời điểm biên dịch, dẫn đến an toàn hơn chương trình. Tài liệu tham khảo về các diễn viên cần thiết để gửi tin nhắn phải có thể làm giả được (tức là chúng có thể được tạo ra một cách tự phát). An toàn kiểu bổ sung thêm những phức tạp mới cho một tác nhân mô hình. Một triển khai tốt sẽ làm cho những điều này không quan trọng hoặc ít nhất là dễ dàng cho người lập trình cần phải làm việc cùng.

1.1 Triển khai mô hình diễn viên hiện có

Một số thư viện diễn viên phổ biến đã tồn tại, nhưng không có thư viện nào đáp ứng được nhu cầu của chúng tôi cho. Trong Erlang [3], ngôn ngữ và BEAM VM tự nó được xây dựng xung quanh các diễn viên, nhưng có thể bị ảnh hưởng bởi hiệu suất kém do kiểu động, cách diễn giải và cô lập không gian bộ nhớ cho các tác nhân cục bộ. Erlang là một ngôn ngữ được gõ động, và không cung cấp loại an toàn mà chúng tôi đang tìm kiếm. Akka là một người mẫu diễn viên được viết cho Java Virtual Machine (JVM). Hiệu suất của nó bị ảnh hưởng bởi JVM biên dịch và thu gom rác đúng lúc. Akka không đi kèm với cùng một đảm bảo an toàn bộ nhớ như Erlang, vì các thông điệp gửi đến các tác nhân cục bộ không được sao chép, nhưng chỉ tham chiếu của họ. Trong trường hợp các đối tượng có thể thay đổi được đang được chuyển giao, vị trí tính minh bạch bị phá vỡ. Trong tình huống này, hiện tại có một trạng thái có thể thay đổi, được chia sẻ giữa các diễn viên địa phương. Chỉ có tuần tự hóa và hủy tuần tự hóa mới cung cấp một

sao chép trong Java, phư ơ ng thức clone() trong Java là nông theo mặc định. Tuần tự hóa trong cục bộ
nhắn tin có thể đư ợc kích hoạt trong Akka [9], như ng hiệu suất đư ợc hư ởng lợi từ việc chia sẻ
bộ nhớ bị mất. Trong Java, tuần tự hóa là một kiểm tra thời gian chạy, làm giảm loại của chúng tôi
sự an toàn.

1.2 Luận đề

Chúng ta có thể phát triển một triển khai an toàn kiểu của mô hình diễn viên an toàn hơ n khi sử dụng không?
trong việc phát triển các giải pháp phân tán và có hiệu suất cao hơ n mô hình diễn viên hiện có
triển khai? Trư ờng hợp sử dụng đư ợc quan tâm đặc biệt là các thiết bị IoT. Chúng tôi sẽ so sánh
màn trình diễn của chúng tôi với ngư ời mẫu diễn viên Akka.

1.3 Ngôn ngữ lập trình Rust

Rust [10] là một ngôn ngữ hệ thống, đủ cấp thấp để dễ dàng tư ơ ng tác với cục bộ
thiết bị. Nó kết hợp các kiểm tra thời gian biên dịch mở rộng để đảm bảo an toàn bộ nhớ.
trình biên dịch ngăn chặn các con trỏ lơ lửng, hầu hết các rò rỉ bộ nhớ và dữ liệu chạy đua qua
theo dõi luồng nào trong phạm vi nào sở hữu một phần bộ nhớ, cho cả hai
ngăn xếp và đống. Đư ợc tận dụng đúng cách, điều này có thể hoạt động như một
thay thế cho việc sắp xếp dữ liệu giữa các tác nhân cục bộ: nhiều triển khai của
Mô hình diễn viên tạo ra các không gian bộ nhớ riêng biệt rõ ràng, nơ i cần phải sao chép.
Quản lý bộ nhớ của Rust đư ợc xử lý hoàn toàn tại thời điểm biên dịch, bộ nhớ đư ợc giải phóng
khi các tham chiếu sở hữu đến nó nằm ngoài phạm vi. Thiếu bộ sư u tập rác thời gian chạy là
có khả năng cải thiện đáng kể hiệu suất. Thay vì các ngoại lệ, Rust xử lý lỗi
thông qua các mẫu khớp trên các giá trị trả về. Xác định luồng điều khiển với giá trị trả về
giá trị nhanh hơ n nhiều so với việc duyệt qua ngăn xếp. Tất cả các tệp thực thi của Rust

Bảng 1.1: So sánh tính năng giữa Rust, Scala và Erlang

Tính năng Rust		thang độ	Tiếng Việt:
Không đồng bộ	Tích hợp với Akka Tích hợp		
Sự phản xạ	KHÔNG	Đúng	Không có
Đồng thời an toàn	Đúng	KHÔNG	Đúng
Tuần tự hóa với Serde	tích hợp		Tích hợp sẵn

mã được tạo ra tại thời điểm biên dịch, giảm thiểu thời gian khởi động chương trình, hữu ích trong tình huống thường xuyên xảy ra sự cố và khởi động lại.

Chúng tôi sẽ so sánh Rust với 2 ngôn ngữ lập trình khác phổ biến trong phân tán máy tính: Scala và Erlang. Akka được triển khai trong Scala và có nghĩa là được sử dụng với Scala (nhưng cũng có Java API). Erlang có các tác nhân được tích hợp sẵn trong ngôn ngữ. Bảng 1.1 so sánh các tính năng ngôn ngữ của Rust, Scala và Erlang có liên quan đến mô hình diễn viên. Việc xử lý Futures của trình biên dịch Rust biến chúng thành các máy trạng thái hiệu quả có thể được quản lý bởi người thực thi và người lập lịch. Việc Rust thiếu phản ánh là vấn đề khi cố gắng triển khai các tác nhân phân tán, nhưng một giải pháp thay thế được triển khai trong thư viện. Rust theo dõi khả năng thay đổi thông qua các loại tham chiếu. Nó đảm bảo đồng thời an toàn sử dụng các đặc điểm Gửi và Đồng bộ. Serde là một thư viện Rust với các đặc điểm và macro để hỗ trợ tuần tự hóa. Rust kiểm tra các triển khai đặc điểm tại thời điểm biên dịch, do đó không có thời gian chạy có thể xảy ra lỗi liên quan đến các kiểu không tuần tự hóa.

1.4 Các hệ thống diễn viên khác cho Rust

Nhiều dự án phần mềm hiện đang triển khai mô hình diễn viên trong Rust. Actix [7] là phát triển tốt nhất. Nó đã trưởng thành và hoạt động đầy đủ, nhưng nó chỉ cho phép diễn viên địa phương. Có plugin mới được gọi là actix-remote, nhưng nó chưa được phát triển và không triển khai các tác nhân từ xa cho actix. Dự án bastion-rs [5] là

được tạo ra với mục tiêu tạo ra một giao diện diễn viên Erlangesque cho Rust, nhưng nó được dùng cho các hệ thống phân tán quy mô nhỏ theo trang web của họ. Bastion cũng sử dụng các tham chiếu diễn viên không được gõ, không thể làm giả, không xử lý tuần tự hóa hoặc hủy tuần tự hóa, không hỗ trợ phân phối tĩnh và không thể gửi tin nhắn từ xa bên ngoài cụm. Dự án Riker [4] được tạo ra để mô phỏng Akka phổ biến thư viện cho Scala [8]. Trong khi Riker có vẻ như có thể phát triển thành Akka-for-Rust thư viện cuối cùng, nhiều tính năng cần thiết vẫn chưa được triển khai như điều khiển từ xa diễn viên, tính minh bạch của vị trí và nhóm. Cuối cùng, Acteur [6] là một dự án khá mới, và không hỗ trợ điều khiển từ xa hoặc nhóm. Không có thư viện nào trong số này đủ để sử dụng ngay để có được chức năng chúng ta mong muốn.

1.5 Bối cảnh

Chúng tôi giả định một số kiến thức về hệ thống phân tán và đồng thời, bao gồm máy chủ, khách hàng và luồng. Hệ thống phân tán của Martin van Steen và Andrew Tanen-baum [29] là một nguồn tài nguyên tốt để tìm hiểu các nguyên tắc của hệ thống phân tán. sách có thể được tải xuống và đọc miễn phí.

CHƯƠNG 2

TÍNH NĂNG VÀ THIẾT KẾ CỦA THƯ VIỆN

Thư viện của chúng tôi sẽ được gọi là Aurum, được đặt theo tên của một nhóm rô-bốt ngoài hành tinh trong video trò chơi Kid Icarus: Uprising. Phần này sẽ đề cập đến nhiều tính năng khác nhau của Aurum và cách chúng được thực hiện.

2.1 Đồng thời cục bộ

Về mặt logic, mỗi diễn viên chạy trên luồng riêng của mình và các thông điệp được nhận nguyên tử. Các diễn viên phải nhẹ và có khả năng mở rộng cục bộ tốt, nghĩa là luồng hệ thống không phải là lựa chọn tốt để chạy các diễn viên. Tokio là một thư viện Rust với một môi trường thời gian chạy để lập lịch các luồng xanh (mà Tokio gọi là “nhiệm vụ”). Nó cần ít nhất một tác vụ Tokio để chạy một diễn viên. Các tác vụ nhẹ và nhiều diễn viên có thể chạy trên một luồng hệ thống duy nhất. Tokio tích hợp với trình biên dịch Rust các tính năng không đồng bộ để lên lịch các tác vụ của mình. Aurum hoạt động theo các quy tắc tương tự như tính bất đồng bộ của Rust (và Tokio): các tác vụ bị ràng buộc bởi IO. Mã chạy trong các tác vụ là dự kiến sẽ phải `chờ đợi` thường xuyên và tình trạng đói có thể xảy ra nếu mã không đồng bộ thực hiện các hoạt động tính toán nặng hoặc IO đồng bộ. Có kế hoạch thêm hỗ trợ cho tính toán nặng nề cho Aurum trong tương lai.

2.2 Tính minh bạch của vị trí

Có 2 loại tham chiếu diễn viên, LocalRef và ActorRef. LocalRef không phải là
có thể tuần tự hóa và chấp nhận các thông điệp triển khai đặc điểm Gửi của Rust. Một LocalRef
trực tiếp đặt tin nhắn vào hàng đợi tin nhắn của diễn viên nhận. Một LocalRef
loại là cần thiết vì một số loại tin nhắn có thể không được tuần tự hóa. ActorRefs làm
không chấp nhận các thông điệp không thể tuần tự hóa, mặc dù ActorRefs có thể mong đợi
các loại tin nhắn không thể tuần tự hóa tồn tại. Tất cả ActorRefs, ngay cả những loại mong đợi không
tin nhắn có thể tuần tự hóa, có thể tuần tự hóa. Chúng chứa thông tin cần thiết để liên hệ với
diễn viên từ xa: vị trí, cổng và tên của nó. ActorRefs cũng có thể chứa LocalRef
nếu họ đề cập đến một diễn viên địa phương. Gửi tin nhắn với ActorRef có thể sử dụng
LocalRef bên trong nếu nó tồn tại. ActorRefs là vị trí trong suốt: lựa chọn của nó giữa
việc gửi dữ liệu cục bộ và từ xa diễn ra mà người dùng không cần phải quan tâm.

2.3 Tham chiếu diễn viên đã nhập

Tham chiếu diễn viên được nhập, nghĩa là họ chỉ có thể nhận được tin nhắn có kiểu cụ thể.
Các tham chiếu được nhập tăng thêm tính an toàn bằng cách chuyển một số lỗi thời gian chạy thành lỗi thời gian biên dịch,
và loại bỏ nhu cầu sử dụng các nhánh khớp lệnh bắt tất cả để xử lý các thông báo không hợp lệ.
Nó cũng làm cho mã dễ đọc hơn, chú thích kiểu cung cấp thêm gợi ý về vai trò của tác nhân
trong chương trình. Tham chiếu diễn viên được nhập tĩnh, nghĩa là loại tin nhắn
họ nhận được phải được xác định tại thời điểm biên dịch. Generic được sử dụng để giao tiếp
trong đó có các loại tin nhắn.

ActorRefs có thể được tuần tự hóa và hủy tuần tự hóa. Khi ActorRef được hủy tuần tự hóa là
được tải vào cấu trúc, chúng ta cần đảm bảo ActorRef thực sự tham chiếu đến một
diễn viên có cùng loại thông điệp mà trình biên dịch nghĩ rằng nó có. Loại thông tin về

tham chiếu diễn viên phải đư ợc tuần tự hóa và gửi cùng với phần còn lại của ActorRef. Khi
thông tin loại đư ợc hủy tuần tự hóa, cần phải thực hiện kiểm tra để đảm bảo diễn viên
tham chiếu có giá trị đối với loại mà nó đang đư ợc tải vào. Nói cách khác, chúng ta đang yêu cầu
một phần dữ liệu về thông tin kiểu của nó tại thời điểm chạy (đư ợc gọi là phản xạ). Vì Rust
không có sự phản chiếu, Aurum giải quyết vấn đề này bằng cách duy trì một
danh sách tất cả các loại tin nhắn đư ợc sử dụng trong hệ thống dư ới dạng enum. Mỗi loại là
đư ợc biểu diễn bằng một biến thể khác trong enum này (sau đây đư ợc gọi là
“loại thống nhất”).

Các tham chiếu diễn viên phải đư ợc hủy tuần tự hóa bằng cách sử dụng đúng loại chung, do đó tên của các diễn viên đư ợc ghép nối với một trư ờng hợp của loại thống nhất truyền tải loại thông điệp nào diễn viên có tên đó nhận đư ợc. Loại tin nhắn tham chiếu của diễn viên cần phải là thành viên của loại thống nhất này và một hệ thống logic duy nhất phải hoạt động chính xác loại thống nhất. Loại thống nhất triển khai một đặc điểm trư ờng hợp cho mọi loại tin nhắn trong biến thể. Đặc điểm trư ờng hợp chứa các hằng số liên quan, cho phép chúng ta truy xuất các biến thể của kiểu thống nhất với thông tin kiểu của Rust để so sánh thời gian chạy.

Một lập trình viên sử dụng Aurum phải tự xác định loại thống nhất. Vì 2
các loại thống nhất khác nhau không thể tồn tại cho cùng một ứng dụng, các thư viện đư ợc xây dựng trên
Chức năng cốt lõi của Aurum (bao gồm cả chức năng phân cụm) không thể xác định một kiểu thống nhất.
Thay vào đó, họ phải định nghĩa logic của mình theo các kiểu chung thực hiện trư ờng hợp
đặc điểm cho các loại tin nhắn của họ. Ngư ời dùng thư viện của họ sẽ xác định enum để bao gồm
tất cả các loại tin nhắn cho tất cả các thư viện đang đư ợc sử dụng.

2.4 Giao diện diễn viên

Không có gì lạ khi ai đó muốn lư u trữ các tham chiếu diễn viên trong một cấu trúc dữ liệu
(ví dụ, để theo dõi những ngư ời đăng ký sự kiện). Bộ sư u tập trong Rust phải

đồng nhất, như ng có thể có nhiều loại diễn viên khác nhau muốn đăng ký
đến cùng một sự kiện. Bạn không thể giữ các tham chiếu diễn viên nhận đư ợc các loại khác nhau trong
cùng một cấu trúc dữ liệu, cũng không phải là tạo ra một cấu trúc khác nhau cho mọi loại có thể
mong muốn, chúng tôi cũng không muốn tạo ra các tác nhân riêng biệt chuyển tiếp các thông điệp sự kiện đến
ngư ời đăng ký thực sự theo loại hình mình ư a thích.

Giao diện diễn viên cho phép lập trình viên xác định một tập hợp con các loại tin nhắn nhận đư ợc
có thể đư ợc thực hiện bởi một diễn viên và tạo các tham chiếu diễn viên chỉ nhận tập hợp con đó. Đối với
LocalRefs nhận đư ợc một thông điệp của kiểu con, tham chiếu sẽ chuyển đổi nội bộ nó
đến loại thực tế mà bên nhận nhận đư ợc và gửi đi.

Giao diện từ xa phức tạp hơ n. Tham chiếu diễn viên có giao diện như
loại chung hư ớng ra bên ngoài của họ, không phải loại tin nhắn cơ bản của ngư ời nhận
diễn viên. Thông tin này vẫn có trong tên của diễn viên, điều này rất quan trọng đối với
tham chiếu giả mạo. Khi một thông điệp đư ợc gửi bởi ActorRef đư ợc tuần tự hóa, giao diện
biến thể đư ợc gửi cùng với nó để cho ngư ời nhận biết cách giải mã tuần tự hóa
byte. Dựa trên biến thể, quá trình giải tuần tự hóa đư ợc phân phối động đến
diễn giải đúng.

2.5 Tài liệu tham khảo có thể làm giả

Theo truyền thống, tham chiếu diễn viên chỉ có thể có đư ợc theo 2 cách: nhận nó trong một tin nhắn
hoặc chịu trách nhiệm tạo ra diễn viên. Tuy nhiên, việc có thể tạo diễn viên là rất hữu ích
tham khảo từ đầu bằng cách tự cung cấp tất cả thông tin cần thiết. Rèn
tham chiếu loại bỏ nhu cầu về một số giao thức khám phá phức tạp trong phân tán
thuật toán. Điều này không thể thực hiện đư ợc với các tham chiếu diễn viên đư ợc gõ trong Akka, bởi vì
Tên diễn viên Akka không chứa bất kỳ thông tin loại nào. Vì Aurum bao gồm loại

thông tin trong tên diễn viên của nó, nó có khả năng tạo ra các tham chiếu diễn viên với một số,
kiến thức thời gian biên dịch về những thông điệp mà diễn viên tham chiếu đến chấp nhận. Ngữ nghĩa
tính hợp lệ của các tham chiếu giả mạo đư ợc đảm bảo tại thời điểm biên dịch bằng cách sử dụng các hằng số liên quan
trong trư ờng hợp đặc điểm cho các loại thống nhất. Chỉ có thể làm giả các tham chiếu từ xa.

2.6 Diễn viên từ xa

Tất cả các tác nhân có thể truy cập từ xa cần phải đư ợc giải quyết độc lập với các tác nhân khác
trên cùng một nút. Mỗi diễn viên có một tên, là sự kết hợp của một chuỗi và một
biến thể loại thống nhất. Biến thể tư ơ ng ứng với loại tin nhắn mà tác nhân nhận đư ợc.
Khi chúng đư ợc bắt đầu, tên của diễn viên và tham chiếu địa phư ơ ng đư ợc gửi đến một cơ quan đăng ký,
mà bản thân nó là một diễn viên. Khi một thông điệp đư ợc tuần tự hóa đư ợc nhận trên một ổ cắm, nó là
đư ợc chuyển tiếp đến sổ đăng ký. Sổ đăng ký giải quyết tên đích có trong
tin nhắn đư ợc tuần tự hóa đến một tham chiếu cục bộ và chuyển tiếp tin nhắn đư ợc tuần tự hóa nếu
nó tồn tại. Để cải thiện tính song song, các diễn viên có trách nhiệm hủy tuần tự hóa của riêng họ
tin nhắn.

2.7 Diễn viên có luồng kép

Hầu hết các diễn viên chạy trên một luồng duy nhất, nhận tin nhắn, hủy tuần tự hóa chúng nếu
cần thiết và chạy phư ơ ng thức recv của Actor trait. Một số actor có thể muốn sử dụng 2
luồng: nhiệm vụ chính để thực thi logic kinh doanh do lập trình viên xác định và
một nhiệm vụ phụ để xử lý các hoạt động đồng thời khác liên quan đến hoạt động chính.
Nhiệm vụ phụ sẽ xử lý những việc này cho nhiệm vụ chính:

- Hàng đợi ư u tiên Chúng tôi không thể thực hiện tốt điều này nếu không có nhiệm vụ thứ cấp.
  nhiệm vụ chính sẽ không thể xây dựng một hàng đợi mà không dừng lại ở giữa

xử lý tin nhắn để xếp hạng các tin nhắn được xếp hàng. Tốt hơn là có mức độ ưu tiên cao nhất
tin nhắn sẵn sàng để gửi khi chính yêu cầu bằng cách ủy quyền cho phụ
nhiệm vụ. Hàng đợi ưu tiên hiện chưa được hỗ trợ cho các tác nhân luồng đôi.

- Hủy tuần tự hóa Nếu một diễn viên nhận được các tin nhắn lớn, từ xa đủ thường xuyên,
  quá trình khử tuần tự hóa có thể chiếm một phần đáng kể thời gian tính toán của nó.
  thứ cấp có thể sử dụng thời gian tính toán của nó để hủy tuần tự hóa thay thế hoặc có thể tạo ra một
  nhiệm vụ riêng biệt nếu hình thức tuần tự đủ lớn để đảm bảo thực hiện như vậy (mặc dù
  (hiện tại thì không phải như vậy).

- Giám sát Thứ cấp có thể theo dõi xem liệu chính có hoảng loạn hay không
  không có nhịp tim. Khi phát hiện lỗi, thứ cấp sẽ khởi động lại
  chính ở cùng trạng thái khi nó bị sập, không có bản sao nào của
  trạng thái. Giám sát vẫn chưa được triển khai cho các tác nhân luồng kép.

Để xử lý tin nhắn tiếp theo, tác vụ chính sẽ gửi yêu cầu qua nhiều
kênh sản xuất-người tiêu dùng đơn lẻ đến kênh thứ cấp, loại bỏ những yếu tố quan trọng nhất
tin nhắn quan trọng từ hàng đợi của nó và truyền lại bằng kênh khác.

2.8 Phân cụm

Có lẽ tính năng quan trọng nhất của Aurum, mô-đun cụm tồn tại để xác định một
nhóm các nút, tất cả đều có kiến thức về nhau. Kiến thức về các nút nào là
một phần của cụm được truyền đến các nút khác thông qua giao thức gossip, trong đó dữ liệu
được gửi ngẫu nhiên cho đến khi mọi nút trong cụm đều có nó. Nếu một nút chưa nhận được
một mẩu chuyện phiếm trong một thời gian, nó sẽ yêu cầu một.

Phát hiện lỗi

Tính đàn hồi của cụm: các nút có thể được thêm vào hoặc xóa khỏi cụm bất kỳ lúc nào và hệ thống sẽ thích ứng. Nếu một nút bị lỗi, nó sẽ được nhận thấy thông qua một xác suất phát hiện lỗi. Mỗi nút giữ một bộ đệm dấu thời gian trên nhịp tim mà nó đã nhận được từ các nút mà nó chịu trách nhiệm giám sát. Khoảng cách giữa các dấu thời gian này được sắp xếp theo phân phối chuẩn. Xác suất phi (do người dùng cung cấp dưới dạng tùy chọn cấu hình) sẽ được so sánh với hàm phân phối tích lũy (CDF) của thời gian kể từ nhịp tim cuối cùng nhận được để xác định xem nút có nên được coi là xuống. Khi thời gian kể từ nhịp tim cuối cùng đạt đến giá trị khi CDF ở hoặc cao hơn phi, nút sẽ được đánh dấu xuống. Thay đổi trạng thái sẽ sau đó được phân tán khắp cụm với tin đồn. Trong trường hợp có sai sót hạ gục, nút bị hạ gục cuối cùng sẽ nhận được thông báo rằng nó đã bị hạ gục (thông qua các yêu cầu tin đồn). Nó sẽ tự gán cho mình một mã định danh mới và tham gia lại cụm.

Vòng nút

Giám sát trong cụm được phân mảnh. Các nút không theo dõi mọi nút khác, nhưng một một tập hợp con nhỏ của chúng. Giải pháp thay thế sẽ dẫn đến sự gia tăng theo cấp số nhân trong mạng lưới giao thông khi cụm phát triển. Thay vào đó, các nút được băm và đặt trong một vòng băm. Vòng băm nhất quán có hàm nghịch đảo, do đó các nút có thể thấy những nút nào chịu trách nhiệm về chúng và những nút nào chúng phải giám sát. Tin đồn giao thức có xu hướng thiên về việc gửi đến các nút lân cận trong vòng, bởi vì các trạng thái vòng phải đồng ý với nhau nếu nhịp tim hoạt động bình thường. Khi nút được thêm vào, xóa hoặc hạ xuống (do đó thay đổi vòng), chỉ có nút đó những người hàng xóm trong vòng bị ảnh hưởng. Sử dụng vòng, toàn bộ cụm không cần phải thay đổi

hành vi giám sát của nó.

Trong vòng, mỗi nút có một số nút ảo (hoặc “vnode”) đại diện cho
gửi nó đến các điểm khác nhau trong vòng. Vnode truyền đạt mức độ tư ơ ng đối
làm việc một nút nên đư ợc chỉ định theo tỷ lệ với phần còn lại của cụm. Một nút đơ n
có 6 vnode trong vòng trong khi mọi nút khác có 3 thư ờng có nghĩa là nút
sẽ thực hiện gấp đôi khối lư ợng công việc cụm. Bởi vì vnode thể hiện một tỷ lệ,
việc đảm bảo mỗi nút có 3 vnode phân bổ cùng một lư ợng công việc tư ơ ng đối cho mỗi nút
như sử dụng 1 vnode. Sử dụng nhiều vnode hơ n cho mỗi nút sẽ làm giảm khả năng có một nút
không nhận đủ công việc hoặc nhận quá nhiều công việc dựa trên giá trị băm không may mắn.
Sử dụng nhiều vnode hơ n cũng làm tăng kích thư ớc của cấu trúc vòng, do đó đây là một sự đánh đổi.
Tuy nhiên, vì chiếc nhẫn dựa trên trạng thái tin đồn nên trạng thái tin đồn sẽ thay đổi
khiến một nút thay đổi chế độ xem vòng của nó. Nó không bao giờ đư ợc gửi qua mạng, vì vậy
việc tăng cư ờng sử dụng bộ nhớ là vấn đề tiềm ẩn duy nhất đối với một chiếc nhẫn lớn hơ n. Tăng
số lư ợng vnode không làm thay đổi kích thư ớc của trạng thái gossip.

Sử dụng Cluster

Các tác nhân cục bộ có thể đăng ký thay đổi trạng thái của cụm cục bộ. Mỗi
bản cập nhật bao gồm vòng nút đầy đủ, danh sách các nút đang hoạt động và danh sách các sự kiện
đã xảy ra kể từ bản cập nhật cuối cùng (thêm, xóa, hạ cấp và thay đổi đối với địa phư ơ ng
định danh của nút). Bản cập nhật đầu tiên mà mỗi ngư ời đăng ký nhận đư ợc (bất kể khi nào họ đăng ký)
chứa mã định danh của nút cục bộ. Thông tin về một nút bao gồm
vị trí, định danh và số lư ợng vnode. Bởi vì ngư ời đăng ký có quyền truy cập đầy đủ vào
vòng nút, họ có thể sử dụng nó để thực hiện phân mảnh của riêng họ. Bất kỳ giá trị băm nào cũng có thể
hỏi vòng xem nút nào sẽ chịu trách nhiệm về nó.

Để tham gia cụm, một phiên bản cụm cục bộ cần đư ợc khởi động. Phiên bản cục bộ trư ờng hợp cần cấu hình, số lư ợng vnode cho nút này và danh sách hạt giống nút. Danh sách các nút hạt giống sẽ đư ợc liên tục ping cho đến khi có phản hồi đầu tiên, khi mà nút đó hiện là một phần của cụm. Nếu các nút hạt giống không phản hồi trư ớc đạt đến giới hạn (đư ợc xác định trong cấu hình), nút sẽ bắt đầu một cụm riêng biệt chỉ bao gồm chính nó.

Cluster có 2 cấu trúc cấu hình: ClusterConfig và HBRConfig . ClusterConfig chứa thông tin mà nút sẽ cần để tư ơ ng tác đúng cách với phần còn lại của cụm, bao gồm các nút hạt giống và replication_factor , có nghĩa là bây giờ nhiều các nút sẽ quản lý từng nút trong cụm. Hệ số nhân bản phải là giá trị giống nhau trên mọi nút trong cụm. HBRConfig cấu hình các bộ thu nhịp tim cho các khoản phí của một nút, bao gồm các giá trị phi . Mỗi cụm đư ợc xác định và phát hiện bằng một tên đư ợc biết đến trên toàn cầu, đư ợc cung cấp khi bắt đầu một phiên bản cụm. Để tham gia cùng một cụm, mỗi trư ờng hợp phải bắt đầu đã biết tên của cụm họ sắp tham gia. Một nút duy nhất có thể tham gia nhiều cụm theo ý muốn của ngư ời dùng.

2.9 Phân tán CRDT Delta-State

Các kiểu dữ liệu sao chép không xung đột (CRDT), để chia sẻ dữ liệu giữa các thành viên của một cụm. Các bản cập nhật cho các loại này không yêu cầu bất kỳ sự phối hợp nào và cuối cùng nhất quán. Các bản sao của dữ liệu đư ợc lư u trữ trên mỗi thành viên của cụm, cập nhật đư ợc áp dụng thông qua các hàm hợp nhất. Các hàm hợp nhất là một phép toán nhị phân thực hiện phiên bản cục bộ cũ và phiên bản mới nhận đư ợc của dữ liệu và trả về một phiên bản cục bộ mới phiên bản. Các phép toán hợp nhất là giao hoán, kết hợp, lũy đẳng và đơ n điệu, vì vậy CRDT hữu ích trên các giao thức truyền thông không có giao hàng hoặc tin nhắn

đảm bảo đặt hàng.

CRDT trạng thái Delta [20] là các cấu trúc đư ợc tối ư u hóa có các hoạt động đột biến đư ợc tái tạo
biến một phiên bản nhỏ hơ n nhiều của trạng thái mà họ vận hành, chỉ chứa những thay đổi đối với
nó. Các delta này có thể đư ợc hợp nhất với nhau giống như các CRDT đầy đủ
đư ợc tạo ra từ, gửi qua mạng và hợp nhất với trạng thái đầy đủ của ngư ời nhận.
kết quả là giảm đáng kể lư u lư ợng mạng, thời gian tuần tự hóa và hủy tuần tự hóa,
và truyền lại. Mỗi nút trong cụm giữ một bộ đệm delta biểu diễn cách
họ phải quay lại xa hơ n nữa để đảm bảo trạng thái của toàn cụm là nhất quán,
cùng với các xác nhận thông báo cho ngư ời gửi về những ngư ời đã đư ợc cập nhật và tại
họ đã đạt tới điểm nào.

Các CRDT này đư ợc triển khai như các cấu trúc dữ liệu liên tục như mảng băm
thử ánh xạ, vectơ cân bằng radix đư ợc nới lỏng và cây b. Im của Rust (viết tắt của “im-
thư viện “mutable”) [18] may mắn thay đã triển khai các cấu trúc này. Chúng có thể đư ợc sao chép
và đư ợc phân phối tại địa phư ơ ng với chi phí thấp, trong khi chỉ đắt hơ n một chút
hoạt động đột biến.

2.10 Theo dõi nút bên ngoài

Đôi khi, bạn muốn tạo một nút có thể truy cập vào cụm mà không cần nút đó tham gia.
Trong trư ờng hợp cần truy vấn độ trễ thấp, thông tin của nút bên ngoài và
trạng thái phải có sẵn ngay lập tức cho mọi thành viên cụm. Mô-đun thiết bị
cung cấp một cách để giới thiệu một nút bên ngoài vào một cụm. Trong số các ứng dụng khác,
điều này hữu ích cho IoT. Các thiết bị IoT không nên là một phần của cụm, biến của chúng
vị trí khiến chúng không phù hợp cho các tư ơ ng tác có sự phối hợp chặt chẽ.

Các máy chủ tự quyết định xem ai sẽ quản lý từng thiết bị bằng cách sử dụng nút

chuông. Khi tin đồn nhóm tụ lại, các máy chủ có cùng kiến thức
của cụm và mỗi máy chủ sẽ biết ai sẽ xử lý từng thiết bị.
Mặc dù không phải tất cả các máy chủ đều có trách nhiệm theo dõi tất cả các thiết bị, tất cả các máy chủ
biết về sự tồn tại của mọi thiết bị. Các thiết bị và trạng thái của chúng được lưu trữ
trong một CRDT toàn cụm, được lưu hành trong cụm bằng cách sử dụng CRDT của Aurum
chức năng.

Các thiết bị có trách nhiệm khởi tạo liên lạc với máy chủ. Chúng phải
được cung cấp danh sách các máy chủ hạt giống khi khởi động. Nó sẽ gửi nhịp tim ưa thích của mình
khoảng thời gian đến mọi máy chủ trong danh sách hạt giống của nó. Nếu máy chủ nhận được yêu cầu khởi tạo
từ một thiết bị mà nó không chịu trách nhiệm, nó sẽ chuyển tiếp yêu cầu đó đến máy chủ
là. Khi máy chủ nhận được yêu cầu khởi tạo từ thiết bị, nó sẽ khởi động một tác nhân cục bộ
dành riêng cho việc gửi yêu cầu nhịp tim đến thiết bị và quyết định thiết bị
trạng thái dựa trên các giá trị phi hiện tại. Khi một thiết bị nhận được yêu cầu nhịp tim từ
máy chủ của nó, nó gửi một nhịp tim để trả lời. Thiết bị cũng theo dõi nhịp tim
yêu cầu nó nhận được từ máy chủ bằng cách sử dụng một máy dò tích lũy phi . Nếu máy chủ bị lỗi
được phát hiện, thiết bị sẽ liên lạc lại với hạt giống. Nếu nhiều người gửi nhịp tim
yêu cầu được phát hiện, thiết bị sẽ gửi tin nhắn đến mọi máy chủ để ra lệnh cho chúng
tự tử nếu họ không phải là người gửi được chỉ định.

2.11 Bộ dụng cụ thử nghiệm

Các thuật toán phân tán không thể đưa ra những giả định giống như các thuật toán cục bộ. Bất kỳ
việc gửi tin nhắn từ xa có thể bị lỗi, bị trùng lặp hoặc bị trì hoãn bất kỳ lúc nào. Khi
thiết kế các thuật toán phân tán, người lập trình phải tính đến bất kỳ thông điệp nào
có bất kỳ lỗi giao hàng nào. Giao hàng không đáng tin cậy dễ dàng kiểm soát từ một ứng dụng

mức độ mà không phụ thuộc hoặc ảnh hư ởng đến phần còn lại của hệ thống. Bộ dụng cụ thử nghiệm cung cấp macro để lựa chọn giữa việc gửi tin nhắn diễn viên một cách bình thư ờng và việc thả ngẫu nhiên chúng. Các macro lấy các biến để chỉ định xem các thông báo có nên bị loại bỏ hay không hoặc không. Trong các mô-đun như cụm và CRDT, các biến này sẽ đư ợc kiểm soát bởi cờ biên dịch. Ngư ời dùng sẽ có thể cấu hình xem việc gửi có thể không thành công hay không bằng cách sử dụng những lá cờ này.

Xác suất gửi tin nhắn không thành công đư ợc cấu hình tại thời gian chạy với các phiên bản của `FailureConfig` . Các phiên bản `FailureConfig` khác nhau đư ợc liên kết với các ổ cắm trong một `FailureConfigMap` , có `FailureConfig` mặc định nếu không tìm thấy trong map. `FailureConfig` cũng chứa các giới hạn trên và dư ới tùy chọn cho sự chậm trễ trong truyền tải nếu kiểm tra ngẫu nhiên thành công.

Bộ kiểm tra cũng chứa một mô-đun ghi nhật ký. Ngư ời ghi nhật ký là một diễn viên, ngư ời có tham chiếu đư ợc lư u trữ trong nút. Mức ghi nhật ký sẽ đư ợc cấu hình theo từng mô-đun bởi cờ biên dịch. Có 7 mức ghi nhật ký (theo thứ tự ư u tiên): Theo dõi, Gỡ lỗi, Thông tin, Cảnh báo, Lỗi, Chết ngư ời và Tắt, và một macro cho mỗi cấp độ. Mỗi macro chấp nhận một bản ghi mức độ như một đối số. Nếu mức độ nhật ký đư ợc truyền vào cao hơ n mức độ nhật ký đư ợc biểu thị bởi macro, thông báo nhật ký không đư ợc gửi. Ghi nhật ký sẽ chấp nhận bất kỳ đối tư ợng đặc điểm Hiển thị nào. Hiện tại, mục tiêu nhật ký duy nhất là stdout.

CHƯƠNG 3

GIAO DIỆN THƯ VIỆN AURUM

3.1 Đặc điểm của diễn viên

Đặc điểm diễn viên đại diện cho đơn vị lập trình cơ bản nhất trong Aurum. Đó là một
Giao diện đơn giản, với 2 kiểu chung và 3 phương thức:

```
pub trait Actor < U : Case <S > + UnifiedType , S : Giao diện cụ thể <U > > {
async fn pre_start (& mut self , ctx : & ActorContext <U , S >) {}
async fn recv (& mut self , ctx : & ActorContext <U , S > , tin nhắn : S ) ;
async fn post_stop (& mut self , ctx : & ActorContext <U , S >) {}
}
```

`pre_start()` chạy trước khi bất kỳ tin nhắn nào được nhận. `recv()` là phản ứng với một
message. `post_stop()` chạy sau khi actor bị chấm dứt. Actor yêu cầu 2 generic
các loại. `U` là loại thống nhất, danh sách trong đó tất cả các loại tin nhắn được nhận bởi
toàn bộ ứng dụng lưu trú. `S` là loại tin nhắn mà tác nhân nhận được.

3.2 Macro AurumInterface

AurumInterface là một macro phái sinh được áp dụng cho các loại tin nhắn cho các tác nhân. Nó tạo ra
tất cả các triển khai cần thiết cho loại tin nhắn để tương tác với thống nhất
giao diện nhập và tạo. Hiện tại, AurumInterface có một chú thích duy nhất:
`vàng`, có một đối số tùy chọn, cho AurumInterface biết liệu inter-

khuôn mặt có phải là cục bộ hoàn toàn hay không. Các loại tin nhắn sử dụng macro AurumInterface không cần biết về một loại thống nhất cụ thể để thuộc về. Ví dụ:

```
#[ lấy ra ( AurumInterface ) ]
#[ aurum ( cục bộ ) ]
enum MyMsgType {
#[ vàng ]
MyStringInterface (Chuỗi) ,
#[ aurum ( địa phư ơ ng ) ]
MyNonserializableInterface (& ' static str )
MyOtherMsg ( sử dụng )
}
```

Trong ví dụ này, vì một trong các tùy chọn tin nhắn không thể tuần tự hóa đư ợc nên tin nhắn type nói chung không thể tuần tự hóa đư ợc. Tuy nhiên, bạn có thể sử dụng `ActorRef<MyUnifiedType, String>` để gửi một chuỗi từ máy từ xa đến bất kỳ tác nhân nào sử dụng loại tin nhắn này. Bạn cũng có thể tạo một `LocalRef<&'static str>` , như ng không phải là `ActorRef` .

3.3 Macro Thống nhất

Macro `unify!` chịu trách nhiệm xây dựng kiểu hợp nhất và triển khai đặc điểm cho nó. Các đối số để `thống nhất!` phải bao gồm tất cả các loại tin nhắn (cho dù chúng là có thể truy cập từ xa hay không), và các loại đư ợc sử dụng cho giao diện từ xa. `unify!` tạo một kiểu, do đó nó chỉ nên đư ợc gọi một lần trong một ứng dụng duy nhất. Ví dụ:

```
thống nhất ! { MyUnifiedType =
Loại tin nhắn của tôi |
Loại tin nhắn khác của tôi |
MsgTypeForSomeThirdPartyThư viện
    ;
```

```
dây |
Giao diện cho một số thư viện bên thứ ba
}
```

3.4 Diễn viên sinh sản

Nút này chịu trách nhiệm tạo ra các tác nhân và quản lý thông tin hệ thống toàn cầu. Nó có thể truy cập đư ợc từ mọi diễn viên mà nó tạo ra và chứa các tham chiếu đến ổ cắm thông tin, thời gian chạy Tokio, sổ đăng ký và trình ghi nhật ký. Việc tạo một nút chỉ yêu cầu socket và biết đư ợc thời gian chạy Tokio sẽ sử dụng bao nhiêu luồng hệ thống. Sinh sản actor rất đơ n giản. Bạn cần một thể hiện ban đầu của bất kỳ loại nào thực hiện đặc điểm của diễn viên, phần chuỗi trong tên diễn viên, liệu diễn viên có nên luồng kép và liệu có nên gửi tham chiếu đến luồng đó đến sổ đăng ký hay không.

```
cấu trúc MyActor {
đầu tiên: Chuỗi,
giây: sử dụng
}
// Đừng quên chú thích này , trình biên dịch sẽ kêu.
#[ đặc điểm async ]
impl Actor <MyUnifiedType, MyMsgType> cho MyActor {...}

let socket = Socket :: mới (...) ;
let node = Node :: new ( socket , 1) ;
let actor = MyActor {
đầu tiên: "Chào bạn",
giây : 42
};
nút . sinh sản (
```

```
sai        ,  // Luồng đôi ?
diễn viên     ,
    "                          "
     diễn viên của tôi rất tuyệt   ,
đúng       ,  // Đăng ký ?
) ;
```

3.5 Gửi tin nhắn

Sử dụng ActorRef ,có một số cách để gửi tin nhắn. remote_send() sử dụng địa chỉ từ xa của diễn viên, cho dù là địa phư ơ ng hay không. send() sao chép tin nhắn nếu nó đang đư ợc gửi cục bộ. move_to() sẽ nắm quyền sở hữu.

ActorRef có 3 thành phần: máy chủ/cổng đích, tên ngư ời nhận actor và một LocalRef có thể . Bạn có thể xây dựng nút mục tiêu và tên actor độc lập. Điều này giống như việc rèn tham chiếu. Hàm udp_msg() lấy những thành phần riêng biệt này.

```
let socket = Socket :: mới (...) ;
hãy để dest = Destination :: <MyUnifiedType, String >:: new :: <MyMsgType >(
    "                          "
     diễn viên của tôi rất tuyệt
) ;
cho msg: Chuỗi =             "Tên tôi là Baloo ";
udp_msg (& ổ cắm       , & đích  , & tin nhắn );
// Rèn luyện tham chiếu
cho phép rèn = ActorRef :: < MyUnifiedType , String >:: new :: < MyMsgType >(
    "                          "
     diễn viên của tôi rất tuyệt   ,
ổ cắm
) ;
giả mạo . remote_send(& msg ) ;
```

Bạn có thể gửi tin nhắn không đáng tin cậy bằng cách sử dụng macro udp_select !. Cho dù tin nhắn có bị loại bỏ một cách giả tạo hay không phụ thuộc vào FailureMode . FailureMode có 3 tùy chọn: Không có, Gói (chưa được triển khai) và Tin nhắn. FailureMode::None sẽ gửi tin nhắn mà không kiểm tra trước để xóa tin nhắn một cách giả tạo. FailureMode::Message sẽ quyết định nguyên tử cho từng tin nhắn xem tin nhắn có được gửi đi hay không và với nội dung gì trì hoãn.

```
// Không nhất thiết phải là const
const MODE : FailureMode = FailureMode :: Thông báo ;
hãy để fail_map = FailureConfigMap { ... };
cho msg: Chuỗi = "Tên tôi là Baloo ";
udp_select !( CHẾ ĐỘ , & nút , & bản đồ lỗi, & ổ cắm , & đích , & tin nhắn );
```

3.6 Ghi nhật ký

Gửi tin nhắn đến trình ghi nhật ký khá đơn giản. Xác định mức nhật ký cho môi trường của bạn, và gọi macro. Trình ghi nhật ký có thể truy cập được từ nút, nhưng không có gì ngăn bạn tạo ra trình ghi nhật ký của riêng bạn, nó chỉ là một tác nhân. Các thông báo nhật ký có thể là bất cứ thứ gì thực hiện Display. Đối số được chuyển thành một đối tượng đặc điểm trong thân vĩ mô.

```
// Không nhất thiết phải là const
const LEVEL: LogLevel = LogLevel :: Gỡ lỗi;
// Không đăng nhập, mức độ là Gỡ lỗi, cái nào ở trên Trace
dấu vết !( CẤP ĐỘ , & nút , "tảo bẹ ") ;
// Điều này được ghi lại, Cảnh báo ở trên Debug
cảnh báo !( CẤP ĐỘ , & nút , "cá mập ") ;
```

3.7 Bắt đầu một cụm

Mỗi cụm có một tên chuỗi, đư ợc sử dụng để xây dựng các tác nhân cần thiết. Trong ngoài các cấu hình, ClusterConfig và HBRConfig , bạn cần một danh sách ban đầu của ngư ời đăng ký vào các sự kiện cụm (có thể trống), một FailureConfigMap và một số của vnodes. Trình xây dựng của cụm trả về một LocalRef gửi lệnh đến cục bộ trư ờng hợp cụm.

```
hãy để mut clr = ClusterConfig :: default () ;
// Thay đổi đối với clr ...
hãy để mut hbr = HBRConfig :: default () ;
// Thay đổi đối với hbr ...
let cluster = Cluster :: mới (
   & nút   ,
    "
     cụm - tên - tuyệt vời - của tôi ". to_string () ,
   3 ,  // số lư ợng vnode
vec ![ ctx . giao diện cục bộ () ]          ,  // Ban đầu
bản đồ lỗi,
lớp      ,
giờ      ,
) ;
```

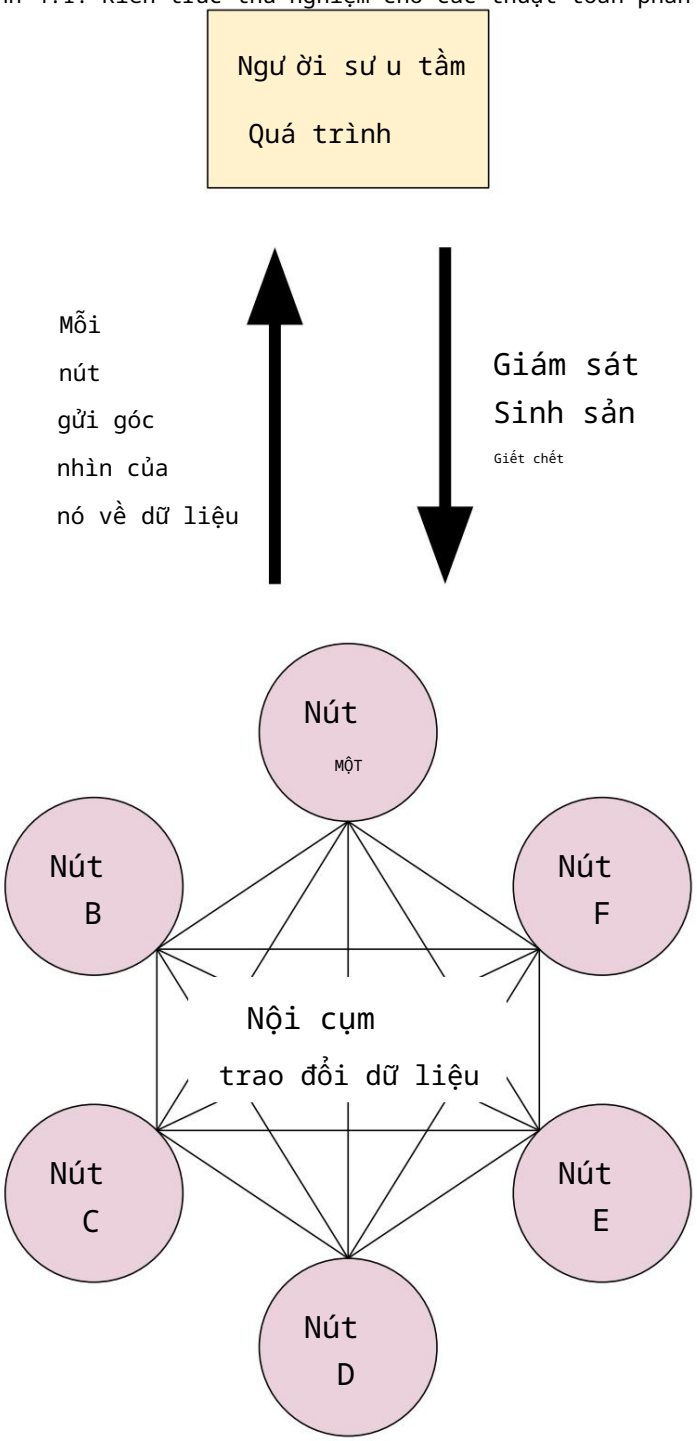
CHƯƠNG 4

KIỂM TRA ĐÚNG ĐẮN

Ngoài những khó khăn liên quan đến việc kiểm tra mã đồng thời cục bộ, chúng ta phải giải thích cho sự không chắc chắn của giao tiếp mạng. Cấu trúc dữ liệu tùy chỉnh giống như thiết bị CRDT, trạng thái tin đồn và vòng nút được kiểm tra đơn vị. Các diễn viên liên quan đến các giao thức phức tạp được kết hợp quá chặt chẽ với nhau để có thể kiểm thử đơn vị hiệu quả. Thay vào đó, các diễn viên được thử nghiệm từ góc nhìn của người dùng. Nhiều tính năng (như nhóm) liên quan đến các diễn viên tương tác với các trường hợp khác của chính họ trên các các nút. Theo dõi cách trạng thái toàn cụm phát triển theo thời gian được thực hiện tốt nhất từ mức độ ứng dụng.

Các thử nghiệm của chúng tôi về phân cụm, theo dõi thiết bị và phân tán CRDT sử dụng testkit để ngẫu nhiên đưa lỗi vào tin nhắn gửi. Các thử nghiệm này có một nút trung tâm tạo ra nhiều tiến trình nhóm lại với nhau. Tiến trình trung tâm đăng ký cập nhật từ mọi nút và thực hiện một loạt các sự kiện (như nút sinh sản, chấm dứt và đột biến dữ liệu). Đối với mỗi sự kiện, quy trình trung tâm xây dựng một giá trị mong đợi cho dữ liệu được chia sẻ và chờ giá trị đó được nhận được từ mọi quy trình trước khi tiến hành đến tập sự kiện tiếp theo. Điều này cho phép kiểm tra được tiến hành mà không sử dụng thời gian chờ chính xác cho mọi tin nhắn. Nó cung cấp tính linh hoạt mà chúng ta cần để sử dụng cùng một cấu hình thử nghiệm cho nhiều thử nghiệm khác nhau giá trị lỗi và độ trễ. Hình 4.1 cung cấp hỗ trợ trực quan.

Hình 4.1: Kiến trúc thử nghiệm cho các thuật toán phân tán



CHƯƠNG 5

SO SÁNH HIỆU SUẤT

5.1 Giới thiệu

Chúng tôi sẽ so sánh hiệu suất của Aurum với Akka. Chúng tôi cần một chuẩn mực để hiển thị sự khác biệt về hiệu suất. Chúng tôi sẽ thảo luận về các chuẩn mực hiện có và lý do tại sao chúng không phù hợp với trường hợp sử dụng của chúng tôi, hãy tự xác định chuẩn mực của riêng mình để so sánh hiệu suất.

5.2 Các chuẩn mực hiện có

Có nhiều chuẩn mực cho một số loại phần mềm phân tán, bao gồm cả phần mềm phân tán hệ thống xử lý luồng (Apache Spark [24], Apache Storm [25] hoặc Apache Flink [26]), công cụ dữ liệu lớn (Apache Hadoop [27], Apache Spark hoặc MapReduce) và diễn viên mô hình (Akka, Proto.Actor hoặc Erlang). Mặc dù hiệu suất của các hệ thống này là được đo lường hiệu quả, tác động của thất bại đến hiệu suất không được khám phá.

MRBench [21] là một chuẩn mực được tạo ra cho MapReduce. Nó triển khai nhiều SQL các tính năng trong MapReduce (như JOIN và WHERE) và chạy nhiều loại phức tạp truy vấn đối với cơ sở dữ liệu có kích thước gigabyte được lưu trữ trong Hệ thống tệp phân tán Hadoop. Vì chuẩn mực của họ hoạt động trên dữ liệu tĩnh nên nó không phù hợp với trường hợp sử dụng IoT, yêu cầu xử lý dữ liệu phải theo thời gian thực.

RIoTBench [22] là một tập hợp các điểm chuẩn vi mô có thể đư ợc kết nối với nhau trong biểu đồ nhiệm vụ để tạo ra các ứng dụng thử nghiệm khá thực tế. Nó sử dụng Apache Storm để thực hiện các nhiệm vụ này, từ chuyển đổi và lọc đến dự đoán. RIoTBench và các bộ tư ơ ng tự khác có mục đích thử nghiệm xử lý luồng phân tán hệ thống thực hiện các hoạt động SIMD trên các luồng dữ liệu có khả năng theo thời gian thực. Trong các luồng này, dữ liệu chỉ truyền theo một chiều: từ thiết bị đến cụm. Các ứng dụng IoT phức tạp hơ n yêu cầu tư ơ ng tác liên tục, toàn song công giữa thiết bị và cụm.

Mặc dù các hệ thống xử lý luồng phân tán đư ợc thiết kế để chịu đư ợc lỗi trong cụm, các chuẩn mực không kiểm tra các hệ thống dư ới loại căng thẳng này. Hiệu ứng của sự thất bại về hiệu suất không đư ợc thiết lập tốt bởi các chuẩn mực này. Cardoso et al [23] đã sử dụng một chuẩn mực vòng đơ n giản để khám phá hiệu suất của Erlang và Scala các diễn viên, như ng không có logic cụm nào có trong thử nghiệm (không có khả năng thêm/xóa nút hoặc phát hiện lỗi. Vì lỗi tư ơ ng đối phổ biến trong cài đặt IoT, hiểu đư ợc tác động của lỗi thiết bị là đặc biệt có giá trị. Lỗi hiệu suất khả năng chịu đựng phụ thuộc vào cả thời gian phục hồi của thiết bị và quy trình của nó như tốc độ mà cụm xử lý thành viên của nó. Một chuẩn mực có chứa tư ơ ng tác thiết bị full-duplex, khả năng chịu lỗi và logic cụm sẽ đư ợc phát triển để giải quyết những yêu cầu này.

5.3 Tiêu chuẩn của chúng tôi

Điểm chuẩn sẽ kiểm tra mô-đun thiết bị so với mô-đun Akka tư ơ ng đư ơ ng. Nó sẽ bao gồm một nhóm các nút bên ngoài định kỳ gửi dữ liệu đến một cụm trong dữ liệu trung tâm. Mỗi lần một nút gửi dữ liệu đến cụm là một “báo cáo”. Báo cáo bao gồm

dữ liệu được tổng hợp giữa báo cáo cuối cùng và thời điểm báo cáo được gửi. Yêu cầu cụm một báo cáo từ mọi nút bên ngoài và khi nhận được nó, sẽ thực hiện một yêu cầu khác ngay sau đó. Điều này cho phép mỗi máy chủ thiết lập tốc độ riêng của mình mà không bị quá tải yêu cầu.

Người ta dự kiến sẽ thường xuyên xảy ra lỗi trong môi trường này. Nói chung, Sự cố có 2 loại: phần cứng và phần mềm. Sự cố phần cứng dẫn đến khởi động lại hoàn toàn cho các thiết bị, lỗi phần mềm khởi động lại daemon mà thiết bị của chúng tôi đang sử dụng nút đang chạy. Các lỗi tương tự có thể xảy ra trong cụm.

MTTR (Thời gian trung bình để sửa chữa) có thể được sử dụng để đo tốc độ thiết bị phục hồi sau sự cố. “Sửa chữa” được định nghĩa là trạng thái của thiết bị đã được phục hồi hiện đã hoạt động cho đến nay và được tất cả các thành viên của nhóm biết đến. MTTR hữu ích để kiểm tra cách hệ thống có thể phục hồi nhanh chóng.

Chúng tôi muốn biết có thể xử lý được bao nhiêu báo cáo mỗi giây (thông lượng) từ đầu đến cuối trong các điều kiện khác nhau. MTTF (Thời gian trung bình đến khi hỏng) là thời gian trung bình tần suất lỗi được đưa vào hệ thống đang chạy để kiểm tra khả năng chịu lỗi của nó. Chúng tôi sẽ kiểm tra thông lượng nào có thể đạt được với các giá trị MTTF khác nhau. Lỗi tiêm có thể là lỗi phần cứng hoặc phần mềm, trong cụm hoặc thiết bị. Các loại nào của các giá trị MTTF bị lỗi sẽ được báo cáo cùng với thông lượng. Chúng tôi sẽ bắt đầu với giá trị MTTF rất cao (tức là lỗi không thường xuyên) và giảm cho đến khi hệ thống không còn có thể hoạt động được nữa. Chúng tôi muốn chứng minh rằng Aurum vẫn khả thi và tiếp tục hoạt động chính xác, mặc dù tốc độ chậm hơn.

Các thông số của chuẩn mực

Để mô phỏng môi trường IoT thực tế, có thể đưa các lỗi và sự chậm trễ vào tin nhắn được gửi. Xác suất gửi không thành công có thể cấu hình được, cùng với mức tối thiểu

và độ trễ tối đa khi gửi tin nhắn. Các tiến trình có thể bị hủy, sau đó khởi động lại
tự động theo các khoảng thời gian được tạo ngẫu nhiên trong một phạm vi được chỉ định. Các thiết bị tất cả
sử dụng cùng một khoảng thời gian nhịp tim (mặc dù giao thức cho phép họ không làm như vậy), đó là
có thể cấu hình được.

Logic kinh doanh

Trong mỗi máy chủ, logic kinh doanh liên hệ với phần mềm trung gian và đăng ký vào danh sách
của các thiết bị mà máy chủ chịu trách nhiệm. Đối với mỗi thiết bị, máy chủ sẽ gửi báo cáo
yêu cầu đến thiết bị. Thiết bị phản hồi bằng một báo cáo, chỉ là một mảng 64 bit
số nguyên có dấu được đánh số từ 1 đến 1000. Nếu không nhận được báo cáo trong một thời gian, yêu cầu
được gửi lại. Khi các báo cáo đến máy chủ, các con số được cộng lại, sau đó được cộng lại
tổng cộng và một yêu cầu báo cáo khác được gửi đi. Mỗi thiết bị trên mỗi máy chủ có
tổng số riêng. Bộ thu thập (một quy trình duy nhất) sẽ thực hiện các yêu cầu định kỳ tới từng máy chủ,
yêu cầu tổng số của từng thiết bị. Người thu thập chắt lọc tổng số thành một số
báo cáo được nhận bởi toàn bộ cụm. Thông tin này sẽ được sử dụng để cho biết cách
nhiều báo cáo mà cụm có thể xử lý như một tổng thể.

5.4 Kết quả

Điểm chuẩn của chúng tôi được chạy trên cụm Onyx của Đại học Boise State. Các nút chạy
quy trình của khách hàng thường chạy nhiều hơn một.

Aurum đấu với Akka

Bộ thử nghiệm đầu tiên của chúng tôi so sánh thư viện của chúng tôi với Akka. Thuật toán chạy trên máy chủ
được thiết kế không bị hỏng khi tải nặng hơn mà chỉ làm chậm thông lượng.

hiệu ứng có thể đư ợc nhìn thấy khi thông lư ợng ổn định ngay cả khi tải tăng cao hơ n nhiều hơ n là khi đạt đư ợc thông lư ợng tối đa. Các thử nghiệm này đã đư ợc thực hiện không có sự tiêm lỗi.

Chuỗi thử nghiệm đầu tiên chúng tôi chạy sử dụng 1 máy chủ và 3 nút máy khách, tăng số lư ợng khách hàng trên mỗi nút với mỗi lần kiểm tra. Thông lư ợng của Akka đạt đến mức ổn định khá nhanh ở mức 14.000 báo cáo mỗi giây, như ng vẫn giữ nguyên khi số lư ợng khách hàng tăng lên. Aurum mất nhiều thời gian hơ n để đạt đến đỉnh điểm, như ng cuối cùng đã đạt đến mức 42.000. Kết quả đã có Bảng 5.1 và Hình 5.1.

Dòng thứ hai hoàn toàn giống nhau, như ng đư ợc mở rộng lên tới 11 máy chủ và 18 máy khách nút. Điều này cung cấp trung bình 21 máy khách cho mỗi máy chủ, cho phép Akka đạt đến trạng thái ổn định. Theo giới hạn mà chúng ta thấy trong thử nghiệm trư ớc, thông lư ợng của Akka đã đạt đến mức ổn định ở mức 155.000 báo cáo mỗi giây. Thông lư ợng của Aurum không ngừng tăng lên khi số lư ợng khách hàng tăng lên, lên tới 315.000 báo cáo mỗi giây. Kết quả đư ợc trình bày trong Bảng 5.2 và Hình 5.2.

Aurum có thể liên tục vư ợt trội hơ n Akka. Tải càng cao, sự khác biệt này trở nên rõ rệt. Đối với một máy chủ duy nhất, thông lư ợng đỉnh của Aurum là gấp ba lần Akka.

Thả tin nhắn

Để mô phỏng một ứng dụng IoT thực tế, chúng tôi đã thử nghiệm Aurum với các thông báo ngẫu nhiên bị bỏ rơ i thay vì đư ợc gửi đi, với một xác suất cụ thể. Tin nhắn chỉ bị bỏ rơ i khi đư ợc gửi giữa máy khách và máy chủ, không phải giữa các máy chủ. Để giữ tải cao, không có sự chậm trễ nào đư ợc đư a vào. Nếu máy chủ không nhận đư ợc báo cáo khi đư ợc yêu cầu, máy chủ sẽ gửi lại yêu cầu sau thời gian chờ 3 mili giây. Kiểm tra này sử dụng 11 máy chủ và 18 nút máy khách với 13 máy khách trên mỗi nút. Chúng tôi dần dần tăng

xác suất tin nhắn không được gửi đi và theo dõi thông lượng cùng với nó. Như
xác suất tăng lên, chúng ta sẽ mong đợi thông lượng sẽ tiến tới bằng không trong một
giảm dần đơn điệu, lõm lên theo kiểu. Kết quả trong Bảng 5.3 và Hình 5.3
cho thấy kỳ vọng của chúng tôi là đúng. Aurum vẫn có thể sử dụng được (mặc dù chậm hơn) trên
tỷ lệ tin nhắn bị mất cao hơn.

Khách hàng thất bại

Một ứng dụng IoT thực tế có thể bao gồm các lỗi máy khách thường xuyên. Để mô phỏng điều đó, mỗi
nút máy khách có một quy trình giám sát sẽ tắt và khởi động lại máy khách khi thử nghiệm chạy,
với một MTTF nhất định. Thời gian giữa lúc bắt đầu một tiến trình và kết thúc nó là ngẫu nhiên
được tạo ra giữa MTTF*0.25 và MTTF*1.75. Một máy khách luôn được khởi động lại 5
vài giây sau khi nó bị giết, bất kể nó còn sống bao lâu. Khi MTTF giảm,
chúng tôi mong đợi thông lượng sẽ tiến tới bằng không, giảm dần và lõm
xuống. Kết quả trong Bảng 5.4 và Hình 5.4 cho thấy đường cong giảm dần đơn điệu
và không lõm hoàn toàn xuống dưới, nhưng đó là xu hướng chung.

Máy chủ bị lỗi

Mặc dù máy chủ thường xuyên bị lỗi không phải là một phần của ứng dụng thực tế, chúng tôi đã chạy
chuẩn mực vì mục đích hoàn thiện. Bài kiểm tra này có thiết lập chính xác giống như
các lỗi của máy khách. Giao thức theo dõi nút bên ngoài không được thiết kế cho mức cao như vậy
tỷ lệ thất bại, vì vậy chúng tôi mong đợi sự không nhất quán đối với các giá trị MTTF cao hơn. Kết quả trong Bảng
5.5 và Hình 5.5 cho thấy thông lượng không đơn điệu, với một đột biến nhỏ
ở MTTF là 21 giây. Tất cả các máy khách đều sử dụng danh sách toàn diện các nút hạt giống, vì vậy
nhiều máy chủ có thể nhận được từ một máy khách đôi khi. Sự gia tăng này là
được khắc phục bằng việc mất thông lượng do các tiến trình máy chủ bị hủy,

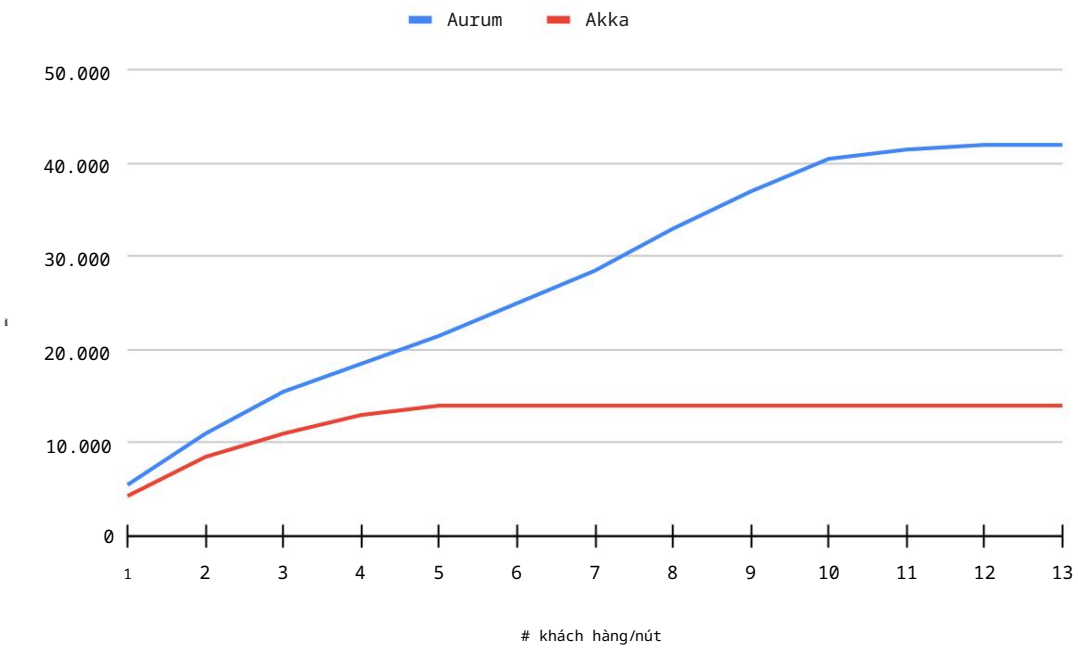
cung cấp cho chúng ta sự không nhất quán ở các giá trị cao hơn. Khi các giá trị thấp hơn, quá trình chết trở thành yếu tố chi phối và thông lượng bắt đầu tiến tới mức bằng không theo cùng một cách như là thử nghiệm thất bại của khách hàng.

Phân tích

Khoảng cách hiệu suất giữa Aurum và Akka vẫn chưa được hiểu đầy đủ. Tuy nhiên, chúng ta có thể suy đoán những người có thể đóng góp vào sự khác biệt. Trình lập lịch của Tokio là đồng hoạt động và được thông báo rõ ràng bởi mã bất cứ khi nào await được gọi, có khả năng tăng tốc chuyển đổi ngữ cảnh. Phản xạ được thực hiện hiệu quả trong Aurum. Các biến thể Enum được biểu diễn bằng một số nhỏ và việc phân phối đến chức năng deserialization thích hợp có thể được thực hiện nhanh chóng. Là một khuôn khổ JVM, Akka phân bổ động nhiều bộ nhớ hơn Aurum và phải giải phóng bộ nhớ đó với việc thu gom rác. Tầm quan trọng của những yếu tố này có thể được nắm bắt trong tương lai thí nghiệm. Akka sử dụng ExecutionContext của Scala, có thể so sánh với Tokio. Sự phản chiếu khó so sánh hơn, nhưng các bài kiểm tra có thể thực hiện được khi nó được kết hợp với tuần tự hóa. Việc thu gom rác khó kiểm soát, do đó tác động chính xác của việc tăng phân bổ bộ nhớ động theo hiệu suất sẽ là một thách thức để thử nghiệm.

# khách hàng/nút	Aurum báo cáo/giây	Akka báo cáo/giây
1	5.500	4.300
2	11.000	8.500
3	15.500	11.000
4	18.500	13.000
5	21.500	14.000
6	25.000	14.000
7	28.500	14.000
	33.000	14.000
	37.000	14.000
8	40.500	14.000
9 10 11	41.500	14.000
12	42.000	14.000
13	42.000	14.000

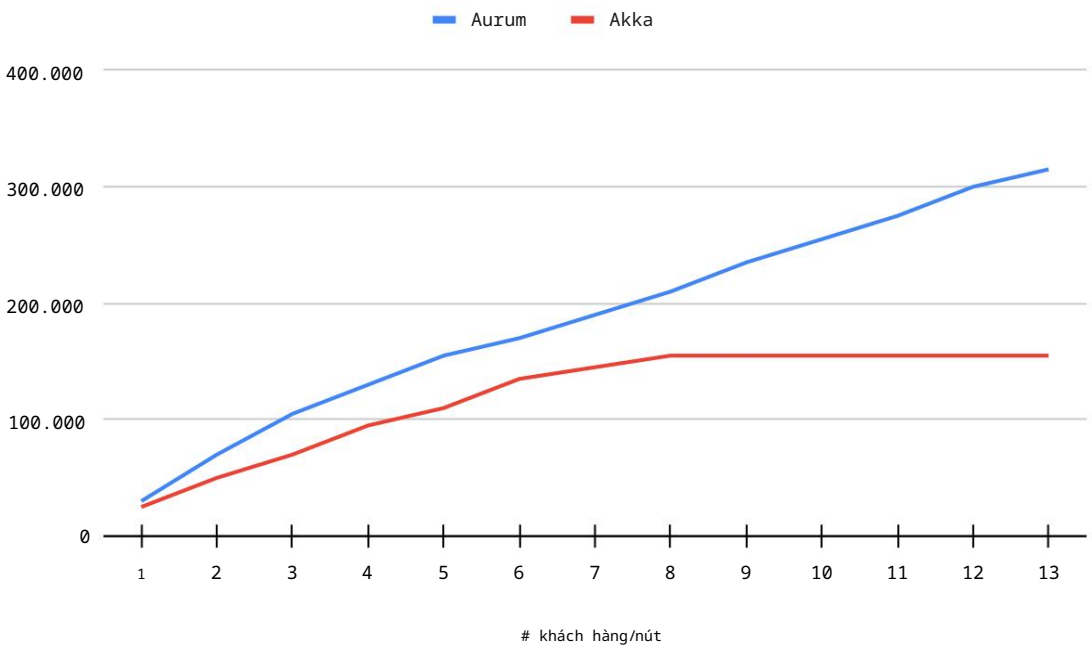
Bảng 5.1: Onyx Cluster - 1 Máy chủ - 3 Nút Máy khách - Không Có Lỗi



Hình 5.1: Onyx Cluster - 1 Máy chủ - 3 Nút Máy khách - Không có Lỗi

# khách hàng/nút	Aurum báo cáo/giây	Akka báo cáo/giây
1	30.000	25.000
2	70.000	50.000
3	105.000	70.000
4	130.000	95.000
5	155.000	110.000
6	170.000	135.000
7	190.000	145.000
	210.000	155.000
	235.000	155.000
8	255.000	155.000
9 10 11	275.000	155.000
12	300.000	155.000
13	315.000	155.000

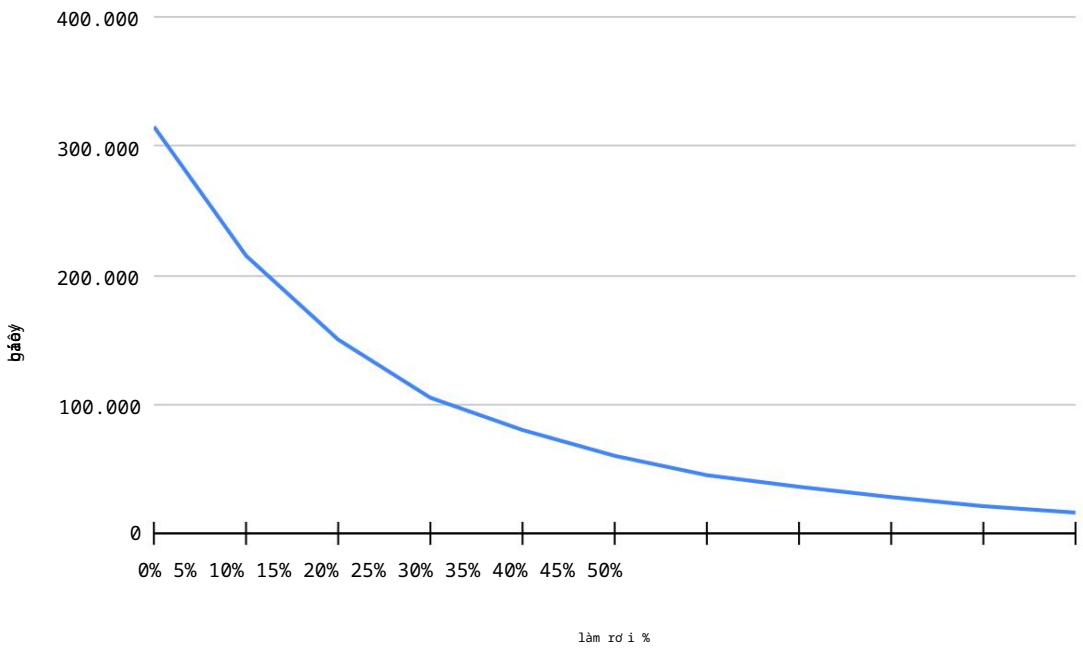
Bảng 5.2: Onyx Cluster - 11 Máy chủ - 18 Nút Máy khách - Không Có Lỗi



Hình 5.2: Onyx Cluster - 11 Máy chủ - 18 Nút Máy khách - Không Có Lỗi

thả % báo cáo/giây
0% 315.000
5% 215.000
10% 150.000
15% 105.000
20% 80.000
25% 60.000
30% 45.000
35% 36.000
40% 28.000
45% 21.000
50% 16.000

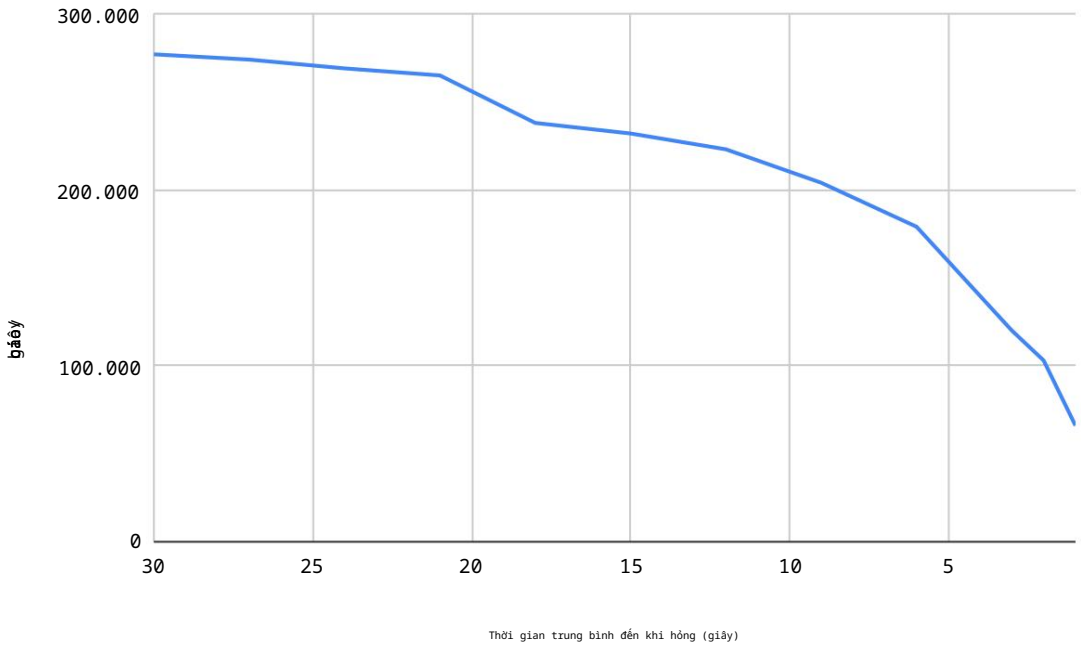
Bảng 5.3: Onyx Cluster - Tin nhắn bị xóa



Hình 5.3: Onyx Cluster - Tin nhắn bị xóa

Báo cáo Thời gian trung bình đến	khi hỏng (giây)/giây
30 277.000	
27 274.000	
24	269.000
21	265.000
18	238.000
15	232.000
12	223.000
	204.000
9 6	179.000
3	120.000
2	103.000
1	66.000

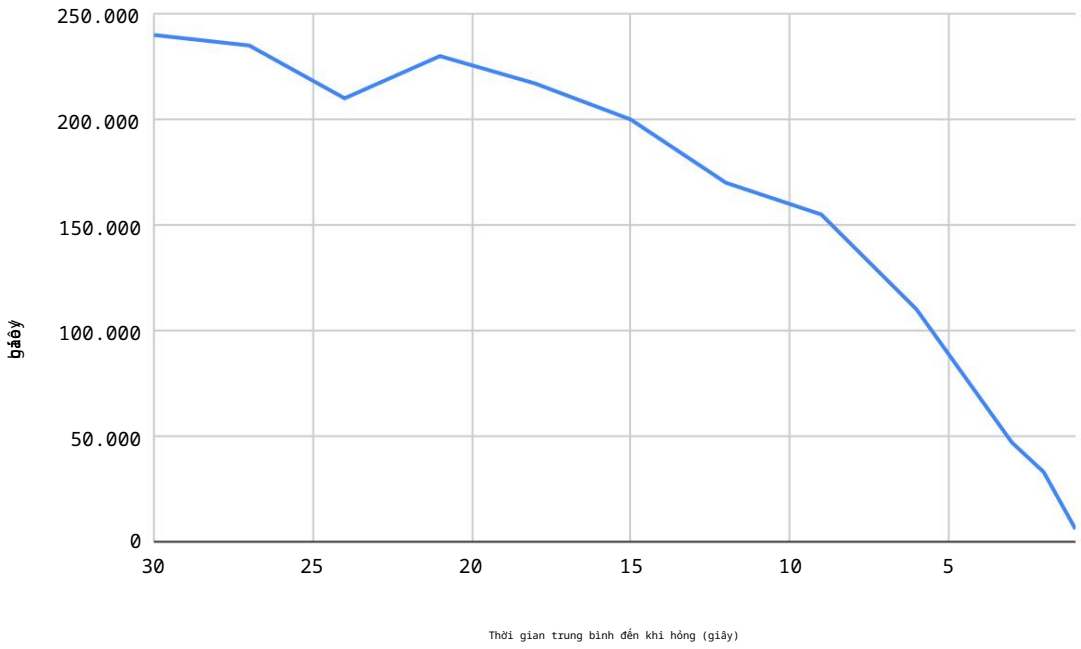
Bảng 5.4: Onyx Cluster - Lỗi máy khách



Hình 5.4: Onyx Cluster - Lỗi máy khách

Báo cáo Thời gian trung bình đến	khi hỏng (giây)/giây
30 240.000	
27 235.000	
24	210.000
21	230.000
18	217.000
15	200.000
12	170.000
	155.000
9 6	110.000
3	47.000
2	33.000
1	6.000

Bảng 5.5: Onyx Cluster - Lỗi máy chủ



Hình 5.5: Onyx Cluster - Lỗi máy chủ

CHƯƠNG 6

PHẦN KẾT LUẬN

6.1 Tóm tắt

Aurum là một thư viện an toàn về kiểu và hiệu suất. Nó vượt trội hơn Akka trong các chuẩn mực của chúng tôi. Lợi ích về hiệu suất khi sử dụng Aurum trở nên rõ ràng hơn khi chúng tôi mở rộng quy mô. Máy chủ Aurum có thể xử lý hàng chục máy khách ở tốc độ tối đa. Giống như Akka, Aurum là một hệ thống chịu lỗi và tiếp tục hoạt động bình thường ngay cả khi các quy trình bị sập và tin nhắn bị loại bỏ. Giao diện của Aurum đơn giản và mạnh mẽ. Nó cung cấp dễ dàng cách để tạo tham chiếu, cấu hình và khởi chạy cụm và chèn tin nhắn thả và sự chậm trễ vào mọi phần của ứng dụng. Các ứng dụng IoT dễ dàng và an toàn hơn khi viết Aurum, nhờ vào các tài liệu tham khảo có thể làm giả, bản dịch loại tin nhắn và kiểm tra thời gian biên dịch.

6.2 Công việc trong tương lai

Mặc dù thư viện Aurum đã có khởi đầu tốt, nhưng vẫn còn nhiều tính năng hơn phải được triển khai để có thể sử dụng đầy đủ trong sản xuất. Dưới đây chúng tôi mô tả một số tính năng này.

Hỗ trợ TCP và TLS

Aurum giao tiếp độc quyền thông qua UDP tại thời điểm phát triển này. Ngư ời dùng có thể muốn cung cấp đáng tin cậy với TCP và bảo mật thông qua TLS [17]. Actor mes-
các nhà hiền triết là những ngư ời chỉ biết bắn và quên, vì vậy ngư ời gửi không nên là ngư ời theo dõi TCP kết nối. Nút sẽ quản lý các tác nhân sở hữu kết nối TCP.

Mã hóa cho tin nhắn UDP

TLS cần một giao thức vận chuyển đáng tin cậy, có thứ tự như TCP bên dư ới nó để hoạt động đúng cách, vì vậy nó không thể đư ợc sử dụng qua UDP. Tuy nhiên, giao tiếp UDP an toàn sẽ sẽ cực kỳ hữu ích khi có. Giao diện phù hợp cho tính năng này vẫn chư a rõ ràng. Hầu hết sự phức tạp nằm ở việc quản lý các khóa. Có thể có lợi khi sử dụng một khóa duy nhất, trên toàn cụm khóa đư ợc chia sẻ bằng mã hóa bất đối xứng hoặc mỗi nút có khóa riêng.
Hệ thống phân tán [29] thảo luận về các cách quản lý khóa. Các khóa cũng sẽ cần có thể dễ dàng truy cập bởi mọi tác nhân trên một nút bất cứ lúc nào.

Nhiều lựa chọn hơ n cho Downing

Mô-đun cụm chỉ có một cơ chế để quyết định xem một nút có bị hỏng không. Nếu một màn hình đơ n phát hiện lỗi, phát hiện đó sẽ đư ợc truyền đến các nút khác, những nút này cũng sẽ xem xét nút xuống. Điều này không giải quyết hiệu quả với phân vùng mạng
Các phân vùng mạng có thể đặc biệt có vấn đề đối với các tính năng đảm bảo chỉ có một trư ờng hợp duy nhất, toàn cụm của một thực thể cụ thể tồn tại. Các tác nhân ảo và mutex phân tán là một ví dụ.

Đầy đủ tính năng của diễn viên Double Threaded

Hàng đợi ư u tiên sẽ yêu cầu một số đặc điểm bổ sung. Các loại tin nhắn của diễn viên sẽ cần một đặc điểm mới với phư ơ ng pháp trả về mức độ ư u tiên của thông điệp đó. Một đặc điểm khác sẽ là cần cung cấp một phư ơ ng pháp chung cho lập trình viên để cấu trúc hàng đợi ư u tiên.
Việc giám sát có thể tự nhiên theo sau các tính năng trong việc truyền tin của Tokio. Khi thứ cấp phát hiện sự cố thông qua ngư ời gửi tin nhắn, nó sẽ khởi động lại chính. Trạng thái chính có thể đư ợc sở hữu bởi một mutex, để cho phép truy cập có thể thay đổi trên dữ liệu đư ợc chia sẻ giữa chính và phụ. Việc khôi phục sau sự cố sẽ chỉ cần liên quan đến việc sao chép một tham chiếu đến mutex và chuyển nó đến mutex mới bắt đầu sơ đẳng.

Mở rộng Mô-đun CRDT

Có nhiều CRDT chư a đư ợc thêm vào mô-đun, bao gồm các bộ chỉ phát triển, bộ đếm, bản đồ và bộ quan sát-loại bỏ, sổ đăng ký và nhiều hơ n nữa. Hỗ trợ trực tiếp CRDT trạng thái delta đư ợc đặt tên cũng cần đư ợc thêm vào giao diện.

Hoạt động tính toán nặng

Aurum đư ợc hỗ trợ bởi thời gian chạy không đồng bộ, Tokio. Các tính năng không đồng bộ của Rust đư ợc thiết kế dành riêng cho các tác vụ liên kết IO. Tạo ra các tác vụ liên kết CPU trên một thời gian chạy đồng bộ sẽ dẫn đến tình trạng chết đói. Bằng cách thêm nhóm thứ hai vào nút đó Lên lịch các tác vụ liên quan đến CPU, Aurum có thể xử lý nhiều ứng dụng hơ n.
Một nhóm như vậy đư ợc tìm thấy trong thư viện Rayon [19], có giao diện thuận tiện cho tính toán song song. Nhóm luồng của Rayon đư ợc quản lý độc lập với Tokio,

như ng kết quả từ các hoạt động tính toán đư ợc thực hiện với Rayon có thể đư ợc chuyển cho một diễn viên không đồng bộ mà không ảnh hư ởng đến lịch trình của Tokio dành cho các diễn viên khác.

Diễn viên ảo

Có thể hữu ích khi chỉ tham chiếu đến một diễn viên bằng tên logic của họ, thay vì tên vật lý của họ. vị trí. Nếu máy chủ của một diễn viên bị sập, nó phải đư ợc khởi động lại trên một nút khác trong cụm. Các tin nhắn đư ợc gửi đến diễn viên đó đư ợc định tuyến lại cho phù hợp. Các diễn viên như vậy là đư ợc gọi là diễn viên ảo và các thư viện diễn viên khác như Akka và Orleans đã có đã triển khai chúng. Các diễn viên ảo có tiềm năng cung cấp cho ứng dụng tốt hơ n khả năng lập trình và phục hồi.

Điều tra sự khác biệt về hiệu suất

Chúng tôi muốn tiến hành nhiều thí nghiệm hơ n trong tư ơ ng lai để hiểu rõ hơ n lý do tại sao Aurum đạt đư ợc hiệu suất tốt hơ n. Bản chất của các chuẩn mực của chúng tôi có thể thay đổi tùy theo Bộ tính năng của Aurum đư ợc mở rộng và hiệu suất cũng đư ợc cải thiện hơ n nữa.

TÀI LIỆU THAM KHẢO

[1] Hewitt, C. (2015, ngày 21 tháng 1). Mô hình tính toán của Actor: Hệ thống thông tin mạnh mẽ có thể mở rộng. Truy cập ngày 03 tháng 12 năm 2020, từ https://arxiv.org/ abs/1008.1459

[2] S´anchez, DD, Sherratt, RS, Arias, P., Almenarez, F., Mar´ın, A. (2015, ngày 24 tháng 6). Mô hình diễn viên cho phép cảm biến đám đông và IoT. Truy cập ngày 2 tháng 12 năm 2020, từ http://centaur.reading.ac.uk/43187/1/Enabling%20Actor%20Model%20for%20Crowd%20Sensing%20and%20IoT%20Full%20text.pdf

[3] Erlang Reference Manual User's Guide. (2020, ngày 22 tháng 9). Truy cập ngày 02 tháng 12 năm 2020, từ https://erlang.org/doc/reference_manual/users_guide.html

[4] Riker-Rs. (2020, ngày 30 tháng 11). Riker-Rs. Truy cập ngày 02 tháng 12 năm 2020, từ https://github.com/riker-rs/riker/

[5] Bastion. (2020, ngày 6 tháng 11). Truy cập ngày 02 tháng 12 năm 2020, từ https://github.com/bastion-rs/bastion

[6] Bonet, D. (2020, ngày 29 tháng 11). Acteur. Truy cập ngày 03 tháng 12 năm 2020, từ https://github.com/DavidBM/acteur-rs

[7] Actix. (2020, ngày 20 tháng 11). Truy cập ngày 03 tháng 12 năm 2020, từ https://github.com/actix/actix

[8] Tài liệu Akka. (2020). Truy cập ngày 03 tháng 12 năm 2020, từ https://doc.akka.io/docs/akka/current/index.html

[9] Akka Serialization Between Local Actors. (2020). Truy cập ngày 03 tháng 12 năm 2020, từ https://doc.akka.io/docs/akka/current/serialization.html#
bộ nối tiếp akka bên ngoài

[10] The Rust Reference. (2020). Truy cập ngày 03 tháng 12 năm 2020, từ https://doc.rust-lang.org/stable/reference/

[11] Dart Isolate. (nd). Truy cập ngày 03 tháng 12 năm 2020, từ https://api.dart. dev/stable/2.10.4/dart-isolate/Isolate-class.html

[12] Barney, B. (2020, ngày 2 tháng 12). Giao diện truyền tin nhắn. Truy cập ngày 03 tháng 12 năm 2020, từ https://computing.llnl.gov/tutorials/mpi/

[13] Kubernetes. (2020). Truy cập ngày 07 tháng 12 năm 2020, từ https://kubernetes.
tôi/

[14] Futures RS. (2020, ngày 2 tháng 12). Truy cập ngày 07 tháng 12 năm 2020, từ https:
//github.com/rust-lang/futures-rs

[15] Tokio. (2020, ngày 7 tháng 12). Truy cập ngày 7 tháng 12 năm 2020, từ https://
github.com/tokio-rs/tokio

[16] Serde. (2020, ngày 5 tháng 12). Truy cập ngày 07 tháng 12 năm 2020, từ https://
github.com/serde-rs/serde

[17] Rescorla, E. (2018, tháng 7). Giao thức bảo mật lớp truyền tải (TLS) phiên bản 1.3. Truy cập ngày 07 tháng 5 năm 2021, từ https://tools.ietf.org/html/rfc8446

[18] Stokke, B. (nd). Bodil/im-rs. Truy cập ngày 07 tháng 5 năm 2021, từ https://github.
com/bodil/im-rs

[19] Stone, J. (nd). Rayon-rs/rayon. Truy cập ngày 07 tháng 5 năm 2021, từ https:// github.com/
rayon-rs/rayon

[20] Almeida, PS, Shoker, A., & Baquero, C. (2018). Các kiểu dữ liệu sao chép trạng thái Delta. Tạp chí tính toán song song và phân tán, 111, 162-173. doi:10.1016/
j.jpdc.2017.08.003

[21] Kim, K., Jeon, K., Han, H., Kim, S., Jung, H., & Yeom, HY (2008). MR-Bench: Một chuẩn mực cho khuôn khổ mapreduce. Hội nghị quốc tế lần thứ 14 của IEEE về Hệ thống song song và phân tán năm 2008. doi:10.1109/icpads.2008.70

[22] Shukla, A., Chaturvedi, S., & Simmhan, Y. (2017). RIoTBench: Một chuẩn mực IoT cho các hệ thống xử lý luồng phân tán. Đồng thời và tính toán: Thực hành và kinh nghiệm, 29(21). doi:10.1002/cpe.4257

[23] Cardoso, RC, H¨ubner, JF, & Bordini, RH (2013). So sánh chuẩn giao tiếp trong Ngôn ngữ dựa trên tác nhân và tác nhân. Kỹ thuật Hệ thống đa tác nhân, 58-77. doi:10.1007/978-3-642-45343-4 4 _

[24] Apache Spark™ - công cụ phân tích thống nhất cho dữ liệu lớn. (nd). Truy cập ngày 07 tháng 5, 2021, từ https://spark.apache.org/

[25] Apache Storm. (nd). Truy cập ngày 07 tháng 5 năm 2021, từ https://storm.apache.
tổ chức/

[26] Apache Flink: Tính toán trạng thái trên luồng dữ liệu. (nd). Truy cập tháng 5 07, 2021, từ https://flink.apache.org/

[27] Apache Hadoop. (nd). Truy cập ngày 07 tháng 5 năm 2021, từ https://hadoop.apache.org/

[28] Cardoso, RC, H¨ubner, JF, & Bordini, RH (2013). So sánh chuẩn giao tiếp trong ngôn ngữ dựa trên tác nhân và tác nhân. Kỹ thuật hệ thống đa tác nhân, 58-77. doi:10.1007/978-3-642-45343-4 4 –

[29] Steen, M. van, & Tanenbaum, AS (2017). Hệ thống phân tán. Pearson Giáo dục.

PHỤ LỤC A

PHỤ LỤC A: MÃ NGUỒN

Mã nguồn đầy đủ có thể đư ợc tìm thấy tại đây: https://github.com/arjunlalshukla/nghiên cứu/cây/thạc sĩ/luận văn. Phiên bản chuẩn mực của Akka đư ợc tìm thấy trong thư mục akka-benchmark . Mã nguồn của Aurum nằm trong thư mục aurum . các dự án đư ợc cấu trúc tư ơ ng ứng như các dự án sbt và hàng hóa tiêu chuẩn . Hư ớng dẫn Hư ớng dẫn về cách sử dụng mã có trong kho lư u trữ.

PHỤ LỤC B

PHỤ LỤC B: CÁC THÔNG SỐ KỸ THUẬT CỦA MÔI TRƯỜNG THỬ NGHIỆM

Phiên bản ngôn ngữ:

- Phiên bản Java: 11.0.11
- Phiên bản Scala: 2.13.1
- Phiên bản Rust: 1.51

Onyx /etc/os-release :

```
TÊN =" Red Hat Enterprise Linux Server "
PHIÊN BẢN ="7.9 ( Maipo ) "
ID = "rhel "
ID_LIKE =" fedora "
BIẾN THỂ =" Máy chủ "
VARIANT_ID =" máy chủ "
PHIÊN BẢN_ID = "7.9"
PRETTY_NAME =" Red Hat Enterprise Linux "
ANSI_COLOR = "0;31"
CPE_NAME =" cpe :/ o : redhat : enterprise_linux :7.9: GA : máy chủ "
TRANG CHỦ_URL =" https://www.redhat.com/"
BUG_REPORT_URL =" https://bugzilla.redhat.com/"

REDHAT_BUGZILLA_PRODUCT =" Red Hat Enterprise Linux 7"
```

```
15 REDHAT_BUGZILLA_PRODUCT_VERSION =7.9
16 REDHAT_SUPPORT_PRODUCT =" Red Hat Enterprise Linux                                                    "
17 PHIÊN BẢN REDHAT_SUPPORT_PRODUCT_VERSION = "7.9"
```

Nút đầu Onyx lscpu :

```
1 Kiến trúc:                          x86_64
2 Chế độ hoạt động của CPU (s):       32-bit    ,  64-bit
Thứ tự 3 Byte:                        Little Endian
4 CPU (s):                            48
5 Danh sách CPU trực tuyến:           0-47
6 Luồng (s) trên mỗi lõi:             2
7 Lõi (giây) cho mỗi ổ cắm:           12
8 Ổ cắm (s):                          2
9 nút NUMA (các nút):                 2
10 ID nhà cung cấp:                   Intel chính hãng
11 Họ CPU:                            6
12 Mô hình:                           85
13 Tên mẫu:                           Bộ vi xử lý Intel (R) Xeon (R) Silver 4116 @ 2,10 GHz
14 Bư ớc:                             4
15 MHz CPU:                           919.006
CPU tối đa 16 MHz:                    3000.0000
17 CPU tối thiểu MHz:                 800.0000
18 BogoMIPS:                          4200,00
19 Ảo hóa:                            VT-x
20 Bộ nhớ đệm L1d:                    32 nghin
Bộ nhớ đệm 21 L1i:                    32 nghin
22 Bộ nhớ đệm L2:                     1024K
23 Bộ nhớ đệm L3:                     16896K
24 NUMA node0 CPU (s):
     0 ,2 ,4 ,6 ,8 ,10 ,12 ,14 ,16 ,18 ,20 ,22 ,24 ,26 ,28 ,30 ,32 ,34 ,36 ,38 ,40 ,42 ,44 ,46
```

```
NUMA node1 CPU (s):
     1 ,3 ,5 ,7 ,9 ,11 ,13 ,15 ,17 ,19 ,21 ,23 ,25 ,27 ,29 ,31 ,33 ,35 ,37 ,39 ,41 ,43 ,45 ,47

lá cờ:                           fpu vme de pse tsc msr pae mce cx8 apic sep
     mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss
      ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art
     arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc
     aperfmperf eagerfpu pni pclmulqdq màn hình dtes64 ds_cpl vmx smx
     est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2
     x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand
     lahf_lm abm 3 dnowprefetch epb cat_l3 cdp_l3 invpcid_single
     intel_pt ssbd mba ibrs ibpb stibp tpr_shadow vnmi flexpriority
     ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid
      rtm cqm mpx rdt_a avx512f avx512dq rdseed adx smap clflushopt
     clwb avx512cd avx512bw avx512vl xsaveopt xsavec xgetbv1 cqm_llc
     cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm ida arat pln pts
      pku ospke md_clear spec_ctrl intel_stibp flush_l1d
```

Nút Onyx `lscpu` :

```
Kiến trúc:                       x86_64
Chế độ hoạt động của CPU (s):    32-bit   ,  64-bit
Thứ tự Byte:                     Little Endian
CPU (s):                         6
Danh sách CPU trực tuyến: 0 -5
Luồng (s) trên mỗi lõi:          1
Lõi (s) cho mỗi ổ cắm: 6
Ổ cắm (s):                       1
nút NUMA (các nút):              1
ID nhà cung cấp:                 Intel chính hãng
```

```
Họ CPU:                          6
Mô hình:                         158
Tên mẫu:                         Bộ vi xử lý Intel (R) Core (TM) i5 -8500 T @ 2,10 GHz
Bước:                            10
MHz CPU:                         3402.976
CPU tối đa MHz:                  3500.0000
CPU tối thiểu MHz:               800.0000
BogoMIPS:                        4224.00
Ảo hóa:                          VT-x
Bộ nhớ đệm L1d:                  32 nghìn
Bộ nhớ đệm L1i:                  32 nghìn
Bộ nhớ đệm L2:                   256K
Bộ nhớ đệm L3:                   9216K
NUMA node0 CPU (s):              0-5
lá cờ:                           fpu vme de pse tsc msr pae mce cx8 apic sep
                                 mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss
                                  ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art
                                 arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid
                                 aperfmperf tsc_known_freq pni pclmulqdq màn hình dtes64 ds_cpl vmx
                                  smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2
                                 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand
                                 lahf_lm abm 3 dnowprefetch cpuid_fault epb invpcid_single pti ssbd
                                  ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid ept_ad
                                 fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm mpx
                                 rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1
                                 xsaves dtherm ida arat pln pts hwp hwp_notify hwp_act_window
                                 hwp_epp md_clear xóa_l1d
```