

ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA ĐIỆN TỬ - VIỄN THÔNG



ĐỒ ÁN TỐT NGHIỆP
NGÀNH: KỸ THUẬT MÁY TÍNH

ĐỀ TÀI:

**PHẦN MỀM GIÁM SÁT VÀ
TÙY CHỈNH TẢI HỆ THỐNG IVI**

Sinh viên thực hiện:

Thực hiện 1: **HỒ ĐỨC VŨ - 106200284 - 20KTMT2**

Thực hiện 2: **NGUYỄN MINH PHƯƠNG - 106200241 - 20KTMT1**

Người hướng dẫn:

Hướng dẫn 1: **TS. NGÔ MINH TRÍ**

Hướng dẫn 2: **KS. PHAN HỒNG SANG**

Hướng dẫn 3: **KS. NGUYỄN VIỆT ĐỨC**

Đà Nẵng, 24 / 06 / 2025

NHẬN XÉT ĐỒ ÁN TỐT NGHIỆP

1. Thông tin chung:

1. Họ và tên sinh viên: Hồ Đức Vũ - Nguyễn Minh Phương
2. Lớp: 20KTMT2 - 20KTMT1 Số thẻ SV: 106200284 - 106200241
3. Tên đề tài: Phần mềm giám sát và tùy chỉnh tải hệ thống IVI
4. Người hướng dẫn: Ngô Minh Trí Học hàm/ học vị: Tiến sĩ

II. Nhận xét, đánh giá đồ án tốt nghiệp:

1. Về tính cấp thiết, tính mới, khả năng ứng dụng của đề tài: (điểm tối đa là 2đ)

.....
.....

2. Về kết quả giải quyết các nội dung nhiệm vụ yêu cầu của đồ án: (điểm tối đa là 4đ)

.....
.....

3. Về hình thức, cấu trúc, bô cục của đồ án tốt nghiệp: (điểm tối đa là 2đ)

.....
.....

4. Đề tài có giá trị khoa học/ có bài báo/ giải quyết vấn đề đặt ra của doanh nghiệp hoặc nhà trường: (điểm tối đa là 1đ)

.....
.....

5. Các tồn tại, thiếu sót cần bổ sung, chỉnh sửa:

.....
.....

III. Tinh thần, thái độ làm việc của sinh viên: (điểm tối đa 1đ)

.....

IV. Đánh giá:

1. Điểm đánh giá:/10 (lấy đến 1 số lẻ thập phân)

2. Đề nghị: Được bảo vệ đồ án Bổ sung để bảo vệ Không được bảo vệ

Đà Nẵng, ngày tháng năm 2025

Người hướng dẫn

NHẬN XÉT PHẢN BIỆN ĐỒ ÁN TỐT NGHIỆP

I. Thông tin chung:

1. Họ và tên sinh viên: Hồ Đức Vũ - Nguyễn Minh Phương
2. Lớp: 20KTMT2 - 20KTMT1 Số thẻ SV: 106200284 - 106200241
3. Tên đề tài: Phần mềm giám sát và tùy chỉnh tải hệ thống IVI
4. Người phản biện: Nguyễn Văn Hiếu Học hàm/ học vị: Tiến sĩ

II. Nhận xét, đánh giá đồ án tốt nghiệp:

TT	Các tiêu chí đánh giá	Điểm tối đa	Điểm đánh giá
1	Sinh viên có phương pháp nghiên cứu phù hợp, giải quyết đủ nhiệm vụ đồ án được giao	80	
1a	- Tính mới (nội dung chính của ĐATN có những phần mới so với các ĐATN trước đây). - Đề tài có giá trị khoa học, công nghệ; có thể ứng dụng thực tiễn.	15	
1b	- Kỹ năng giải quyết vấn đề; hiểu, vận dụng được kiến thức cơ bản, cơ sở, chuyên ngành trong vấn đề nghiên cứu. - Chất lượng nội dung ĐATN (thuyết minh, bản vẽ, chương trình, mô hình,...).	50	
1c	- Có kỹ năng vận dụng thành thạo các phần mềm ứng dụng trong vấn đề nghiên cứu; - Có kỹ năng đọc, hiểu tài liệu bằng tiếng nước ngoài ứng dụng trong vấn đề nghiên cứu; - Có kỹ năng làm việc nhóm;	15	
2	Kỹ năng viết:	20	
2a	- Bố cục hợp lý, lập luận rõ ràng, chặt chẽ, lời văn súc tích	15	
2b	- Thuyết minh đồ án không có lỗi chính tả, in ấn, định dạng	5	
3	Tổng điểm đánh giá theo thang 100: Quy về thang 10 (lấy đến 1 số lẻ)		

- Các tồn tại, thiếu sót cần bổ sung, chỉnh sửa:
.....
- Câu hỏi đề nghị sinh viên trả lời trong buổi bảo vệ:
.....

- Đề nghị: Được bảo vệ đồ án Bổ sung để bảo vệ Không được bảo vệ

Đà Nẵng, ngày tháng năm 2025

Người phản biện

TÓM TẮT

Tên đề tài: Phần mềm giám sát và tùy chỉnh tải hệ thống IVI

Sinh viên thực hiện: Hồ Đức Vũ

Số thẻ SV: 106200284 Lớp: 20KTMT2

Sinh viên thực hiện: Nguyễn Minh Phương

Số thẻ SV: 106200241 Lớp 20KTMT1

Hệ thống IVI (In-Vehicle Infotainment) là nền tảng công nghệ tích hợp trong ô tô, đảm nhận vai trò giải trí, kết nối và hỗ trợ lái xe an toàn. Tuy nhiên, IVI đối mặt với thách thức lớn về quản lý tài nguyên do đặc thù phần cứng hạn chế, nhiều chức năng phải vận hành đồng thời, dễ dẫn tới quá tải, giảm hiệu năng, thậm chí gây nguy hiểm cho người dùng khi hệ thống mất kiểm soát. Các giải pháp khả thi đề ra bao gồm: tối ưu ứng dụng, giám sát hiệu năng hệ thống và điều phối mức sử dụng tài nguyên cho phù hợp. Việc nâng cấp phần cứng thường không khả thi do chi phí cao và phức tạp kỹ thuật.

Do đó, đề tài này hướng đến xây dựng một phần mềm giám sát và tùy chỉnh tải hệ thống IVI, hỗ trợ cả hai giai đoạn: phát triển (giúp nhà phát triển đánh giá, tối ưu mức tiêu thụ tài nguyên của từng tiến trình, thực hiện kiểm thử tải) và vận hành (giám sát tải tài nguyên theo thời gian thực, cảnh báo quá tải, tự động xử lý khi hệ thống gặp sự cố, lưu trữ dữ liệu lịch sử để phục vụ phân tích, tối ưu và bảo trì).

Giải pháp này giúp các bên phát triển, vận hành chủ động theo dõi, tối ưu hệ thống IVI theo điều kiện tài nguyên thực tế, nâng cao tính an toàn, ổn định, góp phần giảm thiểu những nguy cơ xấu có thể xảy ra, hỗ trợ công tác bảo trì và vận hành hệ thống IVI tốt hơn trong thực tế.

Cấu trúc đồ án bao gồm 6 chương:

Chương 1: Tổng quan đề tài

Chương 2: Thiết kế tổng quan phần mềm

Chương 3: Thiết kế và triển khai phần hệ Windows App

Chương 4: Thiết kế và triển khai phần hệ Linux App

Chương 5: Kiểm thử và đánh giá kết quả

Chương 6: Kết luận và hướng phát triển

NHIỆM VỤ ĐỒ ÁN TỐT NGHIỆP

TT	Họ tên sinh viên	Số thẻ SV	Lớp	Ngành
1	Hồ Đức Vũ	106200284	20KTMT2	Kỹ Thuật Máy Tính
2	Nguyễn Minh Phương	106200241	20KTMT1	Kỹ Thuật Máy Tính

- Tên đề tài đồ án: Phần mềm giám sát và tùy chỉnh tải hệ thống IVI.*
- Đề tài thuộc diện: Có ký kết thỏa thuận sở hữu trí tuệ đối với kết quả thực hiện*
- Nội dung phân công đồ án:*

a. *Phân chung:*

TT	Họ tên sinh viên	Nội dung
1	Hồ Đức Vũ	<ul style="list-style-type: none">- Phân tích yêu cầu đề tài- Vẽ sơ đồ thiết kế phần mềm (sơ đồ Use case, sơ đồ lớp, sơ đồ Activity, sơ đồ Sequence)- Xây dựng thuật toán phát hiện và xử lý quá tải hệ thống IVI- Kiểm tra và đánh giá phần mềm- Viết báo cáo M1, M2 và M3
2	Nguyễn Minh Phương	

b. *Phân riêng:*

TT	Họ tên sinh viên	Nội dung
1	Hồ Đức Vũ	<ul style="list-style-type: none">- Thu thập thông số tải hệ thống IVI- Xây dựng module truyền dữ liệu từ Linux App đến Windows App- Xây dựng các hành động xử lý quá tải tại Linux App- Xây dựng chức năng Stress test tại Linux App- Xử lý luồng tại Linux App
2	Nguyễn Minh Phương	<ul style="list-style-type: none">- Xây dựng giao diện (UI) trực quan trên Windows App- Xây dựng khôi phục năng gởi/nhận dữ liệu- Xây dựng các khôi phục năng xử lý liên quan đến hiển thị dữ liệu- Xây dựng chức năng lưu dữ liệu

<i>4. Họ tên người hướng dẫn:</i>	<i>Phản/ Nội dung:</i>
TS. Ngô Minh Trí	Toàn bộ đồ án
KS. Phan Hồng Sang	
KS. Nguyễn Việt Đức	Tìm hiểu đề tài của đồ án

5. Ngày giao nhiệm vụ đồ án:/...../202

6. Ngày hoàn thành đồ án:/...../202

Đà Nẵng, ngày tháng năm 2025

Trưởng Bộ môn.....

Người hướng dẫn

LỜI NÓI ĐẦU

Trường Đại học Bách khoa – Đại học Đà Nẵng là một trong những cơ sở đào tạo uy tín hàng đầu trong lĩnh vực kỹ thuật, là sinh viên thuộc Khoa Điện tử – Viễn thông của trường, chúng tôi luôn ý thức được vai trò và trách nhiệm trong quá trình học tập, rèn luyện và phát triển. Chúng tôi đã xác định cho mình mục tiêu rõ ràng, không ngừng trau dồi kiến thức chuyên môn và kỹ năng thực tiễn ngay từ khi còn ngồi trên ghế giảng đường. Đồ án này là minh chứng rõ nét cho những kiến thức, kỹ năng mà chúng tôi đã tích lũy được trong những năm vừa qua. Chúng tôi xin chân thành cảm ơn nhà trường và quý thầy cô đã tạo nên một môi trường học tập chuyên nghiệp, chất lượng và truyền cảm hứng. Những bài học đắt giá và nền tảng kiến thức vững chắc chính là hành trang quan trọng giúp chúng tôi vững bước trên con đường sự nghiệp sau này.

Trong quá trình học tập, đặc biệt là giai đoạn thực hiện đồ án tốt nghiệp, chúng tôi đã có cơ hội được tiếp cận thực tế, áp dụng những kiến thức đã học để giải quyết bài toán thực tế tại doanh nghiệp. Đồ án “**Phần mềm giám sát và tuỳ chỉnh tải hệ thống IVI**” là kết quả của quá trình đó, vừa là minh chứng cho những nỗ lực học tập, vừa là bước đầu làm quen với môi trường làm việc chuyên nghiệp.

Chúng tôi xin bày tỏ lòng biết ơn sâu sắc đến TS. Ngô Minh Trí, giảng viên Khoa Điện tử – Viễn thông, người đã tận tình hướng dẫn, định hướng và đóng góp nhiều ý kiến chuyên môn quý báu giúp chúng tôi hoàn thành tốt đê tài này.

Chúng tôi cũng xin chân thành cảm ơn Công ty FPT Software Đà Nẵng đã tạo điều kiện cho chúng tôi thực tập và triển khai đề tài trong môi trường thực tế. Đặc biệt, chúng tôi rất biết ơn Kỹ sư Phan Hồng Sang và Kỹ sư Nguyễn Việt Đức, những người đã đồng hành, hỗ trợ kỹ thuật và góp ý xuyên suốt quá trình thực hiện đồ án.

Chúng tôi cũng không quên gửi lời cảm ơn đến gia đình, người thân và bạn bè - những người đã luôn cỗ vũ, động viên tinh thần để chúng tôi vượt qua những khó khăn trong quá trình học tập và hoàn thiện đồ án.

Dù đã cố gắng hết mình, song do kiến thức và kinh nghiệm còn hạn chế, chắc chắn rằng đồ án này sẽ không tránh khỏi những thiếu sót. Chúng tôi rất mong nhận được sự đóng góp, phản biện và chỉ dẫn từ quý thầy cô để giúp chúng tôi học hỏi thêm nhiều kinh nghiệm, bổ sung những thiếu sót từ đó hoàn thiện đồ án này, cũng như có thể vận dụng vào công việc sau này.

Một lần nữa, chúng tôi xin chân thành cảm ơn.

Đà Nẵng, ngày ... tháng ... năm 2025

Nhóm sinh viên thực hiện
(Ký và ghi rõ họ tên)

Hồ Đức Vũ Nguyễn Minh Phương

LỜI CAM ĐOAN

Chúng tôi xin cam đoan rằng đồ án “**Phần mềm giám sát và tuỳ chỉnh tải hệ thống IVI**” là kết quả nghiên cứu và làm việc nghiêm túc của chúng tôi dưới sự hướng dẫn của TS. Ngô Minh Trí, giảng viên Khoa Điện tử – Viễn thông, Trường Đại học Bách khoa – Đại học Đà Nẵng và Kỹ sư Phan Hồng Sang, Kỹ sư Nguyễn Việt Đức, người hướng dẫn tại công ty FPT Software Đà Nẵng.

Các dữ liệu, kết quả nêu trong báo cáo là trung thực, không sao chép từ bất kỳ nguồn nào mà không trích dẫn. Trong quá trình thực hiện, chúng tôi đã tuân thủ nghiêm túc các quy định về đạo đức học thuật và sở hữu trí tuệ. Các tài liệu tham khảo được liệt kê rõ ràng.

Nếu phát hiện có bất kỳ hành vi gian lận, sao chép, đạo văn hoặc vi phạm bản quyền nào trong quá trình thực hiện đồ án, chúng tôi xin hoàn toàn chịu trách nhiệm trước nhà trường và các cơ quan có thẩm quyền.

Chúng tôi xin cam đoan và chịu trách nhiệm với lời cam đoan này.

Đà Nẵng, ngày ... tháng ... năm 2025

Nhóm sinh viên thực hiện

(Ký và ghi rõ họ tên)

Hồ Đức Vũ Nguyễn Minh Phương

MỤC LỤC

Tóm tắt

Nhiệm vụ đồ án

Lời nói đầu và cảm ơn	i
-----------------------	---

Lời cam đoan liêm chính học thuật	iii
-----------------------------------	-----

Mục lục	iv
---------	----

Danh sách các bảng biểu	vii
-------------------------	-----

Danh sách các hình vẽ	viii
-----------------------	------

Danh sách các cụm từ viết tắt	xi
-------------------------------	----

Mở đầu	1
--------	---

Chương 1: Tổng quan đề tài	3
-----------------------------------	---

1.1 Nêu vấn đề	3
1.2 Giải pháp đặt ra	5
1.3 Phần mềm giám sát và tùy chỉnh tải hệ thống IVI là gì?	6
1.4 Kết luận chương	6

Chương 2: Thiết kế tổng quan phần mềm	8
--	---

2.1 Mô hình tổng quan Phần mềm	8
2.2 Một số yêu cầu chung của Phần mềm	9
2.3 Yêu cầu chức năng của các phân hệ	10
2.3.1 Phân hệ Linux App	10

2.3.2	Phân hệ Windows App	11
2.4	Nguyên lý hoạt động của Phần mềm	13
2.5	Nền tảng lý thuyết	17
2.5.1	Kiến trúc hệ điều hành Linux	17
2.5.2	Các giao thức mạng truyền thông	19
2.5.3	Công nghệ phát triển phần mềm dựa trên Qt Framework	21
2.6	Kết luận chương	24
Chương 3: Thiết kế và triển khai phân hệ Windows App		25
3.1	Sơ đồ tổng quan của phân hệ Windows App	25
3.1.1	Sơ đồ trường hợp sử dụng (Use Case Diagram)	25
3.1.2	Sơ đồ lớp (Class Diagram)	25
3.2	Triển khai phân hệ Windows App	29
3.2.1	Kiến trúc mã nguồn sử dụng QML cho UI và C++ cho backend	29
3.2.2	Kết nối C++ với QML	31
3.2.3	Giao tiếp giữa Windows App và Linux App	32
3.2.4	Thiết kế các luồng hoạt động trong Windows App	36
3.2.5	Giao diện của phân hệ Windows App	41
3.3	Triển khai thuật toán phát hiện hệ thống quá tải	46
3.4	Triển khai thuật toán xử lý khi hệ thống quá tải	49
3.5	Kết luận chương	51
Chương 4: Thiết kế và triển khai phân hệ Linux App		52
4.1	Mục tiêu và vai trò của phân hệ Linux App	52
4.2	Sơ đồ tổng quan của phân hệ Linux App	52
4.3	Triển khai Linux App	61
4.3.1	Cách tiếp cận	61
4.3.2	Thu thập dữ liệu tài nguyên	63
4.3.3	Kết thúc tiến trình và cảnh báo đến người dùng	65

4.3.4	Vai trò của Linux App trong Stress test hệ thống IVI	67
4.4	Thiết kế các luồng hoạt động	68
4.5	Kết luận chương	69
Chương 5: Kiểm thử và đánh giá kết quả		71
5.1	Kiểm thử đơn vị	71
5.1.1	Kiểm thử khả năng xử lý dữ liệu	71
5.1.2	Kiểm thử chức năng lưu trữ dữ liệu	72
5.1.3	Kiểm thử khả năng phát hiện quá tải	73
5.1.4	Kiểm thử khả năng xử lý quá tải	73
5.1.5	Kiểm thử kết nối	73
5.1.6	Kiểm thử kích hoạt Stress test	74
5.2	Kiểm thử một số tính năng	76
5.2.1	Đánh giá trong ĐKKT1	76
5.2.2	Đánh giá trong ĐKKT2	79
5.2.3	Đánh giá trong ĐKKT3	81
5.3	Kết luận chương	83
Kết luận và hướng phát triển		84
6.1	Kết quả đã đạt được	84
6.2	Những thiếu sót và vấn đề cần cải thiện	84
6.3	Hướng phát triển trong tương lai	84
6.4	Kết luận chương	85
Tài liệu tham khảo		86
Phụ lục		87

DANH SÁCH CÁC BẢNG

3.1	Mô tả các trọng số, cấu hình nguồn giám sát hệ thống	44
4.1	Mô tả chi tiết các thuộc tính và phương thức trong sơ đồ lớp của Linux App.	53
4.2	Mô tả các thành phần chính trong cấu trúc mã nguồn phân hệ Linux App.	62

DANH SÁCH HÌNH VẼ

2.1	Mô hình tổng quan và cách thức giao tiếp giữa các đối tượng trong <i>Phần mềm</i>	8
2.2	Nguyên lý hoạt động của <i>Phần mềm</i>	14
2.3	Thu thập và hiển thị dữ liệu tải hệ thống IVI (trích xuất và mở rộng từ Hình 2.2).	14
2.4	Phát hiện và xử lý quá tải hệ thống IVI (trích xuất và mở rộng từ Hình 2.2).	15
2.5	Stress test hệ thống IVI (trích xuất và mở rộng từ Hình 2.2).	16
2.6	General Linux system organization [1].	18
2.7	Cách hoạt động của các giao thức mạng.	20
3.1	Sơ đồ Use Case của Windows App.	26
3.2	Sơ đồ lớp của Windows App.	28
3.3	Hộp thoại tạo dự án mới với Qt Creator.	29
3.4	Cấu trúc mặc định sau khi khởi tạo dự án mới với Qt Creator.	30
3.5	Cấu trúc mặc định sau khi khởi tạo dự án mới với Qt Creator.	31
3.6	Thiết lập kết nối TCP giữa Windows App và Linux App.	33
3.7	Thiết lập kết nối UDP giữa Windows App và Linux App.	34
3.8	Cấu trúc dữ liệu dạng JSON nhận từ Linux App.	35
3.9	Cấu trúc yêu cầu bắt đầu chạy Stress test dạng JSON.	36
3.10	Cấu trúc yêu cầu dừng Stress test dạng JSON.	36
3.11	Cấu trúc yêu cầu đóng một tiến trình dạng JSON.	36
3.12	Luồng hoạt động tổng quan của Windows App.	37
3.13	Luồng hoạt động xác định tình trạng của hệ thống.	38
3.14	Luồng hoạt động xử lý dựa vào tình trạng được xác định của hệ thống.	39
3.15	Luồng hoạt động lưu trữ dữ liệu	40

3.16 Giao diện màn hình hiển thị thông tin về tiến trình cùng với tài nguyên chiếm dụng	42
3.17 Giao diện màn hình hiển thị các biểu đồ trực quan	43
3.18 Giao diện màn hình Stress test	43
3.19 Giao diện màn hình hiển thị và thiết lập các thông số cần thiết	44
3.20 Giao diện màn hình hiển thị thông báo	46
3.21 Mô tả trường hợp xác nhận hệ thống quá tải	49
3.22 Mô tả trường hợp cảnh báo hệ thống có khả năng quá tải	49
4.1 Sơ đồ Use case của Linux App.	52
4.2 Sơ đồ lớp của Linux App.	53
4.3 Các tầng thiết kế của sơ đồ lớp Linux App.	58
4.4 Cấu trúc mã nguồn của phân hệ Linux App.	62
4.5 Các thông số dữ liệu Linux App thu thập	64
4.6 Luồng kết thúc tiến trình trên Linux App.	66
4.7 Luồng hoạt động trên Linux App	68
5.1 Một phần dữ liệu thô nhân được từ Linux App	71
5.2 Dữ liệu sau khi được xử lý	72
5.3 Dữ liệu được lưu trữ vào thư mục	72
5.4 Đánh giá tình trạng quá tải dựa vào các trạng thái	73
5.5 Chọn ra tiến trình phù hợp để kết thúc	73
5.6 Kết nối giữa Windows App và Linux App	74
5.7 Kiểm thử kích hoạt Stress test	74
5.8 Thay đổi đầu vào Stress test	75
5.9 Thông tin tiến trình và tài nguyên chiếm dụng trong ĐKKT1	77
5.10 Xu hướng sử dụng tải hệ thống IVI thể hiện tại Windows App trong ĐKKT1	78
5.11 Xu hướng sử dụng tải hệ thống IVI thể hiện tại System Monitor trong ĐKKT1.	78
5.12 Kết thúc tiến trình ”obs” chiếm nhiều tài nguyên trong ĐKK2	79

5.13 Kết thúc tiến trình ”brave” chiếm nhiều tài nguyên trong ĐKK2	80
5.14 Kiểm tra dữ liệu được lưu trữ trong file (mở rộng cho Hình 5.12)	80
5.15 Kiểm tra dữ liệu được lưu trữ trong file (mở rộng cho Hình 5.13)	81
5.16 Tiến trình ”stress” và mức chiếm dụng tài nguyên của nó	82
5.17 Kết thúc tiến trình ”stress” chiếm nhiều tài nguyên trong ĐKK3	83
0.1 Mô hình giao tiếp Client - Server	96

DANH SÁCH CÁC CHỮ VIẾT TẮT

Chữ viết tắt	Nghĩa tiếng Anh	Nghĩa tiếng Việt
IVI	In-Vehicle Infotainment	Thông tin giải trí trong xe
GPS	Global Positioning System	Hệ thống định vị toàn cầu
UDP	User Datagram Protocol	Giao thức datagram người dùng
TCP	Transmission Control Protocol	Giao thức điều khiển truyền tải
HTTP/HTTPS	HyperText Transfer Protocol/Secure	Giao thức truyền tải siêu văn bản/ bảo mật
SSL	Secure Sockets Layer	Lớp ẩn định kết nối bảo mật
CPU	Central Processing Unit	Đơn vị xử lý trung tâm
RAM	Random Access Memory	Bộ nhớ truy cập ngẫu nhiên
Swap	Swap Space	Bộ nhớ hoán đổi
MEM	Memory	Bộ nhớ
PID	Process Identification	Mã nhận dạng tiến trình
UID	User Identification	Mã nhận dạng người dùng
JSON	JavaScript Object Notation	Cú pháp đối tượng JavaScript
UML	Unified Modeling Language	Ngôn ngữ mô hình hóa thống nhất
MVVM	Model-View-ViewModel	Mô hình - Giao diện - Mô hình xem
Windows App	Windows Application	Ứng dụng Windows
Linux App	Linux Application	Ứng dụng Linux
MB	Megabyte	Megabyte
UI	User Interface	Giao diện người dùng
OOP	Object-oriented programming	Lập trình hướng đối tượng

MỞ ĐẦU

1. Mục đích thực hiện đề tài

Hệ thống IVI (In-Vehicle Infotainment) là nền tảng công nghệ tích hợp trong ô tô, đảm nhận vai trò giải trí, kết nối và hỗ trợ lái xe an toàn. Tuy nhiên, IVI đối mặt với thách thức lớn về quản lý tài nguyên do đặc thù phần cứng hạn chế, nhiều chức năng phải vận hành đồng thời, dễ dẫn tới quá tải, giảm hiệu năng, ảnh hưởng đến trải nghiệm người dùng, thậm chí gây nguy hiểm cho người dùng khi hệ thống mất kiểm soát.

Do đó, đề tài này hướng đến xây dựng một phần mềm giám sát và tùy chỉnh tải hệ thống IVI, giúp các bên phát triển, vận hành, chủ động theo dõi sự thay đổi bất thường trong dữ liệu tải tài nguyên hệ thống. Qua đó, kịp thời đưa ra các biện pháp xử lý phù hợp, góp phần giảm thiểu nguy cơ xấu nhất có thể xảy ra.

2. Mục tiêu đề tài

Đề tài hướng đến các mục tiêu cụ thể sau:

1. Xây dựng một ứng dụng chạy trên nền tảng Linux, có nhiệm vụ thu thập dữ liệu liên quan đến tải tài nguyên hệ thống. Đồng thời cung cấp các tác vụ điều khiển hệ thống, cho phép chủ động can thiệp như kết thúc tiến trình tiêu tốn tài nguyên, chạy lệnh kiểm thử tải (stress) hoặc phát cảnh báo âm thanh khi phát hiện tình trạng bất thường.
2. Xây dựng một ứng dụng có giao diện hiển thị trực quan, giúp người dùng dễ dàng quan sát thông tin về các tiến trình đang hoạt động, các xu hướng thay đổi dữ liệu tải như CPU, MEM theo thời gian thực.
3. Khả năng đánh giá tình trạng hiện tại của hệ thống và xác định ứng dụng nào chiếm nhiều tài nguyên.

3. Phạm vi và đối tượng nghiên cứu

1. Phạm vi nghiên cứu: tập trung vào việc xây dựng và triển khai một phần mềm giám sát và tùy chỉnh tải hệ thống IVI trên nền tảng hệ điều hành Linux. Đề tài chỉ khảo sát, thu thập và xử lý các thông tin về tài nguyên hệ thống (CPU, MEM, tiến trình) và các tác vụ điều khiển cơ bản như đóng tiến trình, chạy Stress test, phát cảnh báo âm thanh. Việc thử nghiệm được thực hiện trên thiết bị mô phỏng hệ thống IVI (chạy trên một laptop cá nhân dùng hệ điều hành Linux) trong thời gian thực hiện đồ án.
2. Đối tượng nghiên cứu: giải pháp phần mềm giám sát và tùy chỉnh tải hệ thống IVI

chạy trên nền tảng hệ điều hành Linux. Nghiên cứu tập trung vào việc theo dõi và điều phối tài nguyên hệ thống, đặc biệt là các tiến trình và tài nguyên hệ thống (CPU, MEM).

4. Phương pháp nghiên cứu

Phương pháp nghiên cứu của đề tài bao gồm các bước cụ thể như sau:

1. Khảo sát, phân tích tài liệu: tìm hiểu về hệ thống IVI, các nguyên nhân và tác động của tình trạng quá tải, tham khảo các giải pháp giám sát tài nguyên hiện có.
2. Thiết kế tổng quan: đề xuất mô hình kiến trúc phần mềm, xác định các chức năng cần có.
3. Hiện thực giải pháp: lập trình xây dựng một ứng dụng chạy trên nền tảng Linux và xây dựng một ứng dụng với giao diện giám sát trực quan, điều khiển trên Windows.
4. Kiểm thử và đánh giá: triển khai kiểm thử phần mềm trên môi trường mô phỏng (một laptop chạy hệ điều hành Linux), thực hiện các kịch bản thử nghiệm quá tải, đánh giá khả năng giám sát, kiểm tra chức năng cảnh báo và xử lý khi có sự cố.

5. Cấu trúc của đồ án tốt nghiệp

Cấu trúc đồ án bao gồm 6 chương:

Chương 1: Trình bày tổng quan đề tài

Chương 2: Thiết kế tổng quan phần mềm

Chương 3: Thiết kế và triển khai phân hệ Windows App

Chương 4: Thiết kế và triển khai phân hệ Linux App

Chương 5: Kiểm thử và đánh giá kết quả

Chương 6: Kết luận và hướng phát triển

CHƯƠNG 1: TỔNG QUAN ĐỀ TÀI

1.1. Nêu vấn đề

Hệ thống IVI (In-Vehicle Infotainment) là một nền tảng công nghệ tích hợp trong ô tô, kết hợp các chức năng giải trí, kết nối và hỗ trợ người lái. Với vai trò ngày càng quan trọng trong ngành công nghiệp ô tô hiện đại, IVI không chỉ nâng cao trải nghiệm người dùng mà còn góp phần vào sự phát triển của các công nghệ xe thông minh. Theo thống kê, giá trị thị trường của hệ thống IVI đã đạt khoảng 29 tỉ USD vào năm 2024 và được dự báo sẽ vượt mốc 70 tỉ USD vào năm 2034, với tốc độ tăng trưởng trung bình hàng năm đạt 9,32% [5], những con số này cho thấy IVI đang dần trở thành một thành phần không thể thiếu trong xu hướng phát triển của ngành ô tô toàn cầu.

Hệ thống IVI góp phần nâng cao trải nghiệm người dùng bằng cách tích hợp các dịch vụ như điều hướng GPS, phát nhạc, video, kết nối mạng và hỗ trợ lái xe an toàn như nhận diện biển báo, gợi ý nhắc nhở tài xế trong quá trình vận hành phương tiện. Tuy nhiên, việc quản lý tài nguyên và giám sát hiệu suất hệ thống IVI là một thách thức lớn đối với các nhà sản xuất ô tô và các nhà phát triển phần mềm trên hệ thống này, với đặc điểm là một hệ thống có tài nguyên phần cứng hạn chế, vì vậy có những trường hợp khi hệ thống phải thực thi nhiều ứng dụng và chức năng cùng lúc có thể dễ dẫn đến tình trạng hệ thống bị quá tải. Một số lỗi khi hệ thống IVI bị quá tải như hiệu năng giảm sút, chập chờn, không phản hồi kịp thời các thao tác của người dùng, gây ra lỗi vận hành và có thể đưa ra những chỉ dẫn/hoạt động sai, trong trường hợp xấu có thể dẫn đến khởi động lại hệ thống hoặc làm treo cả giao diện điều khiển xe, điều này có thể gây ra nguy hiểm lớn đến an toàn của những người trong xe. Những chức năng phổ biến trong hệ thống IVI có thể hoạt động sai lệch khi bị quá tải như điều hướng bản đồ sai hoặc trễ, báo sai tốc độ di chuyển, báo sai biển báo giao thông,... Những lỗi đó có thể gây ra hậu quả khiến tài xế đi sai luật và có thể dẫn đến tai nạn giao thông, tạo ra sự khó chịu khi sử dụng hệ thống. Đôi khi trong trường hợp hệ thống IVI bị quá tải nghiêm trọng dẫn đến nhiều chức năng bị dừng đột ngột, điều này rất nguy hiểm khi xe đang di chuyển ở tốc độ cao.

Từ vấn đề nêu trên, có thể xác định ba yếu tố chính dẫn đến tình trạng quá tải hệ thống IVI:

1. Giới hạn tài nguyên phần cứng của hệ thống IVI (CPU, RAM, bộ nhớ lưu trữ,...)
2. Hệ thống phải xử lý đồng thời nhiều ứng dụng và chức năng.
3. Phần mềm tích hợp trên hệ thống chưa được tối ưu tốt, dẫn đến tiêu tốn tài nguyên

không cần thiết (rò rỉ bộ nhớ, xử lý nền kém hiệu quả, chiếm dụng thừa CPU,...)

Hệ thống IVI là một hệ thống có tài nguyên phần cứng hạn chế, trong trường hợp chạy quá nhiều ứng dụng/chức năng cùng lúc có thể dẫn đến quá tải hệ thống, và khi những ứng dụng/chức năng đó được thiết kế kém tối ưu thì tình huống hệ thống IVI bị quá tải lại càng dễ dàng xảy ra. Từ góc nhìn này, có thể đề xuất một số hướng tiếp cận nhằm giảm thiểu và phòng tránh tình trạng quá tải:

1. Cải thiện phần cứng hệ thống IVI.
2. Tối ưu phần mềm tích hợp trên hệ thống.
3. Giám sát hiệu năng sử dụng tài nguyên hệ thống.
4. Quản lý và điều phối mức tài nguyên hệ thống được sử dụng tại một thời điểm.

Việc nâng cấp phần cứng là không khả thi trong đa số trường hợp vì chi phí cao, phức tạp về mặt kỹ thuật và không phù hợp trong bối cảnh thương mại. Những đề xuất còn lại thì mang tính khả thi cao và chi phí thấp, cụ thể, với *tối ưu phần mềm* giúp giảm thiểu tình trạng các ứng dụng chiếm dụng tài nguyên không cần thiết, từ đó tiết kiệm tài nguyên và duy trì hiệu năng ổn định. Với *giám sát hiệu năng hệ thống* cho phép phát hiện sớm các vấn đề quá tải và cung cấp dữ liệu để cải thiện hiệu quả hoạt động của các phần mềm trên hệ thống (nhằm tối ưu phần mềm), và cuối cùng là *quản lý tài nguyên tại thời điểm thực thi* giúp đảm bảo rằng chỉ có một lượng ứng dụng vừa đủ được chạy trong cùng một thời điểm, phù hợp với khả năng xử lý của hệ thống, từ đó tránh vượt ngưỡng tài nguyên và giảm thiểu nguy cơ treo hệ thống.

Từ những hướng tiếp cận trên, chúng tôi đưa ra giải pháp với mục tiêu giúp nhà sản xuất và nhà phát triển phần mềm tạo ra các dịch vụ đảm bảo phù hợp với điều kiện tài nguyên phần cứng của hệ thống IVI. Đồng thời, giải pháp này giúp cho đội ngũ kỹ thuật và vận hành theo dõi tài nguyên của hệ thống IVI trong quá trình vận hành thực tế, nhằm khắc phục kịp thời sự cố khi hệ thống bị quá tải, đồng thời cung cấp thông tin để đội ngũ vận hành có thể nhận biết và chẩn đoán nguyên nhân lỗi một cách chuẩn xác. Vậy cần làm như thế nào để nhà sản xuất và nhà phát triển phần mềm có thể tối ưu dịch vụ của họ phù hợp với tài nguyên phần cứng của hệ thống IVI, đồng thời có thể theo dõi hệ thống trong quá trình vận hành? Nội dung tại phần tiếp theo sẽ lý giải rõ về câu hỏi này.

1.2. Giải pháp đặt ra

Với những gì đã phân tích, có thể thấy rằng việc đảm bảo hệ thống IVI hoạt động ổn định và hạn chế bị quá tải đòi hỏi sự kết hợp giữa tối ưu phần mềm, giám sát hiệu năng và điều phối hợp lý tài nguyên hệ thống. Để hiện thực hóa điều đó, chúng tôi đề xuất một giải pháp *hệ thống phần mềm* nhằm hỗ trợ cả hai giai đoạn chính trong vòng đời hệ thống IVI: **giai đoạn phát triển** và **giai đoạn vận hành**.

Tại **giai đoạn phát triển**, mục tiêu là giúp các nhà phát triển đánh giá mức tiêu thụ tài nguyên (CPU, bộ nhớ,...) của từng dịch vụ, phần mềm (từ giờ chúng tôi sẽ gọi chung là *tiến trình*) đang được phát triển. Từ đó có thể phát hiện các vấn đề tiềm ẩn như rò rỉ bộ nhớ, chiếm dụng CPU bất thường, hoặc xung đột giữa các tiến trình trên hệ thống. Giải pháp đề xuất đó là thiết kế và phát triển hệ thống phần mềm có thể thu thập dữ liệu tài nguyên, hiển thị dữ liệu đó theo thời gian thực để mô tả xu hướng, lưu lượng tài nguyên được tiêu thụ trên hệ thống IVI, giúp nhà phát triển dễ dàng đánh giá và điều chỉnh các tiến trình đang phát triển sao cho phù hợp với giới hạn phần cứng của hệ thống IVI. Ngoài ra, trong môi trường kiểm thử, *hệ thống phần mềm* này còn hỗ trợ thiết lập các kịch bản mô phỏng tải nhằm kiểm tra khả năng chịu đựng của hệ thống IVI trong các điều kiện khác nhau (còn được gọi là Stress test).

Tại **giai đoạn vận hành**, lúc này hệ thống đã được đưa vào sử dụng thực tế, nên yêu cầu chính là đảm bảo hoạt động ổn định, phát hiện và xử lý kịp thời các tình huống quá tải. *Hệ thống phần mềm* mà chúng tôi đề xuất có khả năng giám sát tài nguyên trên hệ thống IVI theo thời gian thực, đưa ra cảnh báo khi phát hiện dấu hiệu quá tải và hỗ trợ cơ chế phản ứng trong trường hợp hệ thống bị quá tải như tự động kết thúc tiến trình tiêu tốn tài nguyên vượt ngưỡng. Ngoài ra, *hệ thống phần mềm* còn có thể lưu dữ liệu tài nguyên vào cơ sở dữ liệu, giúp đội ngũ kỹ thuật theo dõi xu hướng sử dụng tài nguyên, phân tích nguyên nhân gây ra lỗi, từ đó đề xuất giải pháp bảo trì và tối ưu hệ thống IVI hiệu quả hơn.

Tóm lại, giải pháp *hệ thống phần mềm* do chúng tôi đề xuất đóng vai trò hỗ trợ trong cả hai giai đoạn chính của hệ thống IVI. Trong giai đoạn phát triển, công cụ này giúp nhà phát triển tối ưu các phần mềm đang phát triển trên hệ thống dựa trên dữ liệu thực tế, giảm thiểu nguy cơ mắc các lỗi gây ra quá tải ngay từ khâu thiết kế. Trong giai đoạn vận hành, công cụ này giúp giám sát, cảnh báo và điều chỉnh kịp thời khi hệ thống IVI xảy ra tình trạng quá tải, đảm bảo hệ thống IVI hoạt động ổn định, an toàn và hiệu quả, cũng như hỗ trợ cho cả quá trình bảo trì hệ thống IVI.

Chúng tôi gọi *hệ thống phần mềm* này là **”Phần mềm giám sát và tùy chỉnh tải hệ thống IVI”**. Đây chính là giải pháp cốt lõi trong đề tài này của chúng tôi.

1.3. Phần mềm giám sát và tùy chỉnh tải hệ thống IVI là gì?

Phần mềm giám sát và tùy chỉnh tải hệ thống IVI (từ giờ chúng tôi sẽ gọi là *Phần mềm*) là một hệ thống phần mềm được thiết kế với mục tiêu theo dõi, phân tích và quản lý việc sử dụng tài nguyên hệ thống (CPU, bộ nhớ, tiến trình) trong các thiết bị IVI, nhằm đảm bảo hệ thống hoạt động ổn định, an toàn và hiệu quả.

Phần mềm bao gồm hai thành phần chính: Performance Service hoạt động trên hệ điều hành Linux (gọi là Linux App) và PerformanceViewer App hoạt động trên hệ điều hành Windows (gọi là Windows App).

Với Linux App, thành phần này chạy trên thiết bị mô phỏng hệ thống IVI¹, có nhiệm vụ thu thập dữ liệu tài nguyên của hệ thống IVI theo thời gian thực như mức sử dụng CPU, bộ nhớ RAM và Swap, thông tin tiến trình, đồng thời có khả năng thực hiện các hành động như kết thúc tiến trình tiêu tốn nhiều tài nguyên trên hệ thống IVI, gửi cảnh báo thông qua loa trên hệ thống khi vượt ngưỡng tải và tạo tải giả (tạo các tiến trình ảo tiêu tốn tài nguyên) cho quá trình kiểm thử và đánh giá khả năng chịu đựng của hệ thống IVI.

Với Windows App, thành phần giao diện người dùng chạy trên máy tính có hệ điều hành Windows, có nhiệm vụ hiển thị trực quan dữ liệu tài nguyên từ hệ thống IVI, cho phép người phát triển/vận hành (người dùng) quan sát và phân tích xu hướng tiêu thụ tài nguyên của hệ thống IVI để can thiệp khi cần thiết. Đồng thời, ứng dụng này còn có nhiệm vụ đánh giá và tùy chỉnh tải cho hệ thống IVI thông qua các thuật toán phát hiện quá tải hệ thống và chọn lọc các tiến trình tiêu thụ tài nguyên quá mức để kết thúc, từ đó cân bằng tải cho hệ thống IVI. Windows App có thể kết nối từ xa với Linux App thông qua kết nối mạng để trao đổi dữ liệu và lệnh điều khiển.

1.4. Kết luận chương

Qua những nội dung mà chúng tôi đã trình bày trong chương này, có thể thấy rằng hệ thống IVI đang ngày càng trở nên phổ biến và đóng vai trò quan trọng trong các phương

¹ Thiết bị mô phỏng hệ thống IVI là một máy tính nhúng hoặc hệ thống máy tính chạy hệ điều hành Linux, được sử dụng để mô phỏng lại môi trường phần cứng và phần mềm gần giống với hệ thống IVI thực tế trên ô tô, giúp nhà phát triển kiểm thử và đánh giá các phần mềm tích hợp lên hệ thống trong điều kiện gần giống thực tế trước khi triển khai lên hệ thống IVI thực.

tiện hiện đại. Tuy nhiên, với đặc điểm là một hệ thống có tài nguyên phần cứng hạn chế, việc vận hành nhiều tiến trình cùng lúc có thể dẫn đến tình trạng quá tải, ảnh hưởng xấu đến hiệu năng, độ ổn định và tính an toàn của hệ thống IVI.

Từ những nhận định trên, chúng tôi xác định được vấn đề cốt lõi cần giải quyết, đó là: làm thế nào để giám sát, đánh giá và kiểm soát việc sử dụng tài nguyên trên hệ thống IVI một cách hiệu quả, từ đó giúp nhà phát triển tối ưu phần mềm phù hợp với giới hạn phần cứng và hỗ trợ đội ngũ vận hành theo dõi hệ thống trong thực tế.

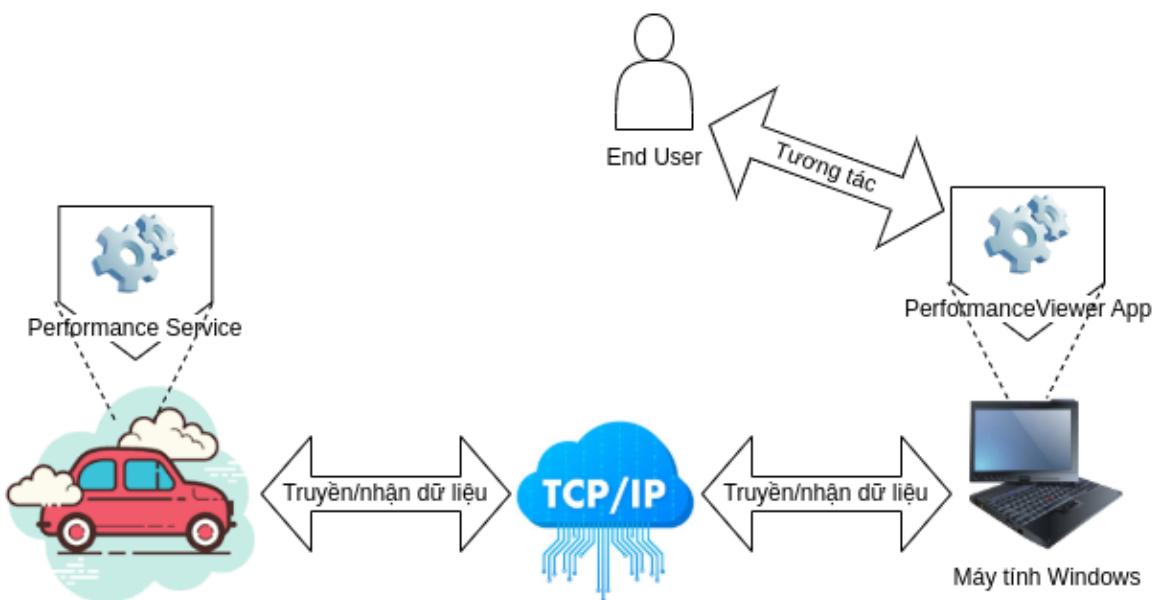
Giải pháp được đề xuất là *Phần mềm* gồm hai thành phần: Performance Service (Linux App) và PerformanceViewer App (Windows App), hợp tác với nhau thông qua kết nối mạng để giám sát, phân tích và tùy chỉnh tải hệ thống IVI theo thời gian thực.

Sản phẩm đầu ra là một hệ thống phần mềm hoàn chỉnh, có khả năng hỗ trợ cả trong giai đoạn phát triển và giai đoạn vận hành hệ thống IVI. Tại chương tiếp theo, chúng tôi sẽ đi vào chi tiết về thiết kế tổng quan cấu trúc, chức năng và nguyên lý hoạt động của *Phần mềm*.

CHƯƠNG 2: THIẾT KẾ TỔNG QUAN PHẦN MỀM

2.1. Mô hình tổng quan Phần mềm

Dựa trên những vấn đề đã phân tích và giải pháp đã đề xuất trong Chương 1, chúng tôi xây dựng một hệ thống phần mềm bao gồm hai thành phần chính: **Performance Service** (Linux App) chạy trên thiết bị mô phỏng hệ thống IVI và **PerformanceViewer App** (Windows App) chạy trên máy tính Windows. Hai thành phần này giao tiếp với nhau thông qua kết nối mạng, trong đó giao thức TCP/IP được sử dụng làm chuẩn giao tiếp trong đề tài này. Lý do lựa chọn giao thức này sẽ được chúng tôi trình bày tại §2.5. Việc sử dụng giao thức TCP/IP giúp kết nối các phân hệ thành một hệ thống phần mềm thống nhất nhằm thực hiện chức năng giám sát, phân tích và điều phối tài nguyên trên hệ thống IVI.



Hình 2.1 Mô hình tổng quan và cách thức giao tiếp giữa các đối tượng trong *Phần mềm*.

Hình 2.1 mô tả tổng quan cấu trúc và cơ chế hoạt động của *Phần mềm*. Trong đó, Linux App² đảm nhận vai trò thu thập dữ liệu tải và thực thi các hành động cân bằng tải tại phía hệ thống IVI - trong Hình 2.1 mô tả hệ thống IVI trong thực tế khi hệ thống này được tích hợp trên phương tiện và đang trong quá trình vận hành. Còn Windows App đóng vai trò hiển thị, điều khiển và phản hồi từ người dùng thông qua việc người dùng tương tác với Windows App trên máy tính Windows. Các thành phần được thiết kế với

²Nhu đã đề cập trước đó, chúng tôi gọi Performance Service là Linux App cho ngắn gọn và dễ hiểu, tương tự như vậy, Windows App là cách gọi ngắn gọn cho PerformanceViewer App.

mục tiêu có thể hoạt động trong cả môi trường phát triển và vận hành thực tế. Trong đề tài này, chúng tôi sử dụng một máy tính Linux để mô phỏng hệ thống IVI cho quá trình phát triển và đánh giá *Phần mềm*, như đã đề cập thì hệ thống IVI hoạt động dựa trên hệ điều hành Linux, vì vậy việc sử dụng một máy tính Linux là lựa chọn phù hợp để mô phỏng hệ thống này, đảm bảo rằng Linux App không chỉ có thể hoạt động trong môi trường phát triển mà còn có khả năng hoạt động trong cả môi trường vận hành thực tế khi chúng tôi tích hợp Linux App lên một hệ thống IVI thực tế.

Tại phần tiếp theo, chúng tôi sẽ trình bày những yêu cầu, chức năng mà *Phần mềm* nói chung và từng phân hệ nói riêng được triển khai - với hai thành phần là Linux App và Windows App trong *Phần mềm*, từ giờ chúng tôi sẽ gọi chúng là các phân hệ như *phân hệ Linux App* và *phân hệ Windows App* cho dễ phân biệt.

2.2. Một số yêu cầu chung của Phần mềm

Dựa trên các mục tiêu đã đặt ra trong chương trước và các chức năng tổng quan của phần mềm đã được trình bày ở mục trước, trong phần này chúng tôi xác định các yêu cầu cốt lõi mà *Phần mềm* cần đáp ứng nhằm đảm bảo khả năng giám sát hiệu quả, xử lý kịp thời tình trạng quá tải và hỗ trợ phát triển các ứng dụng tích hợp lên hệ thống IVI phù hợp với đặc điểm tài nguyên hạn chế của hệ thống này. Trong đó, có một số khái niệm chính cần được làm rõ để hiểu đúng vai trò và chức năng của từng khôi, phân hệ trong *Phần mềm*.

Phát hiện quá tải hệ thống là quá trình đánh giá mức tiêu thụ tài nguyên (CPU, bộ nhớ) của hệ thống IVI trong thời gian thực và xác định khi nào hệ thống rơi vào trạng thái sử dụng vượt quá ngưỡng tài nguyên cho phép. Việc phát hiện quá tải là rất quan trọng vì hệ thống IVI có giới hạn phần cứng, nếu không kiểm soát tốt mức tài nguyên tiêu thụ sẽ dẫn đến mất ổn định, ảnh hưởng đến quá trình vận hành của hệ thống, phát hiện quá tải đóng vai trò quan trọng trong việc kích hoạt chức năng *xử lý quá tải hệ thống*, từ đó đảm bảo hệ thống IVI hạn chế hoạt động trong vùng quá tải, cải thiện hiệu suất và mức độ ổn định của hệ thống.

Xử lý quá tải hệ thống là việc thực hiện các hành động can thiệp vào hệ thống IVI để giảm lượng tài nguyên đang tiêu thụ, cụ thể như sau khi phát hiện hệ thống đang quá tải, chức năng này sẽ chọn lọc ra tiến trình cần kết thúc - là tiến trình tiêu tốn nhiều tài nguyên và có mức ưu tiên sử dụng thấp - và tiến hành kết thúc nó trên hệ thống IVI đồng thời phát cảnh báo đến người dùng. Việc xử lý quá tải giúp đảm bảo hệ thống tiếp tục vận hành ổn định, tránh tình trạng quá tải kéo dài dẫn đến treo, giật lag hoặc sập hệ thống

VI.

Stress test là quá trình tạo tải giả lập - bằng cách khởi chạy các tiến trình tiêu tốn tài nguyên - để kiểm thử khả năng chịu tải và đánh giá độ ổn định của hệ thống IVI trong điều kiện khắc nghiệt. Điều này đặc biệt cần thiết trong giai đoạn phát triển, giúp nhà phát triển phát hiện sớm các lỗi tiềm ẩn và đảm bảo hệ thống IVI hoạt động ổn định trước khi đưa vào vận hành thực tế.

Để hiểu rõ chi tiết hơn về các yêu cầu cho *Phần mềm*, chúng tôi sẽ trình bày chúng trong hai phân hệ chính: *phân hệ Linux App* và *phân hệ Windows App*.

2.3. Yêu cầu chức năng của các phân hệ

2.3.1. Phân hệ Linux App

Phân hệ Linux App là thành phần trực tiếp chạy trên thiết bị mô phỏng hệ thống IVI, chịu trách nhiệm thu thập dữ liệu tài nguyên và thực thi các hành động xử lý quá tải theo yêu cầu từ phân hệ Windows App. Các yêu cầu chức năng chính của phân hệ này bao gồm:

1. Thu thập dữ liệu tài nguyên hệ thống theo thời gian thực. Những dữ liệu tài nguyên mà Linux App thu thập bao gồm các dữ liệu về CPU - CPU toàn cục và trên từng lõi (core) - bộ nhớ RAM và bộ nhớ Swap - mức phần trăm sử dụng và tổng dung lượng - và danh sách tiến trình đang hoạt động trên hệ thống IVI bao gồm các chỉ số như PID (process ID), phần trăm CPU, phần trăm MEM và đối tượng sở hữu tiến trình này³ (User/UID). Lý do thu thập các dữ liệu trên là vì khi xét hệ thống có quá tải hay không thì tài nguyên ảnh hưởng lớn nhất chính là bộ nhớ, đây là tài nguyên được sử dụng nhiều nhất và có nguy cơ bị quá tải nhất, tiếp đến là mức tiêu thụ CPU, bao gồm mức phần trăm tiêu thụ CPU, nhiệt độ và tần số hoạt động của CPU, các thông số này ảnh hưởng lớn đến hiệu suất của hệ thống. Khi các thông số này trên hệ thống ở mức cao liên tục, nó sẽ dẫn đến hiệu suất của hệ thống giảm, hoạt động của các tiến trình bị sai lệch đi và có thể khiến hệ thống bị quá tải. Dữ liệu được cập nhật định kỳ theo chu kỳ 1 giây và gửi về phân hệ Windows App để hiển thị, lưu trữ và đánh giá.
2. Tiến hành kết thúc các tiến trình theo yêu cầu từ phân hệ Windows App và phát cảnh báo đến người dùng. Khi phân hệ Windows App gửi lệnh - là gói tin chứa lệnh kết

³Điều này có liên quan đến quản lý tiến trình trên hệ điều hành Linux, nơi mà mỗi tiến trình đều được gắn với một người dùng cụ thể (user) – thường được xác định bởi UID. Việc xác định đối tượng sở hữu tiến trình giúp phân biệt các tiến trình hệ thống và tiến trình của người dùng thông thường, từ đó đưa ra các chính sách quản lý, ưu tiên hoặc xử lý phù hợp.

thúc tiến trình và tên tiến trình cần kết thúc - nhằm cân bằng tải trên hệ thống IVI đến phân hệ Linux App, phân hệ này sẽ tiến hành can thiệp vào hệ thống IVI để kết thúc tiến trình theo thông tin nhận được và phát cảnh báo đến người dùng - chúng tôi sẽ trình bày về cách làm thế nào mà Linux App có thể can thiệp vào hệ thống IVI để có thể thực hiện các chức năng tương tự như thế này trong §2.5.

3. Tạo tải giả lập trên hệ thống IVI thông qua yêu cầu từ phân hệ Windows App. Phân hệ Linux App hỗ trợ khả năng khởi tạo các tiến trình giả lập để tạo tải nặng (stress) lên hệ thống IVI, phục vụ cho mục đích kiểm thử khả năng chịu đựng tải cao trong môi trường phát triển. Việc bắt đầu và dừng quá trình Stress test sẽ do phân hệ Windows App điều khiển từ xa.

Bên cạnh các yêu cầu chức năng chính nêu trên, một yêu cầu tối quan trọng không thể thiếu đối với phân hệ Linux App đó là chính bản thân Linux App không tiêu thụ nhiều tài nguyên khi hoạt động trên hệ thống IVI, phân hệ này phải được thiết kế tối ưu, đảm bảo chiếm dụng rất ít tài nguyên hệ thống IVI trong quá trình vận hành. Lý do là vì bản thân hệ thống IVI đã có giới hạn nghiêm ngặt về phần cứng, và phân hệ Linux App cũng đang đóng vai trò là ứng dụng giám sát và điều phối tài nguyên cho hệ thống, nên việc bổ sung Linux App vào hệ thống IVI không được phép tạo thêm gánh nặng đáng kể lên tài nguyên của hệ thống. Do đó, Linux App cần được viết gọn nhẹ, tối ưu hiệu suất và đảm bảo hoạt động ổn định ngay cả trong điều kiện hệ thống đang gần đến ngưỡng quá tải.

2.3.2. **Phân hệ Windows App**

Phân hệ Windows App là thành phần có giao diện người dùng, được phát triển và hoạt động trên máy tính có hệ điều hành Windows. Phân hệ có nhiệm vụ trong việc hiển thị dữ liệu thu thập được từ hệ thống IVI bởi Linux App, cho phép người dùng (nhà phát triển hoặc người vận hành/bảo trì) quan sát, phân tích và kiểm soát tải hệ thống khi cần thiết. Các yêu cầu chức năng chính của phân hệ Windows App bao gồm:

1. Hiển thị trực quan dữ liệu tài nguyên hệ thống IVI. Dữ liệu nhận được từ Linux App cần được trực quan hóa dưới dạng biểu đồ, chuyển động, bảng số liệu nhằm giúp người dùng dễ dàng theo dõi tình trạng hệ thống IVI. Cụ thể chúng tôi xây dựng hai biểu đồ đường cho CPU và bộ nhớ để biểu diễn sự thay đổi của chúng theo thời gian thực, thêm vào đó là xây dựng bảng tiến trình để người dùng có thể quan sát các thông số của các tiến trình (như đã đề cập tại phân hệ Linux App) đang hoạt động trên hệ thống IVI, và tất nhiên là theo thời gian thực.

2. Đánh giá tình trạng và phát hiện quá tải hệ thống. Sau khi nhận dữ liệu tải, bên cạnh việc hiển thị thì Windows App còn sử dụng dữ liệu này để tiến hành đánh giá tình trạng hiện tại của hệ thống IVI. Tại đây, chúng tôi sẽ phát triển một thuật toán dùng để đánh giá tình trạng tải hiện tại của hệ thống IVI, sau đó so sánh với mức ngưỡng được thiết lập - mức ngưỡng này chúng tôi thiết kế cho phép người dùng tùy chỉnh theo mục đích và phù hợp với tình huống cá nhân - từ đó đưa ra kết quả chẩn đoán hệ thống IVI đang hoạt động trong mức tải phù hợp hay bị quá tải.
3. Xử lý quá tải hệ thống IVI. Đây là một chức năng rất quan trọng, cho phép *Phần mềm* chủ động cân bằng tải cho hệ thống IVI, từ đó khắc phục các tình trạng như treo, giật lag hoặc phản hồi chậm khi hệ thống bị quá tải.

Cụ thể, khi phân hệ Windows App đánh giá hệ thống IVI đang trong trạng thái quá tải (dựa trên kết quả từ *Đánh giá tình trạng và phát hiện quá tải hệ thống* được nêu ở trên), phần mềm sẽ tiến hành chọn lọc và quyết định kết thúc các tiến trình tiêu thụ nhiều tài nguyên nhưng không quan trọng, nhằm giải phóng tài nguyên cho hệ thống IVI.

Về nguyên tắc, các tiến trình trên hệ điều hành Linux có thể được phân thành hai nhóm chính: tiến trình hệ thống và tiến trình thường. Tiến trình hệ thống là các tiến trình lõi có vai trò điều phối và vận hành hoạt động trong hệ điều hành. Nếu vô tình kết thúc các tiến trình này, có thể gây ra lỗi vận hành nghiêm trọng hoặc khiến hệ thống bị crash. Trong khi đó, tiến trình thường là những tiến trình mà khi kết thúc sẽ không ảnh hưởng nghiêm trọng đến quá trình vận hành hệ thống. Tuy nhiên, vẫn cần lưu ý rằng có những tiến trình thường nhưng đóng vai trò quan trọng với hệ điều hành như tiến trình điều khiển bàn phím, màn hình, mặc dù có thể kết thúc nó và không gây ra lỗi vận hành của hệ điều hành, xong việc này có thể khiến cho hệ thống của chúng ta gặp phải lỗi khi tương tác với người dùng hoặc các vấn đề tương tự.

Ngoài ra, *Phần mềm* còn cân nhắc đến mức độ ưu tiên từ phía người dùng. Ví dụ, trong hệ thống IVI, tiến trình *điều hướng bản đồ GPS* thường được xem là quan trọng và cần duy trì hoạt động hơn so với tiến trình *phát nhạc hoặc video*. Do đó, nếu hai tiến trình này tiêu thụ lượng tài nguyên tương đương nhau trong điều kiện quá tải hệ thống IVI, *Phần mềm* sẽ ưu tiên giữ lại tiến trình *điều hướng bản đồ GPS* và lựa chọn kết thúc tiến trình *phát nhạc hoặc video*.

Sau khi quá trình đánh giá và chọn lọc hoàn tất, phân hệ Windows App sẽ gửi lệnh đến phân hệ Linux App để thực hiện việc kết thúc các tiến trình đã được lựa chọn trên hệ thống IVI. Nhờ đó, lượng tài nguyên tiêu thụ sẽ được giảm xuống, giúp hệ

thống IVI phục hồi trạng thái cân bằng và có thể hoạt động ổn định, trơn tru hơn.

4. Lưu trữ dữ liệu lịch sử để phục vụ phân tích. Windows App ghi nhận và lưu trữ các dữ liệu tài nguyên của hệ thống IVI theo thời gian vào cơ sở dữ liệu. Lưu trữ bao gồm hai loại: thứ nhất là toàn bộ dữ liệu nhận được từ phân hệ Linux App và thứ hai là dữ liệu khi phát hiện hệ thống IVI đang bị quá tải.

Việc lưu trữ dữ liệu là một yêu cầu quan trọng nhằm cho phép người dùng truy xuất lại các thời điểm cụ thể để kiểm tra tình trạng tài nguyên của hệ thống IVI, hỗ trợ phân tích xu hướng sử dụng tài nguyên trên hệ thống IVI theo thời gian, từ đó phát hiện các vấn đề tiềm ẩn hoặc xác định nguyên nhân gây ra hiện tượng quá tải hoặc hiệu năng giảm sút. Ngoài ra chức năng này còn có thể giúp cung cấp dữ liệu đầu vào cho quá trình đánh giá, tối ưu phần mềm trong *giai đoạn phát triển*, hoặc điều chỉnh cấu hình hệ thống trong *giai đoạn vận hành*.

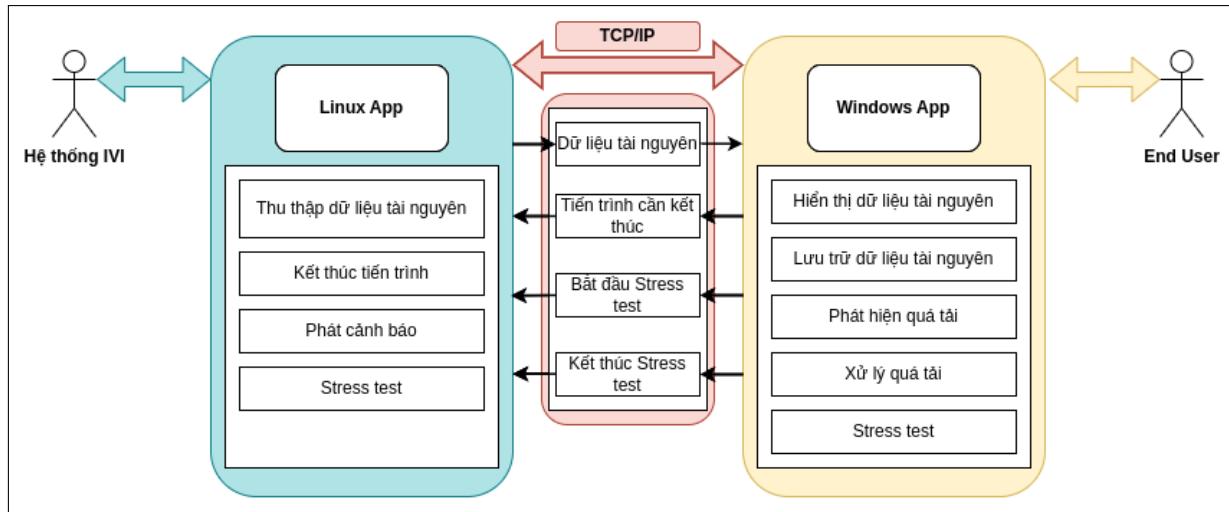
Dữ liệu được lưu trữ cần được tổ chức một cách có cấu trúc, đồng thời đảm bảo không chiếm dụng quá nhiều bộ nhớ lưu trữ của máy tính người dùng trong quá trình phân hệ này vận hành. Việc trích xuất dữ liệu trong cơ sở dữ liệu cần đơn giản, dễ thao tác và có thể phục vụ việc đánh giá và phân tích khi cần thiết.

Việc thực hiện các xử lý logic này tại phân hệ Windows App giúp giảm tải cho phân hệ Linux App, đặc biệt là về kích thước chương trình và mức tiêu thụ tài nguyên của Linux App. Nhờ đó, *Phần mềm* có thể đáp ứng tốt yêu cầu phi chức năng đã đặt ra cho phân hệ Linux App, đó là hoạt động gọn nhẹ, ổn định và không gây ảnh hưởng đến hiệu năng tổng thể của hệ thống IVI mà chúng tôi đã đề cập tại phân hệ Linux App.

2.4. Nguyên lý hoạt động của Phần mềm

Trong phần này, chúng tôi trình bày chi tiết cách thức hoạt động của *Phần mềm*, tập trung vào cơ chế tương tác giữa hai phân hệ chính: *Linux App* và *Windows App*. Thông qua nội dung tại phần này, người đọc có thể hiểu được cách các chức năng được triển khai, vận hành và phối hợp với nhau để đáp ứng các yêu cầu đã nêu trong phần trước.

Hình 2.2 mô tả nguyên lý hoạt động tổng thể của *Phần mềm*, thể hiện các thành phần chính cùng các kênh giao tiếp dữ liệu giữa chúng. Trong mô hình này, hai phân hệ Linux App (biểu diễn thông qua màu xanh) và Windows App (biểu diễn thông qua màu vàng) giao tiếp với nhau thông qua giao thức TCP/IP (biểu diễn thông qua màu hồng đỏ). Mỗi phân hệ chịu trách nhiệm một phần công việc cụ thể, đóng vai trò bổ trợ cho nhau để tạo thành một hệ thống phần mềm hoàn chỉnh và thống nhất.

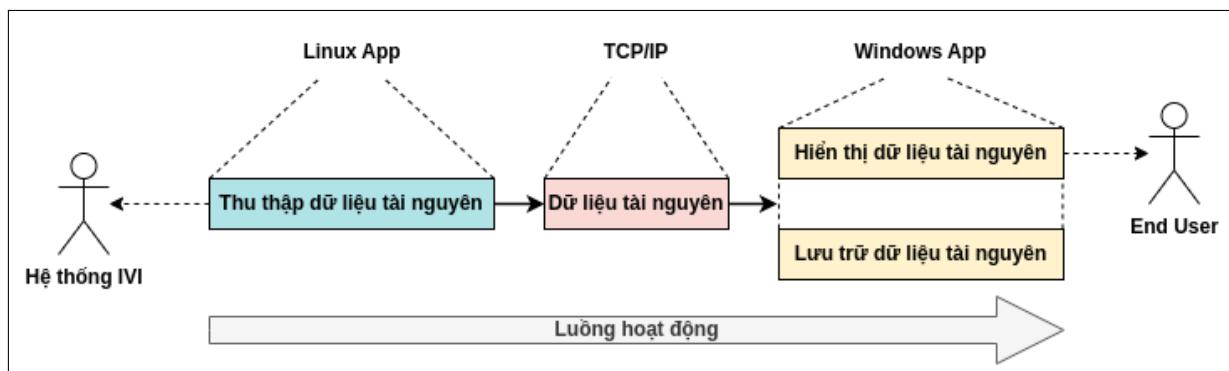


Hình 2.2 Nguyên lý hoạt động của *Phần mềm*.

Trong phân hệ Linux App, chúng tôi thiết kế các chức năng bao gồm: *Thu thập dữ liệu tài nguyên*, *Kết thúc tiến trình*, *Phát cảnh báo* và một phần trách nhiệm trong chức năng *Stress test*. Trong khi đó, phân hệ Windows App đảm nhiệm chức năng bao gồm: *Hiển thị dữ liệu*, *Lưu trữ dữ liệu*, *Phát hiện* và *xử lý quá tải* và phần điều khiển trong chức năng *Stress test*.

Để minh họa rõ ràng hơn cách thức vận hành, chúng tôi sẽ trình bày chi tiết cách hoạt động của *Phần mềm* thông qua các luồng chức năng chính được trích xuất từ sơ đồ tổng thể trên (Hình 2.2), bao gồm:

1. Luồng chức năng thu thập và hiển thị dữ liệu tải của hệ thống IVI.
2. Luồng chức năng phát hiện và xử lý quá tải hệ thống IVI.
3. Luồng chức năng Stress test hệ thống IVI.

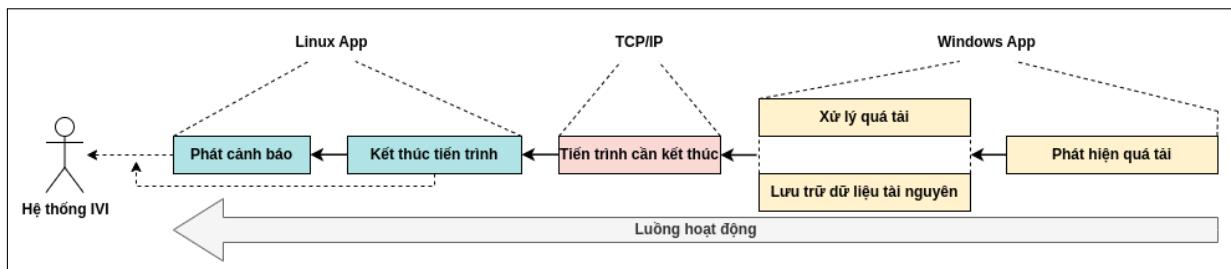


Hình 2.3 Thu thập và hiển thị dữ liệu tải hệ thống IVI (trích xuất và mở rộng từ Hình 2.2).

Hình 2.3 mô tả luồng chức năng *thu thập và hiển thị dữ liệu tải* của hệ thống IVI. Đây

là luồng chức năng cơ bản và quan trọng nhất của *Phần mềm*, thực hiện việc thu thập, truyền và hiển thị dữ liệu tài nguyên hệ thống IVI. Cách hoạt động của luồng chức năng này theo trình tự như sau:

1. **Thu thập dữ liệu tài nguyên:** đầu tiên tại Linux App, phân hệ này sẽ tiến hành thu thập dữ liệu tài nguyên trên hệ thống IVI (bao gồm các thông số về CPU, bộ nhớ RAM, bộ nhớ Swap và các dữ liệu tiến trình như đã đề cập ở nội dung trước đó) theo thời gian thực.
2. **Truyền dữ liệu tài nguyên:** sau khi thu thập xong dữ liệu tài nguyên, Linux App tiến hành đóng gói dữ liệu này (Cấu trúc/định dạng các gói dữ liệu truyền và nhận giữa hai phân hệ sẽ được chúng tôi giải thích rõ trong các phần tiếp theo) và truyền đến Windows App thông qua giao thức TCP/IP.
3. **Hiển thị và lưu trữ dữ liệu tài nguyên:** tại Windows App, sau khi nhận gói dữ liệu tài nguyên, phân hệ này sẽ thực hiện song song hai chức năng. Thứ nhất, nó sẽ hiển thị dữ liệu tài nguyên theo định dạng biểu đồ và bảng dữ liệu, giúp người dùng có thể quan sát một cách trực quan về trạng thái tiêu thụ tài nguyên hiện tại của hệ thống IVI. Thứ hai, Windows App sẽ tiến hành lưu trữ dữ liệu vừa nhận được vào cơ sở dữ liệu, giúp người dùng có thể truy xuất được lịch sử tiêu thụ tài nguyên của hệ thống IVI khi cần thiết.



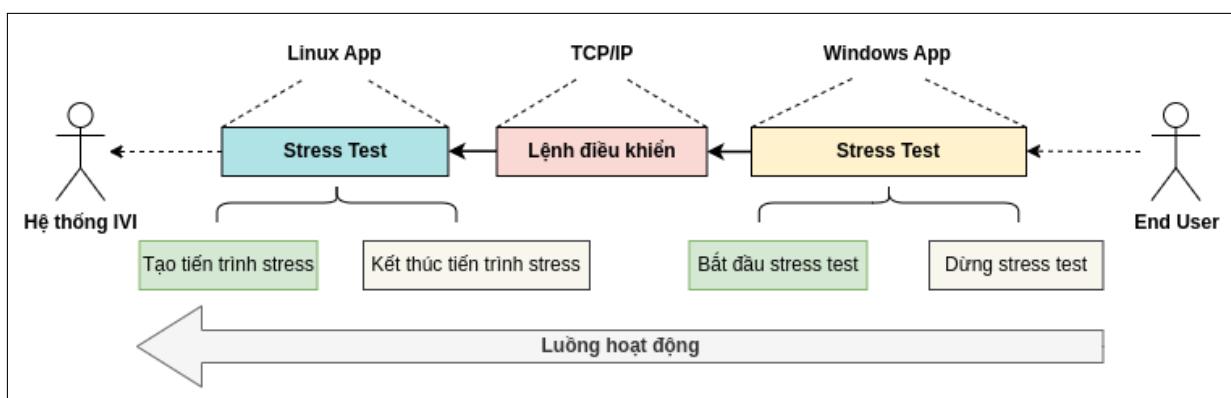
Hình 2.4 Phát hiện và xử lý quá tải hệ thống IVI (trích xuất và mở rộng từ [Hình 2.2](#)).

Luồng chức năng tiếp theo trong *Phần mềm* là *phát hiện và xử lý quá tải hệ thống IVI* được mô tả chi tiết tại [Hình 2.4](#). Luồng này đóng vai trò then chốt trong việc duy trì hiệu suất và độ ổn định của hệ thống IVI khi chạy thực tế. Khi hệ thống IVI bị quá tải, *Phần mềm* sẽ thực hiện xử lý theo luồng chức năng như sau:

1. **Phát hiện quá tải:** tại phân hệ Windows App, sau khi tiếp nhận dữ liệu tài nguyên, phân hệ này sẽ sử dụng thuật toán đánh giá quá tải để phân tích tình trạng hiện tại của hệ thống IVI. Chúng tôi sử dụng mô hình kết hợp giữa thuật toán định ngưỡng (Threshold) và kiểm tra liên tiếp (Consecutive Count), nhằm đảm bảo phát hiện quá

tài không chỉ dựa vào dao động tức thời mà còn tính đến sự ổn định của trạng thái tài cao.

- Xử lý quá tải:** Khi xác định hệ thống IVI đang bị quá tải, Windows App lựa chọn các tiến trình tiêu thụ nhiều tài nguyên nhưng không quá quan trọng và gửi lệnh kết thúc tiến trình đến phân hệ Linux App. Đồng thời Windows App tiến hành lưu trữ dữ liệu tài nguyên quá tải với mục đích giúp nhà phát triển hay người vận hành có thể tìm hiểu được nguyên nhân gây ra quá tải và đưa ra biện pháp để hợp lý để khắc phục nếu cần thiết.
 - Kết thúc tiến trình và phát cảnh báo:** Linux App tiếp nhận yêu cầu, tiến hành can thiệp vào hệ thống IVI để tiến hành kết thúc tiến trình tương ứng sau đó phát cảnh báo thông qua loa trên hệ thống IVI.



Hình 2.5 Stress test hệ thống IVI (trích xuất và mở rộng từ Hình 2.2).

Luồng chức năng cuối cùng là *Stress test hệ thống IVI*. Luồng này hỗ trợ kiểm thử hệ thống IVI trong giai đoạn phát triển, giúp kiểm tra khả năng chịu tải và phát hiện lỗi tiềm ẩn trong điều kiện hoạt động khắc nghiệt, được mô tả chi tiết tại [Hình 2.5](#). Quy trình thực hiện diễn ra như sau:

- Khởi tạo Stress test:** Người dùng thao tác trên Windows App để gửi lệnh khởi động Stress test đến Linux App. Tại đây, người dùng có thể tùy chỉnh cấu hình mức tài nguyên tiêu thụ trên hệ thống như điều chỉnh mức tiêu thụ CPU, bộ nhớ (công thêm vào mức tiêu thụ hiện tại của hệ thống IVI) và thời gian thực hiện Stress test.
 - Tạo tải giả lập:** Sau khi nhận yêu cầu từ Windows App, Linux App tạo các tiến trình tiêu tốn tài nguyên giả lập (CPU, bộ nhớ), giúp mô phỏng điều kiện tải nặng.
 - Kết thúc Stress test:** Khi người dùng muốn kết thúc Stress test trước thời điểm thiết lập, có thể kích hoạt dừng tại Windows App, sau đó Windows App sẽ gửi lệnh kết thúc đến Linux App để dừng toàn bộ các tiến trình giả lập.

Ba luồng chức năng chính được mô tả ở trên minh họa cách *Phần mềm* vận hành một cách thống nhất và linh hoạt. Với sự phân tách rõ ràng vai trò giữa hai phân hệ, *Phần mềm* vừa có thể đáp ứng được nhu cầu giám sát chuyên sâu trong giai đoạn phát triển, vừa có thể đảm bảo tính ổn định cho hệ thống IVI trong vận hành thực tế. Thiết kế này đồng thời giúp cân bằng giữa hiệu năng và chi phí tài nguyên khi triển khai và vận hành *Phần mềm* - một yếu tố then chốt trong các hệ thống IVI có cấu hình phần cứng hạn chế.

Sau khi đã phân tích nguyên lý hoạt động tổng thể và từng luồng chức năng cụ thể của *Phần mềm*, có thể thấy rõ ràng rằng sự phối hợp chặt chẽ giữa hai phân hệ Linux App và Windows App chính là yếu tố then chốt để đảm bảo tín hiệu quả và ổn định của toàn hệ thống. Tuy nhiên, để có thể hiện thực hóa được toàn bộ luồng chức năng như đã mô tả, việc xây dựng *Phần mềm* cần phải dựa trên những nền tảng lý thuyết vững chắc về hệ điều hành, giao thức mạng và công nghệ phát triển phần mềm. Chính vì vậy, trong mục sau đây, chúng tôi sẽ trình bày các nền tảng lý thuyết liên quan mà *Phần mềm* đã dựa vào để triển khai.

2.5. Nền tảng lý thuyết

Để triển khai thành công phần mềm giám sát và điều phối tải hệ thống IVI, chúng tôi cần dựa trên một số nền tảng lý thuyết cốt lõi. Các nền tảng này không chỉ giúp phần mềm hoạt động đúng chức năng mà còn đảm bảo tính hiệu quả và ổn định khi vận hành trên thiết bị có tài nguyên hạn chế. Ba trụ cột lý thuyết chính được chúng tôi sử dụng gồm:

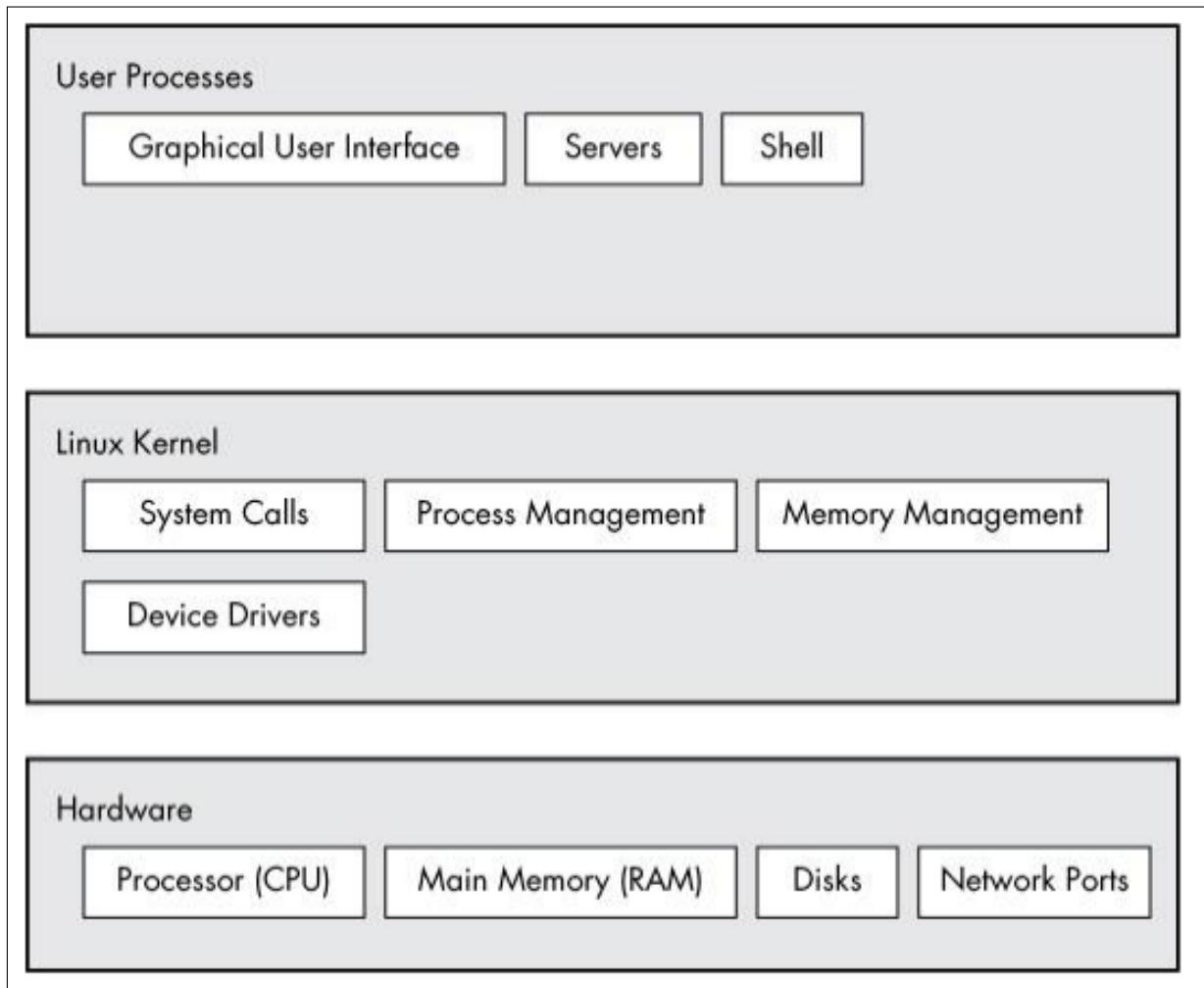
1. Kiến trúc hệ điều hành Linux [1].
2. Các giao thức mạng truyền thông [2].
3. Công nghệ phát triển phần mềm dựa trên Qt Framework [6].

2.5.1. Kiến trúc hệ điều hành Linux

Hệ thống IVI chủ yếu hoạt động trên hệ điều hành Linux, với cấu trúc tài nguyên bao gồm CPU đa nhân, các bộ nhớ lưu trữ dữ liệu, module mạng, disk I/O... Để triển khai phân hệ Linux App thu thập dữ liệu tài nguyên đã được chọn lọc thì trước tiên chúng ta cần hiểu về cấu trúc của hệ điều hành Linux.

Hệ điều hành Linux là hệ điều hành mã nguồn mở, có khả năng tùy biến cao, một trong những điểm mạnh nhất của nó là khả năng quản lý tài nguyên hệ thống hiệu quả. Hình 2.6 mô tả cấu trúc cơ bản của hệ điều hành Linux. Linux chia thành hai không gian

bao gồm Kernel space và User space, với Kernel space đây là nơi chứa kernel dùng để quản lý tài nguyên phần cứng, tiến trình, bộ nhớ, hệ thống file... Với User space, đây là nơi các ứng dụng thông thường chạy, tương tác với kernel thông qua System Calls. Để chi tiết hơn, chúng tôi sẽ trình bày chi tiết từng phần, phân tích các yếu tố và nền tảng để ứng dụng vào *Phần mềm*.



Hình 2.6 General Linux system organization [1].

Linux Kernel là phần cốt lõi của hệ điều hành, có thể truy cập và xử lý mọi tiến trình và tài nguyên trong hệ điều hành, Linux Kernel có bốn đảm nhiệm chính bao gồm:

- Quản lý tiến trình - Process Management:** Có nhiệm vụ xác định các tiến trình nào được phép sử dụng CPU.
- Quản lý bộ nhớ - Memory Management:** Theo dõi toàn bộ bộ nhớ, bao gồm các phần như bộ nhớ được phân bổ (allocated memory) cho các tiến trình, bộ nhớ chia sẻ (shared memory) giữa các tiến trình và bộ nhớ trống (free memory). Chúng tôi quyết định thu thập các loại bộ nhớ cho *Phần mềm* dựa trên đặc điểm này.

3. **Device Drivers:** Hoạt động như một giao diện giúp các tiến trình có thể tương tác với phần cứng. Trong đề tài của này, Linux App sẽ thực hiện lệnh để điều khiển phần cứng như Speaker, thu thập các thông số từ CPU và MEM dựa trên đặc điểm này của Linux Kernel để tương tác với hệ thống IVI.
4. **System Calls:** Các tiến trình tại User space dùng System Calls để giao tiếp với Kernel. Đây là cách mà Linux App có thể triển khai các lệnh để gửi đến Kernel và thực thi lệnh đó.

User Space là nơi mà các tiến trình ở mức ứng dụng được thực thi, đặc điểm của các tiến trình này là bị hạn chế truy cập vào các quyền và tài nguyên trong hệ điều hành, đây cũng là nơi mà Linux App hoạt động. Tại đây, chúng tôi triển khai Linux App dựa trên **Shell**, đặc điểm là sử dụng các commands shell và các lệnh này sẽ được triển khai tại Kernel thông qua **System Calls**, đối với các lệnh *thu thập dữ liệu tải* thì sẽ được tiến hành tại **Process Management** và **Memory Management**, và thông qua **Device Drivers**, các thông số này sẽ có thể "chạm" đến phần cứng CPU và bộ nhớ chính (Main Memory). Còn đối với các lệnh như *kết thúc tiến trình* hay thực hiện *Stress test* thì các commands shell sẽ "đi" từ **Shell** tại User Space đến Kernel với **System Calls** và đến hệ thống phần cứng thông qua **Device Drivers** để can thiệp vào hệ thống từ đó có thể điều chỉnh theo yêu cầu.

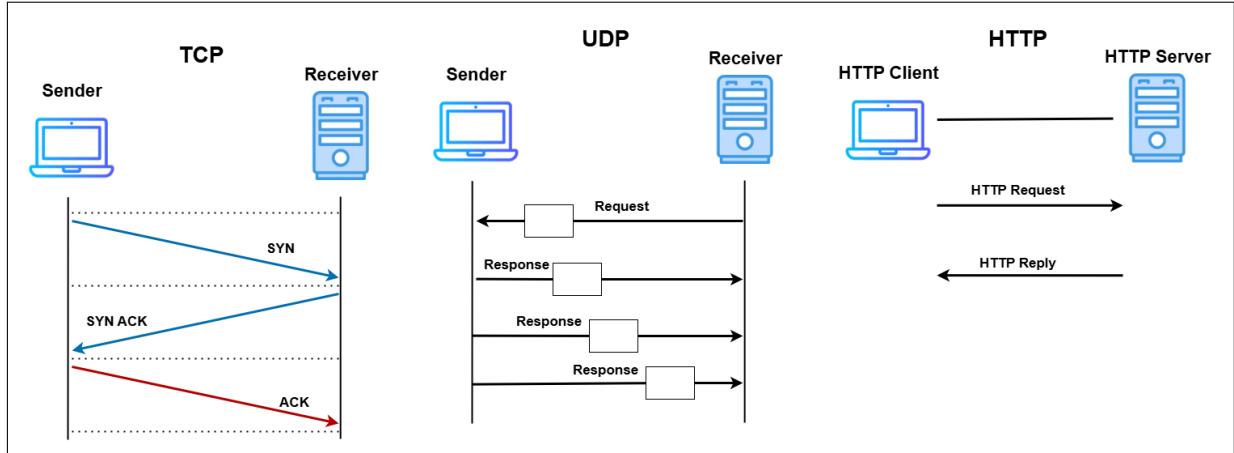
2.5.2. Các giao thức mạng truyền thông

Trong *Phần mềm*, chúng tôi có sử dụng giao thức mạng để truyền/nhận dữ liệu giữa Linux App và Windows App, cụ thể chúng tôi lựa chọn giao thức TCP/IP cho chức năng truyền và nhận dữ liệu giữa hai phân hệ: Linux App đóng vai trò như client, trong khi đó thì Windows App đóng vai trò là server. Chúng tôi lựa chọn giao thức mạng để sử dụng cho *Phần mềm* dựa trên các yếu tố bao gồm:

1. Tính bảo mật.
2. Tính đáng tin cậy khi truyền/nhận dữ liệu.
3. Tính chính xác trong dữ liệu truyền tin.

Các yếu tố này rất quan trọng đối với *Phần mềm*, vì suy cho cùng chúng ta có thể thu thập dữ liệu và điều khiển cả hệ thống IVI thông qua các dữ liệu truyền/nhận đó. Từ các yếu tố trên, chúng tôi nhận thấy rằng giao thức TCP/IP là phù hợp nhất. Để hiểu rõ hơn, chúng tôi sẽ phân tích một số giao thức mạng, các đặc trưng, ưu và nhược của chúng để từ đó có cái nhìn trực quan về nhận định này.

Vì Phần mềm được xây dựng dựa trên Qt Framework và trong Qt Framework (Qt Network) chỉ hỗ trợ các giao thức bao gồm TCP, UDP và HTTP/HTTPS [2]. Vì vậy khi triển khai chức năng giao tiếp mạng trong phần mềm thì chúng tôi chỉ cần đánh giá các giao thức có hỗ trợ trong Qt Framework và đưa ra lựa chọn phù hợp nhất với đề tài.



Hình 2.7 Cách hoạt động của các giao thức mạng.

Hình 2.7 biểu diễn cách hoạt động của các giao thức mạng được đề cập trong đề tài này, mô tả chi tiết chúng tôi sẽ trình bày ở nội dung bên dưới.

TCP (Transmission Control Protocol) là giao thức hướng kết nối, trước khi trao đổi dữ liệu, giữa client và server (Phụ lục 5) phải thiết lập kết nối thông qua tiến trình "bắt tay ba bước" (three-way handshake). TCP đảm bảo dữ liệu được truyền đi theo đúng thứ tự gửi và nếu gói tin bị mất, nó sẽ tự động yêu cầu gửi lại. Ngoài ra TCP còn khả năng gửi lại dữ liệu khi mất kết nối, dữ liệu sẽ được lưu trữ vào bộ nhớ đệm và đến khi có kết nối lại, nó gửi dữ liệu trong bộ nhớ đệm đó cho server/client. Ưu điểm của TCP là tính chính xác và tin cậy rất cao, đảm bảo dữ liệu không thất lạc hoặc bị sai thứ tự, để đảm bảo an toàn khi truyền/nhận, dữ liệu cần được mã hóa để tăng tính bảo mật, tại đây chúng ta có thể tích hợp SSL để mã hóa thông tin trước khi truyền đi. Nhược điểm của TCP là mức tiêu tốn chi phí cao hơn so với một số giao thức khác (như UDP) vì nó tiêu tốn băng thông và thời gian overhead để đảm bảo tính toàn vẹn dữ liệu và sắp xếp thứ tự gói tin.

UDP (User Datagram Protocol) là giao thức không hướng kết nối, mỗi gói tin được gửi động lập và không có đảm bảo về thứ tự hay xác nhận đã nhận. Khi sử dụng giao thức này thì không cần thiết lập kết nối trước, dữ liệu được gửi (từ địa chỉ IP cổng này đến địa chỉ IP cổng khác) mà không cần giai đoạn "bắt tay" (chúng tôi tận dụng yếu tố này để thực hiện kết nối tự động từ Linux App đến Windows App một cách tự động mỗi khi Linux App khởi chạy, cụ thể thì Linux App sẽ quét địa chỉ IP của Windows App, khi

Windows App bắt được tín hiệu thì phân hệ này sẽ gửi địa chỉ IP của nó đến Linux App, từ đó Linux App sẽ sử dụng địa chỉ IP nhận được để tiến hành khởi tạo kết nối TCP/IP, đảm bảo được yêu cầu kỹ thuật trong truyền/nhận dữ liệu của *Phần mềm*). Ưu điểm của UDP là có tốc độ truyền tải cao, ít overhead, lý do là vì không cần thiết lập đường truyền trước, không có cơ chế "xử lý lại" khi gói tin bị hỏng hay mất, đây cũng chính là yếu điểm lớn của UDP khi xét theo góc nhìn của đề tài này. Ngoài ra thì UDP không đảm bảo tính chính xác hay thứ tự gói tin, cơ chế bảo mật cũng rất thấp. Từ nhận định đó, chúng tôi thấy rằng UDP không phải là lựa chọn phù hợp để triển khai cho *Phần mềm*.

HTTP (Hypertext Transfer Protocol) là giao thức tầng ứng dụng dùng cho truyền tải tài liệu siêu văn bản, tập tin và dữ liệu web, trong khi đó HTTPS là bản mở rộng của HTTP thêm lớp mã hóa SSL/TSL để tăng bảo mật. Mặc dù tính bảo mật của HTTPS rất cao nhưng tính chất của giao thức này không phù hợp với nguyên lý hoạt động của *Phần mềm*. Cụ thể, HTTP chỉ hoạt động theo nguyên lý *request* được gửi từ client và server chỉ có nhiệm vụ *reponse*, nó không cho phép server chủ động gửi dữ liệu nếu client chưa yêu cầu, điều này khiến cho việc triển khai các chức năng như *Phát hiện và cân bằng tải hệ thống IVI* hay *Stress test* không thể triển khai được khi dùng giao thức này, vì với các chức năng đó thì Windows App là bên chủ động gửi dữ liệu đi mà không chờ *request* từ phía Linux App.

Từ những phân tích ở trên, chúng tôi có thể dễ dàng đưa ra kết luận rằng việc triển khai giao thức TCP/IP cho *Phần mềm* cho lựa chọn phù hợp và hiệu quả nhất khi mà giao thức này đảm bảo:

1. Tính toàn vẹn và thứ tự của dữ liệu, dữ liệu truyền đi không bị mất mát hay trùng lặp, từ đó đáp ứng được yêu cầu tính chính xác trong dữ liệu truyền tin.
2. Có thể xác minh tính thành công của quá trình truyền nhận, giúp cho quá trình giao tiếp giữa hai phân hệ trở nên đáng tin cậy và an toàn hơn.
3. Bản thân TCP/IP có độ bảo mật không quá cao, nhưng có thể khắc phục được vấn đề này bằng cách tích hợp SSL để mã hóa dữ liệu trước khi truyền đi, từ đó đáp ứng được yêu cầu bảo mật dữ liệu của *Phần mềm*.

2.5.3. Công nghệ phát triển phần mềm dựa trên Qt Framework

Để xây dựng giao diện người dùng và triển khai các chức năng giao tiếp mạng trong *Phần mềm*, chúng tôi lựa chọn sử dụng **Qt Framework** [6] - một bộ công cụ phát triển ứng dụng đa nền tảng, hỗ trợ cả lập trình hướng đối tượng với C++ và ngôn ngữ mô tả giao diện QML. Qt cung cấp hệ sinh thái phong phú bao gồm các module cho giao diện

đồ họa, truyền thông mạng, cơ sở dữ liệu và nhiều thành phần chức năng khác.

Việc sử dụng Qt Framework trong đề tài mang lại nhiều lợi ích, nổi bật gồm:

1. Hỗ trợ giao diện đồ họa trực quan.
2. Giao diện mạng ổn định và đơn giản hóa việc xử lý các gói tin truyền/nhận.
3. Hệ thống signal-slot giúp tách biệt logic xử lý và giao diện.
4. Có thể mở rộng và tái sử dụng mã nguồn tốt.
5. Qt Frameworks hỗ trợ cho nhiều hệ điều này, đồng thời cũng hỗ trợ rất tốt trong môi trường đa nền tảng.

Trong phạm vi đề tài, chúng tôi sử dụng các module chính sau của Qt Framework bao gồm: Qt Core, Qt Quick, Qt Network, Qt Charts. Để có cái nhìn tổng quan và hiểu về các module này, chúng tôi sẽ trình bày chúng là gì và cách mà chúng tôi ứng dụng chúng ra sao trong đề tài này.

Qt Core là một module cốt lõi trong Qt, cung cấp các thành phần nền tảng để phát triển ứng dụng như hệ thống signal-slot, xử lý luồng (multithreading), quản lý thời gian, chuỗi ký tự (QString), và các kiểu dữ liệu cơ bản (QVariant, QList, QMap,...). Trong đề tài này, Qt Core được sử dụng để:

1. Xây dựng các phần logic trong *Phần mềm* cho cả phân hệ Linux App và Windows App.
2. Thiết lập các bộ định thời (QTimer) để cập nhật dữ liệu theo chu kỳ.
3. Khởi tạo các luồng thực thi (threads) để đáp ứng các yêu cầu của *Phần mềm*.
4. Kết nối các thành phần giao diện với xử lý logic thông qua cơ chế signal-slot, giúp *Phần mềm* có tính phản ứng cao mà không cần ràng buộc trực tiếp giữa UI và logic.

Qt Quick và QML được sử dụng để xây dựng giao diện người dùng hiện đại, sinh động và dễ tương tác. Qt Quick cho phép thiết kế các thành phần giao diện như biểu đồ, bảng dữ liệu, nút nhấn, thanh tiến trình (progress bar),... một cách trực quan và dễ dàng mở rộng. Chúng tôi sử dụng Qt Quick để:

1. Xây dựng giao diện hiển thị trên phân hệ Windows App.
2. Cho phép người dùng tương tác với *Phần mềm*.
3. Dễ dàng chỉnh sửa giao diện người dùng khi cần thiết.

Qt Network là module xử lý kết nối mạng trong Qt, hỗ trợ giao thức TCP/IP, UDP, SSL,... Qt Network cung cấp các lớp như `QTcpSocket`, `QTcpServer`, `QNetworkAccessManager` để thiết lập và duy trì kết nối mạng. Trong đề tài này, Qt Network đóng vai trò trung tâm trong việc:

1. Thiết lập kết nối TCP giữa Windows App và Linux App.
2. Truyền và nhận dữ liệu tài nguyên hệ thống IVI theo chuẩn định dạng.
3. Truyền lệnh điều khiển từ Windows App tới Linux App (kết thúc tiến trình, bắt đầu/dừng Stress test).

Qt Charts là module chuyên dùng để trực quan hóa dữ liệu bằng biểu đồ. Nó hỗ trợ nhiều loại biểu đồ như: line chart, bar chart, pie chart,... với khả năng cập nhật dữ liệu động. Chúng tôi sử dụng Qt Charts để:

1. Hiển thị biểu đồ đường cho CPU và bộ nhớ theo thời gian thực.
2. Giúp người dùng dễ dàng quan sát xu hướng tải hệ thống một cách trực quan.

Ngoài những module trên, chúng tôi còn sử dụng kiến trúc **MVVM (Model-View-ViewModel)** để tăng khả năng mở rộng, bảo trì và tái sử dụng mã nguồn. Chúng tôi xây dựng phân hệ Windows App theo mô hình MVVM:

1. Model: xử lý dữ liệu và logic nghiệp vụ (bao gồm kết nối mạng, đánh giá quá tải, xử lý dữ liệu tiến trình).
2. View: giao diện người dùng được xây dựng bằng Qt Quick/QML.
3. ViewModel: đóng vai trò trung gian, nhận dữ liệu từ Model và truyền về View thông qua signal-slot.

Mô hình này giúp phần mềm tách biệt rõ ràng giữa logic và giao diện, đồng thời dễ dàng nâng cấp giao diện hoặc logic xử lý mà không ảnh hưởng đến các phần còn lại.

Tổng thể, việc sử dụng các module trong Qt Framework không chỉ giúp tăng tốc độ phát triển mà còn đảm bảo phần mềm có kiến trúc rõ ràng, dễ bảo trì và có khả năng mở rộng cao trong tương lai. Một điểm đặc biệt quan trọng là Qt hỗ trợ phát triển phần mềm đa nền tảng một cách hiệu quả, cho phép xây dựng và triển khai ứng dụng trên nhiều hệ điều hành khác nhau như Windows, Linux, và các hệ điều hành nhúng.

Trong phạm vi đề tài này, Qt cho phép chúng tôi phát triển đồng thời cả hai phân hệ: Windows App chạy trên hệ điều hành Windows (dành cho người dùng dùng để vận hành,

theo dõi hệ thống IVI) và Linux App chạy trên hệ điều hành Linux (mô phỏng hệ thống IVI thực tế).

Nhờ khả năng đa nền tảng của Qt, chúng tôi có thể sử dụng cùng một bộ công cụ, ngôn ngữ và kiến trúc phần mềm để xây dựng hai phân hệ trên hai hệ điều hành khác nhau, giúp giảm thời gian phát triển, đồng bộ hóa mã nguồn và tăng tính linh hoạt trong việc bảo trì, nâng cấp về sau. Điều này đặc biệt phù hợp với đặc thù của hệ thống IVI, nơi mà các thành phần phần mềm cần tương thích và hoạt động ổn định trên nhiều nền tảng phần cứng và hệ điều hành khác nhau.

2.6. Kết luận chương

Trong chương này, chúng tôi đã trình bày tổng quan kiến trúc phần mềm bao gồm hai phân hệ chính là *Linux App* và *Windows App*, đồng thời làm rõ các yêu cầu chức năng, cơ chế hoạt động và mối quan hệ phối hợp giữa hai phân hệ thông qua giao thức TCP/IP. Ba luồng chức năng quan trọng nhất – thu thập dữ liệu, phát hiện và xử lý quá tải, cùng với Stress test – đã được mô tả cụ thể nhằm minh họa cách mà *Phần mềm* vận hành thống nhất để hỗ trợ giám sát và tối ưu tài nguyên trên hệ thống IVI.

Bên cạnh đó, chúng tôi cũng đã phân tích các nền tảng lý thuyết quan trọng làm cơ sở cho việc triển khai *Phần mềm*, bao gồm kiến trúc hệ điều hành Linux, các giao thức truyền thông mạng và công nghệ Qt Framework – bộ công cụ chính giúp phát triển cả giao diện người dùng lẫn xử lý logic trên đa nền tảng. Đặc biệt, khả năng phát triển đa hệ điều hành của Qt giúp đảm bảo tính linh hoạt, dễ mở rộng và thuận tiện bảo trì, phù hợp với đặc thù của hệ thống IVI trong thực tế.

Những nội dung trong chương này tạo nền móng vững chắc để bước sang các chương tiếp theo, nơi chúng tôi sẽ trình bày chi tiết về quá trình hiện thực hóa *Phần mềm* thông qua việc thiết kế giao diện, triển khai chức năng trên từng phân hệ chính cho đến kiểm thử và đánh giá kết quả thực tế.

CHƯƠNG 3: THIẾT KẾ VÀ TRIỂN KHAI PHÂN HỆ WINDOWS APP

Chương 3 trình bày chi tiết các bước thiết kế và triển khai phân hệ Windows App với các mô hình trực quan như sơ đồ trường hợp sử dụng (Use Case Diagram), sơ đồ lớp (Class Diagram). Việc xây dựng các mô hình thiết kế nhằm xác định rõ các chức năng chính, mối quan hệ giữa các thành phần trong hệ thống cũng như hỗ trợ việc triển khai phần mềm một cách logic và hiệu quả.

Với đề tài này, phân hệ Windows App được triển khai bằng công cụ Qt Creator - một môi trường phát triển tích hợp đa nền tảng (IDE) mạnh mẽ, tạo ứng dụng chạy trên nhiều nền tảng khác nhau. Ngoài ra nó còn cho phép viết mã bằng C++, QML,...

3.1. Sơ đồ tổng quan của phân hệ Windows App

3.1.1. Sơ đồ trường hợp sử dụng (Use Case Diagram)

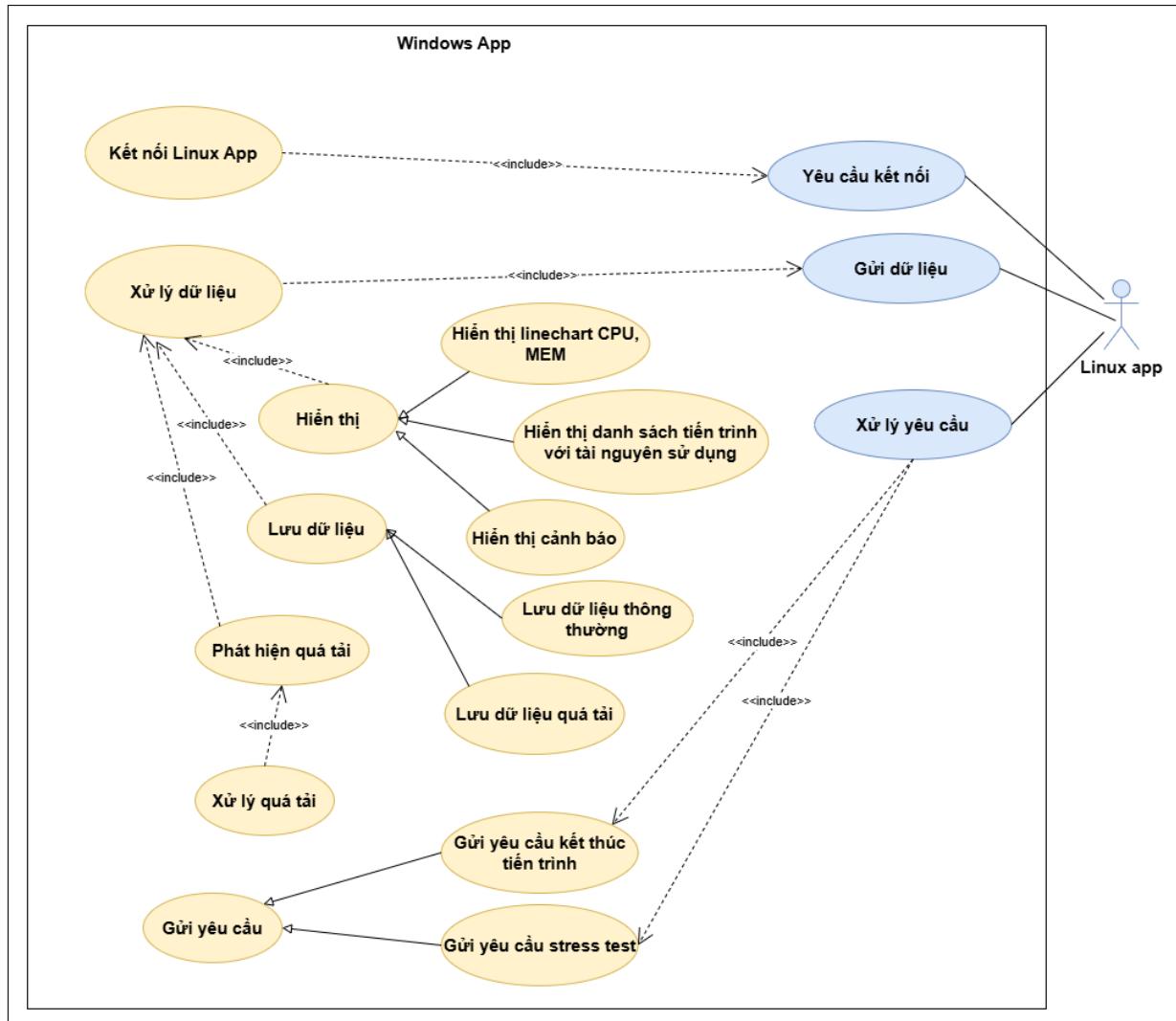
Trong Hình 3.1 là sơ đồ Use Case của ứng dụng Windows App. Được thiết kế nhằm mô hình hóa các yêu cầu chức năng một cách trực quan, giúp xác định được các chức năng chính mà nó cung cấp cũng như cách mà các tác nhân (actors) tương tác với nó. Nhờ đó mà nhà phát triển và người dùng có thể hình dung được những gì mà ứng dụng đó sẽ thực hiện.

Theo như sơ đồ đã thiết kế (Hình 3.1), có hai tác nhân chính đó là Windows App và Linux App. Trong đó, Windows App đóng vai trò là ứng dụng giao diện người dùng với các chức năng cơ bản như hiển thị giao diện giám sát trực quan, lưu trữ dữ liệu cần thiết, phát hiện và xử lý tình trạng quá tải của hệ thống IVI. Linux App chịu trách nhiệm thu thập các giá trị tải, tài nguyên bị chiếm dụng rồi gửi đến Windows App. Ngoài ra, nó còn có nhiệm vụ thực thi các yêu cầu nhận được từ Windows App như kết thúc một tiến trình, chạy Stress test.

3.1.2. Sơ đồ lớp (Class Diagram)

Dựa trên những chức năng đã xác định ở sơ đồ Use Case trên (Hình 3.1) mà chúng tôi đã thiết kế một sơ đồ lớp (Class Diagram) hoàn chỉnh theo mô hình hướng đối tượng (OOP). Sơ đồ này được sử dụng để biểu diễn các lớp (Class), thuộc tính, phương thức của từng lớp, cũng như mối quan hệ giữa chúng.

Sơ đồ lớp của hệ thống (Hình 3.2) được chia thành bốn nhóm chức năng chính: mô hình dữ liệu, xử lý logic, giao tiếp mạng và hiển thị. Mỗi nhóm đóng vai trò cụ thể trong



Hình 3.1 Sơ đồ Use Case của Windows App.

việc đảm bảo luồng xử lý của hệ thống được phân tách rõ ràng, module hóa và dễ mở rộng.

Nhóm lớp mô hình dữ liệu (Model - màu cam), chứa các cấu trúc để lưu trữ các giá trị về tải hệ thống và thông tin các tiến trình. Lớp SystemStats đóng vai trò tổng hợp, lưu trữ trạng thái CPU và bộ nhớ (RAM) tại một thời điểm nhất định thông qua hai thuộc tính chính là CPUStats và MEMStats. Cấu trúc CPU được chia thành hai phần: CpuCore – lưu thông tin chi tiết từng lõi như mức sử dụng, tần số, nhiệt độ; và CpuGeneral – mức trung bình của toàn bộ hệ thống. Lớp SystemCPU là thành phần trung gian, kết hợp cả CpuCore và CpuGeneral. Trong khi đó, lớp SystemMEM chứa các thông số liên quan đến dung lượng RAM, bộ nhớ swap cùng với mức sử dụng hiện tại. Lớp ProcessInfo biểu diễn thông tin chi tiết của từng tiến trình đang hoạt động, cùng với tài nguyên mà chúng đang chiếm dụng. Đặc biệt, lớp OverloadConfig cung cấp các ngưỡng và tham số để cấu hình việc phát hiện tình trạng quá tải như: trong số CPU, RAM, nhiệt độ, các

ngưỡng cảnh báo khác,... Nhóm lớp này đóng vai trò lưu trữ và cung cấp dữ liệu khi cần.

Nhóm xử lý logic (Processing Logic - màu xanh dương) bao gồm các lớp chịu trách nhiệm phân tích dữ liệu, phát hiện quá tải và xử lý quá tải. Lớp SystemMonitor đóng vai trò điều phối chính, quản lý vòng đời giám sát hệ thống, xử lý tín hiệu và phối hợp các thành phần phụ như DataProcessor, OverloadDetector, ProcessManager và IviSocketServer. Lớp DataProcessor thực hiện phân tích dữ liệu thô nhận được từ Linux App, chuyển đổi thành các đối tượng SystemStats và danh sách ProcessInfo. Lớp OverloadDetector sử dụng cấu hình từ OverloadConfig để đánh giá trạng thái hệ thống, phát tín hiệu cảnh báo khi phát hiện tình trạng hệ thống đang vượt quá ngưỡng cho phép. Trong khi đó, ProcessManager sẽ có nhiệm vụ cân bằng tải khi hệ thống quá tải bằng cách gửi yêu cầu kết thúc tiến trình đang chiếm nhiều tài nguyên nhưng không quá quan trọng. Nhóm lớp này thực hiện các tác vụ xử lý cốt lõi, đảm bảo hệ thống vận hành ổn định và phản ứng kịp thời khi quá tải xảy ra.

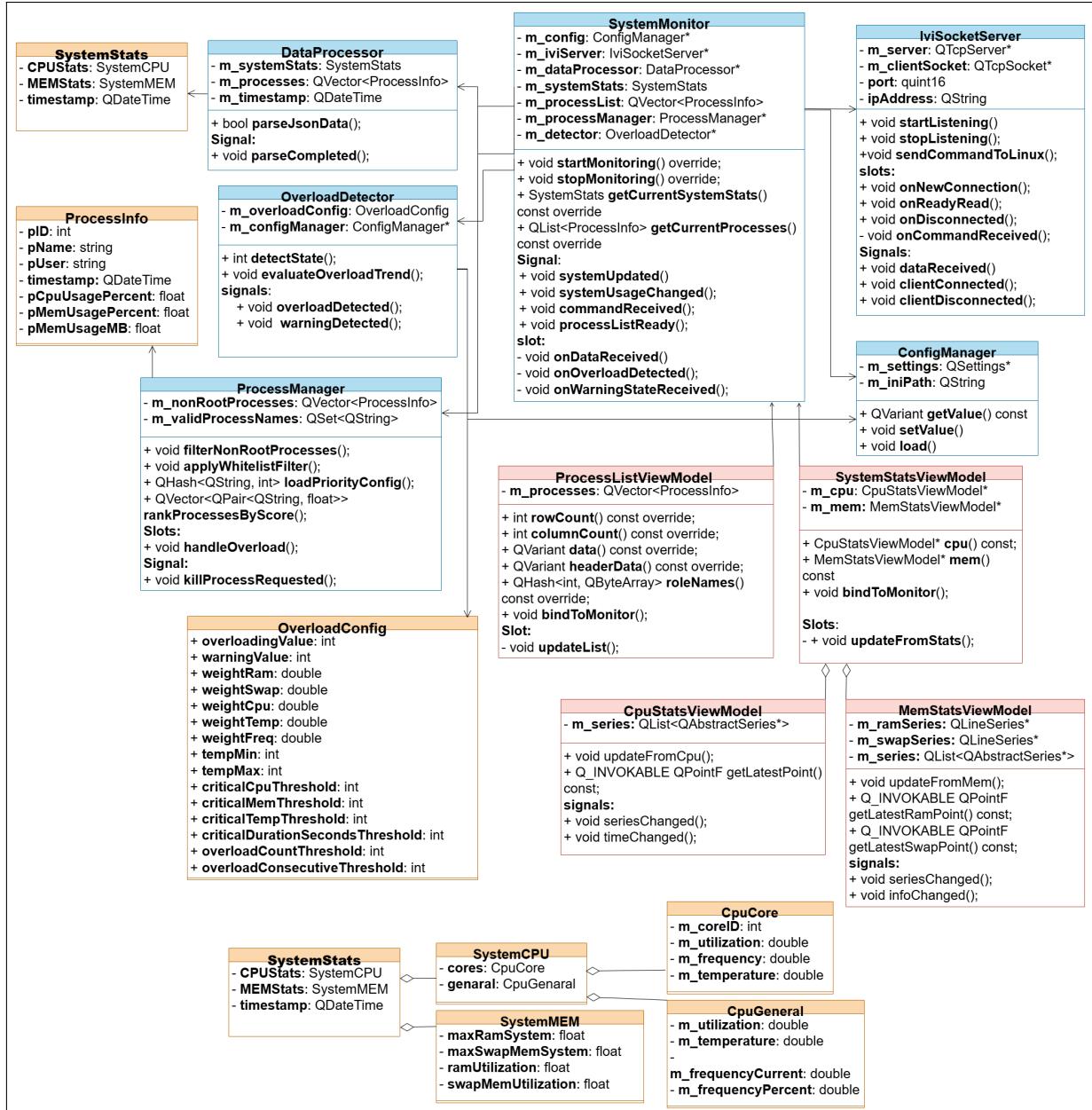
Nhóm giao tiếp mạng (Communication) được đại diện bởi lớp IviSocketServer. Đây là thành phần phụ trách kết nối và truyền nhận dữ liệu giữa hai phân hệ Windows App và Linux App. Lớp này sử dụng giao thức socket TCP để thiết lập kết nối, nhận yêu cầu và gửi phản hồi. Các phương thức như onNewConnection() và onDataReceived() cùng các tín hiệu như dataReceived() hay clientConnected() đảm bảo hệ thống có thể giao tiếp một cách không đồng bộ, nhanh chóng và ổn định. Nhóm lớp giao tiếp giúp kết nối các thành phần của hệ thống hoạt động ở các môi trường khác nhau một cách hiệu quả.

Cuối cùng, nhóm hiển thị (ViewModel) tuân theo mô hình MVVM trong Qt, giúp tách biệt logic xử lý khỏi giao diện người dùng. Lớp SystemStatsViewModel chịu trách nhiệm cập nhật và truyền dữ liệu hệ thống tới hai lớp con là CpuStatsViewModel và MemStatsViewModel, được thiết kế để vẽ biểu đồ line chart hiển thị mức sử dụng CPU và MEM theo thời gian. Bên cạnh đó, lớp ProcessListViewModel đảm nhận việc cung cấp dữ liệu danh sách tiến trình dưới dạng bảng. Đây là nhóm lớp cho phép hiển thị thông tin thời gian thực và đảm bảo sự tương tác mượt mà giữa người dùng và hệ thống.

Tổng thể, sơ đồ lớp đã hiện thực hóa rõ ràng các chức năng phân tích trong sơ đồ Use Case trước đó, đồng thời đảm bảo tính đóng gói, tái sử dụng và dễ bảo trì trong thiết kế phần mềm. Các thành phần được tổ chức rõ ràng theo vai trò, kết hợp cơ chế signal-slot của Qt giúp hệ thống phản ứng linh hoạt với các sự kiện theo thời gian thực.

Đặc biệt, phân hệ Windows App được thiết kế theo kiến trúc MVVM (Model – View – ViewModel) – một mô hình phổ biến trong phát triển phần mềm hiện đại với Qt. Trong kiến trúc này, Model giữ vai trò lưu trữ và quản lý dữ liệu (như SystemStats,

ProcessInfo, OverloadConfig), ViewModel là lớp trung gian có nhiệm vụ xử lý logic hiển thị và cung cấp dữ liệu cho giao diện (như CpuStatsViewModel, MemStatsViewModel, ProcessListViewModel), còn View được hiện thực bằng QML và chỉ chịu trách nhiệm hiển thị.

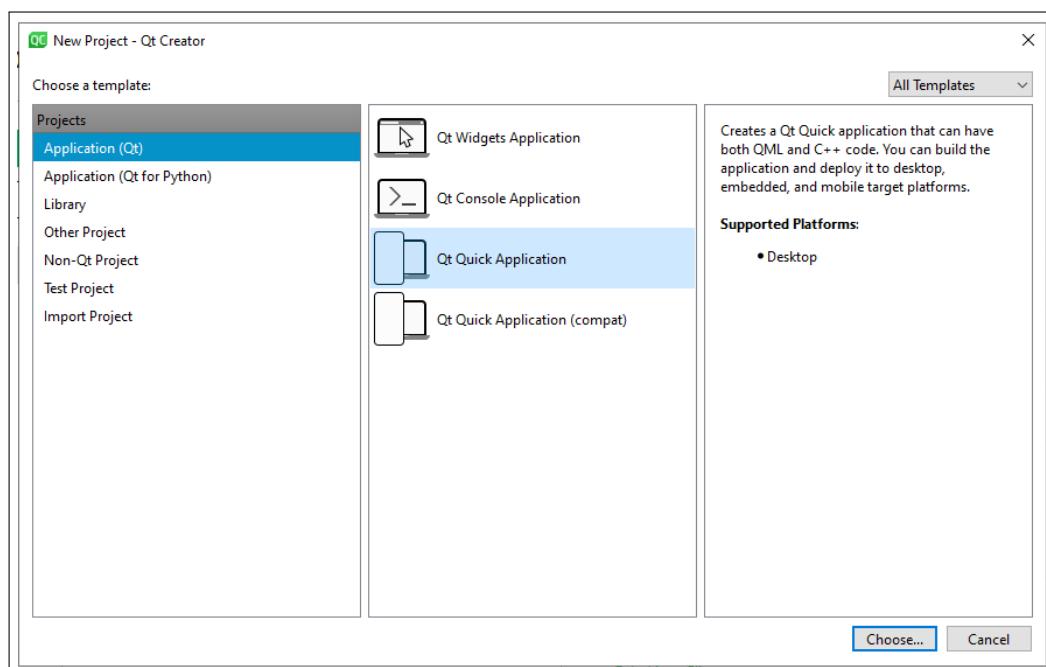


Hình 3.2 Sơ đồ lớp của Windows App.

3.2. Triển khai phần mềm Windows App

3.2.1. Kiến trúc mã nguồn sử dụng QML cho UI và C++ cho backend

Để tạo mới một dự án⁴ với ứng dụng Qt Creator, từ giao diện chính của phần mềm, chọn File -> New Project hoặc nhấn chọn trực tiếp vào nút Create Project. Hộp thoại tạo dự án mới sẽ hiện ra (như Hình 3.3) và cung cấp nhiều loại dự án khác nhau. Trong danh sách bên trái, chọn mục Application (Qt), sau đó chọn mẫu Qt Quick Application - Empty (C++ / QML). Mẫu này cho phép tạo dự án Qt Quick sử dụng cả C++ (backend) và QML (frontend), phù hợp với kiến trúc MVVM. Nhấn Choose để tiếp tục. Tiếp theo sẽ đặt tên cho dự án. Sau đó nhấn Next ở các bước tiếp theo và cuối cùng nhấn Finish để hoàn tất tạo dự án.

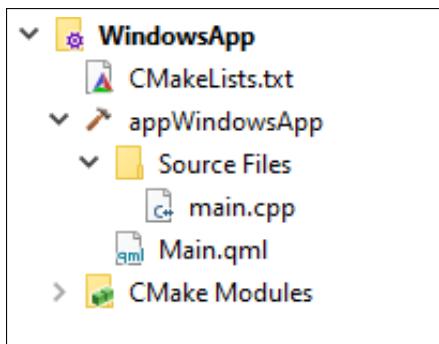


Hình 3.3 Hộp thoại tạo dự án mới với Qt Creator.

Sau khi hoàn tất các bước trên, Qt Creator sẽ sinh ra cấu trúc dự án mặc định (như Hình 3.4) bao gồm:

1. main.cpp: tập tin điểm khởi đầu của chương trình, nơi đăng ký các lớp C++, khởi tạo QApplication, QQmlApplicationEngine và nạp file giao diện main.qml.
2. main.qml: tạo diện người dùng gốc của ứng dụng, nơi khởi tạo cửa sổ chính và định nghĩa các thành phần QML (bố cục, biểu đồ, bảng...).

⁴Để tránh hiểu lầm cách gọi giữa "dự án" và "đồ án", cần lưu ý rằng "dự án" được nhắc đến ở đây là một project được khởi tạo bằng công cụ phát triển phần mềm (trong báo cáo này là Qt Creator), hoàn toàn không liên quan đến "đồ án" như cách được nhắc đến trong báo cáo.



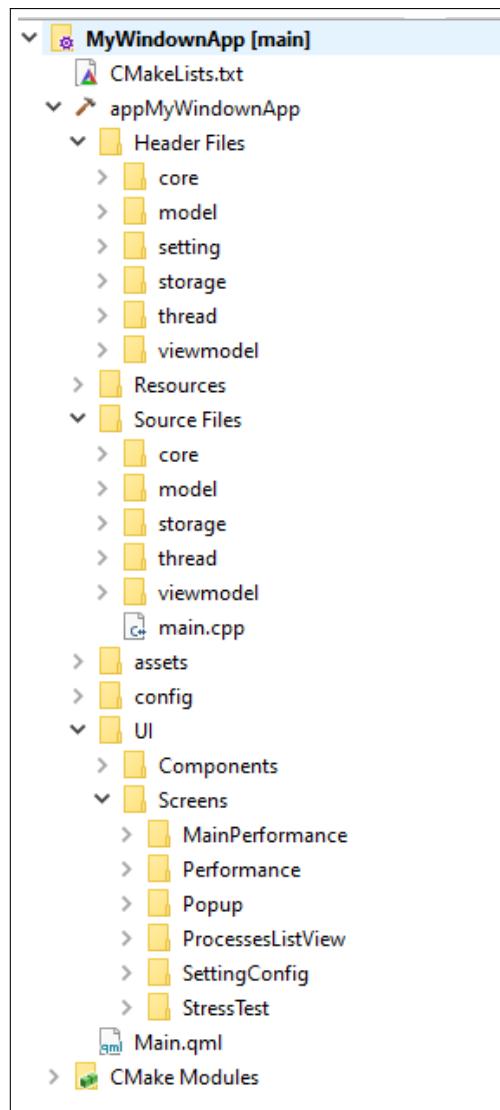
Hình 3.4 Cấu trúc mặc định sau khi khởi tạo dự án mới với Qt Creator.

3. CMakeLists.txt: tập tin cấu hình hệ thống build, quản lý các thư viện liên kết (ví dụ: Qt6::Quick, Qt6::Charts, Qt6::Network...)

Dựa vào sơ đồ lớp đã thiết kế trước đó, xác định rõ vai trò của từng thành phần và tiến hành xây dựng các thư mục mã nguồn tương ứng của dự án (Hình 3.5). Việc tổ chức thư mục như thế này đảm bảo khả năng mở rộng dễ dàng, dễ bảo trì hay sửa lỗi và duy trì sự tách biệt rõ ràng giữa logic xử lý và giao diện người dùng.

Về phần xử lý logic, các thư mục này chứa toàn bộ mã nguồn C++ và được chia thành hai phần: Header Files (tệp .h) và Source Files (tệp .cpp). Mỗi phần lại được chia thành các module tương ứng với từng lớp trong sơ đồ lớp. Trong đó, thư mục core chứa các lớp xử lý chính (core logic) như SystemMonitor, DataProcessor, OverloadDetector, ProcessManager. Thư mục model định nghĩa các lớp mô hình dữ liệu như SystemStats, SystemCPU, CpuCore, SystemMEM, ProcessInfo. Thư mục setting quản lý cấu hình hệ thống, tương ứng với lớp ConfigManager, nơi đọc/ghi thông tin từ file. Thư mục storage nơi lưu trữ dữ liệu cần thiết. Thư mục thread quản lý các luồng chạy thực thi độc lập. Cuối cùng, thư mục.viewmodel là lớp kết nối giữa dữ liệu và QML, như SystemStatsViewModel, CpuStatsViewModel, ProcessListViewModel, cho phép cập nhật và phản ánh dữ liệu thời gian thực từ tầng xử lý đến giao diện người dùng.

Về phía giao diện người dùng, thư mục UI chưa toàn bộ giao diện được viết bằng QML. Thư mục này chia thành các module như thư mục Components chưa các thành phần giao diện sử dụng chung. Trong khi đó, thư mục Screens chưa giao diện màn hình chính của phần mềm, chẳng hạn như MainPerformance hiển thị CPU, RAM dạng line chart. ProcessesListView hiển thị bảng thông tin các tiến trình và mức sử dụng tài nguyên của nó. Màn hình SettingConfig cho phép người dùng tùy chỉnh những cảnh báo, cấu hình hệ thống và các tùy chọn liên quan. Thư mục Popup chứa các hộp thoại cảnh báo quá tải, trong khi StressTest cung cấp giao diện gửi yêu cầu kiểm tra khả năng chịu tải của hệ thống.



Hình 3.5 Cấu trúc mặc định sau khi khởi tạo dự án mới với Qt Creator.

3.2.2. Kết nối C++ với QML

Trong mô hình MVVM được áp dụng cho cấu trúc phân hệ Windows App, lớp ViewModel đóng vai trò trung gian kết nối giữa dữ liệu từ C++(Model/logic) tới giao diện người dùng QML (View). Và để QML có thể sử dụng các thuộc tính, các phương thức ta cần phải đưa đối tượng C++ vào QML. Để thực hiện việc kết nối này, Qt cung cấp hai cơ chế chính:

1. Phương pháp thứ nhất: đăng ký kiểu lớp C++ vào QML (qmlRegisterType<T>())
2. Phương pháp thứ hai: thiết lập đối tượng có sẵn (setContextProperty()).

Phương pháp thứ nhất, qmlRegisterType<T>(), cho phép người lập trình đăng ký một lớp C++ dưới dạng kiểu dữ liệu (type), cho phép lớp đó được sử dụng như một kiểu dữ

liệu trong mã QML. Sau khi đăng ký, QML có thể khởi tạo đối tượng từ lớp này thông qua cú pháp như MyViewModel và truy cập các thuộc tính, phương thức của nó.

```
1 qmlRegisterType<ProcessListViewModel>("MyApp.ProcessListVM", 1, 0,
2                                         "ProcessListViewModel");
```

Trong đó, "MyApp.ProcessListVM" là tên module QML mà lớp sẽ được đăng ký. "1, 0" là phiên bản của module. "ProcessListViewModel" là tên kiểu được sử dụng trong QML. Sau khi đăng ký xong, lớp này được sử dụng trong QML như sau:

```
1 import MyApp.ProcessListVM 1.0
2
3 MyClass {
4     id: myObject
5     property string name: "Qt"
6 }
```

Sau khi khai báo MyClass trong QML, đối tượng myObject có thể được dùng để truy cập các thuộc tính và phương thức của đối tượng được đăng ký từ C++.

Phương pháp thứ hai, setContextProperty(), cho phép người phát triển khởi tạo và quản lý đối tượng trực tiếp từ C++, sau đó gán đối tượng này vào QML như một biến toàn cục. Nhờ đó, QML có thể truy cập và tương tác với đối tượng như với một thành phần có sẵn.

```
1 engine.rootContext()->setContextProperty("SystemStatsVM", systemStatsViewModel);
2 engine.rootContext()->setContextProperty("ProcessListVM", processListViewModel);
```

Sau đó, trong QML, các đối tượng SystemStatsVM, ProcessListVM có thể được truy cập như một biến toàn cục:

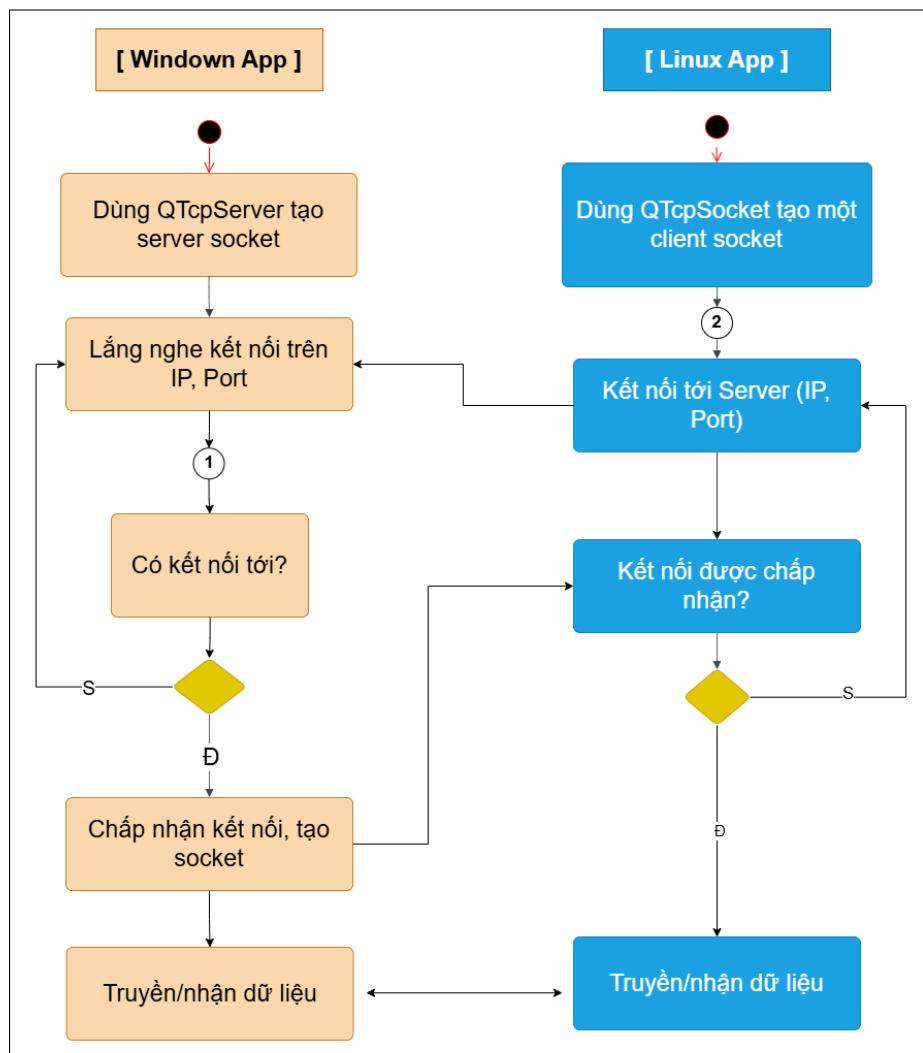
```
1 Text {
2     text: SystemStatsVM.value
3 }
```

3.2.3. Giao tiếp giữa Windows App và Linux App

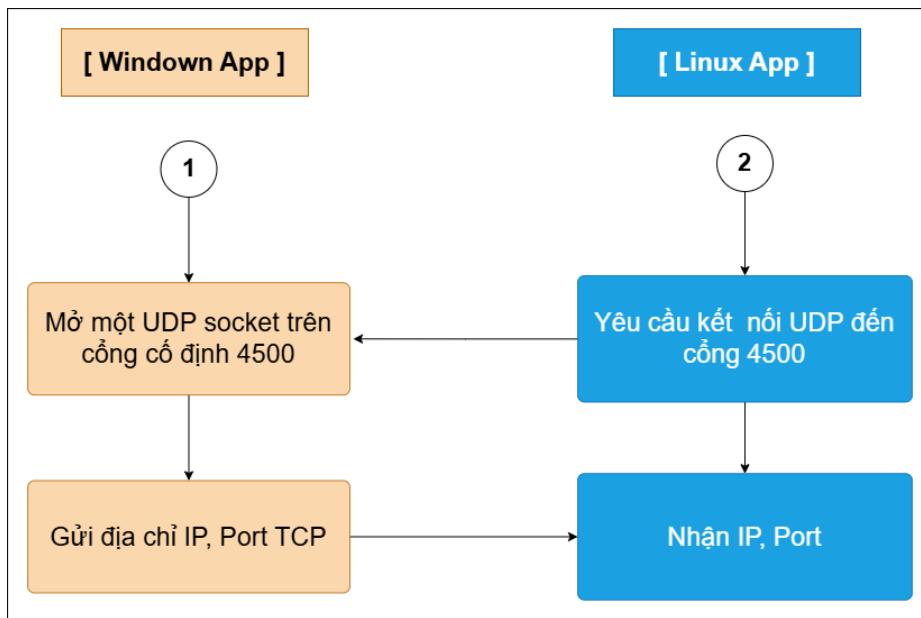
1. Thiết lập kết nối giữa Windows App và Linux App

Windows App và Linux App trao đổi dữ liệu với nhau thông qua giao thức TCP, tại Hình 3.6 đã mô tả các bước để thiết lập kết nối giữa chúng. Đầu tiên, phía Windows App sẽ phải mở một Server Socket và lắng nghe kết nối trên một địa chỉ IP và Port cụ thể. Đối với Linux App, sẽ tạo một Client Socket, yêu cầu kết nối đến đúng địa chỉ IP và Port mà Windows App cung cấp. Khi nhận được yêu cầu kết nối, phía Windows App chấp nhận kết nối đó và bắt đầu quá trình truyền/nhận dữ liệu.

Vấn đề đặt ra là làm sao phía Linux App biết được địa chỉ IP, Port. Trong phạm vi đề tài này, chúng tôi chỉ chạy trong mạng LAN nên mỗi lần kết nối mạng thì phía Windows App sẽ cho một địa chỉ IP khác nhau. Điều này khá là bất tiện khi mỗi lần kết nối đều phải cung cấp lại địa chỉ IP cho Linux App. Để giải quyết vấn đề này, trước khi thiết lập kết nối TCP, chúng tôi sẽ thiết lập một kết nối UDP trước với mục đích là cung cấp địa chỉ IP, Port cho Linux App mỗi khi có sự thay đổi. Điều này được mô tả trong Hình 3.7, phía Windows App, khi khởi động, sẽ mở một UDP Socket trên một Port cố định (trong hình là Port 4500), phía Linux App chỉ cần kết nối UDP đến đúng Port mặc định đó để có thể nhận IP và Port, rồi kết nối TCP đến đúng địa chỉ vừa nhận được.



Hình 3.6 Thiết lập kết nối TCP giữa Windows App và Linux App.



Hình 3.7 Thiết lập kết nối UDP giữa Windows App và Linux App.

2. Mô hình hoá dữ liệu được từ Linux App

Windows App và Linux App trao đổi dữ liệu dưới dạng JSON (Phụ lục 4). Tuy nhiên, do khác biệt về môi trường thực thi và cơ chế truy cập tài nguyên, dữ liệu JSON này không thể được xử lý trực tiếp, mà cần thông qua một lớp trung gian chịu trách nhiệm phân tách, chuyển đổi và ánh xạ dữ liệu phù hợp. Qt hỗ trợ sẵn các lớp xử lý chuỗi JSON như QJsonDocument, QJsonObject, QJsonArray, cho phép phân tách và truy xuất các thành phần trong chuỗi JSON trở nên thuận tiện và hiệu quả. Dựa vào đó, Windows App thiết lập một chức năng chuyên xử lý chuỗi JSON – được hiện thực thông qua lớp DataProcessor. Lớp này tiếp nhận chuỗi JSON, sau đó tiến hành phân tích cú pháp và chuyển đổi dữ liệu thành các dữ liệu tự định nghĩa (class) như SystemStats, SystemCPU, CpuCore, SystemMEM và ProcessInfo (các kiểu dữ liệu này đã trình bày ở §3.1.2).

Cụ thể, khi nhận được chuỗi JSON như minh họa trong Hình 3.8, dữ liệu bao gồm các trường: thời gian (timestamp), thông tin CPU tổng quát (GeneralCPU), danh sách lõi CPU (coresCPU), bộ nhớ (MEM) và danh sách tiến trình (ProcessesStats). Sau khi được phân tích, các trường dữ liệu này sẽ được ánh xạ vào các đối tượng tương ứng trong C++. Ví dụ như GeneralCPU ánh xạ thành một đối tượng CpuGeneral, coresCPU được chuyển thành danh sách CpuCore, MEM ánh xạ thành một đối tượng SystemMEM, ProcessesStats chuyển thành danh sách các ProcessInfo.

```

1   {
2     "timestamp": "2025-04-03 15:47:12",
3     "SystemStats" : {
4       "GeneralCPU" : {
5         "CPUUtilization" : 30,
6         "CPUTemperature" : 30,
7         "CPUFrequency" : 3000,
8         "CPUFrequencyPercent" : 30,
9       },
10      "coresCPU" : {
11        "0" : {
12          "CPUUtilization" : 30,
13          "CPUTemperature" : 30,
14          "CPUFrequency" : 3000,
15        },
16        ...
17        "7" : {
18          "CPUUtilization" : 30,
19          "CPUTemperature" : 30,
20          "CPUFrequency" : 3000,
21        },
22      },
23      "MEM" : {
24        "RAMUsage" : 12345,
25        "RAMPPercent" : 15,
26        "MaxRAM": 16000;
27        "SWAPUsage" : 0,
28        "SWAPPPercent" : 0,
29        "MaxSWAP": 4000,
30      },
31    },
32    "ProcessesStats" : {
33      "321": {
34        "PID": 321,
35        "User": duc-vu,
36        "PName": chrome,
37        "PCPUUsagePercent": 15,
38        "PMEMUsageMB": 2000,
39        "PMEMUsagePercent": 10,
40      },
41      ...
42    },
43  }

```

Hình 3.8 Cấu trúc dữ liệu dạng JSON nhận từ Linux App.

3. Cấu trúc dữ liệu truyền từ Windows App đến Linux App

Bên cạnh việc tiếp nhận và xử lý dữ liệu nhận được từ Linux App, Windows App còn có thể chủ động gửi lại các yêu cầu điều khiển như kết thúc một tiến trình, yêu cầu chạy Stress test hệ thống,... Các yêu cầu đều được đóng gói dưới dạng một chuỗi JSON, trong đó trường ”type” quy định loại hành động cần thực hiện, đi kèm các tham số cụ thể tùy theo ngữ cảnh. Cụ thể, hiện tại đang có các loại yêu cầu chính như sau:

Thứ nhất là yêu cầu chạy Stress test (Hình 3.9), cho phép người dùng kiểm tra khả năng chịu tải của hệ thống bằng cách tạo ra một số lượng tiến trình giả lập việc sử dụng tài nguyên. Yêu cầu này được mã hóa dưới dạng JSON với các trường như ”numberOfTaskT-

”oRun” - số tiền trình stress cần tạo, ”MEMUsagePercent” - phần trăm bộ nhớ sử dụng, ”numberOfCore” - số lõi CPU được sử dụng tối đa và ”timeout” - thời gian chạy Stress test.

```
{
  "type": "startStress",
  "numberOfTaskToRun": 2,
  "MEMUsagePercent": 70,
  "numberOfCore": 4,
  "timeout": 30,
}
```

Hình 3.9 Cấu trúc yêu cầu bắt đầu chạy Stress test dạng JSON.

Thứ hai là lệnh dừng Stress test (Hình 3.10), nhằm chấm dứt toàn bộ các tiến trình được tạo ra từ lệnh stress trước đó. Đây là một yêu cầu đơn giản, chỉ chứa một trường duy nhất ”type”: ”stopStress”.

```
{
  "type": "stopStress",
}
```

Hình 3.10 Cấu trúc yêu cầu dừng Stress test dạng JSON.

Thứ ba là yêu cầu kết thúc một tiến trình, yêu cầu này được biểu diễn dưới dạng JSON như Hình 3.11, trong đó ”PName” là tên tiến trình mà người dùng muốn kết thúc.

```
{
  "type": "killProcess",
  "PName": "chrome",
}
```

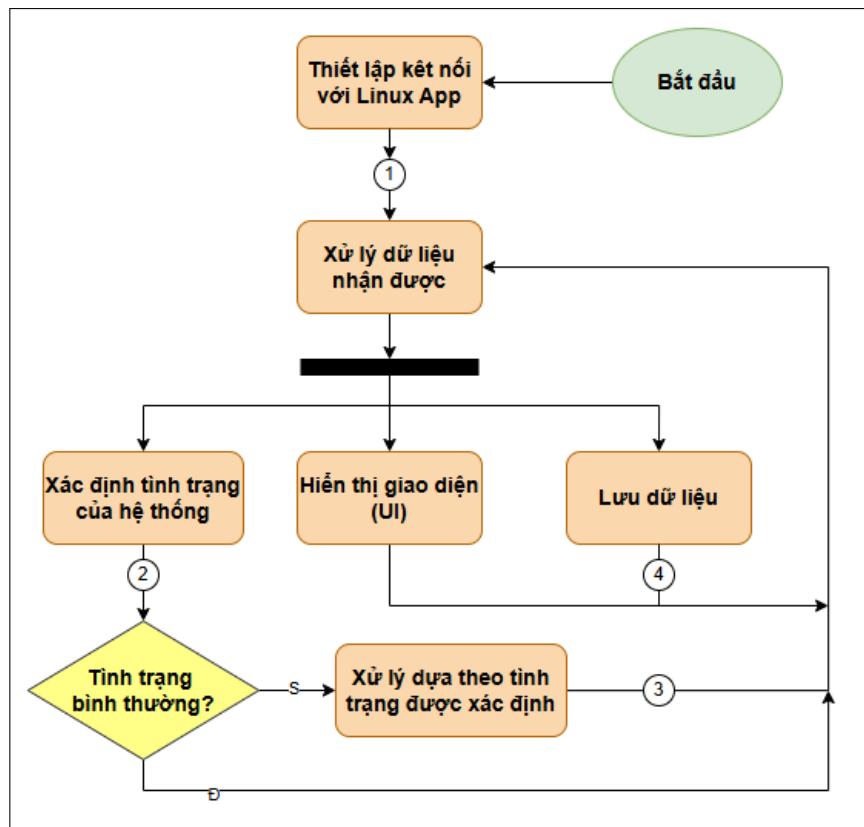
Hình 3.11 Cấu trúc yêu cầu đóng một tiến trình dạng JSON.

3.2.4. Thiết kế các luồng hoạt động trong Windows App

1. Luồng hoạt động tổng thể

Hình 3.12 mô tả các luồng hoạt động chính trong phân hệ Windows App. Và các điểm khoanh tròn 1, 2, 3, 4 là các điểm mốc hoạt động trong luồng chính. Các điểm mốc này nhằm mô tả chi tiết cách hoạt động của các chức năng của Windows App.

Dựa theo mô tả trong Hình 3.12, khi khởi động, Windows App sẽ thiết lập các kết nối với Linux App (đã đề cập trong Mục 2 ở trên) và bắt đầu giao tiếp trao đổi dữ liệu với

**Hình 3.12** Luồng hoạt động tổng quan của Windows App.

nhau. Sau đó dữ liệu sẽ được xử lý để phục vụ các mục đích cụ thể như sau:

Thứ nhất, dữ liệu sẽ được hiển thị trực quan trên giao diện người dùng thông qua các biểu đồ line chart, bảng tiến trình và cảnh báo khi hệ thống xảy ra quá tải. Chẳng hạn, CpuStatsViewModel và MemStatsViewModel sẽ cập nhật biểu đồ sử dụng CPU và MEM theo thời gian, giúp người dùng dễ dàng theo dõi tình trạng hoạt động của hệ thống.

Thứ hai, dữ liệu được dùng để phân tích và phát hiện xem là hệ thống có đang hoạt động ổn định hay chưa, hay là đang có những dấu hiệu bất thường để từ đó đưa ra hướng xử lý phù hợp.

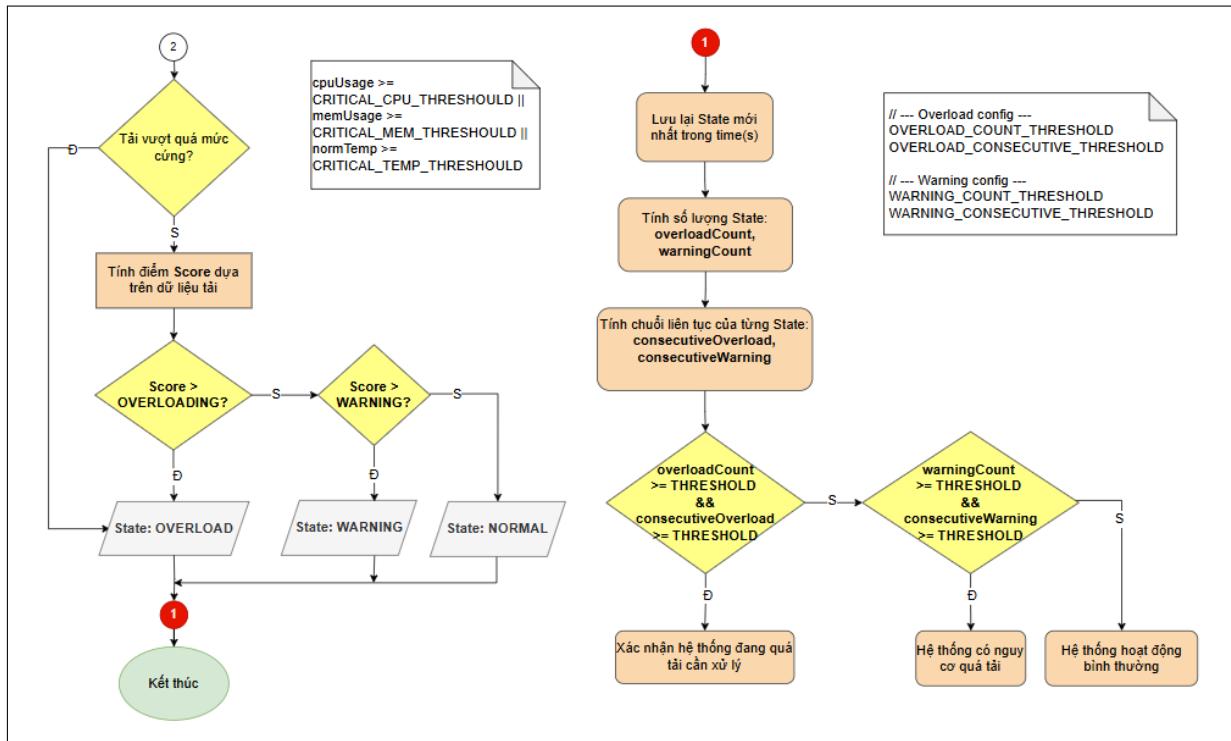
Thứ ba, dữ liệu sau xử lý còn có thể được ghi lại để lưu trữ, phục vụ thống kê, xuất báo cáo hoặc phân tích hiệu suất dài hạn.

2. Luồng xác định tình trạng của hệ thống

Hình 3.13 mô tả cách xác định tình trạng của hệ thống dựa vào các thông số quan trọng như mức sử dụng CPU, bộ nhớ (MEM), nhiệt độ CPU và các yếu tố liên quan khác. Có ba trạng thái cảnh báo chính, cụ thể là:

1. OVERLOAD: đây là trạng thái xác nhận hệ thống được xác định là quá tải.

2. WARNING: đây là trạng thái xác nhận hệ thống có nguy cơ xảy ra quá tải.
3. NORMAL: đây là trạng thái xác nhận hệ thống đang hoạt động bình thường.



Hình 3.13 Luồng hoạt động xác định tình trạng của hệ thống.

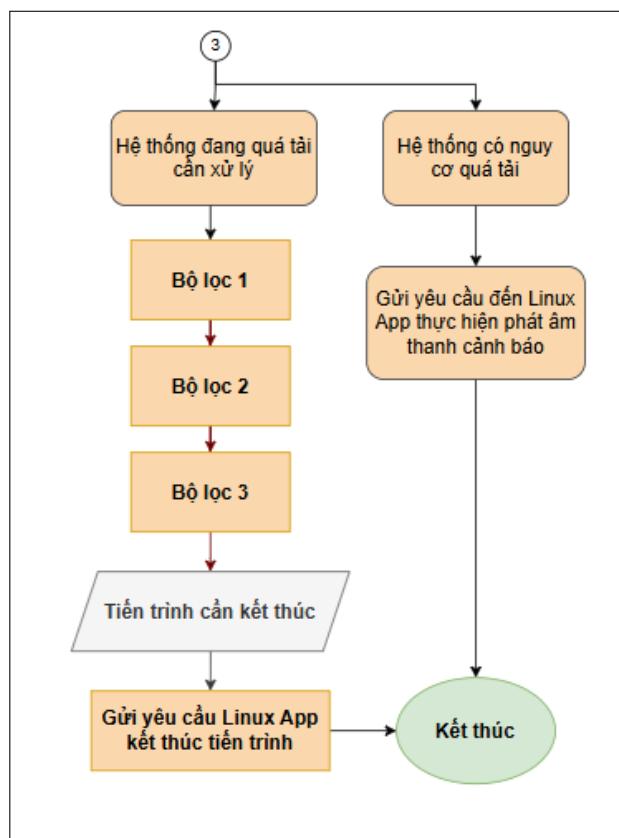
Ban đầu, *Phần mềm* sẽ kiểm tra các thông số cơ bản gồm mức tải CPU (cpuUsage), mức sử dụng bộ nhớ RAM (memUsage), và nhiệt độ CPU (temp). Nếu các thông số này vượt quá các ngưỡng cảnh báo đã được cấu hình (ví dụ: CRITICAL_CPU_THRESHOLD, CRITICAL_MEM_THRESHOLD, hoặc CRITICAL_TEMP_THRESHOLD)⁵ thì xác nhận hệ thống được cho là quá tải. Nếu không, lúc này sẽ tiến hành tính toán điểm tổng hợp (Score) dựa trên dữ liệu tải (phần này được trình bày tại §3.3). Giá trị của điểm này sẽ quyết định trạng thái sơ bộ của hệ thống: Nếu Score lớn hơn ngưỡng quá tải (OVERLOADING)⁵, hệ thống chuyển sang trạng thái OVERLOAD. Nếu Score không vượt ngưỡng quá tải nhưng lớn hơn ngưỡng cảnh báo (WARNING)⁵, hệ thống chuyển sang trạng thái WARNING. Nếu không thỏa cả hai điều kiện trên, hệ thống được coi là ở trạng thái bình thường NORMAL.

Tuy nhiên, để tránh việc phản ứng quá do các biến động ngắn hạn, các trạng thái mới nhất sẽ được lưu lại trong một khoảng thời gian nhất định. Sau đó, tính toán số lần xuất hiện của trạng thái quá tải (overloadCount) hoặc trạng thái cảnh báo (warningCount) trong chuỗi trạng thái được lưu. Đồng thời, kiểm tra chuỗi trạng thái liên tục gần nhất để

⁵Các ngưỡng giá trị này đã được giải thích tại §3.2.5 Mục 5

xác định số lần xuất hiện liên tiếp cho trạng thái quá tải (consecutiveOverload) cũng như trạng thái cảnh báo (consecutiveWarning). Nếu số lần quá tải vượt qua ngưỡng tối đa cho phép (OVERLOAD_COUNT_THRESHOLD)⁵ đồng thời số lần quá tải liên tiếp cũng đạt mức tối thiểu (OVERLOAD_CONSECUTIVE_THRESHOLD)⁵, thì sẽ xác nhận rằng hệ thống đang thực sự ở trạng thái quá tải và cần tiến hành xử lý ngay lập tức. Tương tự, nếu các thông số liên quan đến trạng thái cảnh báo vượt qua các ngưỡng cảnh báo tương ứng (WARNING_COUNT_THRESHOLD, WARNING_CONSECUTIVE_THRESHOLD)⁵, thì sẽ ghi nhận rằng đang tồn tại nguy cơ xảy ra quá tải, cần cảnh báo sớm và theo dõi tình hình. Ngược lại, nếu các điều kiện trên không thỏa mãn, hệ thống được xác nhận là đang ở trạng thái bình thường, không cần xử lý thêm vào thời điểm hiện tại. Phương pháp này giúp hệ thống đưa ra quyết định chính xác, tránh phản ứng quá nhanh hoặc bỏ sót các trường hợp thực sự nguy hiểm.

3. Luồng hoạt động xử lý dựa vào tình trạng được xác định của hệ thống



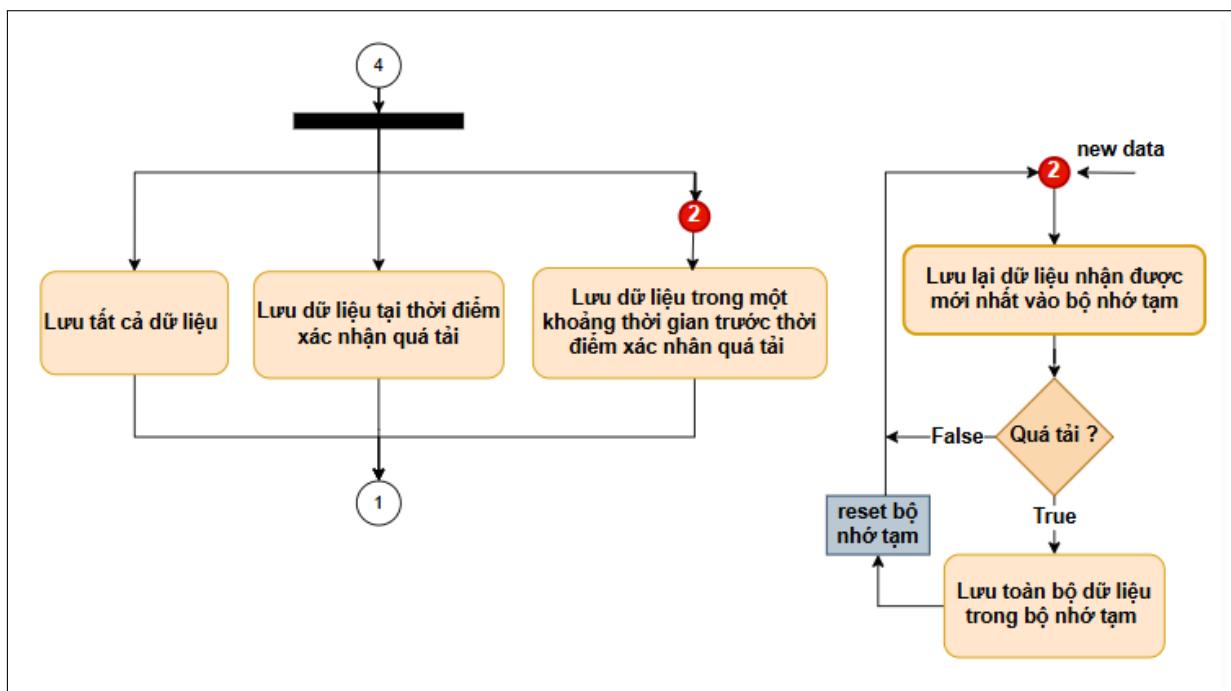
Hình 3.14 Luồng hoạt động xử lý dựa vào tình trạng được xác định của hệ thống.

Sau khi đã xác định được trạng thái hoạt động của hệ thống (như đã mô tả ở §3.2.4 Mục 2), các hành động xử lý tương ứng sẽ được tiến hành thực thi. Hình 3.14 minh họa luồng xử lý dựa trên hai trạng thái chính: quá tải cần xử lý và cảnh báo nguy cơ quá tải.

Nếu hệ thống được xác nhận là đang ở trạng thái quá tải thực sự, luồng xử lý sẽ đi vào nhánh bên trái. Với mục đích xác định các tiến trình chiếm dụng tài nguyên cao nhưng không thực sự cần thiết, *Phần mềm* tiến hành chọn ra tiến trình phù hợp (cách chọn ra tiến trình phù hợp sẽ được trình bày rõ tại §3.4) và gửi yêu cầu kết thúc tiến trình đó (kill process) đến Linux App, dưới dạng chuỗi JSON truyền qua TCP như đã trình bày ở §3.2.3 Mục 3.

Ngược lại, nếu hệ thống chỉ ở mức cảnh báo, Windows App sẽ gửi một yêu cầu đơn giản tới Linux App để phát âm thanh cảnh báo. Điều này giúp người giám sát sớm nhận biết được tình trạng của hệ thống và có phương án xử lý.

4. Luồng lưu trữ dữ liệu



Hình 3.15 Luồng hoạt động lưu trữ dữ liệu

Trong các hệ thống giám sát, việc lưu trữ dữ liệu theo thời gian thực đóng vai trò quan trọng nhằm phục vụ cho quá trình phân tích, chẩn đoán sự cố và theo dõi trạng thái hoạt động của hệ thống. Dữ liệu cần được lưu trữ ổn định, có cấu trúc, dễ truy xuất và phân tách giữa dữ liệu thường xuyên và dữ liệu trong quá trình quá tải.

Hình 3.15 mô tả sơ đồ hoạt động của quá trình lưu trữ dữ liệu trong hệ thống. Quy trình được chia thành ba luồng chính:

1. Luồng lưu trữ liên tục: toàn bộ dữ liệu sẽ được ghi lại theo định kỳ mà không phân biệt trạng thái của hệ thống. Đây là cơ chế lưu trữ mặc định giúp đảm bảo tính liên

tục và toàn vẹn dữ liệu hệ thống.

2. Luồng lưu trữ tại thời điểm xác nhận hệ thống quá tải: khi hệ thống được cho là quá tải và cần xử lý, dữ liệu ngay tại thời điểm này sẽ được lưu lại.
3. Luồng lưu dữ liệu trong một khoảng thời gian trước thời điểm xác nhận hệ thống quá tải: mỗi dữ liệu mới nhận được sẽ được lưu vào bộ nhớ đệm, nơi chứa tập hợp các mẫu dữ liệu gần nhất trong tối đa 60 giây trước thời điểm hệ thống rơi vào trạng thái quá tải. Khi hệ thống được xác định là quá tải, toàn bộ dữ liệu trong bộ nhớ đệm sẽ lập tức được lưu lại, và vùng nhớ tạm này sẽ được xóa để chuẩn bị cho chu kỳ ghi tiếp theo. Quá trình này sẽ diễn ra liên tục như vậy.

3.2.5. Giao diện của phần mềm Windows App

Giao diện được thiết kế theo hướng tối giản, trực quan và thân thiện với người sử dụng. Ở góc trên cùng bên trái của cửa sổ giao diện là tên phần mềm "IVI System Monitor" được hiển thị rõ ràng. Ngay bên dưới tiêu đề là thanh chọn (tab bar) đặt ở trung tâm giao diện, đóng vai trò là khu vực điều hướng chính của phần mềm. Thanh này bao gồm ba thẻ chức năng chính: Processes (Tiến trình), Performance (Hiệu năng), Setting (Cài đặt).

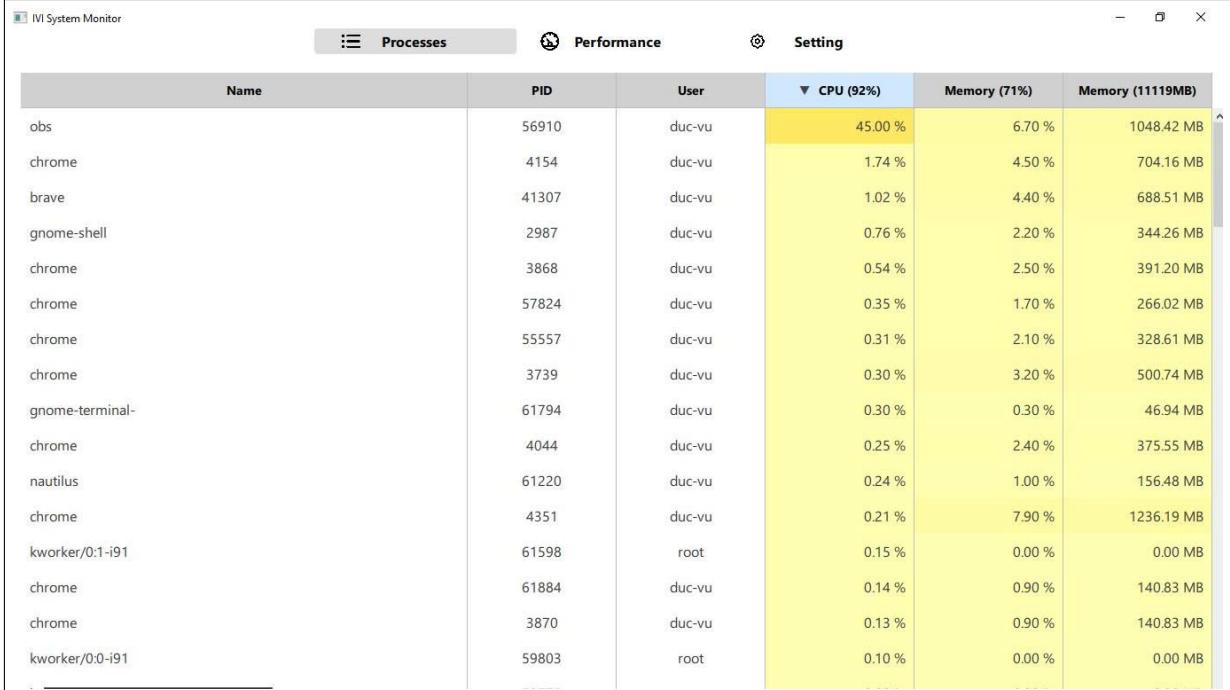
Khi người dùng lựa chọn một trong ba thẻ này, nội dung màn hình hiển thị phía dưới sẽ thay đổi tương ứng, cụ thể:

1. Thẻ Processes cho phép theo dõi chi tiết danh sách tiến trình đang chạy và mức tiêu thụ tài nguyên của từng tiến trình.
2. Thẻ Performance cung cấp các biểu đồ trực quan thể hiện tình trạng sử dụng CPU, MEM theo thời gian thực.
3. Thẻ Setting cung cấp giao diện cấu hình, cho phép người dùng tùy chỉnh các ngưỡng, thông số liên quan.

Nội dung trình bày dưới đây sẽ đi vào giới thiệu chi tiết từng giao diện chính của phần mềm Windows App.

1. Giao diện màn hình hiển thị thông tin về tiến trình cùng với tài nguyên chiếm dụng

Hình 3.16 minh họa giao diện hiển thị thông tin về các tiến trình cùng với tài nguyên mà nó chiếm dụng trong thanh chọn Processes. Cụ thể, với các cột thông tin chi tiết bao gồm: tên tiến trình (Name), mã định danh tiến trình (PID), người dùng khởi tạo tiến trình (User), mức sử dụng CPU (%CPU), mức sử dụng bộ nhớ tính theo phần trăm



The screenshot shows the IVI System Monitor application window. At the top, there are three tabs: 'Processes' (selected), 'Performance', and 'Setting'. The main area is a table with the following columns: Name, PID, User, CPU (92%), Memory (71%), and Memory (11119MB). The table lists various processes along with their resource consumption. The 'CPU' column is sorted by value, with 'obs' at 45.00% being the highest consumer.

Name	PID	User	CPU (92%)	Memory (71%)	Memory (11119MB)
obs	56910	duc-vu	45.00 %	6.70 %	1048.42 MB
chrome	4154	duc-vu	1.74 %	4.50 %	704.16 MB
brave	41307	duc-vu	1.02 %	4.40 %	688.51 MB
gnome-shell	2987	duc-vu	0.76 %	2.20 %	344.26 MB
chrome	3868	duc-vu	0.54 %	2.50 %	391.20 MB
chrome	57824	duc-vu	0.35 %	1.70 %	266.02 MB
chrome	55557	duc-vu	0.31 %	2.10 %	328.61 MB
chrome	3739	duc-vu	0.30 %	3.20 %	500.74 MB
gnome-terminal-	61794	duc-vu	0.30 %	0.30 %	46.94 MB
chrome	4044	duc-vu	0.25 %	2.40 %	375.55 MB
nautilus	61220	duc-vu	0.24 %	1.00 %	156.48 MB
chrome	4351	duc-vu	0.21 %	7.90 %	1236.19 MB
kworker/0:1-i91	61598	root	0.15 %	0.00 %	0.00 MB
chrome	61884	duc-vu	0.14 %	0.90 %	140.83 MB
chrome	3870	duc-vu	0.13 %	0.90 %	140.83 MB
kworker/0:0-i91	59803	root	0.10 %	0.00 %	0.00 MB

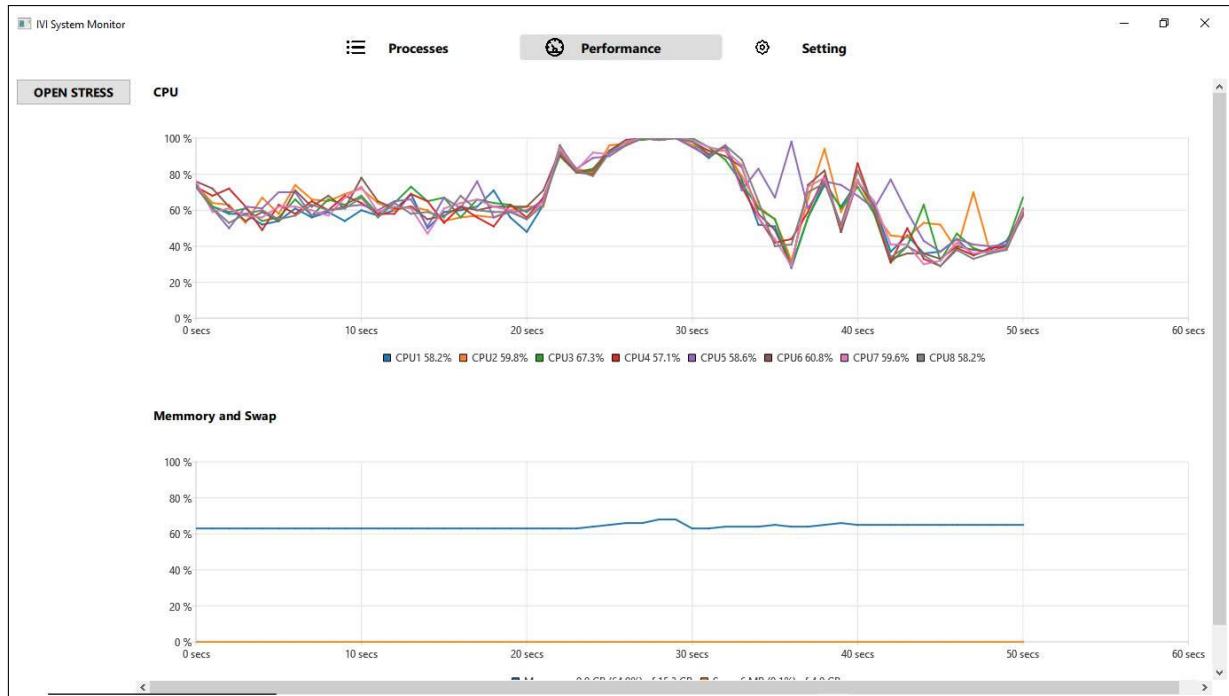
Hình 3.16 Giao diện màn hình hiển thị thông tin về tiến trình cùng với tài nguyên chiếm dụng

(%Memory), dung lượng bộ nhớ sử dụng tính theo megabyte (Memory - MB). Có thể sắp xếp danh sách tiến trình theo độ tăng hay giảm mức sử dụng tài nguyên. Ngoài ra, có sự thay đổi độ đậm nhạt của màu sắc giúp người dùng thấy mức độ sử dụng tài nguyên của các tiến trình trực quan hơn. Màu sắc càng đậm cho thấy tiến trình đó chiếm dụng tài nguyên càng nhiều và ngược lại.

2. Giao diện màn hình hiển thị các biểu đồ đường trực quan

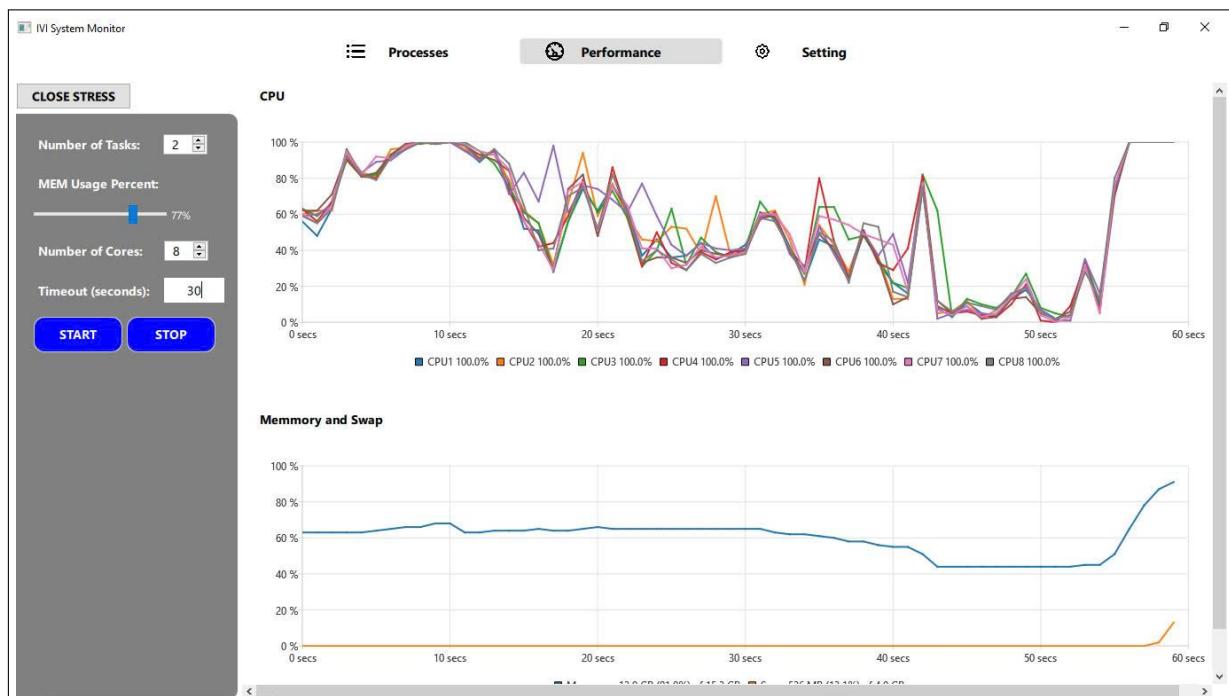
Trong thanh chọn Performance, hiển thị một giao diện trực quan (Hình 3.17) gồm có hai biểu đồ đường (Line Chart) chính: biểu đồ thứ nhất là biểu đồ theo dõi CPU của từng lõi (core) và biểu đồ thứ hai là biểu đồ theo dõi bộ nhớ (RAM và Swap). Cho phép người dùng dễ dàng theo dõi sự biến thiên liên tục về mức sử dụng tài nguyên trong khoảng 60s mới nhất. Đối với biểu đồ thứ nhất, mỗi lõi CPU sẽ được biểu diễn bằng một màu riêng biệt và phía dưới biểu đồ có chú giải (Legend) cho biết chính xác mức sử dụng của từng lõi CPU. Đối với biểu đồ thứ hai cũng tương tự như thế.

Ngoài ra, trong giao diện Performance, ở góc trên bên trái có một nút "OPEN STRESS". Khi kích chọn vào thì nó sẽ hiển thị ra giao diện Stress test (được giới thiệu ở Mục 3 dưới đây).



Hình 3.17 Giao diện màn hình hiển thị các biểu đồ trực quan

3. Giao diện màn hình Stress test



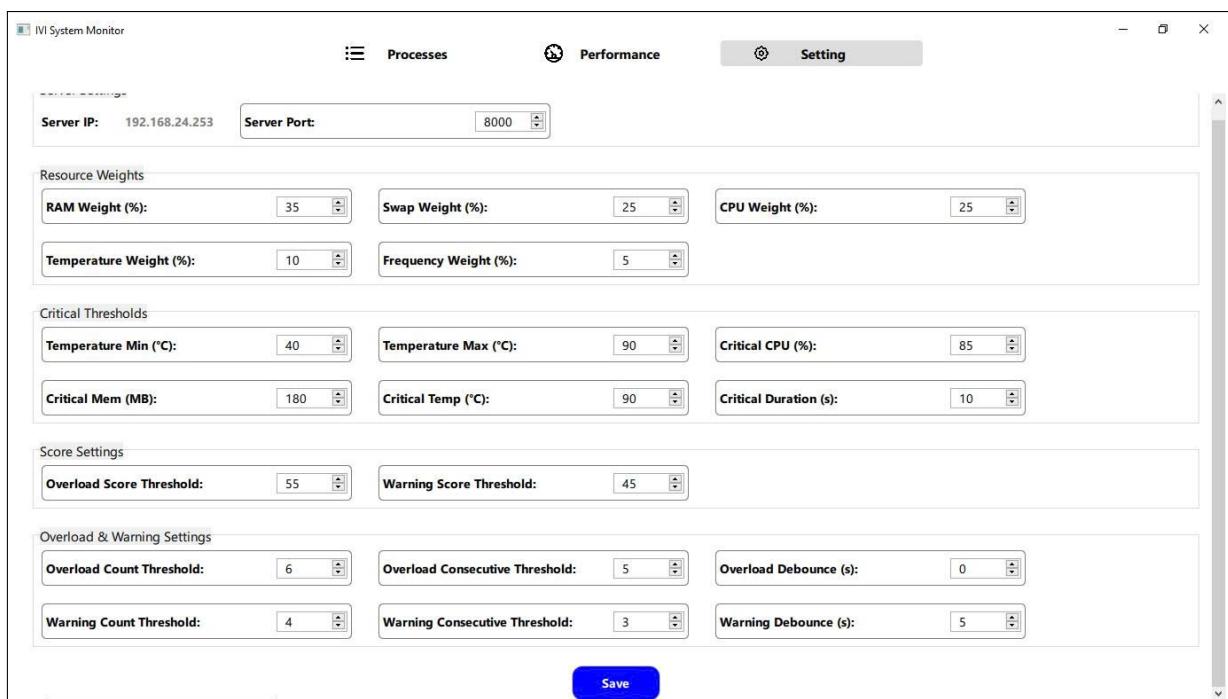
Hình 3.18 Giao diện màn hình Stress test

Sau khi kích chọn vào nút "OPEN STRESS", một giao diện Stress test xuất hiện, cho phép người dùng thiết lập các thông số cần thiết. Mục đích để tạo ra các tiến trình giả lập có tên "stress" được chiếm dụng mức tài nguyên cao. Khi ấn nút "START", các thông

số này sẽ được gửi cho Linux App (trình bày ở §3.2.3 Mục 3) để tiến hành chạy "stress" trên hệ thống. Và cũng có thể bấm nút "STOP" để dừng quá trình "stress" trước thời gian đã thiết lập (Timeout). Trong trường hợp chưa cần thực hiện kiểm thử tải, người dùng có thể đóng cửa sổ "Stress test" bằng cách nhấn vào nút "CLOSE STRESS".

4. Giao diện màn hình hiển thị và thiết lập các thông số cần thiết

Giao diện trong thanh chọn Setting (Hình 3.19), minh họa các ngưỡng, các trọng số mà người dùng có thể thấy được và cũng có thể điều chỉnh theo nhu cầu và yêu cầu của từng hệ thống khác nhau. Các giá trị này được thiết lập cụ thể, phục vụ cho chức năng phát hiện tình trạng hiện tại của hệ thống (được đề cập tại §3.2.4 Mục 2). Sau khi điều chỉnh xong, nhấn nút "Save" để lưu lại các thay đổi. Dưới đây là bảng giải thích rõ về cấu hình ngưỡng và trọng số. (Bảng: 3.1)



Hình 3.19 Giao diện màn hình hiển thị và thiết lập các thông số cần thiết

Bảng 3.1 Mô tả các trọng số, cấu hình ngưỡng giám sát hệ thống

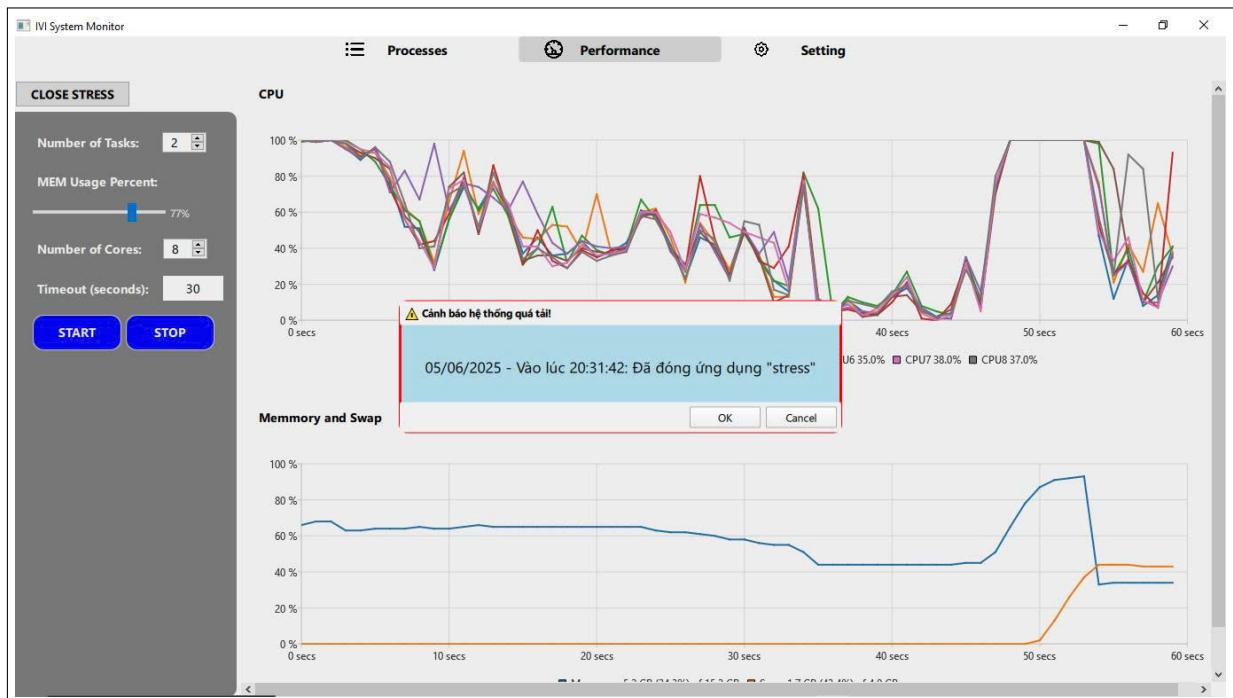
Nhóm	Tên cấu hình	Tham số	Mô tả
Weight - Trọng số	RAM Weight	WEIGHT_RAM	Trọng số đánh giá mức sử dụng RAM.
Weight - Trọng số	Swap Weight	WEIGHT_SWAP	Trọng số đánh giá mức sử dụng Swap.

Weight - Trọng số	CPU Weight	WEIGHT_CPU	Trọng số đánh giá mức sử dụng CPU.
Weight - Trọng số	Temperature Weight	WEIGHT_TEMP	Trọng số đánh giá mức nhiệt độ của CPU.
Weight - Trọng số	Frequence Weight	WEIGHT_FREQ	Trọng số đánh giá mức tần số của CPU.
Threshold - Nguồn	Temperature Min	TEMP_MIN	Nguồn nhiệt độ thấp nhất được chấp nhận.
Threshold - Nguồn	Temperature Max	TEMP_MAX	Nguồn nhiệt độ cao nhất cho phép.
Threshold - Nguồn	Critical CPU	CRITICAL_CPU_THRESHOLD	Mức sử dụng CPU nếu vượt quá sẽ được coi là quá tải.
Threshold - Nguồn	Critical Mem	CRITICAL_MEM_THRESHOLD	Mức sử dụng MEM nếu vượt quá sẽ được coi là quá tải.
Threshold - Nguồn	Critical Temp	CRITICAL_TEMP_THRESHOLD	Mức nhiệt độ mà nếu vượt quá sẽ coi là quá tải.
Threshold - Nguồn	Critical Duration	CRITICAL_DURATION_THRESHOLD	Khoảng thời gian cần thiết để xác nhận chính xác tình trạng hệ thống.
Threshold - Nguồn	Overload Score Threshold	OVERLOADING	Nếu điểm tổng hợp (Score) vượt ngưỡng này, hệ thống xác nhận trạng thái quá tải (trạng thái OVERLOAD).
Threshold - Nguồn	Warning Score Threshold	WARNING	Nếu điểm tổng hợp (Score) vượt ngưỡng này, hệ thống xác nhận trạng thái cảnh báo (trạng thái WARNING).
Threshold - Nguồn	Overload Count Threshold	OVERLOAD_COUNT_THRESHOLD	Số lần xuất hiện trạng thái OVERLOAD trong một khoảng thời gian.
Threshold - Nguồn	Overload Consecutive Threshold	OVERLOAD_CONSECUTIVE_THRESHOLD	Số lần trạng thái OVERLOAD liên tục.
Threshold - Nguồn	Overload Debounce	OVERLOAD_DEBOUNCE_MS	Thời gian chờ để phát tín hiệu xác nhận hệ thống quá tải cần xử lý.

Threshold - Nguồn	Warning Count Threshold	WARNING _COUNT _THRESHOLD	Số lần xuất hiện trạng thái WARNING.
Threshold - Nguồn	Warning Consecutive Threshold	WARNING _CONSECUTIVE _THRESHOLD	Số lần trạng thái WARNING liên tục.
Threshold - Nguồn	Warning Debounce	WARNING _DEBOUNCE_MS	Thời gian chờ để phát tín hiệu xác nhận hệ thống có khả năng quá tải.

5. Giao diện màn hình hiển thị màn hình cảnh báo

Hình 3.20, có một giao diện hiển thị thông báo có viền màu đỏ, xuất hiện ngay chính giữa màn hình, với nội dung là thời gian cụ thể lúc xảy ra quá tải cùng với tên tiến trình đã được kết thúc. Chỉ khi có tín hiệu xảy ra tình trạng quá tải thì thông báo này mới xuất hiện và có thể dễ dàng đóng thông báo này nếu muốn.



Hình 3.20 Giao diện màn hình hiển thị thông báo

3.3. Triển khai thuật toán phát hiện hệ thống quá tải

Tại §3.2.4 Mục 2, đã trình bày rõ ràng quá trình để xác định tình trạng của hệ thống đang thực sự quá tải cần được xử lý, hay đang có nguy cơ quá tải, hay là vẫn hoạt động ổn định. Trong phần này, chúng tôi muốn làm rõ cách tính điểm (Score) và đánh giá tình

trạng thực sự của hệ thống.

1. Phương pháp tính điểm (Score)

Đầu tiên, đối với phương pháp tính điểm, chúng tôi dựa theo mô hình tổng trọng số - Weighted Sum Model [7] với đầu vào là các yếu tố ảnh hưởng và trọng số tương ứng. Công thức được áp dụng trong trường hợp này như sau:

$$\text{Score} = w_{\text{ram}} \times \text{ramUsagePercent} + w_{\text{swap}} \times \text{swapUsagePercent} + w_{\text{cpu}} \times \text{cpuUsage} \\ + w_{\text{temp}} \times \text{normTemp} + w_{\text{freq}} \times \text{cpuFreqPercent} + \text{balancePenalty} \quad (1)$$

Trong đó:

- $w_{\text{ram}}, w_{\text{swap}}, w_{\text{cpu}}, w_{\text{temp}}, w_{\text{freq}}$ lần lượt là các trọng số của các chỉ số: RAM, Swap, CPU, nhiệt độ, và tần số CPU.
- ramUsagePercent: Tỷ lệ bộ nhớ RAM đã sử dụng (chuẩn hóa từ 0 đến 1).
- swapUsagePercent: Tỷ lệ bộ nhớ Swap đã sử dụng (chuẩn hóa từ 0 đến 1).
- cpuUsage: Mức sử dụng CPU trung bình toàn hệ thống (chuẩn hóa từ 0 đến 1).
- normTemp: Nhiệt độ CPU (chuẩn hóa từ 0 đến 1).
- cpuFreqPercent: Tỷ lệ tần số CPU hiện tại (chuẩn hóa từ 0 đến 1).
- balancePenalty là giá trị bổ sung dựa trên mức độ mất cân bằng giữa các lõi CPU (nếu có).

Giá trị các trọng số được lựa chọn dựa vào mức độ ảnh hưởng thực tế của từng yếu tố dẫn đến quá tải. Theo như bài nghiên cứu [3][4], mức sử dụng bộ nhớ RAM và Swap là yếu tố quan trọng quyết định đến hiệu suất của hệ thống. Bên cạnh đó, CPU cũng là một trong những yếu tố quan trọng không kém tuy nhiên xếp sau RAM và Swap một chút. Đối với nhiệt độ và tần số CPU chỉ là các chỉ số bổ trợ trong quá trình đánh giá quá tải hệ thống. Nhiệt độ cao thường là do hệ quả của quá trình sử dụng tài nguyên kéo dài (đặc biệt là CPU và nguồn điện cao) và khi vượt ngưỡng cho phép có thể tự động giảm xung nhịp ảnh hưởng nghiêm trọng đến hiệu năng thực tế. Tuy nhiên, trong điều kiện hoạt động bình thường, nhiệt độ và tần số CPU thường không phải là nguyên nhân trực tiếp gây quá tải. Vì vậy, các yếu tố này thường được ưu tiên trọng số thấp hơn so với RAM, Swap và CPU. Sau nhiều lần thử nghiệm và đánh giá, chúng tôi đã thiết lập giá trị cho các trọng số $w_{\text{ram}}, w_{\text{swap}}, w_{\text{cpu}}, w_{\text{temp}}, w_{\text{freq}}$ lần lượt là: "0.3", "0.3", "0.25", "0.1",

”0.05”. Và các trọng số này có thể được điều chỉnh phù hợp với yêu cầu hệ thống khác nhau (đề cập tại §3.2.5 Mục 4).

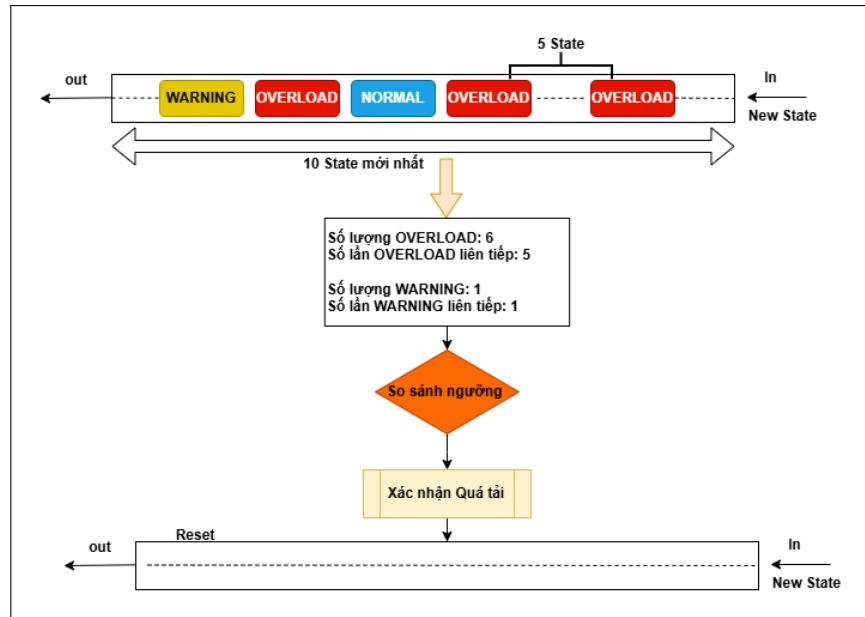
Giá trị ”balancePenalty” là một khoản điểm cộng thêm vào tổng điểm quá tải (Score), phản ánh độ mất cân bằng tải giữa các lõi CPU. Nếu các lõi chia sẻ đều tài nguyên thì giá trị penalty gần như bằng 0. Còn nếu trường hợp một lõi CPU (core) chiếm 100% trong khi các core khác thì nhàn rỗi. Có thể hệ thống chưa quá tải, nhưng thực tế tại các tác vụ tại core quá tải kia khả năng cao bị treo, dẫn đến treo cả hệ thống nếu không phát hiện và xử lý kịp thời.

2. Đánh giá tình trạng hệ thống sau một khoảng thời gian.

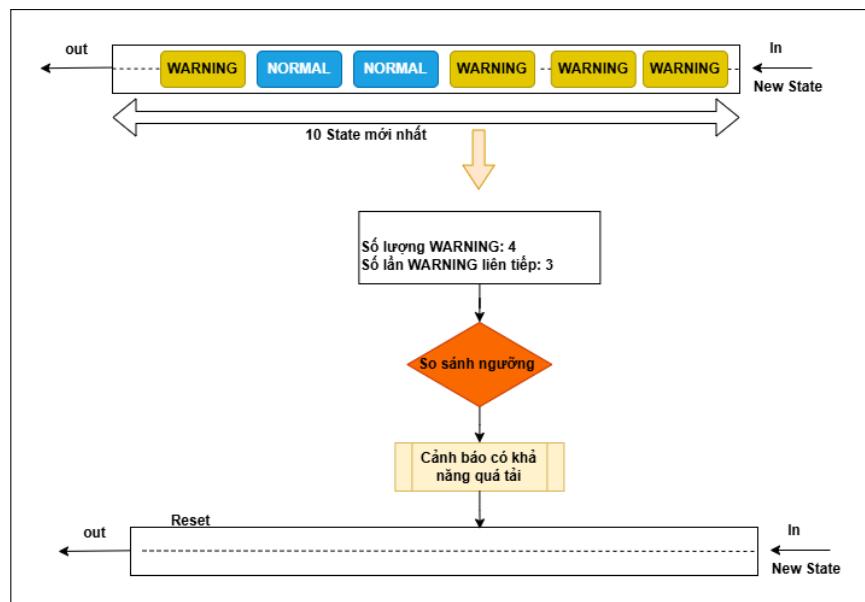
Để đảm bảo tính chính xác và loại bỏ ảnh hưởng của các dao động ngắn hạn. Việc đánh giá trạng thái hiện tại của hệ thống không chỉ dựa vào trạng thái tức thời mà còn phải xét đến sự ổn định của trạng thái đó trong một khoảng thời gian nhất định. Có tất cả là ba trạng thái, cụ thể: OVERLOAD, WARNING, NORMAL (có đề cập tại §3.2.4 Mục 2). Dựa vào số lần xuất hiện và tính liên tục của trạng thái trong một khoảng thời gian để đưa ra chính xác tình trạng hiện tại của hệ thống.

Các trạng thái này sẽ được lưu vào một hàng đợi để phục vụ cho việc tính số lượng và số lần liên tiếp của trạng thái. Sau khi xác nhận tình trạng của hệ thống (đã đề cập ở §3.2.4 Mục 2), hoặc đặt giới hạn số lượng trạng thái cho phép, hàng đợi này sẽ được làm mới và tiếp tục nhận trạng thái tiếp theo.

Để dễ hiểu hơn, hãy xem qua ví dụ về trường hợp xác nhận là hệ thống đang quá tải (Hình 3.21). Hàng đợi sẽ chứa tối đa 10 trạng thái gần nhất, trạng thái vào trước tiên là WARNING, không phát cảnh báo ngay mà chờ xem những trạng thái tiếp theo thế nào. Trạng thái tiếp theo là OVERLOAD, lúc này vẫn không phát cảnh báo ngay, chờ trạng thái tiếp theo là NORMAL, lúc này vẫn đang cho rằng hệ thống đang chạy ổn định. Rồi đến trạng thái tiếp theo nhận được là OVERLOAD, liên tục 5 lần như thế. Lúc này ghi nhận rằng trong vòng 10 trạng thái gần nhất, có tổng cộng là 6 lần OVERLOAD và có chuỗi OVERLOAD liên tục là 5, điều này phù hợp với ngưỡng thiết lập nên nhận định rằng hệ thống đang thực sự quá tải và cần xử lý. Sau đó hàng đợi sẽ được làm mới và tiếp tục quá trình nhận và kiểm tra như vậy. Ví dụ ở Hình 3.22, cũng tương tự vậy, đây là một ví dụ mô tả cho trường hợp cảnh báo hệ thống có khả năng quá tải.



Hình 3.21 Mô tả trường hợp xác nhận hệ thống quá tải



Hình 3.22 Mô tả trường hợp cảnh báo hệ thống có khả năng quá tải

3.4. Triển khai thuật toán xử lý khi hệ thống quá tải

Chúng tôi thực hiện tùy chỉnh tải thông qua việc kết thúc các tiến trình đang chạy trên hệ thống, từ đó giảm tải, đưa trạng thái của hệ thống quay lại bình thường. Vấn đề là làm sao để xác định được tiến trình nào cần và có thể kết thúc. Trong hệ điều hành Linux, có nhiều tiến trình tối quan trọng, nếu kết thúc các tiến trình này thì hệ thống sẽ hoạt động sai lệch và có thể dẫn đến hệ thống ngừng hoạt động đột ngột, ngoài ra còn có các tiến

trình có thể không khiếu cho hệ thống bị lỗi nhưng cũng đóng vai trò rất quan trọng như tiến trình điều khiển bàn phím, màn hình... Những tiến trình như vậy tuyệt đối không được phép kết thúc. Đồng thời việc kết thúc các tiến trình cũng cần xem xét đến trải nghiệm người dùng, như có những tiến trình thông thường không tác động sâu vào hệ điều hành nhưng lại tác động lớn đến người dùng như điều hướng bản đồ, xem video,... nhưng mức độ quan trọng của chúng đối với người dùng là khác nhau, ta cần xem xét nên ưu tiên kết thúc tiến trình nào vừa tối ưu tải cho hệ thống đồng thời tăng trải nghiệm người dùng, trong trường hợp nếu tiến trình "xem video" chiếm dụng mức tải ngang với tiến trình "điều hướng bản đồ", thì ta nên kết thúc tiến trình "xem video". Với ý tưởng đó chúng tôi đã xây dựng nên 3 bộ lọc để xác định nên kết thúc tiến trình nào.

Bộ lọc 1 là bộ lọc được xây dựng để loại bỏ các tiến trình root ra khỏi danh sách các tiến trình có thể kết thúc, các tiến trình "root" là các tiến trình tối quan trọng trong hệ điều hành Linux, kết quả sau khi qua bộ lọc 1 ta sẽ thu được danh sách các tiến trình thông thường.

Bộ lọc 2 là bộ lọc loại bỏ các tiến trình thường nhưng rất quan trọng như tiến trình điều khiển bàn phím, màn hình,... ra khỏi danh sách các tiến trình còn lại sau bộ lọc 1. Các tiến trình này được chúng tôi quy định tại một file cấu hình gọi là "Whitelist.txt" - chứa tên các tiến trình đó, sau khi đi qua bộ lọc 2 ta sẽ thu được danh sách những tiến trình thông thường có thể kết thúc.

Bộ lọc 3 là bộ lọc xác định tiến trình nào cần kết thúc từ danh sách tiến trình thu được sau khi qua bộ lọc 2. Tại đây thực hiện tính điểm (Score) dựa trên ba yếu tố chính: tỷ lệ sử dụng CPU (cpuPercent), tỷ lệ sử dụng bộ nhớ MEM (memPercent) và mức độ ưu tiên tiến trình (priority) - giá trị càng cao thì có khả năng bị kết thúc cao. Mức độ ưu tiên của tiến trình sẽ được cấu hình trong file "Priority.txt" - chứa tên tiến trình cùng với mức ưu tiên tương ứng. Phương pháp tính điểm tiến trình dựa theo mô hình tổng trọng số - Weighted Sum Model, công thức được áp dụng như sau:

$$\text{Score} = 0.35 \times \text{cpuPercent} + 0.45 \times \text{memPercent} + 0.2 \times \text{priority} \quad (2)$$

Trong đó, các giá trị "0.35", "0.45", "0.2" là các trọng số tương ứng. Việc lựa chọn các trọng số này dựa trên mức độ ảnh hưởng của từng yếu tố lên trạng thái chung của hệ thống. CPU và MEM (RAM và Swap) là hai yếu tố chính ảnh hưởng đến hiệu năng do đó được ưu tiên gán hiệu số cao hơn. Tuy nhiên, trọng số của MEM (0.45) được thiết lập nhỉnh hơn so với CPU (0.35) một chút (như đã giải thích tại §3.3 Mục 1), trong khi độ ưu tiên được xem là phụ trợ nên có trọng số thấp hơn (được xét là 0.2).

3.5. Kết luận chương

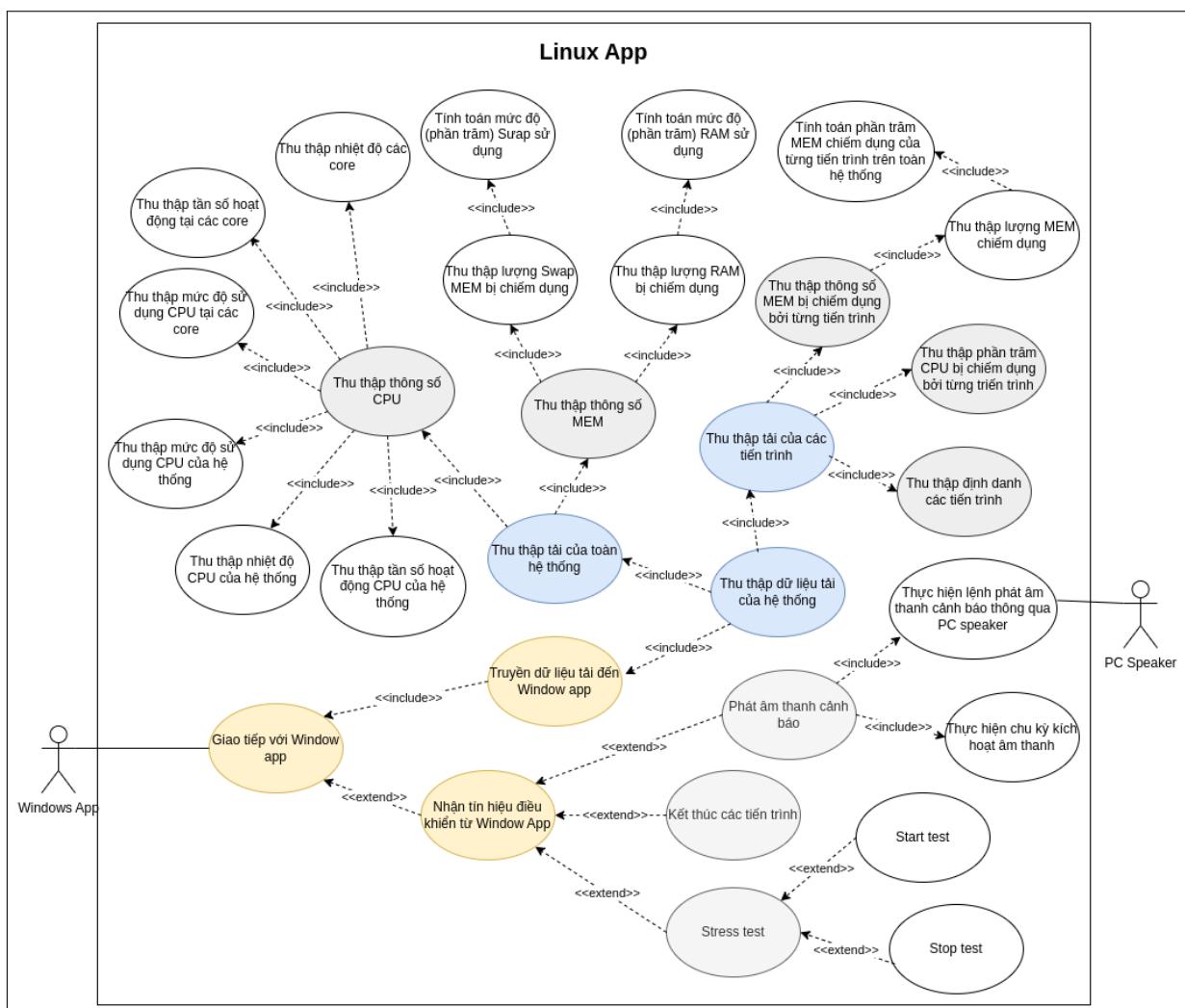
Như vậy, chương này đã trình bày chi tiết về quá trình thiết kế và triển khai phân hệ Windows App. Dựa vào các sơ đồ Use Case và Class Diagram để triển khai cấu trúc tổng thể của một chương trình. Chương này cũng đã mô tả cách đăng ký một đối tượng C++ vào QML để có thể sử dụng, cũng như cơ chế truyền nhận dữ liệu thông qua giao thức TCP/IP. Bên cạnh đó, luồng thực thi các chức năng chính và giao diện của Windows App đã được trình bày trực quan. Đặc biệt, chương đã đi sâu vào phân tích thuật toán phát hiện và xử lý khi hệ thống quá tải.

CHƯƠNG 4: THIẾT KẾ VÀ TRIỂN KHAI PHẦN HỆ LINUX APP

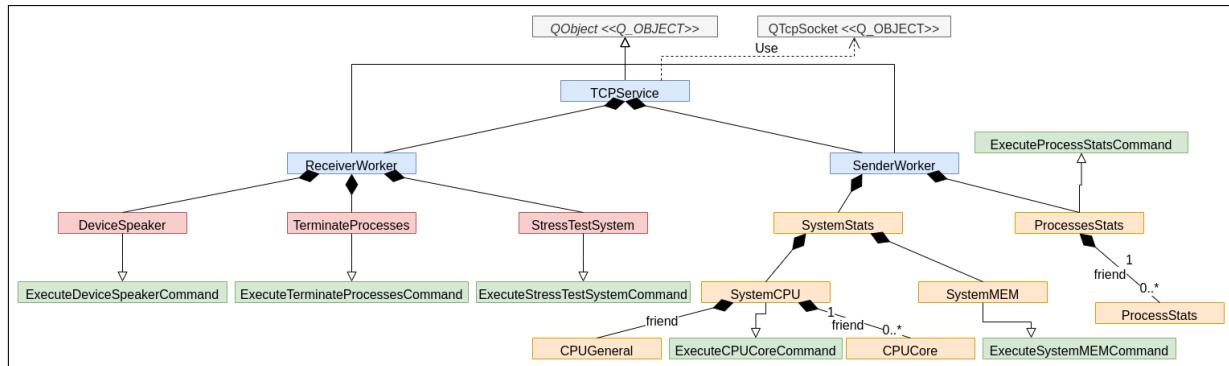
4.1. Mục tiêu và vai trò của phân hệ Linux App

Phân hệ Linux App đóng vai trò là thành phần chính chịu trách nhiệm giám sát, thu thập dữ liệu tài nguyên hệ thống IVI, thực hiện các thao tác kiểm soát như kết thúc tiến trình, cảnh báo người dùng và thực thi Stress test trên thiết bị mô phỏng hệ thống IVI. Nếu ví phân hệ Windows App là "bộ não", thì phân hệ Linux App như là những "cánh tay" thực thi trực tiếp của *Phần mềm*, đảm bảo hệ thống IVI luôn được theo dõi sát sao và phản ứng kịp thời khi phát hiện sự cố quá tải.

4.2. Sơ đồ tổng quan của phân hệ Linux App



Hình 4.1 Sơ đồ Use case của Linux App.



Hình 4.2 Sơ đồ lớp của Linux App.

Bảng 4.1 Mô tả chi tiết các thuộc tính và phương thức trong sơ đồ lớp của Linux App.

Lớp	Thuộc tính	Phương thức
TCPServier	<ul style="list-style-type: none"> - socket: QTcpSocket* - receiverThread: QThread - senderThread: QThread - stressTestThread: QThread - receiver: ReceiverWorker* - sender: SenderWorker* - stressTest: StressTestSystem* 	<ul style="list-style-type: none"> - establishSocket() - establishReceiverThread() - establishSenderThread() - establishStressTestThread() - onReadRead() - writeToSocket() - startStress() - handleTypeMessage() - killProcess() - stopStress() + start()
ReceiverWorker	<ul style="list-style-type: none"> - terminateProcesses: TerminateProcesses* - speaker: DeviceSpeaker* - stressTest: StressTestSystem* 	<ul style="list-style-type: none"> - terminate() - alterSpeaker() + handleStopStress() + handleKillProcess()
SenderWorker	<ul style="list-style-type: none"> - timer: * QTimer - timestamp: QString - systemStats: SystemStats* - processesStats: ProcessesStats* 	<ul style="list-style-type: none"> - currentDatTime() - collectStats() - collectSystemStats() - collectProcessesStats() - collectCPUStrats() - collectMEMStrats() - sendStats() - systemStatsToJson()

		<ul style="list-style-type: none"> - systemGeneralCPUToJson() - systemCoresCPUToJson() - systemMEMToJson() - processesStatsToJson() - sendMessage() + run()
ProcessStats	<ul style="list-style-type: none"> - processesInfoCommand: QString 	<pre># getprocessesInfoCommand()</pre>
SystemMEM	<ul style="list-style-type: none"> - MEMUtilizationCommand: QString - maxMEMSystemInfo Command:QString 	<pre># getMEMUtilization Command()</pre> <pre># getmaxMEMSystemInfo Command()</pre>
CPUCore	<ul style="list-style-type: none"> - numberOfCoreCPUCommand: QString - CPUUtilizationCommand: QString - coreCPUUtilizationCommand: QString - CPUTemperatureCommand: QString - coreTemperatureCommand: QString - CPUFrequencyMaxMin Command:QString - CPUFrequencyPercent Command:QString - coreFrequencyCommand: QString 	<pre># getnumberOfCoreCPU Command()</pre> <pre># getCPUUtilizationCommand()</pre> <pre># getCPUTemperature Command()</pre> <pre># getcoreTemperature Command()</pre> <pre># getCPUFrequencyMaxMin Command()</pre> <pre># getCPUFrequencyPercent Command()</pre> <pre># getcoreFrequencyCommand()</pre>
StressTestSystem	<ul style="list-style-type: none"> - stressTestCommand: QString - stopStressTestCommand: QString 	<pre># getStressTestCommand()</pre> <pre># getStopStressTestCommand()</pre>
TerminateProcess	<ul style="list-style-type: none"> - terminateProcessCommand: QString 	<pre># getTerminateProcess Command()</pre>
DeviceSpeaker	<ul style="list-style-type: none"> - alarmCommand: QString 	<pre># getAlarmCommand()</pre>

SystemStats	+ CPUStats: SystemCPU + MEMStats: SystemMEM	
SystemCPU	+ cores: CPUCore[numberOfCore] + general: CPUGeneral	+ getCPUUtilizationStats FromDevice() + getCPUTemperatureStats FromDevice() + getcoresTemperatureStats FromDevice() + getCPUFrequencyPercent FromDevice() + getcoresFrequencyStats FromDevice() - getCPUFrequencyMaxMin FromDevice() - getNumberOfCoreCPU FromDevice() - setupCores() - extractCoresCPUUsageInfo() - extractCoresCPUTemperature Info() - extractCoreCPUFrequency Info()
SystemMEM	+ maxRAMSystem: static float + maxSWAPMEMSystem: static float - RAMUtilization: float - SWAPMEMUtilization: float	+ getSystemMEMUtilization FromDevice() - getMaxSystemMEMInfo FromDevice() + getRAMUtilization() + getSWAPMEMUtilization() + getRAMUtilizationPercent() + getSWAPMEMUtilization Percent() - extractMEMUtilizationInfo() - extractMaxMEMSystemInfo()

CPUGeneral	<ul style="list-style-type: none"> - CPUUtilization: float - CPUTemperature: float - CPUFrequencyPercent: float - CPUFrequencyMax: float - CPUFrequencyMin: float 	<ul style="list-style-type: none"> + getCPUUtilization() + getCPUTemperature() + getCPUFrequencyPercent() + getCPUFrequency()
CPUCore	<ul style="list-style-type: none"> + numberofCore : static int - coreID: int - coreCPUUtilization: float - coreTemperature: float - coreFrequency: float 	<ul style="list-style-type: none"> + getCoreID() + getCoreCPUUtilization() + getCoreTemperature() + getCoreFrequency()
ProcessesStats	<ul style="list-style-type: none"> + processes: ProcessStats[] 	<ul style="list-style-type: none"> + getProcessesStats FromDevice() - extractProcessesInfo()
ProcessStats	<ul style="list-style-type: none"> - PID: int - PName: QString - user: QString - PCPUUsagePercent: float - PMEMUsagePercent: float - PMEMUsageMB: float 	<ul style="list-style-type: none"> + getPID() + getPNAME() + getUser() + getPCPUUsagePercent() + getPMEMUsagePercent() + getPMEMUsageMB()
StressTestSystem	<ul style="list-style-type: none"> - numberOfTaskToRun: int - MEMUsage: float - numberofCore: int - timeout: float 	<ul style="list-style-type: none"> + setup() + start() + stop() - run() - createNumberOfTaskToRun() - setupMEMUsagePercent() - setupNumberOfCore() - setupTimeout()
TerminateProcesses		<ul style="list-style-type: none"> + terminateProcessByPName()
DeviceSpeaker		<ul style="list-style-type: none"> + alertUserViaSound()

Chú thích bảng 4.1

- - : Thành viên **private**
- + : Thành viên **public**
- # : Thành viên **protected**
- * : Con trỏ đến đối tượng
- `QString, QThread` : Lớp trong Qt
- `*(name)*` : Execute(name)Command (Ví dụ ExecuteCPUCoreCommand)

Để biểu diễn các chức năng chính và mối quan hệ giữa chúng, chúng tôi sử dụng:

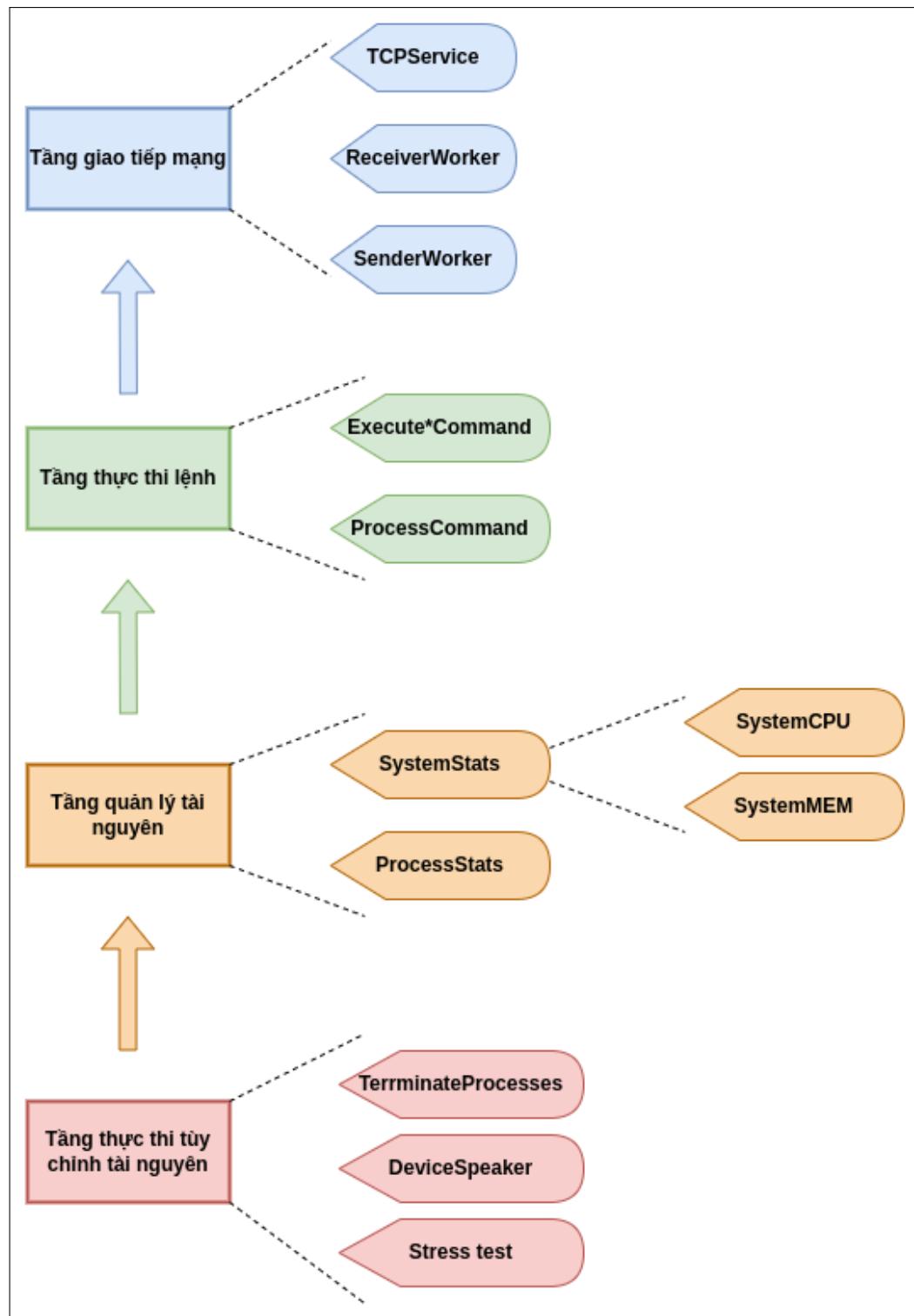
1. **Sơ đồ Use Case - Hình 4.1:** dùng để mô tả các chức năng mà Linux App cung cấp như thu thập dữ liệu, kết nối với Windows App, tạo tải giả lập, kết thúc tiến trình và phát cảnh báo.
2. **Sơ đồ lớp – Hình 4.2:** thể hiện cách tổ chức các thành phần nội bộ cũng như mối quan hệ giữa chúng. Chúng tôi đã chủ động đổi tượng hóa các chức năng, thiết kế các thành phần và xác định rõ ràng mối liên kết giữa chúng nhằm hỗ trợ hiệu quả cho quá trình hiện thực hóa phân hệ Linux App. Chi tiết thiết kế của sơ đồ lớp được trình bày trong Bảng 4.1, trong đó mỗi lớp được mô tả với các thuộc tính và phương thức tương ứng để thực hiện nhiệm vụ riêng. Ý tưởng tổ chức sơ đồ được xây dựng theo hướng phân tầng, cụ thể được minh họa qua Hình 4.3.

Các sơ đồ thiết kế không chỉ giúp hình dung tổng quan Linux App mà còn đóng vai trò định hướng cho quá trình hiện thực hóa mã nguồn, giúp đảm bảo tính rõ ràng, module hóa và khả năng mở rộng của phân hệ. Dưới đây là phần mô tả chi tiết về các sơ đồ dùng để thiết kế cho phân hệ Linux App.

Trong sơ đồ Use Case Hình 4.1, chúng tôi thiết kế phân hệ Linux App xoay quanh ba nhóm chức năng chính: thu thập dữ liệu tài nguyên hệ thống IVI, giao tiếp với Windows App và thực thi các lệnh điều khiển như phát cảnh báo, kết thúc tiến trình, hoặc tạo tải giả lập.

Đối với chức năng thu thập dữ liệu tài nguyên hệ thống IVI, phân hệ Linux App đảm nhận vai trò thu thập chi tiết các thông số bao gồm:

1. **Thông số CPU:** bao gồm nhiều khía cạnh như mức độ sử dụng CPU toàn hệ thống, mức sử dụng CPU trên từng lõi (core), tần suất hoạt động và nhiệt độ của từng lõi



Hình 4.3 Các tầng thiết kế của sơ đồ lớp Linux App.

cũng như của toàn bộ CPU. Các thông số này phản ánh trực tiếp mức độ tiêu thụ tài nguyên xử lý của hệ thống. Do có mối liên hệ chặt chẽ với nhau, chúng tôi gom toàn bộ các chức năng liên quan vào một module thống nhất là *"Thu thập thông số CPU"*.

2. **Thông số bộ nhớ (MEM)**: chúng tôi thu thập thông tin về bộ nhớ RAM và bộ nhớ Swap, bao gồm tổng dung lượng đang của hệ thống IVI và dung lượng đang bị chiếm dụng hiện tại. Sau đó chúng tôi tính toán mức phần trăm dựa dung lượng bộ nhớ đã thu thập được để có cái nhìn trực quan hơn về mức tiêu thụ bộ nhớ hiện tại của hệ thống. Tất cả chức năng này được gom vào module "*Thu thập thông số MEM*" – đại diện cho tài nguyên bộ nhớ của toàn hệ thống.
3. **Thông tin tiến trình**: mỗi tiến trình đang hoạt động trên hệ thống được thu thập thông tin bao gồm phần trăm CPU và MEM mà tiến trình đó tiêu thụ, cùng với các thông tin định danh như PID (Process ID), User (người sở hữu tiến trình), và PName (tên tiến trình). Các dữ liệu này được nhóm trong module "*Thu thập tải của các tiến trình*" và từ đó hợp nhất thành module tổng quát hơn là "**Thu thập tải của toàn hệ thống**".

Tại nhóm chức năng giao tiếp với phân hệ Windows App, dữ liệu tài nguyên sau khi được thu thập sẽ được truyền về phân hệ Windows App để hiển thị, lưu trữ và xử lý. Để phục vụ mục tiêu này, chúng tôi thiết kế module "*Giao tiếp với Windows App*", bao gồm hai chức năng chính:

1. **Truyền dữ liệu đến Windows App**: các gói dữ liệu chứa thông số tài nguyên (CPU, MEM, tiến trình...) được đóng gói và gửi đi định kỳ thông qua kết nối TCP đã thiết lập.
2. **Nhận dữ liệu và tín hiệu điều khiển từ Windows App**: phân hệ Linux App tiếp nhận lệnh điều khiển từ Windows App như: kết thúc một tiến trình cụ thể, phát cảnh báo âm thanh khi quá tải, hoặc khởi tạo và dừng Stress test.

Cuối cùng trong mô tả sơ đồ Use case là nhóm chức năng điều khiển và phản hồi hệ thống, các chức năng thuộc nhóm này giúp Linux App chủ động can thiệp vào hệ thống IVI theo yêu cầu từ người dùng thông qua Windows App hoặc từ chính bản thân Windows App:

1. **Kết thúc tiến trình**: sau khi tiến hành đánh giá mức độ ưu tiên và tài nguyên tiêu thụ, phân hệ Windows App có thể yêu cầu kết thúc tiến trình không quan trọng. Linux App sẽ tiếp nhận lệnh này và thực thi trực tiếp trên hệ thống IVI.
2. **Phát cảnh báo**: khi Windows App phát hiện hệ thống IVI quá tải hoặc sắp chạm ngưỡng quá tải, một tín hiệu điều khiển sẽ được gửi đến Linux App để kích hoạt âm thanh cảnh báo qua loa ngoài (Speaker).

3. **Stress test:** bao gồm hai thao tác, khởi động và dừng quá trình tạo tải giả lập bằng cách sinh ra các tiến trình tiêu tốn CPU và bộ nhớ nhằm mô phỏng trạng thái quá tải trong quá trình thử nghiệm. Những hành động này cũng được kích hoạt thông qua tín hiệu điều khiển từ Windows App.

Sau khi đã phân tích chi tiết các chức năng và mối liên hệ giữa chúng từ sơ đồ Use case Hình 4.1, chúng tôi tiếp tục trình bày nội dung trong sơ đồ lớp (Hình 4.2), nhằm làm rõ cấu trúc tổ chức của phân hệ Linux App, qua đó đặt nền móng vững chắc cho quá trình hiện thực hóa và triển khai phần mềm ở các mục tiếp theo.

Sơ đồ phân tầng ở Hình 4.3 thể hiện rõ cách tổ chức các lớp trong phân hệ Linux App theo từng chức năng, được chia thành bốn tầng rõ ràng:

1. **Tầng giao tiếp mạng:** mục tiêu của tầng này là quản lý kết nối TCP/IP với phân hệ Windows App. Các lớp chính bao gồm `TCPService`, đây là lớp trung tâm quản lý socket TCP, tiếp nhận tín hiệu và phân phối đến các worker phù hợp. `ReceiverWorker`, nhận và xử lý lệnh điều khiển từ Windows App. `SenderWorker` gửi dữ liệu tài nguyên (CPU, RAM, tiến trình...) đến Windows App định kỳ.
2. **Tầng thực thi lệnh:** đây là tầng được thiết kế để quản lý các commands shell, phân phối lệnh đến các chức năng tương ứng và "chạy" các lệnh đó. Cụ thể, các commands shell được triển khai bên trong các lớp `Execute*Command` và được kế thừa bởi các đối tượng cần triển khai tương tác với hệ thống IVI - các đối tượng này được thiết kế tại tầng quản lý tài nguyên và tầng thực thi tùy chỉnh tải. Khi các đối tượng muốn thay đổi trạng thái hệ thống IVI, chúng chỉ cần "gọi" lớp `ProcessCommand` để triển khai chức năng mà chúng đảm nhiệm.
3. **Tầng quản lý tài nguyên:** là tầng sử dụng các câu lệnh được kế thừa từ tầng thực thi để triển khai thu thập và lưu trữ tạm thời dữ liệu tài nguyên của hệ thống, được thiết kế theo nguyên tắc code sạch (clear code), mỗi lớp được thiết kế tinh chỉnh, dễ triển khai, dễ bảo trì và mở rộng, mỗi lớp chỉ thực thi một nhiệm vụ duy nhất. Các lớp chính bao gồm `SystemStats` và `ProcessesStats`, ở mức chi tiết hơn chúng tôi thiết kế thêm các đối tượng con thuộc tính của lớp này thành hai lớp con là `SystemCPU` - đại diện cho các thông số CPU của hệ thống IVI - và `SystemMEM` - đại diện cho các thông số bộ nhớ của hệ thống IVI. Chi tiết trong `SystemCPU` thì tiếp tục được triển khai thêm các đối tượng con bao gồm `CPUGeneral` đại diện cho các thông số CPU của toàn hệ thống và `CPUCore` đại diện cho các thông số của các core CPU. Đối với `ProcessesStats`, chúng tôi đổi tượng hóa mỗi process để có thể dễ dàng

quản lý và mở rộng chức năng.

4. **Tầng thực thi tùy chỉnh tài nguyên:** tại đây có bao gồm các lớp `TerminateProcesses` - dùng để kết thúc các tiến trình theo tín hiệu nhận được - `DeviceSpeaker` - dùng để phát âm thanh cảnh báo khi nhận được tín hiệu hệ thống IVI đang quá tải hoặc chạm ngưỡng quá tải - và `Stress test` để khởi tạo và kết thúc tải giả lập cho kiểm thử khả năng chịu tải của hệ thống IVI.

Sơ đồ đầy đủ và chi tiết được thể hiện tại Hình 4.2 cho thấy rõ mối liên hệ giữa các lớp trong từng tầng, cách thiết kế này cho chúng tôi nhận thức rõ ràng cấu trúc phần mềm của phân hệ Linux App. Các lớp được thiết kế rõ ràng, tuân thủ nguyên lý đơn nhiệm (Single Responsibility) - mỗi lớp đảm nhiệm duy nhất một chức năng cụ thể, giúp giảm thiểu sự phụ thuộc và tăng tính ổn định cho toàn hệ thống. Việc sử dụng các lớp như `SystemCPU`, `SystemMEM`, `ProcessStats` giúp dễ dàng mở rộng khi cần bổ sung thêm loại tài nguyên khác (như disk hay network), cùng với đó việc phân tách dữ liệu hệ thống tổng quát (`SystemStats`) và tiến trình cụ thể (`ProcessStats`) giúp dễ quản lý, tái sử dụng và kiểm thử từng thành phần riêng biệt một cách dễ dàng.

Từ những thiết kế chi tiết ở trên, có thể khẳng định rằng chúng tôi đã xây dựng được một kiến trúc phần mềm vững chắc, sẵn sàng để triển khai Linux App. Trong mục tiếp theo, chúng tôi sẽ trình bày chi tiết quy trình hiện thực hóa phân hệ này, cùng với những khó khăn gặp phải và cách chúng tôi đã giải quyết nhằm đạt được hiệu quả triển khai cao nhất.

4.3. Triển khai Linux App

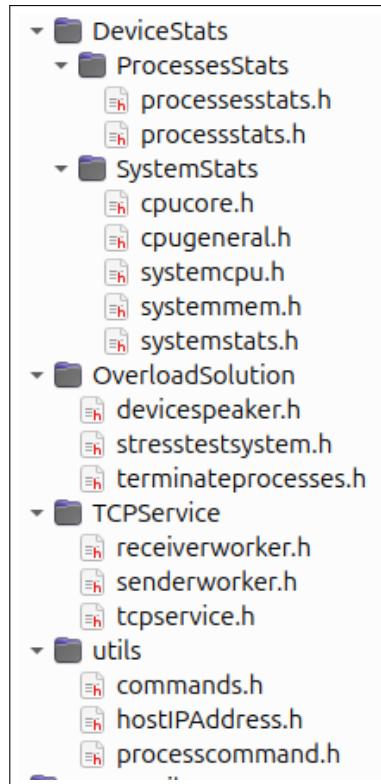
4.3.1. Cách tiếp cận

Trong quá trình triển khai phân hệ Linux App, chúng tôi áp dụng nguyên tắc thiết kế hướng đối tượng (như trình bày ở nội dung trước), tổ chức mã nguồn theo từng module chức năng riêng biệt nhằm đảm bảo khả năng bảo trì, mở rộng và dễ kiểm thử.

Chúng tôi lựa chọn **ngôn ngữ lập trình C++** kết hợp với các module trong **Qt Framework**⁶ và giao tiếp với hệ điều hành thông qua các **commands shell**. Toàn bộ mã nguồn được chia thành các nhóm lớp theo từng chức năng đã trình bày trong sơ đồ lớp.

Hình 4.4 mô tả rõ cấu trúc mã nguồn trong phân hệ Linux, cấu trúc này được giải thích chi tiết tại Bảng 4.2.

⁶Trong khi Windows App được phát triển với Qt Quick nhằm đáp ứng yêu cầu giao diện đồ họa, thì Linux App được triển khai bằng Qt Console do không có yêu cầu về giao diện người dùng, lựa chọn này giúp giảm thiểu kích thước của phân hệ, đồng thời mang lại giải pháp phù hợp và tối ưu hơn khi triển khai Linux App.

**Hình 4.4** Cấu trúc mã nguồn của phân hệ Linux App.**Bảng 4.2** Mô tả các thành phần chính trong cấu trúc mã nguồn phân hệ Linux App.

Module	File	Mô tả
SystemStats	cpucore.h	Lớp đơn vị dùng để đối tượng hóa cho từng lõi CPU.
	cpugeneral.h	Lớp đơn vị dùng để đối tượng hóa cho CPU toàn hệ thống.
	systemcpu.h	Thu thập và xử lý và lưu trữ tạm thời thông số CPU của hệ thống IVI.
	systemmem.h	Thu thập và xử lý và lưu trữ tạm thời thông số bộ nhớ (RAM và Swap) của hệ thống IVI.
	systemStats.h	Lớp khởi tạo và điều phối đối tượng CPU và MEM.
ProcessesStats	processstats.h	Lớp đơn vị dùng để đối tượng hóa tiến trình.
	processsestats.h	Thu thập và quản lý thông tin các tiến trình đang chạy.

OverloadingSolution	terminateprocesses.h devicespeaker.h stresstestsystem.h	Lớp thực thi kết thúc tiến trình. Lớp thực thi phát âm cảnh báo. Lớp thực thi Stress test.
Network	tcpservice.h receiverworker.h senderworker.h	Quản lý kết nối TCP với Windows App, xử lý giao tiếp hai chiều. Điều phối chức năng tương ứng với dữ liệu nhận được. Đóng gói dữ liệu tài nguyên thu thập và điều phối đến lớp TCPSERVICE.
Utility	commands.h processcommand.h	Quản lý các commands shell. Thực thi các commands shell.

4.3.2. Thu thập dữ liệu tài nguyên

Như đã trình bày, việc thu thập dữ liệu tài nguyên được thực hiện thông qua các commands shell, cụ thể các loại tài nguyên được thu thập được minh họa tại Hình 4.5. Các lệnh này được thực thi trong Linux App thông qua module QProcess của Qt, giúp quản lý linh hoạt đầu ra/đầu vào một cách hiệu quả.

Đối với bộ nhớ, chúng tôi cần hai loại dữ liệu là RAM (Mem) và Swap. Trong hệ điều hành Linux, có thể sử dụng lệnh `free -m` để lấy các thông tin này:

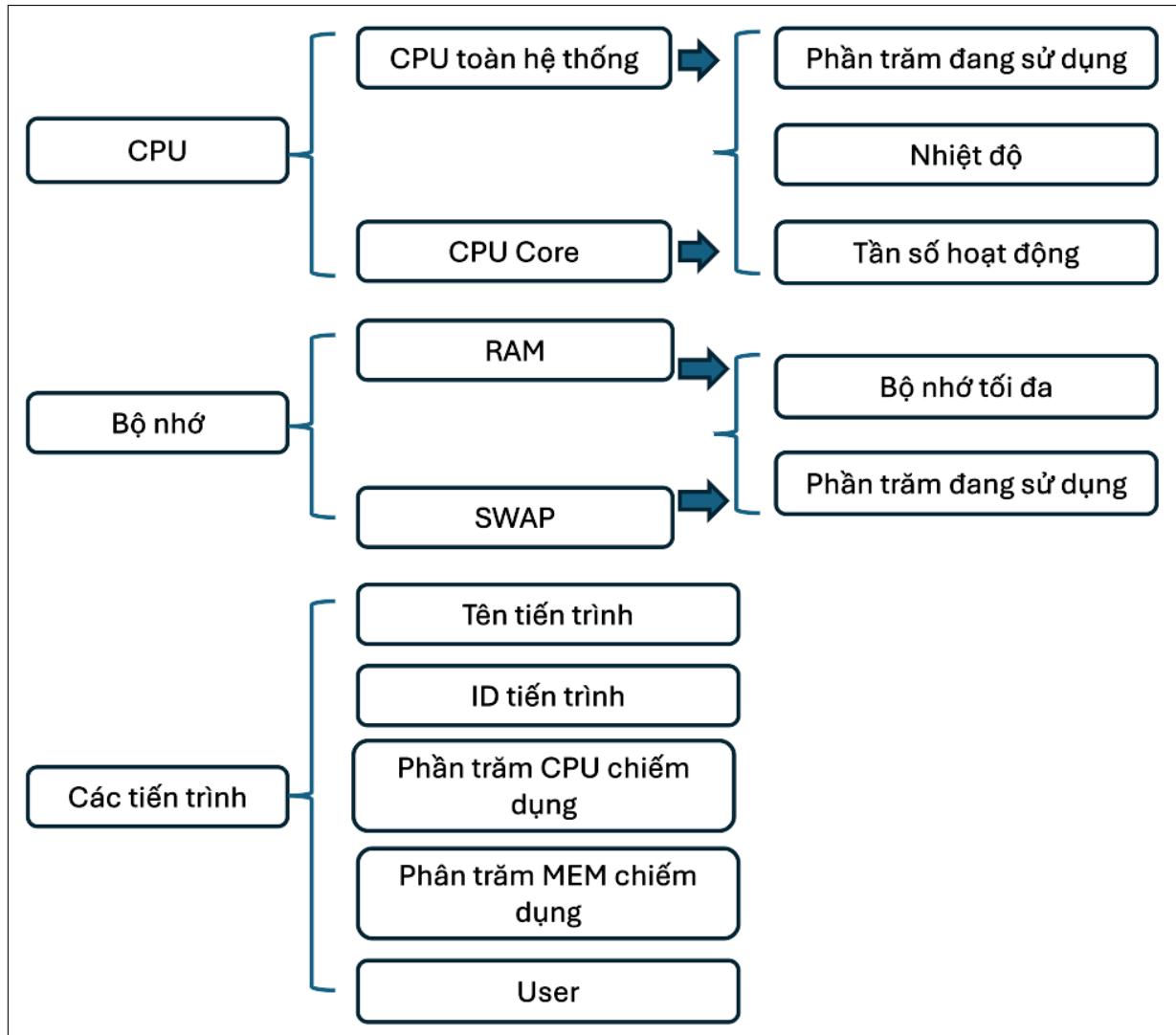
```
$ free -m
      total        used        free      shared  buff/cache   available
Mem:   15648       3193       9018        611       4381      12455
Swap:   4095          0       4095
```

Các thông số có thể truy xuất từ lệnh `free -m` bao gồm tổng bộ nhớ của hệ thống IVI (total), lượng bộ nhớ đang sử dụng (used) và bộ nhớ trống (free), các thông số này đều có đơn vị là MB (megabyte). Như biểu diễn tại Hình 4.5, chúng tôi cần thông số tổng bộ nhớ và bộ nhớ đang bị chiếm dụng của từng hai loại, để trích xuất thông số như mong muốn, có thể sử dụng lệnh `awk 'print $N'`. Kết quả cuối cùng ta sẽ có lệnh như bên dưới để thu thập các thông số về RAM và Swap của toàn hệ thống IVI.

```
$ free -m | awk 'NR >= 2 { print $3 }'
$ free -m | awk 'NR >= 2 { print $2 }'
```

Trong đó:

- `NR>=2` dùng để lấy các hàng lớn hơn 2 (hàng Mem và Swap).

**Hình 4.5** Các thông số dữ liệu Linux App thu thập.

- \$N dùng để chỉ định cột cần trích xuất. Với \$3 để trích xuất dung lượng đang sử dụng và \$2 để trích xuất dung lượng tối đa của RAM và Swap.

Đối với CPU, chúng tôi cần các thông số về phần trăm sử dụng, nhiệt độ và tần số hoạt động của CPU trên toàn hệ thống và trên từng lõi. Các lệnh được sử dụng để thu thập các thông số này như sau:

```

$ nproc
$ mpstat -P ALL 1 1 | awk 'NR>3 && NR<13 {print 100-$NF}'
$ sensors | grep 'CPU' | awk '{print $NF}'
$ sensors | awk '/Core/ {print $3}'
$ lscpu | grep 'MHz' | awk 'NR>1 {print $NF}'
$ lscpu | grep 'MHz' | awk 'NR==1 {print $NF}'
$ cat /sys/devices/system/cpu/cpu*/cpufreq/scaling_cur_freq
  
```

Với nproc, chúng tôi có thể biết được số lượng lõi CPU trên hệ thống IVI, lệnh này

chỉ được thực thi một lần ngay thời điểm Linux App được khởi động. Lệnh `mpstat -P ALL 1 1` dùng để thu thập phần trăm CPU đang sử dụng trung bình trong 1 giây, tăng độ chính xác cho dữ liệu thu thập.

Chúng tôi thu thập nhiệt độ của CPU và nhiệt độ trên từng lõi thông qua lệnh `sensors`, với `sensors | grep CPU` `awk '{print $NF}'` được dùng để lấy nhiệt độ của CPU toàn hệ thống và `sensors | awk /Core/ {print $3}` dùng để lấy nhiệt độ trên từng lõi.

Đối với tần số hoạt động của CPU, chúng tôi tiến hành lấy tần số hoạt động tối đa và tối thiểu thông qua lệnh `lscpu | grep MHz` `awk NR>1 {print $NF}'`, lệnh này chỉ được thực thi tại thời điểm khởi động Linux App, đối với tần số hoạt động hiện tại của CPU thì có thể thu thập thông qua lệnh `lscpu | grep MHz` `awk NR==1 {print $NF}'`, kết quả của lệnh này cho chúng tôi biết CPU đang hoạt động bao nhiêu phần trăm, khi kết hợp với giá trị tối đa và tối thiểu về tần số hoạt động thì có thể biết CPU đang hoạt động với tần số nào. Lệnh cuối cùng là lệnh dùng để thu thập tần số hoạt động trên từng lõi CPU, trên thực tế không có một lệnh trực tiếp để thu thập dữ liệu này, vì vậy chúng tôi đã thực hiện đọc dữ liệu từ tệp hệ thống thông qua lệnh `cat` để lấy dữ liệu tần số hoạt động trên từng lõi CPU.

Dữ liệu cuối cùng cần thu thập là các thông số của các tiến trình đang chạy trên hệ thống IVI. Thông qua lệnh `ps`, chúng tôi có thể thu thập các dữ liệu này, với các tham số `comm`, `user`, `pid`, `%cpu` và `%mem` tương ứng với tên của tiến trình, đối tượng đang thực thi, ID của tiến trình, phần trăm CPU và phần trăm bộ nhớ mà tiến trình đang chiếm dụng.

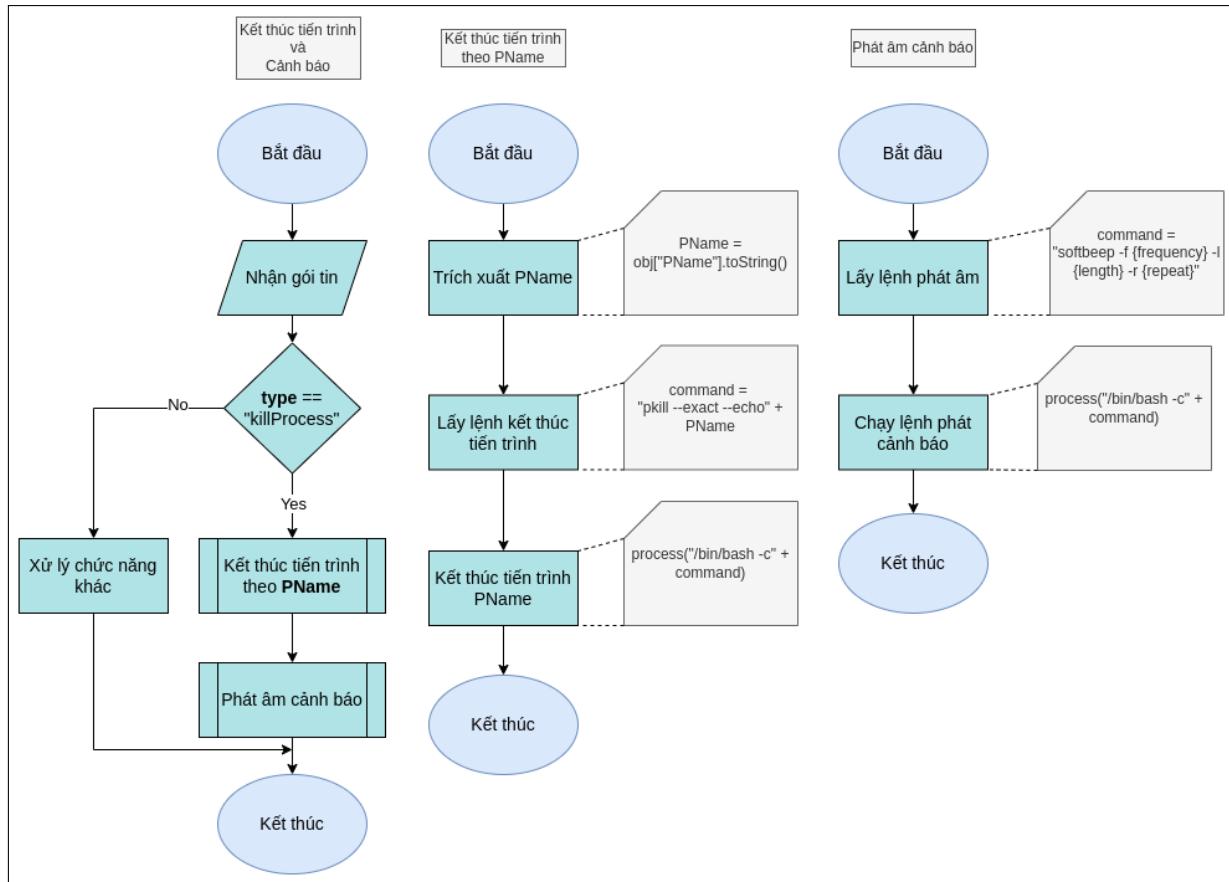
```
$ ps -eo comm,user,pid,%cpu,%mem | awk 'NR>1'
```

Việc áp dụng các lệnh này vào mã nguồn được thực hiện thông qua QProcess, giúp đảm bảo Linux App có thể tự động thu thập dữ liệu theo chu kỳ và gửi về Windows App một cách liên tục và chính xác.

4.3.3. Kết thúc tiến trình và cảnh báo đến người dùng

Chức năng kết thúc tiến trình trong Linux App được thiết kế để xử lý các tiến trình tiêu tốn quá nhiều tài nguyên, góp phần cân bằng tải hệ thống IVI trong trường hợp quá tải.

Luồng hoạt động cụ thể được thể hiện tại Hình 4.6, khi Linux App nhận được một gói tin từ Windows App, phân hệ này sẽ kiểm tra trường `type` trong gói tin. Nếu giá trị này



Hình 4.6 Luồng kết thúc tiến trình trên Linux App.

là "**”killProcess”**", tiến trình kết thúc sẽ được thực hiện như sau:

1. Xác định tiến trình cần kết thúc thông qua trường **PName**. Trong quá trình thử nghiệm, chúng tôi phát hiện rằng việc sử dụng **PID** để kết thúc tiến trình không đảm bảo hiệu quả vì một tiến trình có thể sinh ra nhiều PID (liên quan đến các luồng hoặc tiến trình con). Trong khi đó, việc sử dụng **PName** sẽ đảm bảo tất cả tiến trình con liên quan đều bị kết thúc triệt để.
2. Thực thi lệnh kết thúc tiến trình bằng cách sử dụng command shell **pkill PName**. Lệnh này được triển khai trong mã nguồn bằng cách sử dụng QProcess, đảm bảo quá trình được gọi đúng và xử lý lỗi nếu có.
3. Phát cảnh báo âm thanh sau khi tiến trình được kết thúc, giúp người lái xe (đang sử dụng hệ thống IVI) dễ dàng nhận biết hệ thống IVI đang trong tình trạng quá tải. Chức năng này được thực hiện thông qua lệnh **softbeep -f { frequency } -l { length } -r { repeat }**.
4. Nếu gói tin không phải loại **killProcess**, Linux App sẽ chuyển hướng đến xử lý các chức năng khác như Stress test (được thể hiện rõ tại §4.4).

Nhờ vào thiết kế cấu trúc gói tin hợp lý và xử lý tập trung dựa trên `type`, việc tích hợp các chức năng xử lý trên Linux App trở nên đơn giản, hiệu quả, dễ mở rộng và có thể tái sử dụng trong nhiều trường hợp thực tế khác nhau trong vận hành hệ thống IVI.

4.3.4. Vai trò của Linux App trong Stress test hệ thống IVI

Stress test được chúng tôi triển khai thông qua lệnh `stress` — một command shell thường được sử dụng để kiểm tra khả năng chịu tải của hệ thống trong môi trường Linux — từ đó cho phép mô phỏng tình trạng tài nguyên hệ thống (CPU, bộ nhớ) bị chiếm dụng bằng cách tạo ra các tiến trình giả lập tiêu tốn tài nguyên.

Một vấn đề mà chúng tôi gặp phải trong quá trình phát triển chức năng này là khi triển khai Stress test trên luồng thực thi chính của Linux App. Trong lúc kiểm thử tải, các tiến trình giả lập khiến luồng chính bị chặn hoàn toàn, dẫn đến toàn bộ logic giám sát và điều khiển bị ngưng lại. Để khắc phục, chúng tôi đã thiết kế lại cơ chế xử lý, cho phép Stress test được thực thi trên một luồng riêng biệt, tách biệt hoàn toàn với các luồng xử lý chính, nhằm đảm bảo các chức năng khác vẫn hoạt động ổn định trong suốt quá trình kiểm thử. Khi thời gian kiểm thử kết thúc hoặc khi có tín hiệu dừng từ Windows App, Linux App sẽ kết thúc toàn bộ tiến trình kiểm thử bằng lệnh:

```
$ killall stress
```

Khi nhận được gói tin với trường `type` là `startStress`, Linux App sẽ khởi chạy lệnh `stress` với các tham số như sau:

```
$ stress --vm {numberOfTask} --vm-bytes {memUsage} \
    --cpu {numberOfCore} --timeout {seconds}
```

Lệnh trên sẽ tạo ra các tiến trình giả lập nhằm chiếm dụng tài nguyên bộ nhớ (`--vm`), CPU (`--cpu`) trong khoảng một thời gian xác định (`--timeout`).

Trong đó:

- `numberOfTask` là số lượng tiến trình giả lập.
- `memUsage` là lượng bộ nhớ mà các tiến trình này chiếm dụng.
- `numberOfCore` là số lượng lõi CPU mà các tiến trình sẽ sử dụng với hiệu suất 100%.
- `seconds` là thời gian thực hiện Stress test, tính bằng giây.

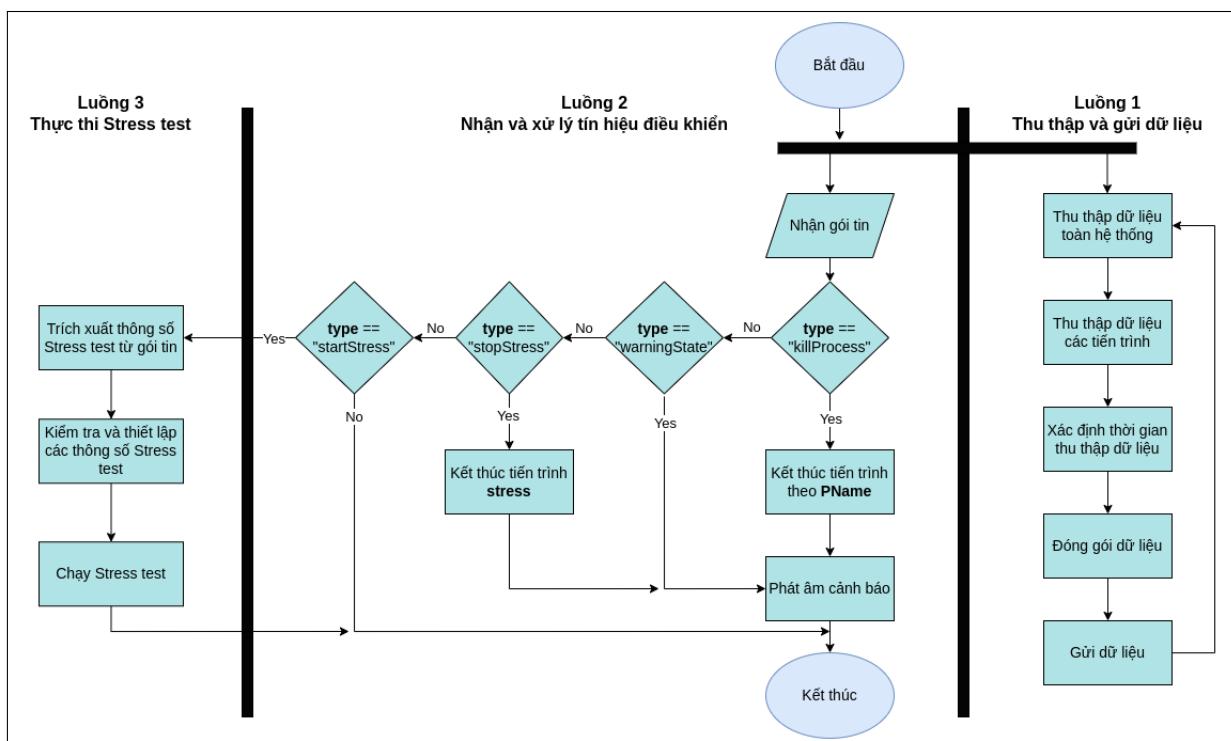
Quá trình thực thi Stress test diễn ra hoàn toàn ở phía Linux App (Windows App chỉ đóng vai trò gửi lệnh điều khiển cho chức năng này), và mặc dù được thực hiện ở luồng

riêng biệt, nhưng kết quả lại ảnh hưởng đến toàn bộ hiệu năng của hệ thống. Qua đó, chúng tôi có thể đánh giá:

1. Mức độ ổn định và khả năng xử lý logic của phần mềm trong điều kiện tải nặng.
2. Khả năng phản hồi và độ trễ của Linux App khi tài nguyên bị chiếm dụng lớn trên hệ thống IVI.
3. Hiệu quả của tính năng cân bằng tải trong phần mềm.

Chức năng này không chỉ giúp đánh giá hiệu suất của Linux App mà còn phản ánh khả năng chịu tải và độ tin cậy tổng thể của hệ thống IVI trong các điều kiện vận hành khắc nghiệt.

4.4. Thiết kế các luồng hoạt động



Hình 4.7 Luồng hoạt động trên Linux App

Linux App được thiết kế với kiến trúc đa luồng nhằm đảm bảo tính ổn định, phản hồi nhanh, và không bị gián đoạn trong quá trình vận hành hệ thống IVI. Cụ thể, phân hệ này được chia thành ba luồng chính với vai trò riêng biệt như thể hiện tại Hình 4.7:

1. **Luồng 1: Thu thập và gửi dữ liệu** - Luồng này thực hiện việc thu thập dữ liệu tài

nguyên hệ thống IVI định kỳ, xử lý định dạng và gửi dữ liệu về cho Windows App thông qua giao thức TCP/IP.

2. **Luồng 2: Nhận và xử lý tín hiệu điều khiển** - Luồng này đảm nhiệm việc lắng nghe các gói tin điều khiển gửi từ Windows App, bao gồm các yêu cầu như *kết thúc tiến trình, cảnh báo người dùng* và lệnh *bắt đầu hoặc dừng Stress test*. Sau khi nhận gói tin, luồng này sẽ gọi các chức năng tương ứng để xử lý.
3. **Luồng 3: Thực thi Stress test** - Đây là luồng chỉ khởi chạy khi nhận được lệnh bắt đầu Stress tess (`startStress`) tại luồng 2. Việc tách riêng luồng này giúp quá trình Stress test không ảnh hưởng đến các chức năng giám sát và điều khiển đang hoạt động song song.

Việc phân tách chức năng thành các luồng riêng biệt mang lại các lợi ích quan trọng. Điều này đảm bảo Linux App **phản ứng tức thì** khi có lệnh điều khiển từ Windows App. Cho phép **giám sát hệ thống liên tục**, ngay cả trong lúc đang thực hiện kiểm thử hiệu suất (Stress test). Và tránh hiện tượng **chặn lấn nhau giữa các chức năng**, đặc biệt trong điều kiện tài nguyên bị chiếm dụng cao.

Cách tiếp cận đa luồng này không chỉ tối ưu hóa hiệu năng phân hệ, mà còn giúp cải thiện độ tin cậy và khả năng mở rộng cho Linux App, khắc phục được những vấn đề gặp phải trong quá trình phát triển phân hệ này.

4.5. Kết luận chương

Trong chương này, chúng tôi đã trình bày chi tiết quá trình triển khai phân hệ Linux App, bao gồm từ mục tiêu, kiến trúc phần mềm, cách tổ chức mã nguồn đến cách hiện thực hóa các chức năng chính như thu thập dữ liệu tài nguyên, giao tiếp mạng, điều khiển tiến trình, phát cảnh báo và thực hiện Stress test.

Quá trình triển khai gấp một số vấn đề đáng kể, đặc biệt là hiện tượng chặn luồng (blocking) khi các chức năng được xử lý trên cùng một luồng chính. Điều này làm gián đoạn hoạt động của toàn hệ thống và khiến ứng dụng phản hồi không đúng theo yêu cầu. Để giải quyết, chúng tôi đã thiết kế lại kiến trúc theo hướng đa luồng, phân tách rõ ràng các chức năng độc lập và triển khai mỗi chức năng trên một luồng riêng biệt. Giải pháp này đã giúp đảm bảo hệ thống hoạt động ổn định, phản ứng kịp thời với các lệnh điều khiển, và duy trì khả năng giám sát trong suốt quá trình kiểm thử tái.

Bên cạnh đó, việc áp dụng tư duy thiết kế hướng đối tượng kết hợp với công cụ Qt và các commands shell Linux đã giúp chúng tôi hiện thực hóa các chức năng một cách rõ

ràng, module hóa và có tính mở rộng cao. Kiến trúc phần mềm của Linux App được xây dựng theo từng tầng chức năng, đảm bảo khả năng bảo trì và nâng cấp trong tương lai.

CHƯƠNG 5: KIỂM THỬ VÀ ĐÁNH GIÁ KẾT QUẢ

Chương này sẽ trình bày nội dung của một số trường hợp kiểm thử, từ đó giúp đánh giá khả năng hoạt động của Phần mềm trong các điều kiện khác nhau.

5.1. Kiểm thử đơn vị

Về phía Windows App, chúng tôi tiến hành kiểm thử từng nhóm (module) chức năng cụ thể như: chức năng xử lý dữ liệu, phát hiện quá tải, xử lý quá tải, Stress test.

5.1.1. Kiểm thử khả năng xử lý dữ liệu

```
--Dữ liệu thô nhận được từ Linux App--
[{"ProcessesStats": [{"ID": "1182", "PCPUUsagePercent": 0, "PID": 1182, "PMEMUsageMB": 15.648000717163086, "PMEMUsagePercent": 0, "Name": "snapd", "User": "root"}, {"ID": "1674", "PCPUUsagePercent": 0, "PID": 1674, "PMEMUsageMB": 15.648000717163086, "PMEMUsagePercent": 0.1, "Name": "mariadb", "User": "mysql"}, {"ID": "2509", "PCPUUsagePercent": 0.01250000186264515, "PID": 2509, "PMEMUsageMB": 0, "PMEMUsagePercent": 0, "Name": "teamviewerd", "User": "root"}, {"ID": "2758", "PCPUUsagePercent": 0.1, "PID": 2758, "PMEMUsageMB": 15.648000717163086, "PMEMUsagePercent": 0.1, "Name": "pipewire", "User": "duc-vu"}, {"ID": "2762", "PCPUUsagePercent": 0, "PID": 2762, "PMEMUsageMB": 0, "PMEMUsagePercent": 0, "Name": "wireplumber", "User": "duc-vu"}, {"ID": "2764", "PCPUUsagePercent": 0.025, "PID": 2764, "PMEMUsageMB": 15.648000717163086, "PMEMUsagePercent": 0.1, "Name": "pipewire-pulse", "User": "duc-vu"}, {"ID": "2987", "PCPUUsagePercent": 0.7749999761581421, "PID": 2987, "PMEMUsageMB": 140.8319854736328, "PMEMUsagePercent": 0.8999999761581421, "Name": "gnome-shell", "User": "duc-vu"}, {"ID": "3170", "PCPUUsagePercent": 0.0375, "PID": 3170, "PMEMUsageMB": 0, "PMEMUsagePercent": 0, "Name": "ibus-daemon", "User": "duc-vu"}, {"ID": "3233", "PCPUUsagePercent": 0, "PID": 3233, "PMEMUsageMB": 15.648000717163086, "PMEMUsagePercent": 0.1, "Name": "evolution-alarm", "User": "duc-vu"}, {"ID": "3555", "PCPUUsagePercent": 0.025, "PID": 3555, "PMEMUsageMB": 31.296001434326172, "PMEMUsagePercent": 0.2, "Name": "tracker-miner-f", "User": "duc-vu"}, {"ID": "3558", "PCPUUsagePercent": 0, "PID": 3558, "PMEMUsageMB": 1156.47999572753906, "PMEMUsagePercent": 1, "Name": "xdg-desktop-por", "User": "duc-vu"}, {"ID": "3739", "PCPUUsagePercent": 0.3, "PID": 3739, "PMEMUsageMB": 312.9599914550781, "PMEMUsagePercent": 0.2, "Name": "chrome", "User": "duc-vu"}, {"ID": "3799", "PCPUUsagePercent": 0.075, "PID": 3799, "PMEMUsageMB": 15.648000717163086, "PMEMUsagePercent": 0.1, "Name": "Xwayland", "User": "duc-vu"}, {"ID": "3839", "PCPUUsagePercent": 0, "PID": 3839, "PMEMUsageMB": 15.648000717163086, "PMEMUsagePercent": 0.1, "Name": "mutter-x11-fram", "User": "duc-vu"}, {"ID": "384", "PCPUUsagePercent": 0.0125, "PID": 384, "PMEMUsageMB": 0, "PMEMUsagePercent": 0.1, "Name": "systemd-journal", "User": "root"}, {"ID": "3868", "PCPUUsagePercent": 0.05375, "PID": 3868, "PMEMUsageMB": 1.2, "PMEMUsagePercent": 1.2, "Name": "chrome", "User": "duc-vu"}, {"ID": "3870", "PCPUUsagePercent": 0.125, "PID": 3870, "PMEMUsageMB": 78.23999786376953, "PMEMUsagePercent": 0.5, "Name": "chrome", "User": "duc-vu"}, {"ID": "3880", "PCPUUsagePercent": 0, "PID": 3880, "PMEMUsageMB": 15.648000717163086, "PMEMUsagePercent": 0.1, "Name": "chrome", "User": "root"}, {"ID": "3968", "PCPUUsagePercent": 0.0125, "PID": 3968, "PMEMUsageMB": 93.88800811767578, "PMEMUsagePercent": 0.6, "Name": "chrome", "User": "duc-vu"}, {"ID": "4044", "PCPUUsagePercent": 0.26249998807907104, "PID": 4044, "PMEMUsageMB": 281.6639709472656, "PMEMUsagePercent": 1.7999999523162842}, {"ID": "4045", "PCPUUsagePercent": 0.26249998807907104, "PID": 4045, "PMEMUsageMB": 281.6639709472656, "PMEMUsagePercent": 1.7999999523162842}]}]
```

Hình 5.1 Một phần dữ liệu thô nhận được từ Linux App

Trong Hình 5.1, mô tả một phần dữ liệu thô chưa qua xử lý, bao gồm nhiều chuỗi JSON liên tiếp được gửi từ Linux App. Tại thời điểm này, hàm xử lý dữ liệu sẽ tiến hành phân tích và ánh xạ các chuỗi JSON vào các nhóm dữ liệu tương ứng đã định nghĩa trong các lớp mô hình (Model) tại §3.1.2, nhằm phục vụ cho các chức năng xử lý tiếp theo.

Hình 5.2 minh họa kết quả sau khi phân tích dữ liệu, trong đó thông tin đã được chuyển thành dạng hiển thị trực quan và dễ quan sát hơn. Qua đó, có thể thấy được rằng chức năng này đã thực hiện tốt vai trò của nó.

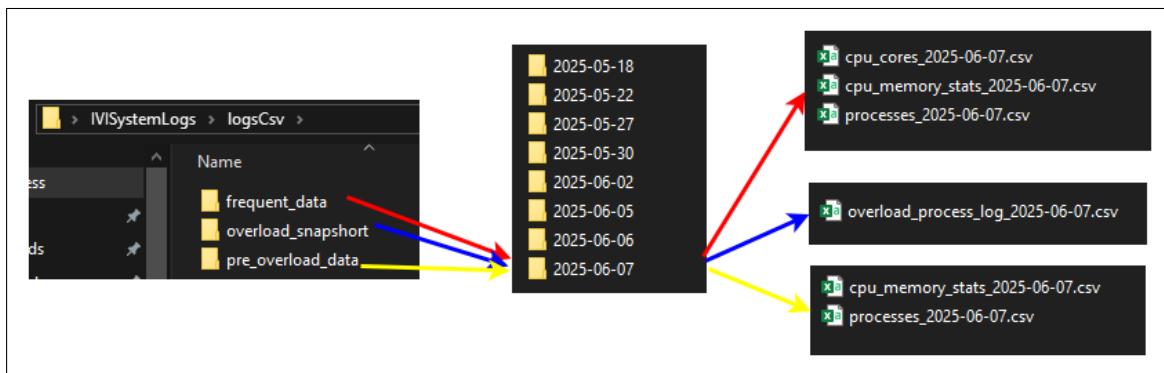
```

---Dữ liệu sau khi được xử lý---
Timestamp: "2025-06-07 11:20:25"
General CPU:
  Utilization: 1.63
  Temperature: 43
  Frequency Current: 1030
CPU Cores:
  Core 0 Usage: 0 Freq: 900.239 Temp: 42
  Core 1 Usage: 2 Freq: 899.991 Temp: 41
  Core 2 Usage: 1 Freq: 900.238 Temp: 41
  Core 3 Usage: 3.96 Freq: 900.348 Temp: 43
  Core 4 Usage: 0 Freq: 900.228 Temp: 42
  Core 5 Usage: 5 Freq: 400 Temp: 41
  Core 6 Usage: 0 Freq: 900.113 Temp: 41
  Core 7 Usage: 1 Freq: 400 Temp: 43
MEM:
  RAM Usage: 6454 MB ( 41.2449 %) of 15648
  SWAP Usage: 3134 MB ( 76.5324 %) of 4095
Processes:
  Name: "snapd" User: "root" PID: 1182 CPU(%): 0 MEM(MB): 15.648 MEM(%): 0.1
  Name: "mariadb" User: "mysql" PID: 1674 CPU(%): 0 MEM(MB): 15.648 MEM(%): 0.1
  Name: "teamviewer" User: "root" PID: 2509 CPU(%): 0.0125 MEM(MB): 0 MEM(%): 0
  Name: "pipewire" User: "duc-vu" PID: 2758 CPU(%): 0.025 MEM(MB): 15.648 MEM(%): 0.1
  Name: "wireplumber" User: "duc-vu" PID: 2762 CPU(%): 0.025 MEM(MB): 0 MEM(%): 0
  Name: "pipewire-pulse" User: "duc-vu" PID: 2764 CPU(%): 0.0625 MEM(MB): 15.648 MEM(%): 0.1
  Name: "gnome-shell" User: "duc-vu" PID: 2987 CPU(%): 0.775 MEM(MB): 140.832 MEM(%): 0.9
  Name: "ibus-daemon" User: "duc-vu" PID: 3170 CPU(%): 0.0375 MEM(MB): 0 MEM(%): 0
  Name: "evolution-alarm" User: "duc-vu" PID: 3233 CPU(%): 0 MEM(MB): 15.648 MEM(%): 0.1
  Name: "tracker-miner-f" User: "duc-vu" PID: 3555 CPU(%): 0.0625 MEM(MB): 31.296 MEM(%): 0.2

```

Hình 5.2 Dữ liệu sau khi được xử lý

5.1.2. Kiểm thử chức năng lưu trữ dữ liệu

**Hình 5.3** Dữ liệu được lưu trữ vào thư mục

Dữ liệu nhận được sẽ được lưu trữ thành các file ".csv", có thể dùng phần mềm Excel để xem nội dung bên trong. Có thể thấy ở [Hình 5.3](#), chức năng này hoạt động đúng theo luồng lưu trữ dữ liệu đã được mô tả tại [§3.2.4 Mục 4](#).

5.1.3. Kiểm thử khả năng phát hiện quá tải

Hình 5.4, minh họa kết quả đầu ra của chức năng xác định trạng thái hệ thống. Cụ thể, 10 trạng thái hoạt động gần nhất của hệ thống được ghi nhận và lưu trữ lại nhằm đưa ra kết luận chính xác nhất. Có 3 trạng thái chính: "0" biểu thị trạng thái bình thường (Normal), "1" là cảnh báo (Warning), "2" là quá tải (Overloaded). Điều này cho thấy chức năng này chạy được và có khả năng xác định được tình trạng hệ thống qua từng thời điểm.

```
[OverloadDetector] ===== Evaluate Overload Trend =====
[OverloadDetector] Last 10 States (0:Normal, 1:Warning, 2:Overloaded):
" 1. State = 0"
" 2. State = 0"
" 3. State = 0"
" 4. State = 0"
" 5. State = 0"
" 6. State = 2"
" 7. State = 2"
" 8. State = 2"
" 9. State = 2"
" 10. State = 2"
```

Hình 5.4 Đánh giá tình trạng quá tải dựa vào các trạng thái

5.1.4. Kiểm thử khả năng xử lý quá tải

Hình 5.5 minh họa kết quả lệnh kết thúc tiến trình của chức năng xử lý quá tải. Cụ thể, khi phát hiện rằng một tiến trình (trong trường hợp này là tiến trình "stress") tiêu tốn quá nhiều tài nguyên hệ thống đã gửi lệnh kết thúc tiến trình ("killProcess") tới Linux App. Qua đó cho thấy rằng chức năng này đã thực hiện đúng yêu cầu và hoạt động tốt.

```
== Kill process ==
[SystemMonitor]== Command kill_process === "{\"PName\":\"stress\",\"type\":\"killProcess\"}"
Sending command to linux: "{\"PName\":\"stress\",\"type\":\"killProcess\"}"
```

Hình 5.5 Chọn ra tiến trình phù hợp để kết thúc

5.1.5. Kiểm thử kết nối

Hình 5.6 minh họa kết quả của việc thiết lập kết nối giữa Windows App và Linux App. Cụ thể quá trình diễn ra như sau:

1. Khởi động Server: Windows App khởi tạo một server TCP, lắng nghe các kết nối

```
[SystemMonitor] Start monitoring...
TCP Server listening on Host: "172.20.10.4" and Port: 8000
[UDP] Listening for requests on port 45000
[DISCOVERY] Responded to QHostAddress("172.20.10.5") : 35843 → "SERVER:8000"
Client connected from "::ffff:172.20.10.5"
```

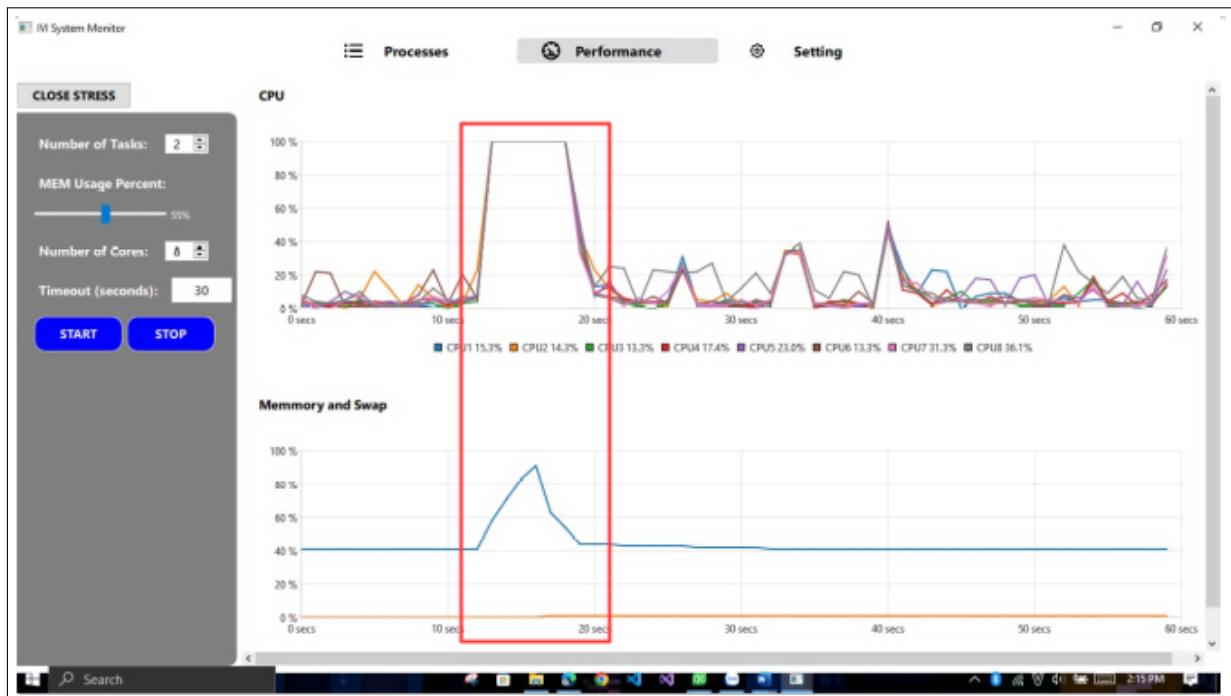
Hình 5.6 Kết nối giữa Windows App và Linux App

đến trên Host 172.20.10.4 và Port 8000. Đồng thời, server cũng mở Port 45000 để nhận yêu cầu từ Linux App.

2. Lắng nghe yêu cầu: Windows App nhận yêu cầu kết nối qua giao thức UDP trên cổng 45000.
3. Đáp ứng yêu cầu Discovery: sau khi nhận yêu cầu từ Linux App (DISCOVERY), server đáp ứng bằng cách gửi địa chỉ IP của mình (tại 172.20.10.5) cùng với cổng server (SERVER:8000).
4. Kết nối thành công: cuối cùng, Linux App đã kết nối thành công với Windows App qua địa chỉ IP 172.20.10.5 trên cổng 8000. Thông tin về kết nối cũng được in ra, xác nhận đã thiết lập thành công kết nối.

Qua đó cho thấy chức năng này hoạt động đúng với yêu cầu.

5.1.6. Kiểm thử kích hoạt Stress test

**Hình 5.7** Kiểm thử kích hoạt Stress test

Hình 5.7 minh họa kết quả của quá trình kiểm thử Stress test trên hệ thống, với mục tiêu đánh giá khả năng phản ứng và giám sát tài nguyên khi hệ thống chịu tải cao trong một khoảng thời gian ngắn.

Trong quá trình kiểm thử, người dùng có thể chủ động kích hoạt chế độ Stress test thông qua giao diện điều khiển bên trái, cụ thể: tăng số lượng tiến trình "stress" giả lập (Number of Tasks) với giá trị là 2, tăng tỷ lệ sử dụng bộ nhớ (MEM Usage Percent) lên 50%, tăng số lượng nhân xử lý CPU được sử dụng (Number of Cores) với giá trị là 8, khoảng thời gian kích hoạt (Timeout) là 30 giây và sau đó nhấn nút "START" để bắt đầu quá trình.

Ngay sau khi Stress test được kích hoạt, các tiến trình giả lập chiếm nhiều CPU và MEM được tạo ra, nhằm mô phỏng một kịch bản quá tải. Điều này được phản ánh rõ ràng qua biểu đồ bên phải: biểu đồ CPU thể hiện mức sử dụng của tất cả các lõi tăng vọt gần như đồng thời, đạt đỉnh trong khoảng thời gian từ giây 13 đến giây 23 (tại giây thứ 18, chúng tôi nhấn nút "STOP" để dừng Stress test). Biểu đồ *Memory and Swap* cũng cho thấy dung lượng bộ nhớ bị chiếm dụng tăng mạnh tương ứng với thời điểm hệ thống chịu tải nặng (khoảng thời gian được đóng khung đỏ trong hình cho thấy rõ điều đó). Khi nhấn nút "STOP", mức sử dụng tài nguyên nhanh chóng quay trở lại trạng thái bình thường, xác nhận rằng quá trình Stress test đã được triển khai đúng với mục tiêu kiểm thử.



Hình 5.8 Thay đổi đầu vào Stress test

Khi thay đổi các thông số đầu vào, mức sử dụng CPU và bộ nhớ của hệ thống IVI cũng biến đổi tương ứng. Sự thay đổi này được thể hiện rõ qua việc so sánh giữa Hình 5.7 và Hình 5.8. Cụ thể, tại Hình 5.8, chúng tôi thiết lập mức sử dụng bộ nhớ lên 80% và giảm số lỗi CPU tham gia kiểm thử xuống còn 4. Khi đó, biểu đồ CPU cho thấy số lỗi đạt công suất tối đa (100%) đã giảm, và một số lỗi chỉ hoạt động ở mức khoảng 60

Đối với bộ nhớ, khi thiết lập giá trị kiểm thử ở mức 80%, cả bộ nhớ chính (MEM) và vùng trao đổi (Swap) đều tăng mạnh. Một đặc điểm dễ nhận thấy là Swap chỉ bắt đầu tăng sau khi MEM đã bị chiếm dụng hoàn toàn (100%). Trên biểu đồ *Memory and Swap*, giá trị cực đại được ghi nhận là 100% đối với MEM và hơn 95% đối với Swap. Những quan sát này cho thấy rõ ràng việc thay đổi các thông số đầu vào dẫn đến sự thay đổi tương ứng về mức tiêu thụ tài nguyên hệ thống, qua đó xác nhận tính đúng đắn của chức năng Stress test.

5.2. Kiểm thử một số tính năng

Trong phần này, chúng tôi sẽ kiểm tra khả năng hoạt động của một số tính năng quan trọng trong các điều kiện khác nhau, cụ thể như: chức năng giám sát tài nguyên hệ thống, khả năng phát hiện quá tải hệ thống và khả năng xử lý tình trạng đó.

1. Điều kiện kiểm thử 1 (ĐKKT1)

Điều kiện: Hệ thống IVI hoạt động trong điều kiện bình thường.

Yêu cầu: Windows App và Linux App kết nối cùng một mạng.

2. Điều kiện kiểm thử 2 (ĐKKT2)

Điều kiện: Hệ thống IVI hoạt động trong điều kiện tải cao (chạy nhiều tiến trình).

Yêu cầu: Windows App và Linux App kết nối cùng một mạng.

3. Điều kiện kiểm thử 3 (ĐKKT3)

Điều kiện: Hệ thống IVI hoạt động trong điều kiện stress.

Yêu cầu: Windows App và Linux App kết nối cùng một mạng.

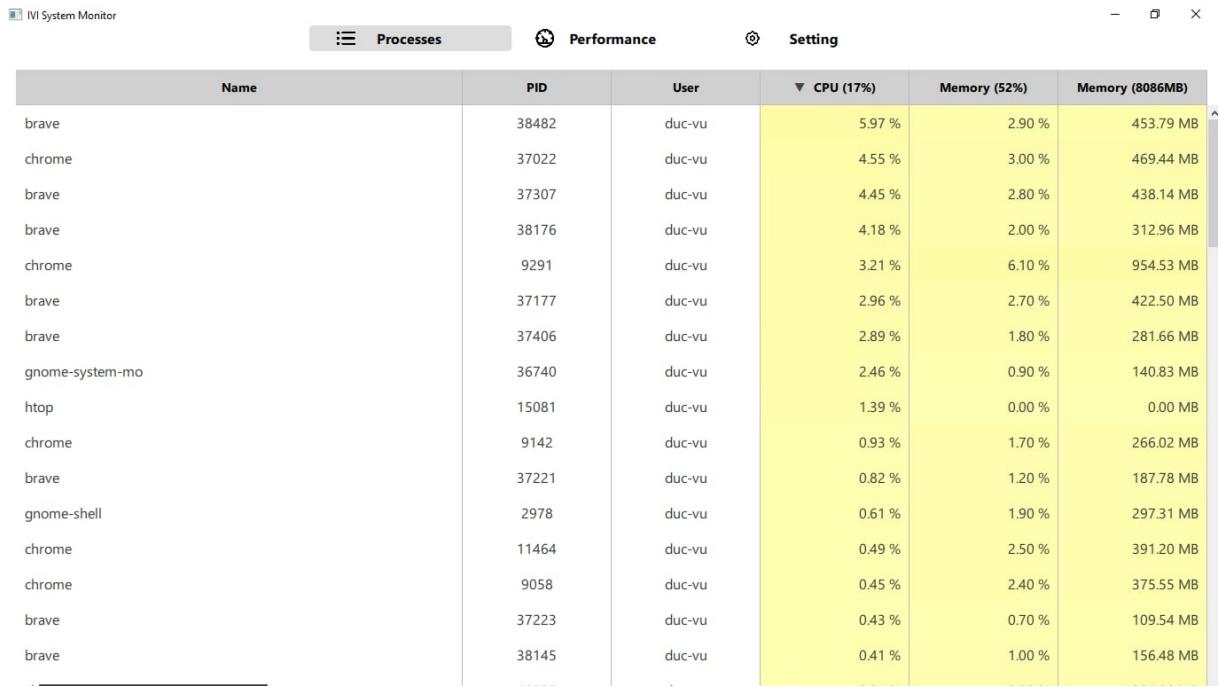
5.2.1. Đánh giá trong ĐKKT1

Tại điều kiện kiểm thử này, chúng tôi tiến hành khởi chạy hệ thống IVI trong trạng thái hoạt động bình thường với mức tải nhẹ đến vừa phải – tương đương với việc chỉ chạy một số tiến trình cơ bản hoặc ứng dụng không tiêu tốn nhiều tài nguyên. Mục tiêu của kiểm thử là đánh giá khả năng của Phần mềm trong việc thu thập và phản ánh chính

xác tình trạng tài nguyên của hệ thống IVI trên phân hệ Windows App.

Để đánh giá độ chính xác của dữ liệu được thu thập và hiển thị, chúng tôi thực hiện so sánh giữa kết quả hiển thị trên Windows App với dữ liệu từ phần mềm mặc định của hệ điều hành Linux – cụ thể là ứng dụng *System Monitor*. Đây là công cụ giám sát hệ thống tích hợp sẵn trên nhiều bản phân phối Linux, cung cấp thông tin chính xác theo thời gian thực về CPU, bộ nhớ, tiến trình và các thông số khác. Chúng tôi tiến hành theo dõi song song cả hai nguồn dữ liệu (Windows App và System Monitor) trong cùng thời điểm và cùng điều kiện chạy hệ thống để kiểm tra tính nhất quán về:

1. Tỷ lệ sử dụng CPU toàn cục và trên từng lõi.
2. Mức sử dụng RAM và Swap.
3. Số lượng và mức tiêu thụ tài nguyên của các tiến trình đang hoạt động (thể hiện tại [Hình 5.9](#)).



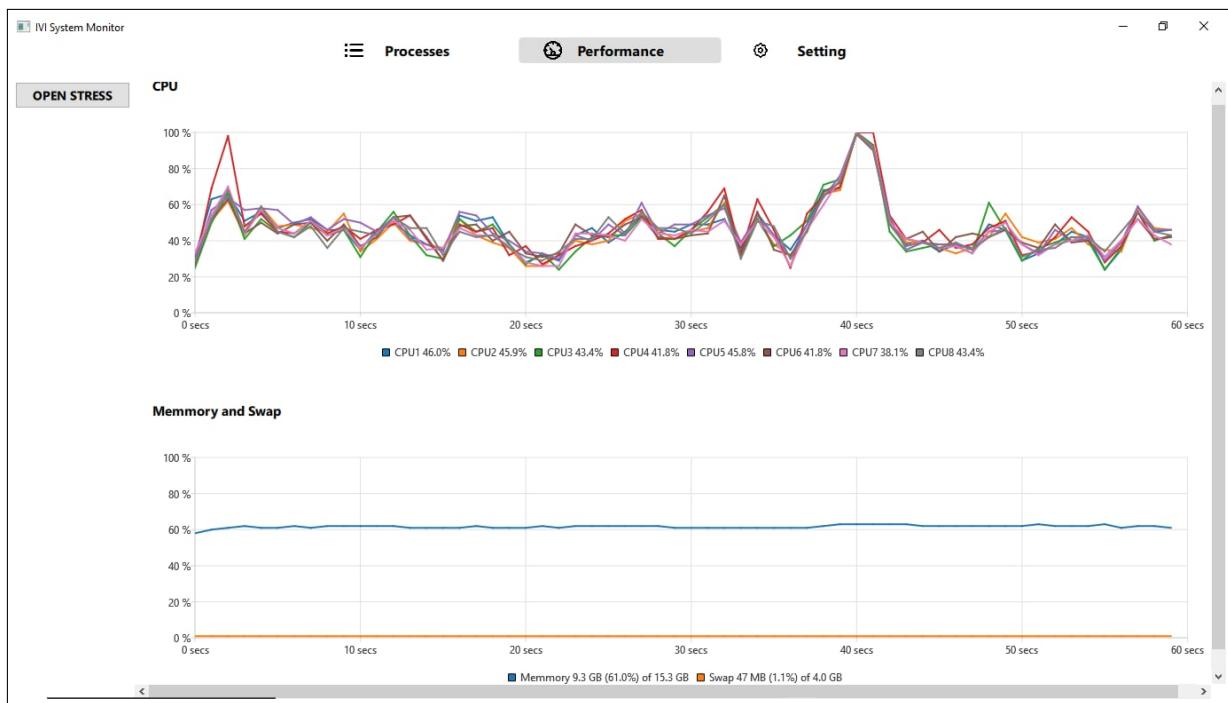
The screenshot shows the 'IVI System Monitor' application window. At the top, there are three tabs: 'Processes' (selected), 'Performance', and 'Setting'. The main area is a table with the following columns: Name, PID, User, CPU (17%), Memory (52%), and Memory (8086MB). The table lists various processes including brave, chrome, gnome-system-mo, htop, and several instances of duc-vu. The 'CPU (17%)' column is highlighted in yellow, indicating it is the current tab selected.

Name	PID	User	CPU (17%)	Memory (52%)	Memory (8086MB)
brave	38482	duc-vu	5.97 %	2.90 %	453.79 MB
chrome	37022	duc-vu	4.55 %	3.00 %	469.44 MB
brave	37307	duc-vu	4.45 %	2.80 %	438.14 MB
brave	38176	duc-vu	4.18 %	2.00 %	312.96 MB
chrome	9291	duc-vu	3.21 %	6.10 %	954.53 MB
brave	37177	duc-vu	2.96 %	2.70 %	422.50 MB
brave	37406	duc-vu	2.89 %	1.80 %	281.66 MB
gnome-system-mo	36740	duc-vu	2.46 %	0.90 %	140.83 MB
htop	15081	duc-vu	1.39 %	0.00 %	0.00 MB
chrome	9142	duc-vu	0.93 %	1.70 %	266.02 MB
brave	37221	duc-vu	0.82 %	1.20 %	187.78 MB
gnome-shell	2978	duc-vu	0.61 %	1.90 %	297.31 MB
chrome	11464	duc-vu	0.49 %	2.50 %	391.20 MB
chrome	9058	duc-vu	0.45 %	2.40 %	375.55 MB
brave	37223	duc-vu	0.43 %	0.70 %	109.54 MB
brave	38145	duc-vu	0.41 %	1.00 %	156.48 MB

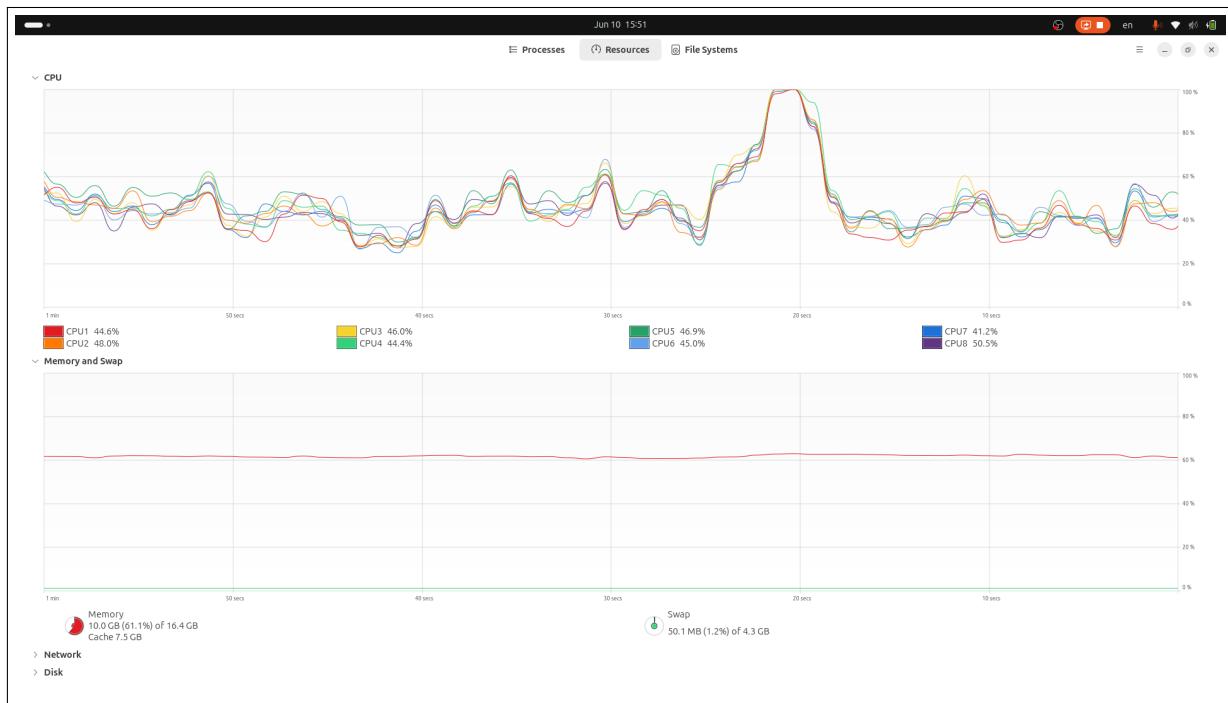
Hình 5.9 Thông tin tiến trình và tài nguyên chiếm dụng trong ĐKKT1

[Hình 5.10](#) (Windows App) và [Hình 5.11](#) (System Monitor) cho thấy dữ liệu được hiển thị trên Windows App gần như trùng khớp với thông tin từ System Monitor. Vì Linux App thu thập dữ liệu với chu kỳ 1s và độ trễ trong quá trình truyền/nhận dữ liệu qua mạng TCP/IP, nên đôi lúc dữ liệu hiển thị tại Windows App không tránh khỏi một ít sai sót nhỏ. Tuy vậy xu hướng biến động của tải hệ thống IVI được cập nhật chính xác và tức thời trên biểu đồ của Windows App, cho thấy rằng cơ chế thu thập và truyền tải dữ

liệu của phân hệ Linux App hoạt động ổn định, chính xác và đáng tin cậy.



Hình 5.10 Xu hướng sử dụng tải hệ thống IVI thể hiện tại Windows App trong ĐKKT1

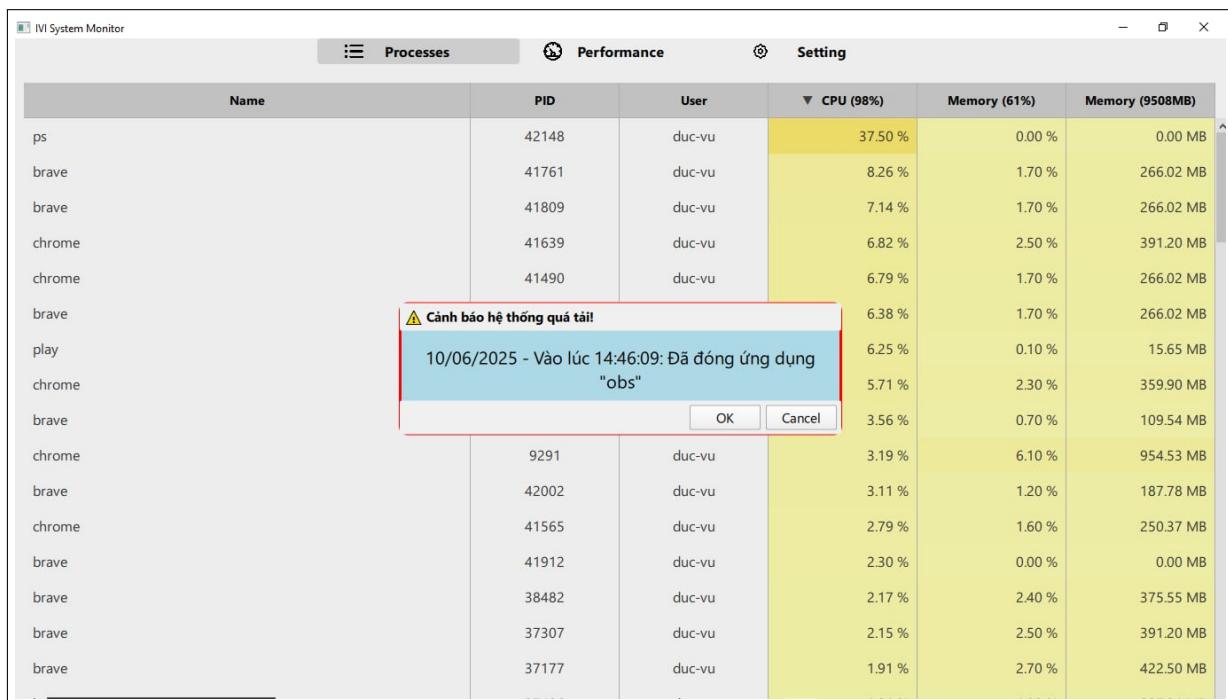


Hình 5.11 Xu hướng sử dụng tải hệ thống IVI thể hiện tại System Monitor trong ĐKKT1.

5.2.2. Đánh giá trong ĐKKT2

Trong điều kiện kiểm thử thứ hai (ĐKKT2), chúng tôi chủ động đặt hệ thống IVI vào trạng thái hoạt động với tải cao bằng cách khởi chạy nhiều tiến trình tiêu tốn tài nguyên cùng lúc (bao gồm cả tiến trình đồ họa và trình duyệt web), nhằm mô phỏng một tình huống quá tải thực tế. Mục tiêu của kiểm thử này là đánh giá hiệu quả hoạt động của *Phần mềm* trong việc phát hiện và xử lý tình trạng quá tải hệ thống IVI – một trong những chức năng quan trọng nhất mà *Phần mềm* cần đáp ứng.

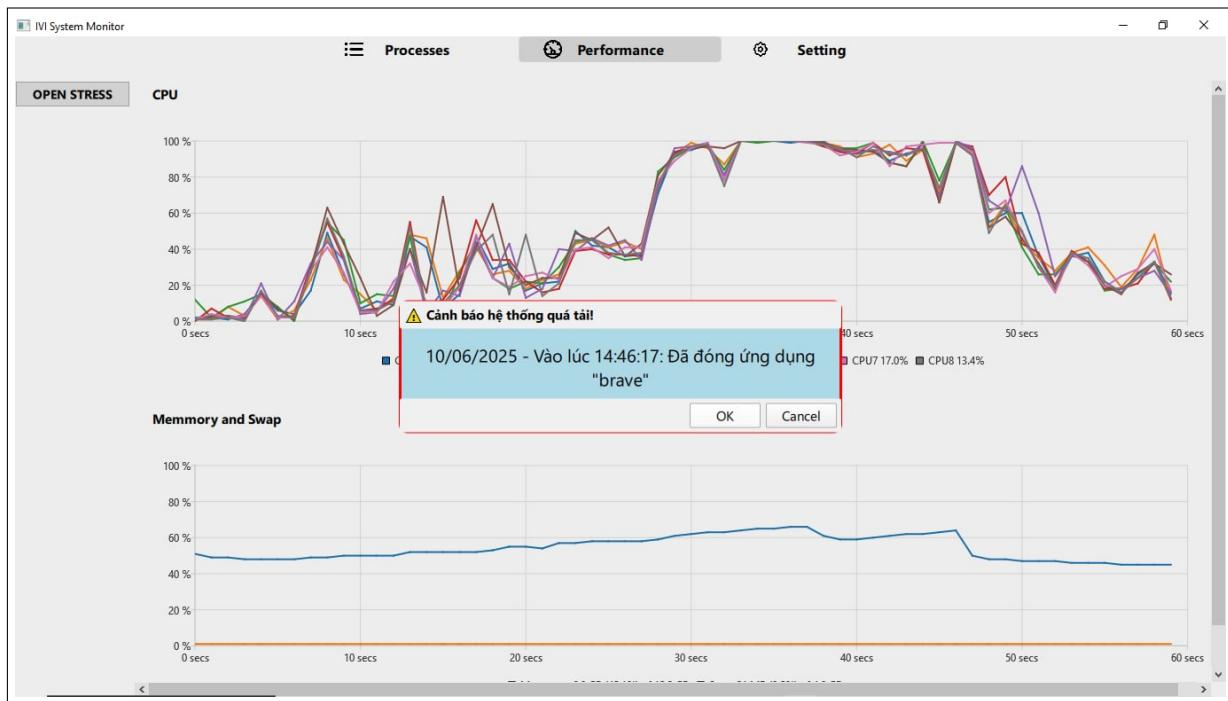
Cụ thể, khi mức sử dụng tài nguyên (CPU và bộ nhớ) vượt ngưỡng cho phép trong khoảng thời gian liên tiếp (như đã giải thích tại các nội dung trước), Windows App sẽ tự động đưa ra chẩn đoán hệ thống đang quá tải, sau đó *Phần mềm* sẽ bắt đầu quá trình xử lý nhằm cân bằng lại tài nguyên hệ thống.



Hình 5.12 Kết thúc tiến trình ”obs” chiếm nhiều tài nguyên trong ĐKK2

Hình 5.12 và Hình 5.13 mô tả kết quả xử lý tình huống quá tải nêu trên. Cụ thể: tại Hình 5.12, khi phát hiện tình trạng quá tải (ngưỡng tải cao với CPU hoạt động gần như 100% công suất và MEM hoạt động khoảng 60% trong khoảng từ giây 26 đến giây 45 - Hình 5.13), *Phần mềm* đã lựa chọn kết thúc tiến trình obs – một tiến trình ghi hình màn hình đang tiêu thụ lượng lớn tài nguyên hệ thống, nhằm cân bằng tải cho hệ thống IVI. Tuy nhiên, như thể hiện trong Hình 5.13, sau khi tiến trình obs được kết thúc, hệ thống IVI vẫn chưa trở lại trạng thái ổn định (*Phần mềm* kết thúc obs vào lúc 14:46:09 nhưng CPU vẫn đang ở mức cao) - nhưng tải trên hệ thống IVI đã có xu hướng giảm. Lúc này,

Phần mềm tiếp tục xác định tiến trình gây tiêu tốn nhiều tài nguyên trên hệ thống IVI, cụ thể là tiến trình *brave*) và tiến hành kết thúc nó. Sau khi *brave* được xử lý, biểu đồ hiển thị tài nguyên cho thấy mức tiêu thụ CPU và bộ nhớ giảm rõ rệt (vào giây thứ 46 trở đi, CPU giảm xuống rõ rệt từ 100% xuống còn 20%), đưa hệ thống trở lại trạng thái ổn định.



Hình 5.13 Kết thúc tiến trình ”brave” chiếm nhiều tài nguyên trong ĐKK2

Hình 5.14 và Hình 5.15 thể hiện dữ liệu được lưu trữ trong cơ sở dữ liệu sau khi quá trình xử lý quá tải diễn ra. Tại đây, người dùng có thể truy xuất lại thông tin chi tiết về các tiến trình đã góp phần gây ra tình trạng quá tải cho hệ thống IVI, bao gồm thời điểm xảy ra sự kiện, mức tiêu thụ tài nguyên (CPU, bộ nhớ), và hành động xử lý tương ứng.

1	Time Stamp	PID	User	ProcessName	CPU(%)	MEM(MB)	MEM(%)	Priority	Score	Is Killed
140	2025-06-10T14:46:09	38145	duc-vu	brave	0.15	156.48	1	2	0.81	
141	2025-06-10T14:46:09	38176	duc-vu	brave	1.85	312.96	2	2	0.81	
142	2025-06-10T14:46:09	38191	duc-vu	brave	0	78.24	0.5	2	0.81	
143	2025-06-10T14:46:09	38482	duc-vu	brave	2.2	375.55	2.4	2	0.81	
144	2025-06-10T14:46:09	40441	duc-vu	chrome	5.75	359.9	2.3	2	0.85	
145	2025-06-10T14:46:09	40597	duc-vu	chrome	0.01	93.89	0.6	2	0.85	
146	2025-06-10T14:46:09	40628	duc-vu	gnome-control-c	0.05	31.3	0.2	2	0.51	
147	2025-06-10T14:46:09	40697	duc-vu	obs	19	970.18	6.2	2	9.84	Yes
148	2025-06-10T14:46:09	41239	duc-vu	obs-ffmpeg-mux	0.06	31.3	0.2	2	0.51	
149	2025-06-10T14:46:09	41243	duc-vu	tracker-extract	0.05	31.3	0.2	2	0	
150	2025-06-10T14:46:09	41490	duc-vu	chrome	7.4	281.66	1.8	2	0.85	

Hình 5.14 Kiểm tra dữ liệu được lưu trữ trong file (mở rộng cho Hình 5.12)

1	Time Stamp	PID	User	ProcessName	CPU(%)	MEM(MB)	MEM(%)	Priority	Score	Is Killed
227	2025-06-10T14:46:17	37406	duc-vu	brave	1.83	312.96	2	2	0.75	Yes
228	2025-06-10T14:46:17	37514	duc-vu	brave	0.09	78.24	0.5	2	0.75	Yes
229	2025-06-10T14:46:17	3758	duc-vu	ibus-x11	0	15.65	0.1	1	0.25	
230	2025-06-10T14:46:17	3763	duc-vu	mutter-x11-fram	0	93.89	0.6	2	0	
231	2025-06-10T14:46:17	38145	duc-vu	brave	0.14	156.48	1	2	0.75	Yes
232	2025-06-10T14:46:17	38176	duc-vu	brave	1.74	328.61	2.1	2	0.75	Yes
233	2025-06-10T14:46:17	38191	duc-vu	brave	0	78.24	0.5	2	0.75	Yes
234	2025-06-10T14:46:17	38482	duc-vu	brave	2.05	375.55	2.4	2	0.75	Yes
235	2025-06-10T14:46:17	40441	duc-vu	chrome	5.4	359.9	2.3	2	0.65	
236	2025-06-10T14:46:17	40597	duc-vu	chrome	0.01	93.89	0.6	2	0.65	
237	2025-06-10T14:46:17	40628	duc-vu	gnome-control-c	0.04	31.3	0.2	2	0.5	
238	2025-06-10T14:46:17	41243	duc-vu	tracker-extract	0.04	31.3	0.2	2	0	
239	2025-06-10T14:46:17	41490	duc-vu	chrome	4.68	266.02	1.7	2	0.65	
240	2025-06-10T14:46:17	41565	duc-vu	chrome	4.09	266.02	1.7	2	0.65	
241	2025-06-10T14:46:17	41639	duc-vu	chrome	5.95	438.14	2.8	2	0.65	

Hình 5.15 Kiểm tra dữ liệu được lưu trữ trong file (mở rộng cho Hình 5.13)

Các tiến trình đã bị kết thúc sẽ được đánh dấu bằng giá trị "Yes" trong cột *Is Killed*. Thông tin này không chỉ giúp minh chứng cho quá trình xử lý đã được kích hoạt đúng lúc, mà còn hỗ trợ người dùng (đặc biệt là nhà phát triển) phân tích nguyên nhân gây ra quá tải, từ đó đưa ra các phương án tối ưu về ứng dụng hoặc cấu hình cho hệ thống IVI.

Qua cơ chế lưu trữ này, *Phần mềm* không chỉ thực hiện chức năng xử lý quá tải theo thời gian thực, mà còn cung cấp khả năng phản hồi sau sự kiện – một yếu tố rất quan trọng trong quy trình kiểm thử và bảo trì hệ thống.

Tình huống kiểm thử này đã chứng minh rằng cơ chế đánh giá và xử lý quá tải của *Phần mềm* hoạt động ổn định và hiệu quả. Phân hệ Windows App không chỉ phát hiện chính xác tình trạng quá tải, mà còn có khả năng lựa chọn tiến trình phù hợp để xử lý – ưu tiên các tiến trình tiêu tốn nhiều tài nguyên nhưng không ảnh hưởng đến hoạt động cốt lõi của hệ thống IVI. Đồng thời, cơ chế cảnh báo người dùng cũng được kích hoạt đúng lúc, giúp cải thiện khả năng tương tác và giám sát hệ thống.

5.2.3. Đánh giá trong ĐKKT3

Cũng tương tự như trong ĐKKT2, trong ĐKKT3 này, hệ thống được đặt trong trạng thái tải nặng thông qua việc kích hoạt các tiến trình giả lập "stress" (như Hình 5.16, các tiến trình giả lập "stress" được tạo ra và chiếm CPU và bộ nhớ ở mức cao). Mục tiêu là đẩy hệ thống đến mức sử dụng tài nguyên cao nhất, từ đó đánh giá khả năng giám sát và phản ứng của *Phần mềm*.

Trong Hình 5.17, minh họa rõ ràng sự thay đổi mức sử dụng CPU và bộ nhớ (RAM và Swap) trước và sau thời điểm kích hoạt Stress test. Giá trị đầu vào để kích hoạt Stress

test được thể hiện rõ ở cửa sổ bên trái, cụ thể như sau: chúng tôi thiết lập mức sử dụng bộ nhớ là 50%, sử dụng 8 lõi CPU, và thời gian kiểm thử là 30 giây.

Trước thời điểm Tress test, hệ thống hoạt động ổn định với mức sử dụng tài nguyên ở ngưỡng trung bình và ít dao động. CPU thường duy trì ổn định dưới 40%, có lúc lên đến 60% - 70% và mức sử dụng RAM ổn định dưới 50%. Tuy nhiên, ngay sau khi các tiến trình “stress” được khích hoạt, biểu đồ tài nguyên cho thấy mức sử dụng CPU và RAM tăng đột ngột và duy trì ở mức cao trong một khoảng thời gian. Đối với CPU, tất cả các lõi đều đạt mức tối đa 100%. Còn đối với RAM, mức sử dụng cũng đã tăng mạnh, có lúc chiếm đến 80%. Điều này cho thấy rằng, hệ thống hiện đang vận hành trong trạng thái tải cao và cần được xử lý kịp thời.

Name	PID	User	CPU (100%)	Memory (60%)	Memory (9419MB)
stress	45519	duc-vu	10.26 %	0.00 %	0.00 MB
stress	45520	duc-vu	9.96 %	13.50 %	2112.48 MB
stress	45522	duc-vu	9.90 %	0.00 %	0.00 MB
stress	45524	duc-vu	9.77 %	0.00 %	0.00 MB
stress	45518	duc-vu	9.68 %	12.00 %	1877.76 MB
stress	45523	duc-vu	9.68 %	0.00 %	0.00 MB
stress	45521	duc-vu	9.55 %	0.00 %	0.00 MB
stress	45517	duc-vu	9.15 %	0.00 %	0.00 MB
stress	45526	duc-vu	8.35 %	0.00 %	0.00 MB
stress	45525	duc-vu	8.10 %	0.00 %	0.00 MB
chrome	9291	duc-vu	3.17 %	4.90 %	766.75 MB
gnome-system-mo	36740	duc-vu	1.83 %	0.90 %	140.83 MB
htop	15081	duc-vu	1.35 %	0.00 %	0.00 MB
chrome	9142	duc-vu	0.93 %	1.60 %	250.37 MB
gnome-shell	2978	duc-vu	0.63 %	1.90 %	297.31 MB
chrome	9058	duc-vu	0.45 %	2.40 %	375.55 MB

Hình 5.16 Tiến trình “stress” và mức chiếm dụng tài nguyên của nó

Để đưa hệ thống trở lại trạng thái ổn định, *Phần mềm* đã chủ động kết thúc tiến trình giả lập “stress”, với thông báo kết thúc hiển thị như minh họa trong Hình 5.17. Trong ĐKKT3 này, chúng tôi đã thiết lập cho hệ thống hoạt động dưới tải nặng trong vòng 30 giây. Tuy nhiên, chỉ sau khoảng 10 giây, *Phần mềm* đã kịp thời phát hiện sự bất thường trong mức sử dụng tài nguyên hệ thống và ngay lập tức thực hiện phương án xử lý bằng cách kết thúc tiến trình “stress”. Có thể rõ ràng trong biểu đồ, mức CPU từ 100% giảm xuống dưới 40% và mức sử dụng RAM giảm từ khoảng 60% xuống còn khoảng 40%. Điều này cho thấy khả năng phát hiện và phản ứng nhanh của *Phần mềm* khi hệ thống gặp tình trạng quá tải.



Hình 5.17 Kết thúc tiến trình ”stress” chiếm nhiều tài nguyên trong ĐKK3

5.3. Kết luận chương

Trong chương này đã kiểm thử khả năng hoạt động của từng nhóm chức năng riêng lẻ. Tiếp đến, chúng tôi kiểm tra luồng hoạt động tổng thể sau khi kết nối Windows App với Linux App, cụ thể như : giám sát tài nguyên hệ thống, khả năng phát hiện quá tải hệ thống và khả năng xử lý khi xảy ra. Dựa vào dữ liệu được lưu lại trong quá trình triển khai để đưa ra đánh giá về sự chính xác về khả năng phát hiện quá tải cũng như xử lý quá tải.

CHƯƠNG 6: KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

6.1. Kết quả đã đạt được

Đề tài “Phần mềm giám sát và tùy chỉnh tải hệ thống IVI” đã thành công hoàn thiện một *Phần mềm* với các chức năng cốt lõi như: theo dõi tình trạng sử dụng tài nguyên hệ thống (CPU, MEM, tiến trình), phát hiện quá tải, và điều phối tài nguyên hệ thống qua việc kết thúc các tiến trình chiếm dụng nhiều tài nguyên.

Phần mềm này có hai thành phần chính, bao gồm Linux App (chạy trên hệ thống IVI) và Windows App (giao diện người dùng). Về phía Linux App, đã thành công xây dựng được các chức năng đảm nhận việc thu thập dữ liệu tài nguyên và can thiệp vào hệ thống IVI thực hiện các hành động theo yêu cầu nhận được từ Windows App. Trong khi đó, Windows App cung cấp một giao diện hiển thị trực quan, có khả năng phát hiện và xử lý khi hệ thống quá tải. Và thành công kết nối hai thành phần này với nhau thông qua giao thức TCP/IP.

6.2. Những thiếu sót và vấn đề cần cải thiện

Mặc dù *Phần mềm* đã hoạt động ổn và đáp ứng các yêu cầu đã đề ra, tuy nhiên vẫn còn tồn tại một vài thiếu sót và vấn đề cần được cải thiện trong các phiên bản tiếp theo. Cụ thể như:

1. Tương tác giữa các phân hệ: giao tiếp giữa Linux App và Windows App đã được thực hiện qua TCP/IP. Tuy nhiên, dữ liệu trong quá trình gửi/nhận chưa được mã hoá và cần phải kết nối trong cùng một mạng.
2. Khả năng phát hiện quá tải: *Phần mềm* hiện tại chủ yếu dựa vào các quy tắc cố định để phát hiện quá tải. Tuy nhiên, trong tương lai, *Phần mềm* có thể được cải tiến để sử dụng dữ liệu lịch sử và phân tích hành vi hoạt động của hệ thống, nhằm dự đoán sớm các tình huống quá tải. Điều này sẽ giúp nâng cao khả năng chủ động trong việc quản lý tài nguyên.

6.3. Hướng phát triển trong tương lai

Với những kết quả đã đạt được, đề tài này có thể tiếp tục phát triển và mở rộng trong tương lai, cụ thể như:

1. Tối ưu hóa thuật toán phát hiện và xử lý quá tải: việc cải thiện các thuật toán phát hiện và xử lý quá tải là điều cần thiết nhằm tăng độ tin cậy về tính hiệu quả của *Phần mềm*.
2. Mở rộng phạm vi giao tiếp: nghiên cứu giải pháp giúp kết nối Windows App và Linux App trên hai mạng LAN khác nhau.
3. Hỗ trợ nhiều nền tảng phần cứng: hiện tại, *Phần mềm* được phát triển và thử nghiệm trên một hệ thống mô phỏng IVI. Tuy nhiên, trong tương lai, nó có thể được triển khai trên nhiều loại phần cứng khác nhau, bao gồm cả các hệ thống IVI thực tế. Điều này đòi hỏi *Phần mềm* phải được thiết kế linh hoạt và tương thích với các loại phần cứng khác nhau.
4. *Phần mềm* có thể giám sát nhiều thiết bị IVI, đồng thời tạo ra một hệ thống phân quyền truy cập (hệ thống tài khoản người dùng/quản lý), giúp tăng tính ứng dụng và khả năng thương mại hóa của *Phần mềm*.

6.4. Kết luận chương

”*Phần mềm giám sát và tùy chỉnh tải hệ thống IVI*” đã đạt được các kết quả đáng kể trong việc giám sát tài nguyên hệ thống và tối ưu hóa hiệu suất của hệ thống IVI. Mặc dù còn một số vấn đề cần được cải thiện, như việc kết nối hai phân hệ khác mạng (không cùng mạng), độ chính xác của thuật toán phát hiện và đánh giá trạng thái, nhưng *Phần mềm* này đã cho thấy được tiềm năng trong việc hỗ trợ phát triển và vận hành hệ thống IVI một cách ổn định và hiệu quả. Với những hướng phát triển trong tương lai, *Phần mềm* sẽ tiếp tục được hoàn thiện và mở rộng nhằm cải thiện các vấn đề còn tồn tại.

TÀI LIỆU THAM KHẢO

- [1] Ward B. How Linux works: What every superuser should know. no starch press; 2021 Apr 19.
- [2] J. F. Kurose and K. W. Ross. Computer Networking: A Top-Down Approach. 6th ed. Boston: Pearson, 2013.
- [3] Goichon, F., Salagnac, G., Frénot, S. (2013). Swap Fairness for Thrashing Mitigation. Inria.
- [4] Schneider, J., Nett, D. (2019). Safety Issues of Integrating IVI and ADAS Functionality via Running Linux and AUTOSAR in Parallel on a Dual-Core-System.
- [5] Precedence Research. In-Vehicle Infotainment Market Size, Share, Growth Report 2024–2034. Precedence Research; 2024. [Online].
- [6] The Qt Company. Qt Documentation. [Online].
- [7] Triantaphyllou, E. Multi-Criteria Decision Making: A Comparative Study. Boston: Springer, 2000.

PHỤ LỤC 1

HỆ THỐNG IVI

Khái niệm hệ thống IVI

Hệ thống IVI (In-Vehicle Infotainment) là một nền tảng tích hợp trong phương tiện giao thông, kết hợp giữa các chức năng giải trí (*entertainment*) và thông tin (*information*) nhằm mang lại trải nghiệm tiện nghi, an toàn và thông minh cho người lái cũng như hành khách. Các hệ thống IVI hiện đại thường tích hợp nhiều công nghệ như màn hình cảm ứng, định vị GPS, trình phát nhạc/video, kết nối không dây (Wi-Fi, Bluetooth), điều khiển bằng giọng nói, camera lùi và nhiều tiện ích hỗ trợ điều khiển xe.

Thành phần của một hệ thống IVI điển hình

Một hệ thống IVI thông thường bao gồm các thành phần sau:

- **Bộ xử lý trung tâm (SoC hoặc MCU)**: điều khiển và xử lý toàn bộ dữ liệu, logic hoạt động trong hệ thống.
- **Bộ nhớ và lưu trữ**: bao gồm RAM, bộ nhớ flash hoặc ổ cứng để chứa hệ điều hành và dữ liệu người dùng.
- **Màn hình hiển thị**: thường là màn hình cảm ứng LCD/LED dùng để tương tác với người dùng.
- **Hệ điều hành**: đa phần sử dụng Linux hoặc Android do tính chất mã nguồn mở, dễ tùy biến.
- **Các module giao tiếp**: GPS, Bluetooth, Wi-Fi, LTE, USB, HDMI, CAN Bus,...
- **Giao diện người dùng (UI)**: thiết kế trực quan, dễ sử dụng khi đang lái xe.
- **Thiết bị ngoại vi**: micro, loa, camera, cảm biến,...

Vai trò của hệ điều hành trong hệ thống IVI

Hệ điều hành trong hệ thống IVI đóng vai trò trung gian giữa phần mềm và phần cứng, quản lý tài nguyên, tiến trình và cung cấp môi trường cho các ứng dụng hoạt động. Những hệ điều hành thường gặp gồm:

- **Linux**: phổ biến trong các hệ thống IVI nhúng, nhẹ, bảo mật cao.

- **Android Automotive**: một biến thể của Android tối ưu hóa cho ô tô.

Hệ điều hành đảm bảo các yếu tố:

- Quản lý tài nguyên hiệu quả (CPU, RAM, thiết bị I/O,...).
- Hỗ trợ đa nhiệm và phản hồi thời gian thực.
- Tương thích với phần cứng và ứng dụng nhúng.

Yêu cầu trong phát triển phần mềm cho hệ thống IVI

Việc phát triển phần mềm cho hệ thống IVI cần đáp ứng các yêu cầu nghiêm ngặt:

- **Tối ưu hiệu suất**: hệ thống IVI thường có giới hạn phần cứng.
- **Phản hồi nhanh**: đảm bảo an toàn khi người lái tương tác.
- **Ôn định và tin cậy**: phần mềm không được phép treo hoặc crash.
- **Dễ cập nhật và bảo trì**: hỗ trợ cập nhật OTA hoặc bằng USB.
- **Tương thích thiết bị ngoại vi**: đảm bảo kết nối với cảm biến, camera,...

Mối liên hệ giữa đề tài và hệ thống IVI

Trong phạm vi đề tài, chúng tôi sử dụng một máy tính chạy hệ điều hành Linux để mô phỏng hệ thống IVI. Phân hệ *Linux App* được thiết kế và triển khai để chạy trên thiết bị này, đảm nhiệm chức năng thu thập dữ liệu, theo dõi tài nguyên và thực hiện các lệnh kiểm soát tài.

Thiết kế này giúp phần mềm có thể được kiểm thử trong môi trường phát triển mà vẫn đảm bảo tính tương thích với các hệ thống IVI thực tế. Qua đó, *Phần mềm* có thể được tích hợp vào hệ thống IVI trong tương lai để phục vụ mục đích giám sát và tối ưu tài nguyên trong các phương tiện hiện đại.

PHỤ LỤC 2

HỆ ĐIỀU HÀNH LINUX

Khái quát về hệ điều hành Linux

Linux là một hệ điều hành mã nguồn mở được phát triển dựa trên Unix, nổi tiếng với tính linh hoạt, ổn định và khả năng tùy biến cao. Linux hiện được sử dụng phổ biến trong các hệ thống nhúng, máy chủ, siêu máy tính, thiết bị IoT và đặc biệt là trong hệ thống IVI (In-Vehicle Infotainment) hiện đại.

Không giống như các hệ điều hành thương mại, Linux cho phép người dùng truy cập và chỉnh sửa mã nguồn, từ đó có thể xây dựng các phiên bản tùy biến phù hợp với từng mục đích sử dụng, đặc biệt hữu ích trong môi trường nhúng với yêu cầu tối ưu tài nguyên nghiêm ngặt.

Kiến trúc cơ bản của hệ điều hành Linux

Linux được thiết kế theo mô hình phân lớp, chia thành hai không gian chính:

- **Kernel space (Không gian nhân)**: nơi chứa *Linux kernel* – thành phần cốt lõi chịu trách nhiệm quản lý phần cứng, bộ nhớ, tiến trình, hệ thống file và thiết bị đầu cuối. Kernel có thể giao tiếp trực tiếp với phần cứng thông qua các trình điều khiển thiết bị (device drivers).
- **User space (Không gian người dùng)**: là nơi chạy các ứng dụng thông thường, như các chương trình điều khiển, giao diện người dùng, trình phát nhạc,... Các tiến trình tại user space không thể trực tiếp truy cập phần cứng mà phải thông qua các system call.

Vai trò của Linux trong hệ thống nhúng

Linux rất phù hợp cho các hệ thống nhúng như IVI nhờ vào các yếu tố sau:

- **Tùy biến cao**: Có thể cắt giảm những thành phần không cần thiết để tối ưu dung lượng và hiệu suất.
- **Hỗ trợ phần cứng tốt**: Cộng đồng phát triển rộng lớn và hàng nghìn driver sẵn có.
- **Môi trường phát triển mạnh mẽ**: Hỗ trợ đầy đủ công cụ lập trình, debug và mô phỏng trên PC trước khi triển khai thực tế.

- **Chi phí thấp:** Mã nguồn mở hoàn toàn miễn phí.
- **Khả năng quản lý tài nguyên tốt:** Linux cung cấp nhiều công cụ giám sát và quản lý tiến trình, bộ nhớ, CPU – rất quan trọng trong các hệ thống giới hạn tài nguyên.

System Calls và giao tiếp giữa các tầng

Trong Linux, các tiến trình ở không gian người dùng sử dụng **System Calls** để tương tác với Kernel. Ví dụ:

- `fork()`, `exec()` để tạo và thực thi tiến trình mới.
- `kill()` để kết thúc tiến trình.
- `open()`, `read()`, `write()` để truy xuất file và thiết bị.

Điều này rất quan trọng trong các ứng dụng giám sát hệ thống như đè tài này, vì việc thu thập thông tin CPU, bộ nhớ, tiến trình,... đều thông qua các lệnh shell hoặc đọc từ các file đặc biệt trong `/proc`, vốn được kernel cập nhật liên tục.

File hệ thống và quản lý tài nguyên

Linux sử dụng hệ thống file đặc biệt để quản lý và cung cấp thông tin về tài nguyên hệ thống:

- `/proc/stat` – chứa thông tin về CPU.
- `/proc/meminfo` – thông kê bộ nhớ RAM, Swap.
- `/proc/[PID]/status` – chứa thông tin chi tiết về từng tiến trình.

Ngoài ra, hệ điều hành còn cung cấp các công cụ giám sát như `top`, `htop`, `vmstat`, `ps`, `free`,... Các công cụ này đều có thể được gọi và xử lý thông qua các chương trình tự động như Linux App trong đè tài.

Ứng dụng trong đè tài

Trong phạm vi đè tài, Linux được sử dụng làm nền tảng triển khai phân hệ *Linux App*. Cụ thể:

- Hệ điều hành Linux cho phép truy xuất thông tin tài nguyên một cách hiệu quả.

- Các shell command được sử dụng để thu thập dữ liệu về CPU, bộ nhớ, tiến trình.
- Linux hỗ trợ tốt socket programming để giao tiếp với Windows App thông qua TCP/IP.
- Hệ thống có thể can thiệp trực tiếp để kết thúc tiến trình và điều khiển phần cứng như loa cảnh báo.

Nhờ sử dụng Linux, chúng tôi có thể xây dựng một hệ thống nhẹ, ổn định, dễ tùy biến và dễ tích hợp lên các thiết bị IVI thực tế trong tương lai.

PHỤ LỤC 3

HỆ ĐIỀU HÀNH WINDOWS

Giới thiệu hệ điều hành Windows

Windows là một hệ điều hành thương mại được phát triển bởi Microsoft, nổi tiếng với giao diện đồ họa thân thiện và khả năng hỗ trợ đa dạng phần mềm. Trong đề tài này, hệ điều hành Windows được lựa chọn làm nền tảng phát triển phân hệ **Windows App (PerformanceViewer App)**, chịu trách nhiệm hiển thị dữ liệu, tương tác với người dùng và điều phối các hành động điều khiển đến hệ thống I/O.

Việc sử dụng Windows có các ưu điểm sau:

- Môi trường phát triển phần mềm mạnh mẽ, tích hợp tốt với Qt Framework.
- Giao diện người dùng thân thiện, dễ triển khai các chức năng hiển thị đồ họa.
- Tương thích tốt với các công cụ giám sát, cơ sở dữ liệu và lập trình socket.

Kiến trúc hệ điều hành Windows

Hệ điều hành Windows được tổ chức thành nhiều tầng (layers) bao gồm:

- **User Mode**: nơi thực thi các ứng dụng như trình duyệt, phần mềm văn phòng, chương trình người dùng, bao gồm cả Windows App trong đề tài.
- **Kernel Mode**: nơi xử lý các tác vụ hệ thống như quản lý bộ nhớ, tiến trình, thiết bị và điều phối hệ thống file.
- **Windows API**: cung cấp tập các hàm giao tiếp giữa ứng dụng người dùng và hệ thống (bao gồm Win32 API, .NET API...).

Trong đề tài này, chúng tôi chủ yếu hoạt động trong không gian **User Mode**, sử dụng các thư viện từ Qt Framework để tương tác gián tiếp với hệ thống qua các API được hỗ trợ.

Quản lý luồng và giao diện trong Windows

Windows hỗ trợ tốt cơ chế đa luồng (multithreading), cho phép chương trình thực thi song song nhiều tác vụ:

- Thu nhận dữ liệu từ phân hệ Linux App theo thời gian thực.
- Hiển thị biểu đồ động, bảng dữ liệu cập nhật liên tục.
- Phát hiện và xử lý quá tải mà không làm “đơ” giao diện người dùng.

Qt Framework tận dụng tốt khả năng đa luồng của Windows thông qua các lớp như `QThread`, `QTimer`, giúp tách biệt logic xử lý dữ liệu với giao diện đồ họa.

Socket và mạng TCP/IP trên Windows

Windows hỗ trợ lập trình socket thông qua thư viện **Winsock (Windows Sockets API)**. Tuy nhiên, nhờ sử dụng Qt Framework, lập trình socket TCP/IP được trừu tượng hóa thông qua các lớp như `QTcpSocket`, `QTcpServer` giúp triển khai kết nối dễ dàng và đa nền tảng.

Phân hệ Windows App trong đê tài hoạt động như một **TCP Server**, lắng nghe và nhận dữ liệu từ Linux App (TCP Client). Toàn bộ hoạt động mạng đều được xử lý trong các luồng riêng biệt, đảm bảo tính phản hồi và ổn định của phần mềm.

Khả năng hiển thị đồ họa và trực quan hóa dữ liệu

Một trong những lý do chọn Windows làm nền tảng phát triển giao diện là khả năng hiển thị đồ họa mạnh mẽ:

- Qt hỗ trợ module Qt Charts để vẽ biểu đồ CPU và bộ nhớ thời gian thực.
- Kết hợp bảng dữ liệu và hoạt ảnh giúp người dùng dễ theo dõi tài nguyên.
- Windows hỗ trợ tốt màn hình độ phân giải cao và nhiều kiểu font chữ.

Ngoài ra, khả năng kết hợp với cơ sở dữ liệu SQLite hoặc file lưu tạm thời giúp lưu trữ dữ liệu giám sát dễ dàng trên Windows.

Tóm tắt vai trò của Windows trong đê tài

Tổng kết lại, hệ điều hành Windows đóng vai trò:

- Cung cấp môi trường phát triển phần mềm mạnh mẽ để xây dựng giao diện và xử lý logic điều khiển.

- Thực thi phân hệ Windows App – trung tâm hiển thị, phân tích và điều phối toàn bộ hệ thống IVI.
- Kết nối mạng ổn định và dễ mở rộng với các hệ thống khác trong tương lai.

Nhờ sự ổn định và thân thiện của nền tảng Windows cùng với Qt Framework, chúng tôi có thể phát triển nhanh chóng và triển khai hiệu quả phần mềm giám sát tài nguyên trong hệ thống IVI.

PHỤ LỤC 4

KIẾU DỮ LIỆU JSON

JSON (viết tắt của JavaScript Object Notation) là một định dạng dữ liệu văn bản, phổ biến để lưu trữ và truyền tải thông tin giữa các hệ thống khác nhau. Dữ liệu JSON được tổ chức theo cặp khóa – giá trị (key-value) và có cú pháp đơn giản, dễ đọc đối với người sử dụng và dễ phân tích bởi máy tính. Một tài liệu JSON được bao bọc bởi cặp dấu ngoặc nhọn (đại diện cho một đối tượng – object), bên trong chứa các cặp ”khóa”: giá trị phân tách nhau bằng dấu phẩy. JSON cũng hỗ trợ cấu trúc mảng (array) dùng dấu ngoặc vuông [] để biểu diễn danh sách các giá trị hoặc đối tượng. Khóa phải là chuỗi (string) đặt trong ngoặc kép ”...”, còn giá trị có thể ở dạng chuỗi, số, boolean (true/false), null, đối tượng hoặc mảng lồng nhau. Ví dụ một đối tượng JSON đơn giản:

```
1 {  
2     "name": "Lan",  
3     "age": 25,  
4     "languages": ["Python", "C++", "Java"]  
5 }
```

JSON trở nên rất phổ biến trong phát triển phần mềm ngày nay nhờ các ưu điểm sau:
(i) Nhẹ và dễ đọc: Cú pháp JSON tối giản, không có các thẻ đóng/mở dư thừa. Điều này giúp giảm dung lượng dữ liệu truyền trên mạng và tăng tốc độ xử lý. JSON cũng có cấu trúc trực quan, nên dữ liệu định dạng JSON tương đối dễ đọc hiểu đối với lập trình viên.
(ii) Độc lập ngôn ngữ: JSON không phụ thuộc vào nền tảng hay ngôn ngữ lập trình cụ thể nào – hầu hết các ngôn ngữ (JavaScript, Python, Java, C, v.v.) đều có sẵn thư viện để phân tích và sinh JSON. Việc này giúp JSON trở thành định dạng chung để các hệ thống khác biệt có thể trao đổi dữ liệu với nhau một cách thuận tiện hơn. (iii) Hỗ trợ rộng rãi: Hầu hết các framework và hệ quản trị cơ sở dữ liệu hiện đại (ví dụ NoSQL như MongoDB) đều hỗ trợ JSON, nên việc tích hợp, lưu trữ hoặc xử lý dữ liệu JSON rất thuận tiện

Trong bộ giao thức TCP/IP, JSON không phải là một giao thức mạng mà là định dạng dữ liệu hoạt động ở tầng ứng dụng. TCP/IP đảm nhiệm việc truyền dữ liệu thô (chuỗi byte) một cách tin cậy giữa các thiết bị, còn JSON thường được sử dụng để đóng gói dữ liệu ứng dụng trước khi gửi qua kết nối TCP. Nói cách khác, JSON đảm nhiệm vai trò định dạng thông điệp (message format) cho ứng dụng, giúp hai bên (client và server) hiểu được cấu trúc và ý nghĩa của dữ liệu trao đổi thông qua kết nối TCP.

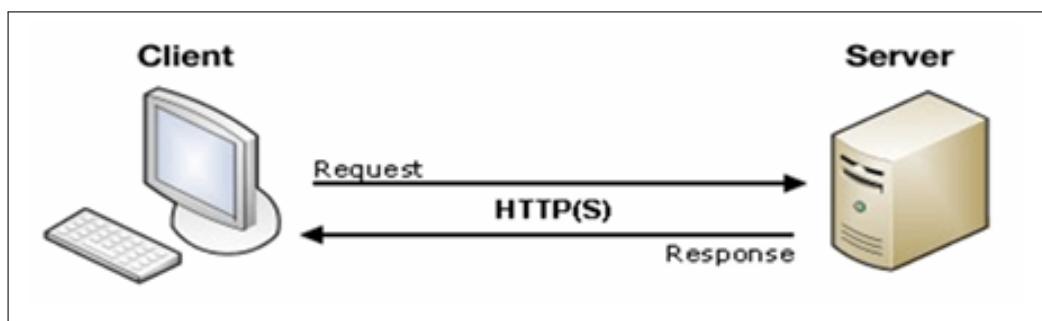
PHỤ LỤC 5

MÔ HÌNH GIAO TIẾP CLIENT-SERVER

Mô hình Client-Server là một kiến trúc phần mềm phổ biến trong công nghệ thông tin, nơi ứng dụng hoạt động theo cơ chế phân tách thành hai phần: client (người dùng hoặc ứng dụng người dùng) và server (máy chủ). Các phần này tương tác với nhau thông qua mạng hoặc giao thức mạng như HTTP. Mô hình Client-Server có hai thành phần chính:

1. Client: Đây là phần sẽ được cài đặt và chạy trên máy tính hoặc thiết bị của người dùng cuối (như máy tính cá nhân, Smartphone, máy tính bảng, trình duyệt web, v.v.). Nhiệm vụ của client là gửi các yêu cầu đến server và chờ phản hồi từ server. Client có trách nhiệm hiển thị dữ liệu hoặc thông tin được nhận từ server và cung cấp giao diện người dùng để tương tác với ứng dụng.
2. Server: Đây là phần mềm hoặc máy chủ được cài đặt trên các máy tính hoặc hệ thống máy chủ chuyên dụng. Server nhận các yêu cầu từ client, xử lý yêu cầu và trả về phản hồi chứa dữ liệu hoặc thông tin được yêu cầu. Server chịu trách nhiệm xử lý các tác vụ phức tạp và thường có tài nguyên cao hơn để đáp ứng nhiều yêu cầu từ client cùng một lúc.

Trong mô hình này, client và server tương tác qua mạng, thông qua giao thức truyền thông như HTTP, TCP/IP, v.v. Client gửi yêu cầu đến server, sau đó server xử lý yêu cầu và trả về phản hồi cho client. Mô hình Client-Server giúp phân tách trách nhiệm giữa client và server, giúp hệ thống dễ dàng mở rộng, bảo trì và phát triển thêm các tính năng mới.



Hình 0.1 Mô hình giao tiếp Client - Server

Mô hình Client-Server là một trong những kiến trúc quan trọng và phổ biến trong việc xây dựng ứng dụng và hệ thống phân tán, đặc biệt là trong lĩnh vực dịch vụ web và ứng dụng web.

PHỤ LỤC 6
MÃ NGUỒN PHẦN MỀM

- [1] Performance-Service App Repository (Linux App):
https://github.com/HODUCVU/Capstone-Project_Linux-App.git
- [2] Performance-Viewer App Repository (Windows App):
https://github.com/Wb1305/Project_Capstone_WindownApp.git