TRƯỜNG ĐẠI HỌC BÁCH KHOA - ĐẠI HỌC ĐÀ NẴNG

KHOA ĐIỆN TỬ VIỄN THÔNG

(Hợp tác giữa Khoa Điện tử Viễn thông, Trường Đại học Bách khoa – Đại học Đà Nẵng
và
Công ty FPT Software)

BÁO CÁO GIỮA KỲ
ĐỒ ÁN TỐT NGHIỆP - CAPSTONE PROJECT

Đề tài:
PHẦN MỀM GIÁM SÁT VÀ TÙY CHỈNH TẢI HỆ THỐNG IVI

Thực hiện 1: Hồ Đức Vũ - 106200284 - 20KTMT2
Thực hiện 2: Nguyễn Minh Phương - 106200241 - 20KTMT1

Hướng dẫn 1: TS. Ngô Minh Trí
Hướng dẫn 2: KS. Phan Hồng Sang
Hướng dẫn 3: KS. Nguyễn Việt Đức

Đà Nẵng, Ngày 7 tháng 5 năm 2025

Mục lục

1 Nền tảng lý thuyết **3**

2 Thiết kế và phát triển phần mềm **5**

2.1 Sơ đồ khối phần mềm 5

2.1.1 Các chức năng của phần mềm 6

2.1.2 Trình tự hoạt động theo thời gian 7

2.2 Kết nối Linux App và Windows App thông qua TCP/IP 8

2.3 Thu thập và hiển thị dữ liệu tải hệ thống 10

2.3.1 Thu thập dữ liệu tải của hệ thống IVI tại Linux App 10

2.3.2 Hiển thị dữ liệu tải của hệ thống IVI tại Windows App 11

2.4 Lưu trữ dữ liệu vào database 15

2.5 Phát hiện và xử lý quá tải 16

2.6 Stress Test hệ thống 20

3 Kết quả đạt được **21**

4 Kế hoạch **24**

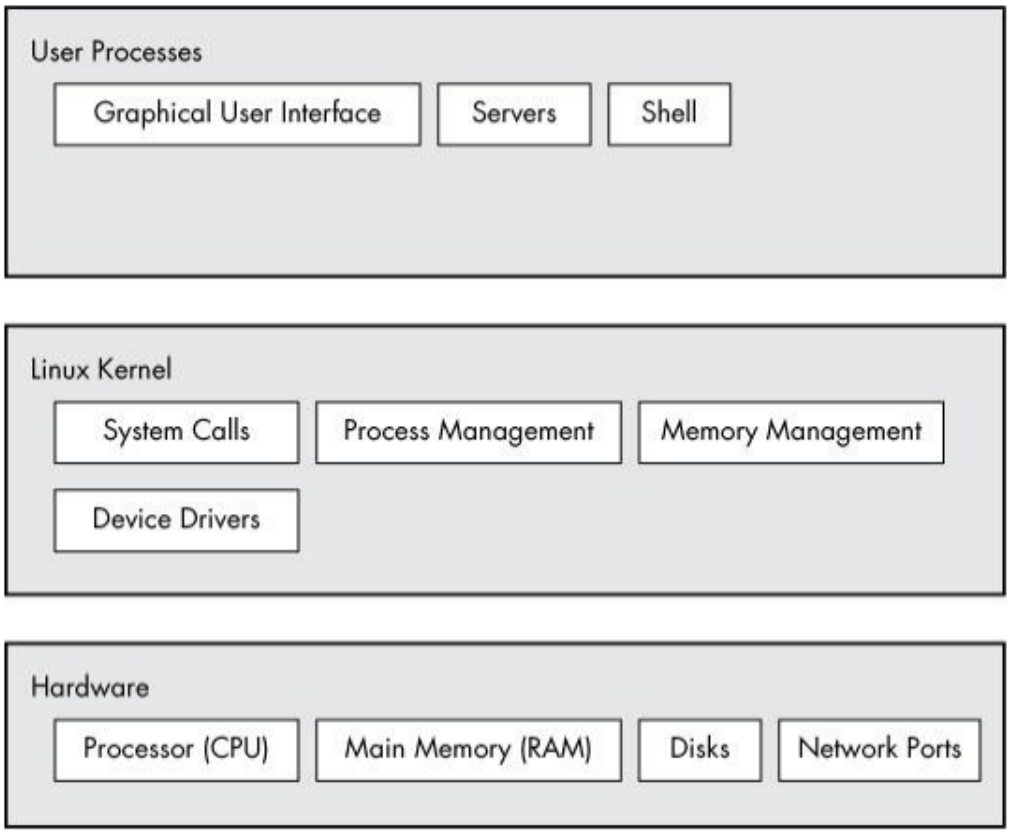
Tài liệu tham khảo **27**

Phụ lục **28**

1 Nền tảng lý thuyết

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... Trong đề tài này, chúng ta sẽ theo dõi tài nguyên của hệ thống và đánh giá trạng thái tải hiện tại, dựa vào tính chất và mức độ ảnh hưởng của từng loại tài nguyên hoạt động trên hệ thống IVI, chúng tôi đã chọn lọc hai tài nguyên có vai trò quyết định về đánh giá tình trạng quá tải của hệ thống đó là CPU và bộ nhớ. Để triển khai phần mềm Linux App thu thập dữ liệu tài nguyên đã được chọn lọc (nội dung này được trình bày chi tiết tại các mục tiếp theo) 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 1 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 call. Để chi tiết hơn, chúng ta cần đi vào từng phần, phân tích các yếu tố và nền tảng để ứng dụng vào đề tài.



Hình 1: 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:

1. Process Management (Quản lý tiến trình): Có nhiệm vụ xác định các tiến trình nào được phép sử dụng CPU.

2. Memory Management (Quản lý bộ nhớ): 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 command 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 Main Memory, còn đối với các lệnh như kết thúc tiến trình hay thực hiện Stress Test thì các command shell sẽ "đi" từ Shell tại User Space đến Kernel với System Calls và đến hệ thống phần cứng thông qua Device Drivers

Trong phần mềm này chúng tôi có sử dụng giao thức mạng để truyền/nhận dữ liệu giữa Linux App và Windows App (chi tiết tại §2.2), cụ thể chúng tôi lựa chọn giao thức TCP/IP cho chức năng truyền và nhận dữ liệu giữa hai ứng dụng. Chúng tôi lựa chọn giao thức mạng để sử dụng cho phần mềm dựa trên các yếu tố bao gồm: tính bảo mật, an toàn và đáng tin cậy khi truyền/nhận dữ liệu cùng với đó là 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 ta sẽ tìm hiểu và phân tích một số giao thức mạng, ưu và nhược của chúng để từ đó có cái nhìn trực quan về nhận định này.

Phần mềm được xây dựng dựa trên Qt Framework [3] và trong Qt Framework (Qt Network) chỉ hỗ trợ các giao thức bao gồm TCP, UDP, HTTP/HTTPS [2], MQTT. 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.

Transmission Control Protocol (TCP) là giao thức hướng kết nối, trước khi trao đổi dữ liệu, giữa client và server 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. Ưu điểm của TCP là tính chính xác và tinh 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 sử dụng SSL/TLS 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.

User Datagram Protocol (UDP) 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 giao đoạn

"bắt tay". Ưu điểm của UDP là có tốc độ truyền tải cao, ít overhead vì không cần thiết lập đường truyền trước, không có cơ chế "xử lý lại" khi gói tin bị hỏng hay mất, ngoài ra thì UDP không đảm bảo tính chính xác hay thứ tự gói tin, cơ chế bảo mật cũng rất thấp. Khi xét từ góc nhìn của đề tài này, chúng tôi thấy rằng UDP không phải là lựa chọn phù hợp cho phần mềm của chúng tôi.

Hypertext Transfer Protocol (HTTP) là giao thức tầng ứng dụng dùng cho truyền tải tài liệu siêu văn bản, file và dữ liệu web, trong khi đó thì 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 truyền những loại dữ liệu so với loại dữ liệu được truyền/nhận trong phần mềm cùng với tốt độ truyền tải không tương thích tối ưu bằng TCP/IP khi xét trên phạm vi mạng LAN, trong tương lai khi chúng tôi mở rộng kết nối giữa Linux App và Windows App ở hai mạng LAN khác nhau thì việc sử dụng HTTP/HTTPS sẽ là lựa chọn tốt nhất.

Message Queuing Telemetry Transport (MQTT) là giao thức publish/subscribe, client sẽ kết nối đến một MQTT broker qua giao thức TCP và subscribe vào các topic. Về cơ bản thì giao thức MQTT được xây dựng dựa trên nền tảng giao thức TCP và không được hỗ trợ chính thức trong Qt Framework, việc cần sử dụng các broker trung gian làm tăng độ phức tạp khi triển khai và có thể khiến cho dữ liệu bị "rò rỉ" làm giảm đi mức an toàn thông tin.

Từ những phân tích ở trên, chúng ta có thể dễ dàng nhận thấy rằng việc triển khai giao thức TCP/IP cho phần mềm cho lựa chọn phù hợp và hiệu quả nhất.

2 Thiết kế và phát triển phần mềm

Dựa trên mục tiêu sản phẩm, chúng tôi xác đinh các chức năng chính bao gồm:

1. Thu thập dữ liệu tải và hiển thị dữ liệu này một cách trực quan, sinh động.
2. Lưu trữ dữ liệu tải với mục đích trích xuất dữ liệu quá khứ để đánh giá chuẩn xác về hiệu suất hoạt động của hệ thống IVI.
3. Phát hiện kịp thời và xử lý tình trạng quá tải hệ thống IVI.
4. Thực hiện kiểm thử phần mềm với Stress Test, đảm bảo phần mềm cũng như hệ thống có thể hoạt động tốt trong điều kiện tải cao.

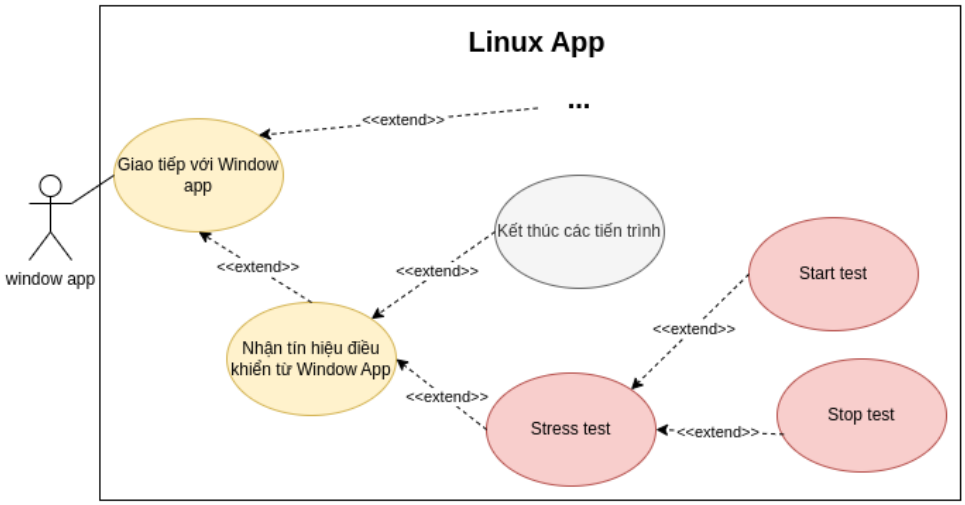
Các mục tiếp theo sẽ mô tả chi tiết về các chức năng này và cách mà chúng tôi triển khai nó trong phần mềm.

2.1 Sơ đồ khối phần mềm

Các sơ đồ khối thiết kế phần mềm chúng tôi đã trình bày trong báo cáo lần 1, vì vậy tại phần này chúng tôi sẽ chỉ đề cập đến những bổ sung mới mà chúng tôi cập nhật trong quá trình triển khai và phát triển phần mềm.

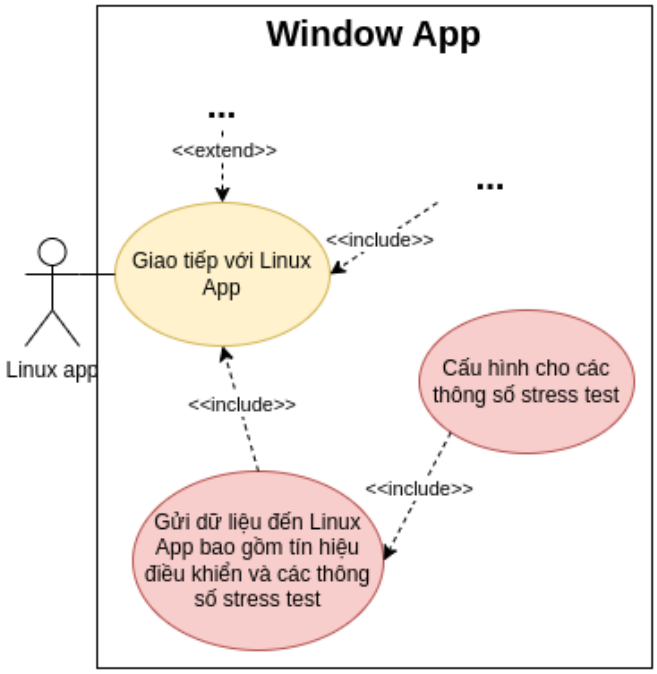
2.1.1 Các chức năng của phần mềm

Đối với Linux App (như mô tả tại Hình 2), chúng tôi thiết kế bổ sung chức năng Stress Test hệ thống (chi tiết tại §2.6), với hai vai trò là bắt đầu Stress Test và dừng Stress Test. Để bắt đầu Stress Test, Linux App sẽ nhận gói dữ liệu chứa các thông số và tín hiệu bắt đầu Stress Test từ Windows App (chi thiết tại §2.2), và khi muốn dừng Stress Test thì Linux App cũng sẽ nhận tín hiệu kết thúc Stress Test từ Windows App hoặc chờ cho đến khi thời gian Stress Test kết thúc như thiết lập.



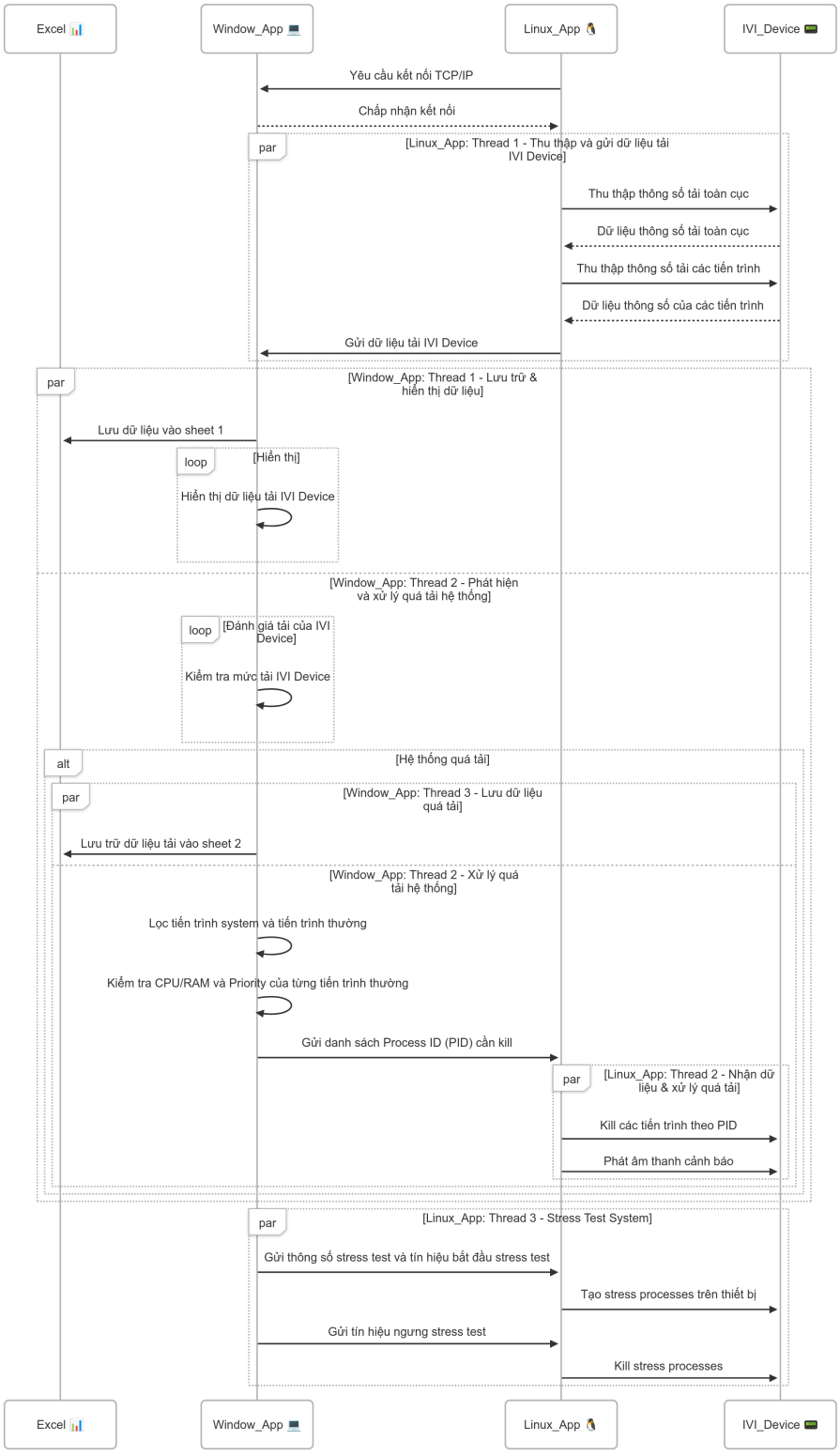
Hình 2: Các chức năng và mối quan hệ của chúng trong Performance Service (Linux App).

Đối với Windows App (như mô tả tại Hình 3), tại đây chúng tôi thiết kế thêm chức năng cấu hình các thông số cho Stress Test và gửi các tín hiệu bắt đầu hoặc kết thúc Stress Test đến Linux App.



Hình 3: Các chức năng và mối quan hệ của chúng trong Performance Viewer App (Windows App).

2.1.2 Trình tự hoạt động theo thời gian



Hình 4: Sơ đồ thể hiện luồng hoạt động theo thời gian của phần mềm.

Trước khi thêm chức năng Stress Test thì Linux App có hai luồng hoạt động, một luồng (luồng thứ 1) thu thập dữ liệu tải và gửi dữ liệu đó đến Windows App, luồng còn lại (luồng thứ 2) nhận tín hiệu điều khiển tải từ Windows App và tiến hành kết thúc các tiến trình theo tín hiệu đó. Việc Stress Test

đòi hỏi một luồng thực thi mới, trong khi chúng tôi cố gắng tích hợp nó vào luồng thứ 2 nhưng khi thực hiện Stress Test thì nó chặn luôn luồng thực thi này khiến cho việc kết thúc các tiến trình phải chờ cho đến khi Stress Test kết thúc, điều này là sai logic đối với cách hoạt động của phần mềm vì vậy chúng tôi đã xây dựng một luồng độc lập cho Stress Test trên Linux App (như thể hiện tại Hình 4) để có thể thực hiện chức năng cân bằng tải của phần mềm ngay cả khi đang thực hiện Stress Test một cách đúng logic.

2.2 Kết nối Linux App và Windows App thông qua TCP/IP

Như đã đề cập trước đó, chúng tôi sử dụng giao thức TCP/IP để truyền/nhận dữ liệu giữa Linux App và Windows App, được triển khai thông qua Qt Framework [3] với QTcpSocket, nơi mà Windows App đóng vai trò như server và Linux App đóng vai trò như client.

Trong phần mềm này, chúng tôi chia ra 4 "Listing" dữ liệu/tín hiệu (gọi chung là gói tin) được truyền/nhận giữa hai ứng dụng. Với mỗi Listing ở bên dưới là một loại gọi tin tương ứng với các chức năng của phần mềm, để hiểu rõ hơn thì chúng ta sẽ đi phân tích từng gói tin và cách mà chúng tôi triển khai các gói tin này trong phần mềm.

Đầy tiên là Listing 1, đây là cấu trúc dữ liệu được truyền từ Linux App đến Windows App, gói tin này chứa thông tin dữ liệu tải đã được thu thập bởi Linux App bao gồm tải của toàn hệ thống như CPU, các lõi CPU, bộ nhớ và tải của các tiến trình đang hoạt động. Với `timestamp` dùng để biết thời gian mà dữ liệu này được thu thập, mục đích là để khi lưu dữ liệu này vào cơ sở dữ liệu thì ta có thể biết rằng nó được thu thập từ khi nào, từ đó có những đánh giá đúng về trạng thái của hệ thống. `SystemStats` bao hàm các thông số của CPU toàn cục (`GeneralCPU`), các lõi CPU (`coresCPU`) và cả bộ nhớ của hệ thống (`MEM`), các thông số này được dùng để đánh giá tải hiện tại của hệ thống IVI đang ở tình trạng nào (như đã đề cập tại báo cáo lần 1). `ProcessesStats` chứa các thông tin về ID của tiến trình (`PID`), đối tượng đang thực thi nó (`User`), tên (`PName`) và các thông số tải của nó bao gồm phần trăm CPU đang chiếm dụng (`PCPUUsagePercent`), phần trăm bộ nhớ (cả private memory và share memory - `MEMUsagePercent`), những thông số này được dùng để đánh giá tiến trình nào nên được kết thúc để cân bằng tải của hệ thống trong trường hợp hệ thống IVI bị quá tải.

Tiếp theo là Listing 2, đây là cấu trúc dữ liệu được truyền từ Windows App đến Linux App, mục đích của gói tin này dùng để kết thúc tiến trình để cân bằng tải khi hệ thống IVI bị quá tải. Listing 2 chứa hai thông tin là kiểu gói tin (`type`) và tên tiến trình cần kết thúc (`PName`). Vì dữ liệu truyền từ Windows App đến Linux App đa dạng với 3 loại gói tin là Listing 2, Listing 3 và Listing 4 nên tại Linux App ta cần xác định gói tin này thuộc loại nào trước khi tiến hành xử lý nó, để làm được điều này thì ta chỉ cần truy xất kiểu gói tin tại `type` là có thể biết được, đối với gói tin dùng để kết thúc tiến trình thì nó được thiết lập mặc định là `type: killProcess` . Trong quá trình phát triển và kiểm thử phần mềm, chúng tôi nhận thấy rằng việc gửi một tiến trình để kết thúc và sau đó kiểm tra lại trạng thái hệ thống sẽ hợp lý hơn là việc gửi một danh sách với nhiều tiến trình để kết thúc một lúc (chi tiết tại §2.5), vì vậy mỗi gói tin này sẽ chứa tên của một tiến trình bị kết thúc (`PName`).

Đối với Listing 3, đây là gói tin dùng để thực hiện chức năng kiểm thử phần mềm thông qua Stress Test. Với kiểu gói tin mặc định là `type: startStress` , các thông số trong gói tin này được thiết lập tại Windows App bao gồm: `numberOfTaskToRun` dùng để tạo ra số lượng tiến trình ảo để chạy trên Linux App, `MEMUsagePercent` là phầm trăm bộ nhớ mà các tiến trình ảo chiếm dụng (tối đa là 90%), `numberOfCore` là số lõi CPU mà các tiến trình ảo này thực hiện chạy tối đa công suất, ví dụ như ta thiết lập `numberOfCore` là 7 thì tiến trình ảo sẽ chạy với tối đa công suất 100% của 7 trên 8 lõi CPU

Listing 1 Cấu trúc dữ liệu truyền từ Linux App đến Windows App - Dữ liệu tải trên hệ thống IVI

```
{
    "timestamp": "2025-04-03 15:47:12",
    "SystemStats" : {
        "GeneralCPU" : {
            "CPUUtilization" : 30,
            "CPUTemperature" : 30,
            "CPUFrequency" : 3000,
            "CPUFrequencyPercent" : 30,
        },
        "coresCPU" : {
            "0" : {
                "CPUUtilization" : 30,
                "CPUTemperature" : 30,
                "CPUFrequency" : 3000,
            },
            ...
            "7" : {
                "CPUUtilization" : 30,
                "CPUTemperature" : 30,
                "CPUFrequency" : 3000,
            },
        },
        "MEM" : {
            "RAMUsage" : 12345,
            "RAMPercent" : 15,
            "MaxRAM": 16000;
            "SWAPUsage" : 0,
            "SWAPPercent" : 0,
            "MaxSWAP": 4000,
        },
    },
    "ProceesesStats" : {
        "321": {
            "PID": 321,
            "User": duc-vu,
            "PName": chrome,
            "PCPUUsagePercent": 15,
            "PMEMUsageMB": 2000,
            "PMEMUsagePercent": 10,
        },
        ...
    },
}
```

Listing 2 Cấu trúc dữ liệu truyền từ Windows App đến Linux App - Kết thúc tiến trình trên hệ thống IVI

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

của hệ thống (giả sử hệ thống IVI có tối đa là 8 lõi), `timeout` là thời gian thực hiện Stress Test, có đơn vị là giây. Tại Linux App, sau khi nhận gói tin này thì Linux App sẽ tiến hành thiết lập các thông số và bắt đầu Stress Test.

Listing 3 Cấu trúc dữ liệu truyền từ Windows App đến Linux App - Thông số và lệnh bắt đầu Strest Test hệ thống

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

Trong thời gian thực hiện Stress Test, có hai cách để kết thúc quá trình này, cách thứ nhất là chờ cho đến khi thời gian Stress Test kết thúc như thiết lập tại `timeout`, cách thứ hai là chúng ta có thể kết thúc nó thông qua lệnh dừng Stress Test tại Windows App với gói tin như Listing 4, gói tin này chỉ có một thông tin duy nhất là kiểu gói tin `type: stopStress`, khi Linux App nhận gói tin này, nó sẽ tiến hành kết thúc Stress Test ngay lập tức.

Listing 4 Cấu trúc dữ liệu truyền từ Windows App đến Linux App - Lệnh dừng Stress Test hệ thống

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

2.3 Thu thập và hiển thị dữ liệu tải hệ thống

2.3.1 Thu thập dữ liệu tải của hệ thống IVI tại Linux App

Các dữ liệu tải được thu thập bao gồm các thông số về CPU, bộ nhớ và các thông số của các tiến trình đang chạy trên hệ thống IVI (như đã đề cập tại §2.2 - Listing 1) thông qua các command shell mà chúng tôi liệt kê bên dưới (Các command shell này được triển khai thông qua QProcess khi chạy trên Linux App).

Đối với bộ nhớ, chúng tôi cần hai loại dữ liệu là RAM (Mem) và Swap, trong hệ điều hành Linux, chúng ta có thể sử dụng lệnh `free -m` để lấy các thông số về hai loại dữ liệu 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 (total), lượng bộ nhớ đang bị chiếm dụng (used) và bộ nhớ trống (free), các thông số này đều có đơn vị là megabyte (MB). Trong phần mềm này, chúng tôi cần thông số tổng bộ nhớ và bộ nhớ đang bị chiếm dụng của từng loại,

để có thể trích xuất các thông số như mong muốn thì ta 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}'
```

Đối với CPU, chúng ta cần các thông số về phần trăm đang sử dụng, nhiệt độ và tần số hoạt động CPU trên toàn hệ thống và trên từng lõi CPU. Các lệnh được sử dụng để thu thập các thống số đó 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`, ta 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 của toàn hệ thống và trên từng lõi, lệnh này sẽ lấy trung bình trong 1 giây, tăng độ chuẩn xác cho dữ liệu thu thập. Ta có thể lấy nhiệt độ của CPU và nhiệt độ trên từng lõi CPU 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 của từng lõi CPU. 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 cũng chỉ được thực thi một lần ngay thời điểm Linux App được khởi động, đối với tần số hoạt động hiện tại của CPU thì ta có thể thu thập thông qua lệnh `lscpu | grep 'MHz' | awk 'NR==1 print $NF'`, lệnh này giúp chúng ta biết được hiện tại CPU đang hoạt động bao nhiêu phần trăm ở mức tần số tối đa nó có thể hoạt động. 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 ta 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'
```

Sau khi tiến hành thu thập các dữ liệu tải ở trên, Linux App sẽ thực hiện "đóng gói" và truyền gói tin như Listing 1 đến Windows App, những dữ liệu này sẽ được hiển thị một cách trực quan trên Windows App cùng với đó nó được dùng để đánh giá tình trạng tải hiện tại của hệ thống IVI và đưa ra biện pháp điều chỉnh tải nếu cần thiết.

2.3.2 Hiển thị dữ liệu tải của hệ thống IVI tại Windows App

Ứng dụng Windows App được thiết kế theo kiến trúc Model-View-ViewModel (MVVM), phân tách rõ ràng giữa giao diện người dùng (View), dữ liệu và logic nghiệp vụ (Model), và lớp trung gian (ViewModel)

giúp kết nối hai phần này. Các thành phần chính bao gồm: SystemMonitor (lớp quản lý việc thu thập dữ liệu hệ thống từ Linux App qua TCP), các lớp Model/ViewModel (đại diện cho trạng thái CPU, RAM, danh sách tiến trình,... để cung cấp cho giao diện), và giao diện QML hiển thị các thông tin này một cách trực quan. Dữ liệu từ thiết bị Linux được gửi qua giao thức TCP đến ứng dụng; SystemMonitor chạy trên Main thread nhằm lắng nghe kết nối TCP, nhận gói tin dữ liệu (như đã đề cập tại §2.2 - Listing 1) và chuyển tiếp cho lớp xử lý tiếp theo.

Quá trình nhận và xử lý sẽ diễn ra như sau: Đầu tiên IviSocketServer (thành phần chứa QTcpServer / QTcpSocket) chấp nhận kết nối từ Linux App và bắt đầu nhận các gói dữ liệu thô (dạng QByteArray). Tiếp đó, dữ liệu thô được chuyển đến lớp DataProcessor để phân tách thành các cấu trúc dữ liệu nội bộ. Cụ thể, DataProcessor sẽ tách QByteArray nhận được thành hai phần: một là các thông số tải của hệ thống, là một cấu trúc `SystemStats` chứa các thống tin về CPU/MEM hiện tại. Hai là thông tin chi tiết của các tiến trình, là một `QList<ProcessInfo>` liệt kê các tiến trình đang chạy cùng thông tin sử dụng tài nguyên của chúng. Kết quả phân tích này được quản lý bởi SystemMonitor – đối tượng chịu trách nhiệm cập nhật trạng thái hiện tại của hệ thống.

Trong SystemMonitor, các số liệu mới được cập nhật sẽ được truyền đến lớp ViewModel tương ứng. SystemMonitor sẽ sử dụng cơ chế signal-slot để phát đi tín hiệu mỗi khi có số liệu mới vừa được cập nhật; các ViewModel lắng nghe tín hiệu này và cập nhật giá trị của mình. Qt cung cấp cơ chế signal-slot cho phép các đối tượng giao tiếp bất đồng bộ theo mô hình phát tín hiệu và xử lý tín hiệu. Ví dụ, lớp SystemMonitor sau khi nhận và xử lý gói tin sẽ emit một tín hiệu `systemStatsUpdated()`. Tín hiệu này được kết nối `(QObject::connect)` tới các slot thích hợp, chẳng hạn slot của SystemStatsViewModel để cập nhật lại các giá trị SystemStats mới và slot của ProcessListViewModel để cập nhật lại danh sách tiến trình. Khi slot thực thi, nó sẽ điều chỉnh dữ liệu trong model/viewmodel (cập nhật giá trị thuộc tính, thực thi các chức năng,...) và thường kích hoạt các notify signal của Q_PROPERTY hoặc các signal dataChanged() của model, nhờ đó giao diện QML biết có sự thay đổi.

```
// Setup connect signal-slot in Qt
// SystemStatsViewModel
connect(monitor, &SystemMonitor::systemUpdated,
        this,
        [this](const SystemStats& systemStats, const QVector<ProcessInfo>&)
        {
            updateFromStats(systemStats);
        });
// ProcessListViewModel
QObject::connect(monitor, &SystemMonitor::systemUpdated,
         this,
         [this](const SystemStats& systemStats,
         const QVector<ProcessInfo>& listProcess)
         {
            updateList(systemStats, listProcess);
         });
```

Q_PROPERTY: Qt sử dụng macro Q_PROPERTY để khai báo các thuộc tính của QObject có thể truy cập từ QML. Mỗi Q_PROPERTY xác định kiểu dữ liệu, phương thức đọc (getter), phương thức ghi (setter, tùy chọn) và một tín hiệu NOTIFY (tùy chọn, nhưng cần có để thông báo thay đổi cho QML). Trong lớp ViewModel (ví dụ SystemStatsViewModel), ta định nghĩa các thuộc tính như:

```
// Setup Context Property in trong Qt
Q_PROPERTY(double ramPercent READ ramPercent NOTIFY infoChanged)
```

Điều này khai báo một property ramPercent có kiểu double, getter ramPercent()và notify signal infoChanged(). Khi giá trị ram thay đổi, code C++ sẽ gọi emit infoChanged(). Khi đó, cơ chế property binding của QML sẽ tự động cập nhật mọi UI binding liên quan đến ramPercent. Nói cách khác, Q_PROPERTY + notify signal giúp giao diện luôn cập nhật kịp thời với những thay đổi. Nếu không khai báo notify, QML sẽ không biết property đã đổi và UI sẽ không tự cập nhật. Macro Q_PROPERTY thực chất cấu hình sẵn các phương thức và tín hiệu cần thiết để QML có thể sử dụng property đó.

Và để QML có thể sử dụng các property, các phương thức ta cần phải Binding đối tượng C++ vào QML. Có hai cách phổ biến để đưa đối tượng C++ sang cho QML sử dụng: (1) Context Property – thêm instance C++ vào context của QML, và (2) Đăng ký kiểu (qmlRegisterType) – để có thể khởi tạo đối tượng C++ trực tiếp trong QML. Cách (1) thường dùng khi có một đối tượng singleton(là một đối tượng duy nhất, tạo ra một lần, dùng chung xuyên suốt chương trình). Các đối tượng như SystemStatsViewModel (giữ trạng thái thống kê CPU, RAM), ProcessListViewModel (quản lý danh sách tiến trình) chỉ cần tạo ra một lần duy nhất, không cần thiết tạo ra nhiều đối tượng ProcessListViewModel hay SystemStatsViewModel, vì chúng chỉ quản lý đúng một trạng thái chung duy nhất. Trong Qt, nó thường được thêm vào QML thông qua `setContextProperty` để mọi thành phần QML đều dùng chung dễ dàng.:

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

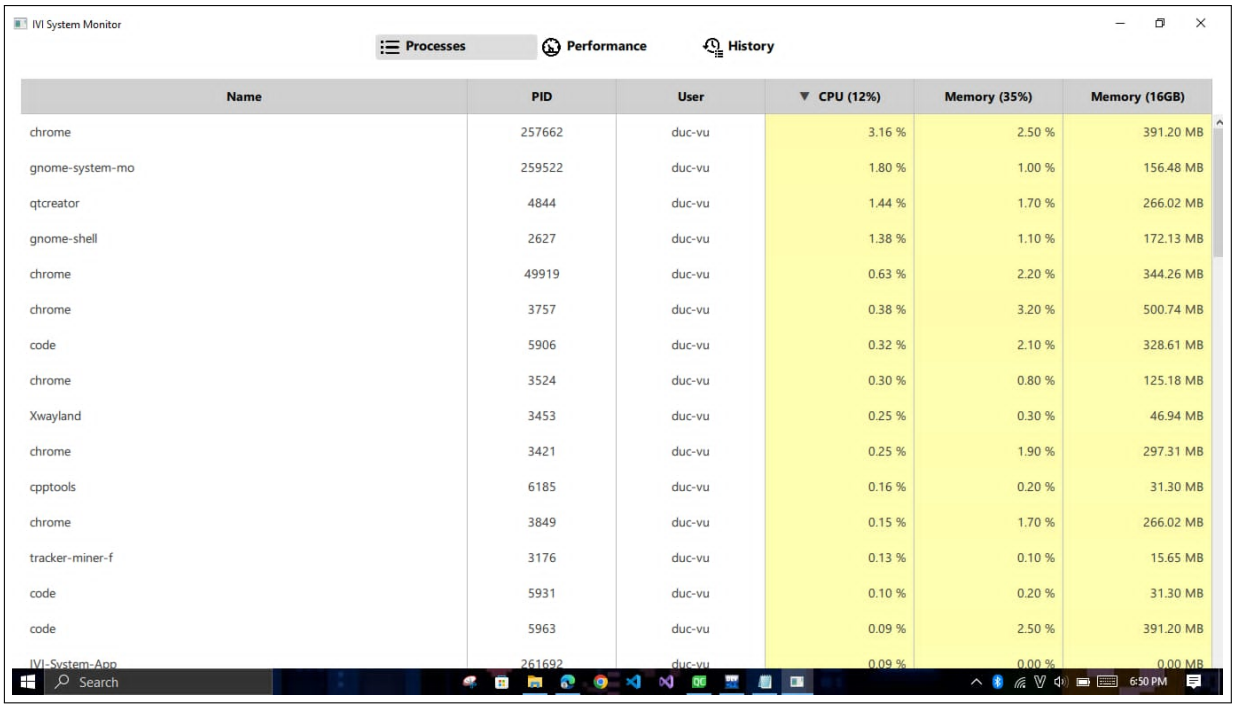
Sau đó trong QML có thể truy cập `systemStatsViewModel.ramPercent`. Cách này đơn giản và phù hợp nếu chỉ có một instance cần dùng ở QML. Cách (2) dùng khi ta muốn tạo nhiều đối tượng hoặc quản lý vòng đời bởi QML. Thay vì tạo sẵn đối tượng ở C++ rồi gửi vào QML như cách singleton, thì ta phải đăng ký một kiểu đối tượng vào QML thông qua `qmlRegisterType`.

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

rồi trong QML có thể làm `ProcessListViewModel  id: procModel` để tạo model mới. Trong đồ án này, hầu hết các thành phần như ViewModel và Model chỉ cần tạo duy nhất một lần rồi dùng chung cho toàn ứng dụng (instance toàn cục) Vì thế, sử dụng instance toàn cục thông qua Context Property là phương pháp hiệu quả và thuận tiện nhất.

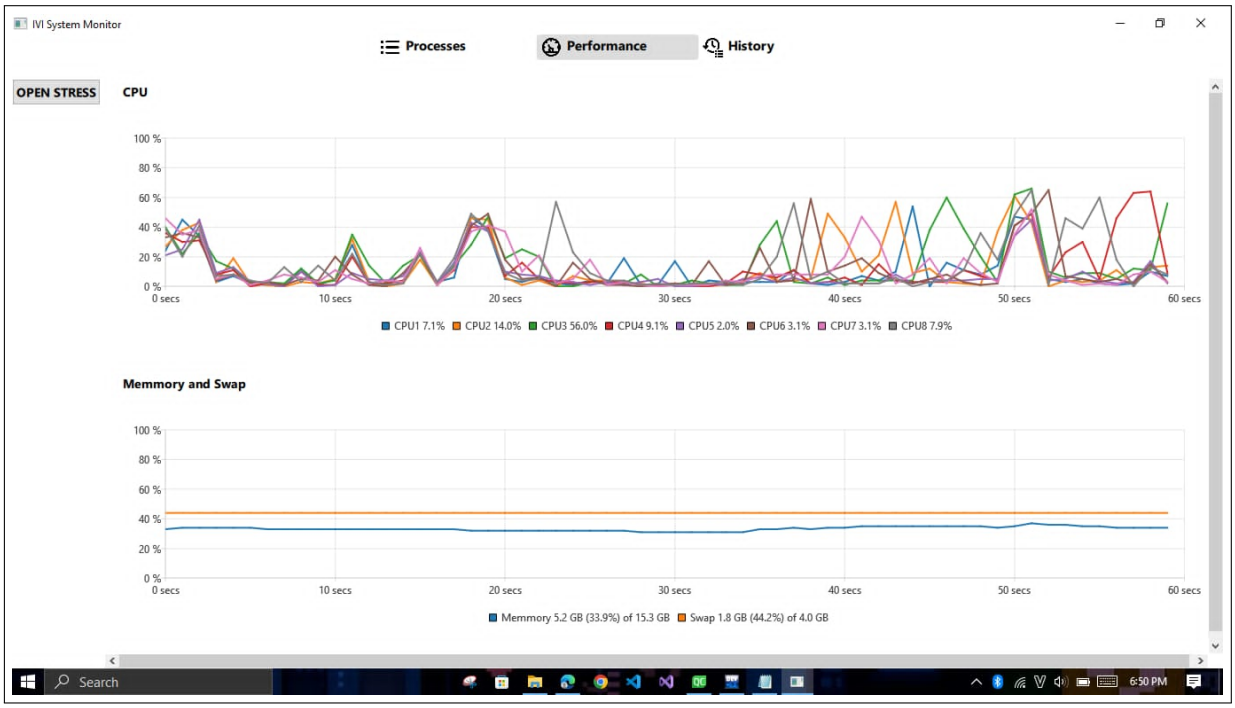
Sử dụng TableView trong QML kết hợp với QAbstractTableModel trong C++ để quản lý dữ liệu và hiển thị danh sách tiến trình trên giao diện người dùng (Hình 5). QAbstractTableModel là lớp trừu tượng Qt cung cấp để tạo mô hình dữ liệu dạng bảng. Trong bối cảnh ứng dụng, ProcessListViewModel kế thừa QAbstractTableModel để quản lý danh sách các ProcessInfo. Việc sử dụng QAbstractTableModel cho phép ta định nghĩa số hàng, số cột và cách truy xuất dữ liệu cho mỗi ô (thông qua phương thức data(index, role)). Mỗi ProcessInfo có thể bao gồm các trường như PID, tên tiến trình, CPU, RAM... sẽ tương ứng với các cột trong bảng. Lớp model này cũng định nghĩa roleNames() để cung cấp tên role cho QML (ví dụ "pid", "name", "cpuUsage", "memUsage"), giúp QML truy cập dữ liệu dễ dàng. Khi SystemMonitor cập nhật danh sách tiến trình mới, ProcessListViewModel có thể phát các tín hiệu như beginResetModel()/endResetModel() hoặc sử dụng các hàm beginInsertRows()/endInsertRows()... để thông báo cho view rằng dữ liệu đã thay đổi. Nhờ đó, TableView trong QML biết để cập nhật lại danh sách. TableView (Qt Quick Controls) được thiết kế cho dữ liệu dạng bảng với nhiều cột. Nó cho phép định nghĩa các cột rõ ràng kèm tiêu đề, và ánh xạ trực tiếp đến các role của model hoặc các cột của QAbstractTableModel.

Đối với các giá trị CPU từng core, MEM, để dễ dàng quan sát xu hướng biến đổi liên tục theo thời



Hình 5: Hiển thị dữ liệu các tiến trình trên hệ thống.

gian. Tôi chọn biểu đồ đường (Line Chart) là cách biểu diễn trực quan nhất (như Hình 6). Qt cung cấp mô-đun QtCharts cho phép tạo biểu đồ 2D (Line, Bar, Pie, Scatter, v.v.) dễ dàng và tích hợp với QML (qua QtCharts::ChartView). Và sử dụng LineSeries trong ChartView, để có thể vẽ đường biểu diễn các thông số tải theo trục thời gian.



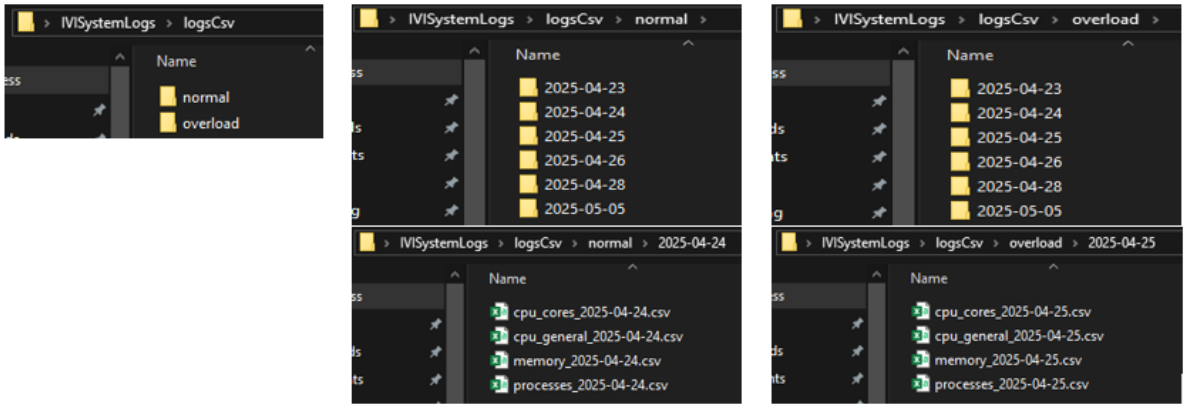
Hình 6: Hiển thị dữ liệu toàn hệ thống.

2.4 Lưu trữ dữ liệu vào database

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. Dưới đây là cấu trúc thư mục nhằm lưu trữ dữ liệu.

```
IVISystemLogs/
     logsCsv/
          normal/yyyy-MM-dd/
          overload/yyyy-MM-dd/
```

Trong Hình 7, thư mục gốc "IVISystemLogs/logCsv" chứa 2 thư mục chính là "normal" và "overload". Thư mục "normal" chứa dữ liệu định kỳ theo từng ngày, trong đó dữ liệu nhận được sẽ lưu vào các file csv tương ứng. Ở đây, ta có 4 file csv lần lượt là "cpu_cores_YYYY-MM-DD.csv" (mức sử dụng CPU trên từng core), "cpu_general_YYYY-MM-DD.csv" (mức sử dụng CPU tổng thể), "memory_YYYY-MM-DD.csv" (trạng thái bộ nhớ RAM và SWAP), "processes_YYYY-MM-DD.csv" (danh sách và chỉ số tài nguyên tiến trình).

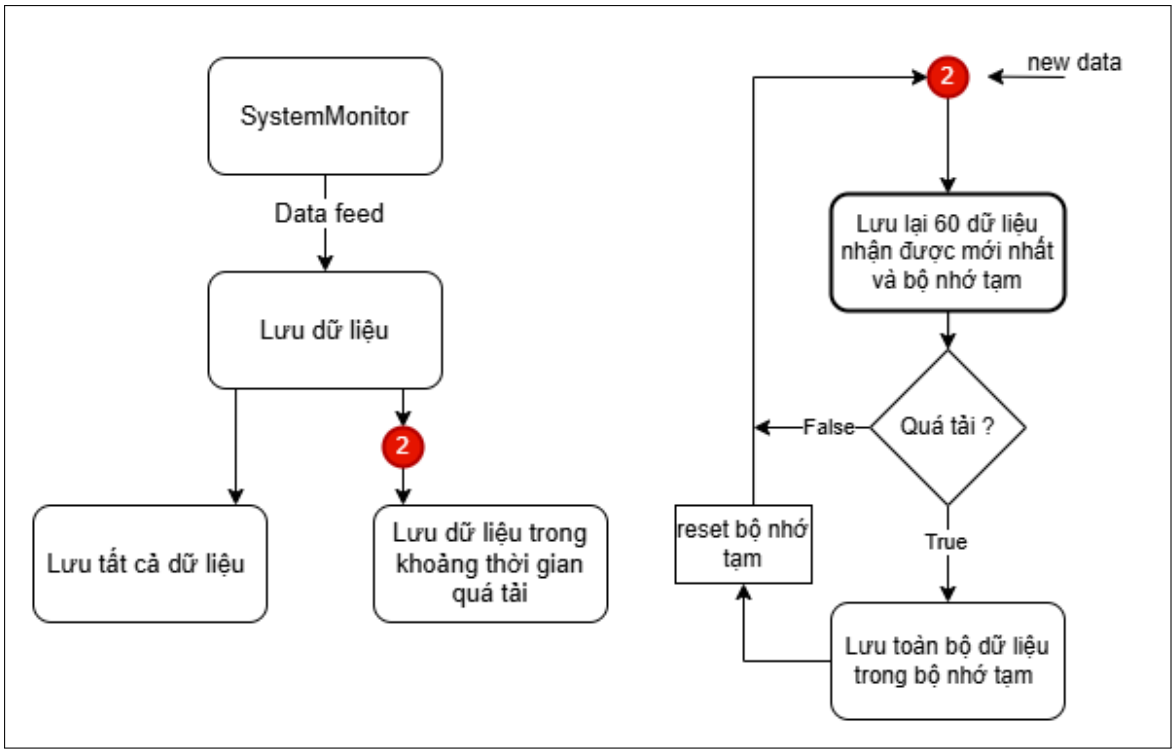


Hình 7: Cấu trúc thư mục lưu file dữ liệu.

Hình 8 mô tả sơ đồ hoạt động của quá trình lưu trữ dữ liệu trong hệ thống. Quy trình được chia thành hai luồng chính:

1. Luồng lưu trữ liên tục: Từ dữ liệu đầu vào do SystemMonitor cung cấp, 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 khi quá tải: Một nhánh song song sẽ xử lý dữ liệu theo hướng kiểm tra trạng thái tải. Với mỗi dữ liệu mới nhận được, đều được lưu vào trong bộ nhớ tạm gồm các dữ liệu mới nhất trong 60 giây trước khi hệ thống quá tải. Khi hệ thống được xác định là quá tải thì ngay lập tức toàn bộ dữ liệu trong bộ nhớ đệm sẽ được lưu lại và bộ nhớ tạm sẽ được xoá toàn bộ giá trị. Và tiếp tục quá trình lưu trữ.



Hình 8: Sơ đồ hoạt động mô tả quá trình lưu trữ dữ liệu.

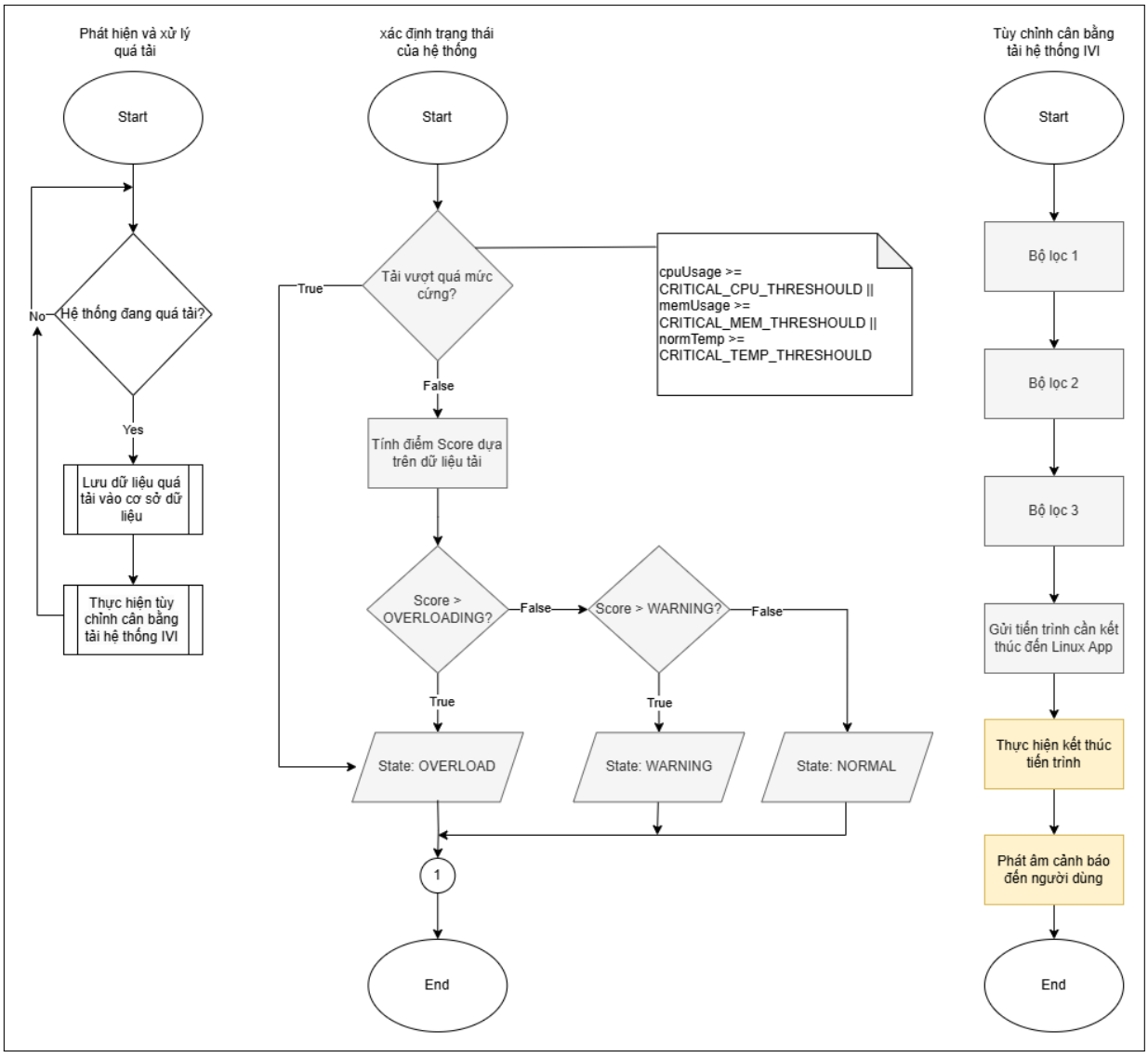
2.5 Phát hiện và xử lý quá tải

Tại Windows App, sau khi nhận dữ liệu tải từ Linux App, bên cạnh việc hiển thị dữ liệu đó thông qua biểu đồ và các task bar thì còn sử dụng các dữ liệu này để chuẩn đoán tình trạng tải hiện tại của hệ thống IVI, nếu kết quả chuẩn đoán kết luận hệ thống IVI đang quá tải thì ta sẽ thực hiện tùy chỉnh để cân bằng lại tải cho hệ thống IVI.

Hình 9 mô tả luồng thực thi chức năng phát hiện và xử lý khi hệ thống IVI bị quá tải. Sau khi nhận dữ liệu tải, Windows App sẽ đánh giá dữ liệu này có đang ở mức quá tải hay không, nếu có thì sẽ tiến hành lưu dữ liệu này vào cơ sở dữ liệu dành riêng cho các thời điểm hệ thống bị quá tải, sau đó tiến hành cân bằng tải như thể hiện qua flow chart thứ nhất tại Hình 9. Để chi tiết hơn nữa, chúng tôi sẽ chia ra làm hai phần để phân tích bao gồm: Phát hiện quá tải hệ thống và tùy chỉnh cân bằng tải hệ thống.

Flow chart thứ hai tại Hình 9 biểu diễn logic phát hiện quá tải hệ thống. Đầu tiên chúng tôi kiểm tra các dữ liệu tải có đang ở mức cao hơn "ngưỡng cứng" hay không, cụ thể các "ngưỡng cứng" bao gồm: Mức CPU toàn hệ thống vượt quá 95%, mức bộ nhớ đang sử dụng với cả RAM và Swap cộng lại vượt quá 190%, nhiệt độ của CPU vượt quá 85°C, nếu một trong các tiêu chí đó thỏa mãn thì chúng tôi kết luận tình trạng hệ thống hiện tại đang bị quá tải. Nếu dữ liệu tải không vượt quá "ngưỡng cứng", chúng tôi tiến hành tính điểm (chuẩn hóa) dữ liệu tải với các trọng số cho mỗi loại dữ liệu tải có tác động đến tình trạng quá tải của hệ thống, các trọng số (các trọng số phát hiện và xử lý quá tải có thể tùy chỉnh trên Windows App như Hình 11) được chúng tôi xác định như bên dưới:

```
WEIGHT_RAM_PERCENT_USAGE 0.35
WEIGHT_SWAP_PERCENT_USAGE 0.25
WEIGHT_CPU_PERCENT_USAGE 0.25
WEIGHT_TEMP_OF_CPU 0.10
WEIGHT_FREQ_PERCENT_OF_CPU 0.05
```

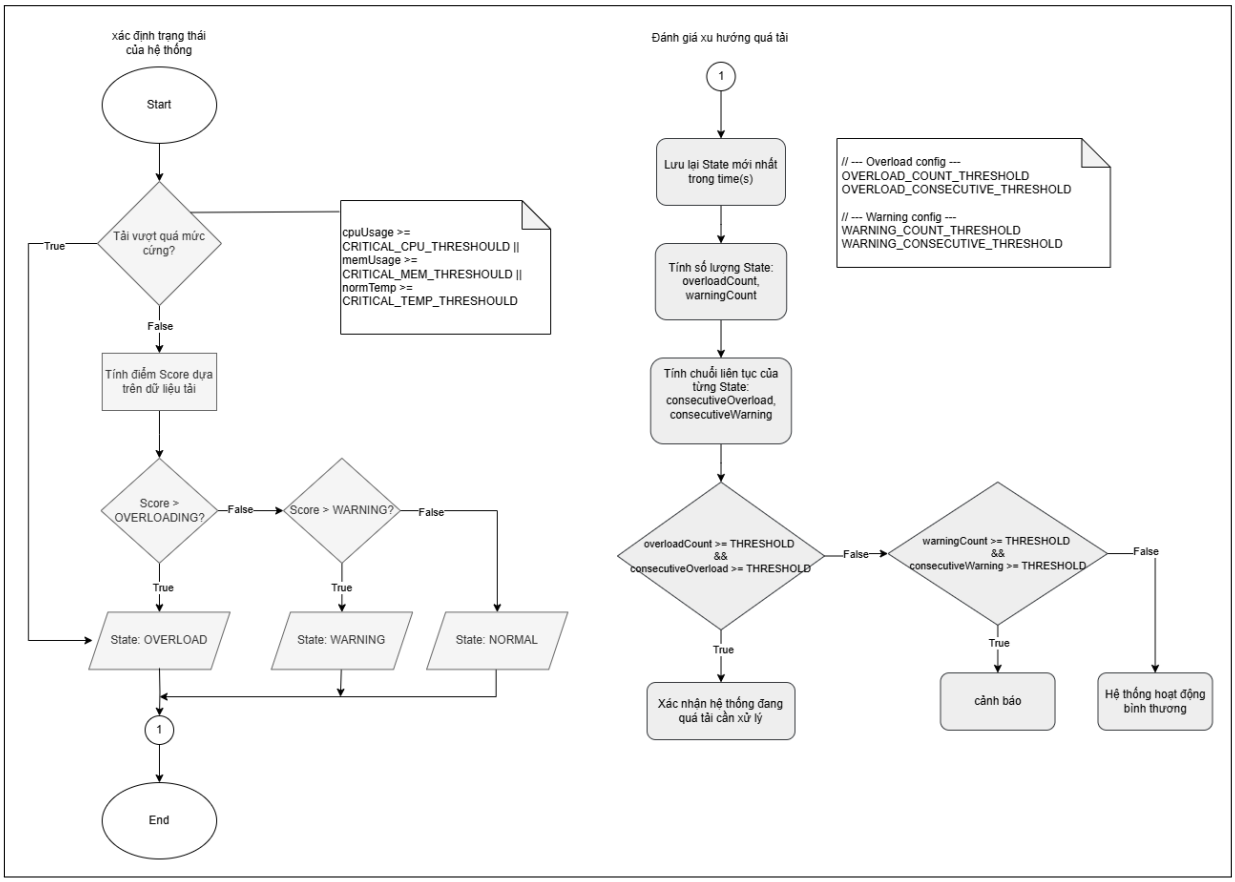


Hình 9: Quá trình phát hiện và xử lý khi hệ thống IVI quá tải.

Tổng của các trọng số là 1, với `WEIGHT_RAM_PERCENT_USAGE` là trọng số cho RAM, tương tự như vậy cho các loại dữ liệu tải khác, đối với nhiệt độ của CPU thì ta cần chuẩn hóa về thang phần trăm trước khi tích hợp nó với trọng số `WEIGHT_TEMP_OF_CPU`, mức nhiệt độ thấp nhất mà ta có thể thu là 40°C và cao nhất là 100°C tương ứng từ 0% đến 100%.

Sau khi tính điểm các dữ liệu tải với các trọng số trên thì ta sẽ thu được kết quả có giá trị từ 0% đến 100%, nếu kết quả này vượt quá 50% (giá trị OVERLOADING làm ngưỡng đánh giá), thì hiện tại hệ thống đang ở trạng thái quá tải, nếu kết quả này vượt quá 45% (giá trị WARNING làm ngưỡng đánh giá), hiện tại hệ thống đang ở trạng thái cảnh báo và nếu không thì tức là hệ thống đang bình thường. Tuy nhiên, nếu chỉ dựa vào một lần phát hiện duy nhất, hệ thống rất dễ đưa ra cảnh báo sai (false alarm) do những dao động tức thời hoặc sự thay đổi đột ngột ngắn hạn, điều này có thể dẫn đến các phản ứng không cần thiết như ngắt tiến trình hoặc cảnh báo người dùng. Để giải quyết vấn đề này, chúng tôi đã áp dụng phương pháp Hybrid – kết hợp giữa số lần vượt ngưỡng (threshold) và độ dài chuỗi liên tiếp (consecutive count) vào quá trình xác định quá tải của hệ thống (như thể hiện tại hình 10). Cụ thể, chỉ khi trạng thái quá tải xuất hiện với tần suất đủ lớn và kéo dài liên tục trong một khoảng thời gian xác định, hệ thống mới phát cảnh báo. Cách tiếp cận này giúp cân bằng giữa độ nhạy và tính ổn định: hệ

thống vẫn phản ứng đủ nhanh khi có nguy cơ quá tải thực sự, nhưng không bị ảnh hưởng bởi các dao động ngắn.



Hình 10: Sơ đồ hoạt động đánh giá quá tải hệ thống.

Trong trường hợp hệ thống bị quá tải, phần mềm sẽ thực hiện tùy chỉnh để cân bằng tải hệ thống, logic của chức năng này được biểu diễn như flow chart thứ ba tại Hình 9. Ta có thể thấy rằng flow chart thứ ba được thể hiện thông qua 2 màu xám và vàng, màu xám tượng trưng cho các quy trình được xử lý tại Windows App và màu vàng tượng trưng cho các quy trình được xử lý tại Linux App.

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 ta kết thúc các tiến trình này thì hệ thống sẽ hoạt động sai lệch và có thể dẫn đến crash toàn bộ hệ thống, ngoài ra còn có các tiến trình có thể không khiến cho hệ thống bị lỗi những cũng đóng vai trò rất quan trọng như tiến trình điều khiển bàn phím, màn hình... Những tiến trình như vậy ta cần đảm bảo rằng phần mềm không được phép kết thúc chúng. 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 động 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 bản 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 Whitelist, sau khi đi qua bộ lọc 2 ta sẽ thu được danh sách những tiến trình thông thường có thể kết thúc.

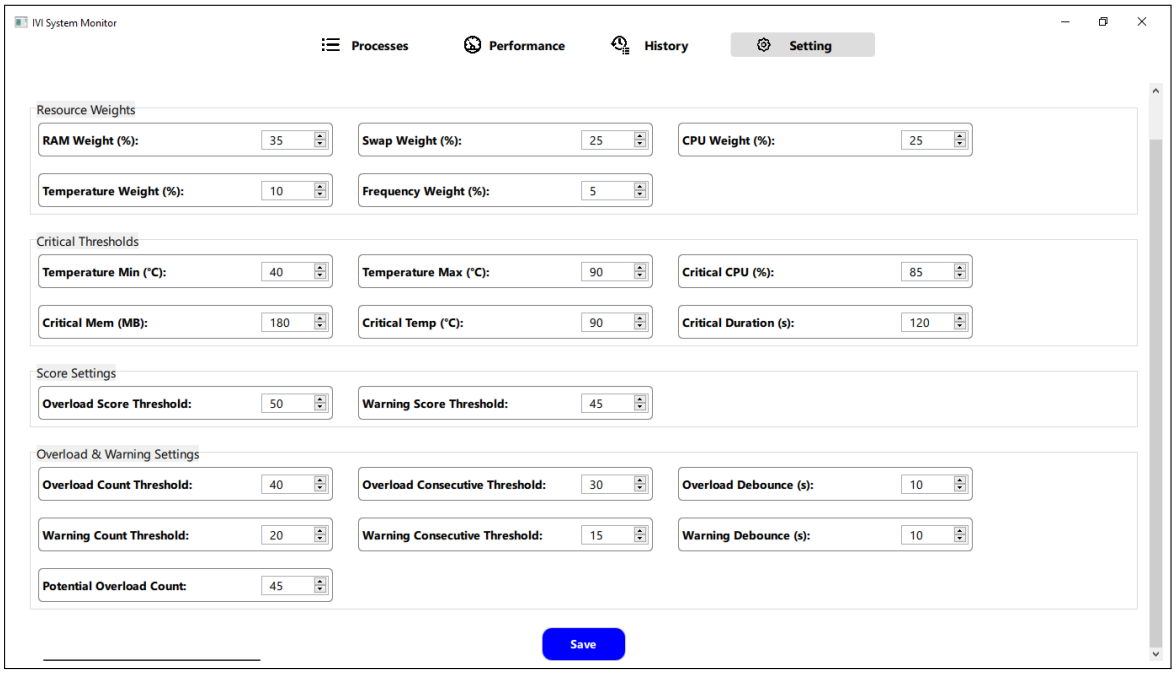
Bộ lọc 3 là bộ lọc xác định tiến trình nào cần kết thúc từ danh sách thu được sau khi qua bộ lọc 2. Tại đây chúng tôi thực hiện tính điểm dựa trên mức CPU, mức bộ nhớ mà tiến trình chiếm dụng kèm theo là mức độ ưu tiên của tiến trình này (giá trị càng cao thì độ ưu tiên càng thấp - mức ưu tiên được dùng để tăng trải nghiệm cho người dùng như đã phân tích ở trên), với trọng số tương ứng là 0.45, 0.45 và 0.1, từ đó chúng tôi tìm ra tiến trình nào có điểm cao nhất và thực hiện gửi gói tin như Listing 2 đến Linux App.

Tại Linux App, sau khi nhận gói tin này, ứng dụng sẽ tiến hành kết thúc tiến trình tương ứng trên hệ thống IVI, ta có thể thực hiện điều này thông qua lệnh bên dưới:

```
$ pkill --exact --echo {PNAME}
```

Khi kết thúc một tiến trình nào đó, thì Linux App đồng thời cũng cảnh báo đến người dùng rằng hệ thống đang quá tải, cần lưu ý khi mở thêm tiến trình hay thao tác các tác vụ nặng. Lệnh bên dưới giúp Linux App có thể tương tác với Speaker trên hệ thống IVI để phát ra âm thanh cảnh báo.

```
$ softbeep -f {frequency} -l {length} -r {repeat}
```



Hình 11: Bảng điều chỉnh các thông số cho chức năng phát hiện và xử lý quả tải

Lý do chúng tôi thực hiện kết thúc một tiến trình mỗi lần thay vì một danh sách tiến trình là vì điều này làm tăng trải nghiệm người dùng hơn, giả sử rằng ta sẽ gửi một danh sách tiến trình cần kết thúc từ Windows App đến Linux App thì danh sách đó nên chứa bao nhiêu tiến trình? Và liệu rằng từng đó trong danh sách là đủ hay thừa/thiếu để giúp hệ thống IVI cân bằng tải? Việc kết thúc một tiến trình

mỗi lần, sau đó quay lại kiểm tra trạng thái tải của hệ thống IVI, cứ như vậy cho đến khi hệ thống cân bằng tải, mặc dù sẽ mất thời gian hơn một chút nhưng mức hiệu quả thì lại cao hơn nhiều.

Các thông số trong Hình 11 được sử dụng để xác định được trang thái của hệ thống đang là NORMAL, WARNING hay OVERLOAD. Các thông số này sẽ được thiết lập mặc định trong file "my_ivi_settings.ini" và nó cũng có thể được tuỳ chỉnh lại giá trị tại màn hình "Setting" của Windows App. Cuối cùng ấn nút "Save" để lưu thay đổi cũng như cập nhật lại toàn bộ giá trị các tham số trong file.

2.6 Stress Test hệ thống

Chức năng 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 chúng tôi mô phỏng tình trạng tải tài nguyên hệ thống như CPU và bộ nhớ bằng cách tạo ra các tiến trình giả lập tiêu tốn tài nguyên.

Như đã đề cập đến các mục trước đó, chúng tôi tích hợp Stress Test như một luồng xử lý riêng biệt trong Linux App nhằm đảm bảo việc khởi động và dừng Stress Test không ảnh hưởng đến các chức năng logic khác đang chạy. Tại Windows App, ta có thể điều khiển chức năng này như thể hiện tại Hình 12. Khi người dùng từ giao diện Windows App chọn thực hiện Stress Test, Windows App sẽ gửi tín hiệu điều khiển và các tham số đến Linux App để khởi chạy lệnh `stress` với các tham số đó 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, số lượng tiến trình giả lập chính là `numberOfTask` , lượng bộ nhớ mà các tiến trình này chiếm dụng là `memUsage` , `numberOfCore` là số lương lõi CPU mà các tiến trình này sẽ chạy 100% hiệu suất và `seconds` là thời gian chạy Stress Test được tính bằng giây. Các tham số này có thể được cấu hình tùy theo yêu cầu kiểm thử từ Windows App (như Hình 12) giúp chúng tôi dễ dàng điều chỉnh mức độ căng tải hệ thống IVI.

Việc thực thi Stress Test diễn ra hoàn toàn ở phía Linux App (Windows App dùng để điều khiển lệnh cho chức năng này), độc lập với các luồng xử chính, tuy nhiên kết quả của quá trình này lại ảnh hưởng đến toàn bộ hiệu năng của hệ thống, qua đó giúp chúng tôi đánh giá:

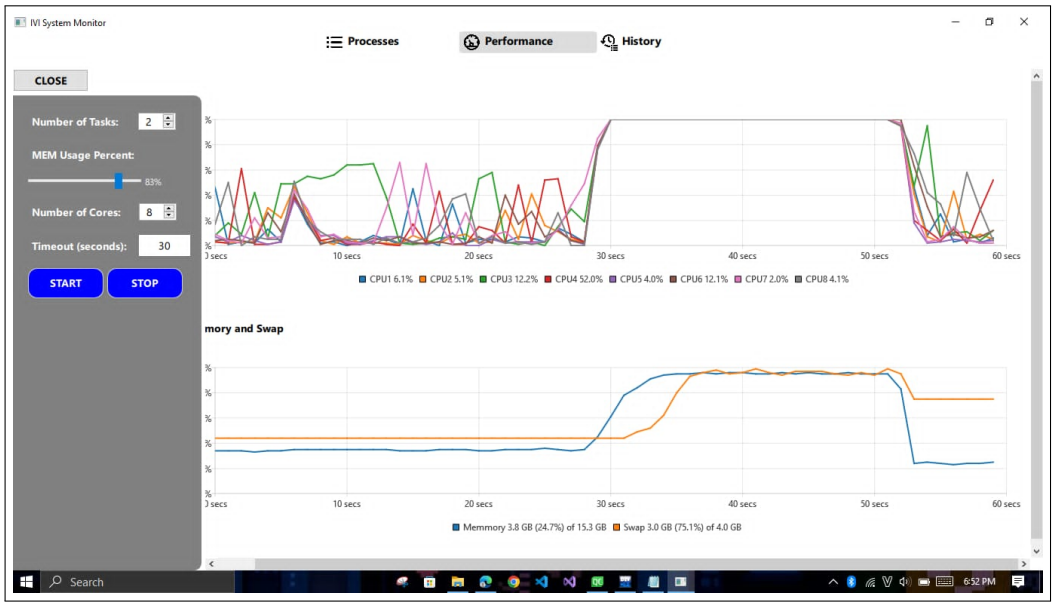
1. Mức độ ổn định và khả năng xử lý logic của phần mềm trong điều kiện tải nặng.
2. Khả năng phản hồi và độ trễ của Linux App khi tài nguyên bị chiếm dụng lớn trên hệ thống IVI.
3. Tính năng cân bằng tải của phần mềm.

Khi cần dừng kiểm thử, Windows App có thể gửi lệnh đến Linux App để thực thi việc kết thúc các tiến trình `stress` . Mặc dù chung stôi triển khai Stress Test như một luồng độc lập, nhưng bản chất việc dừng lại vẫn tương tự như việc kết thúc bất kỳ tiền trình nào khác trong Linux, cụ thể thông qua lệnh sau:

```
$ killall stress
```

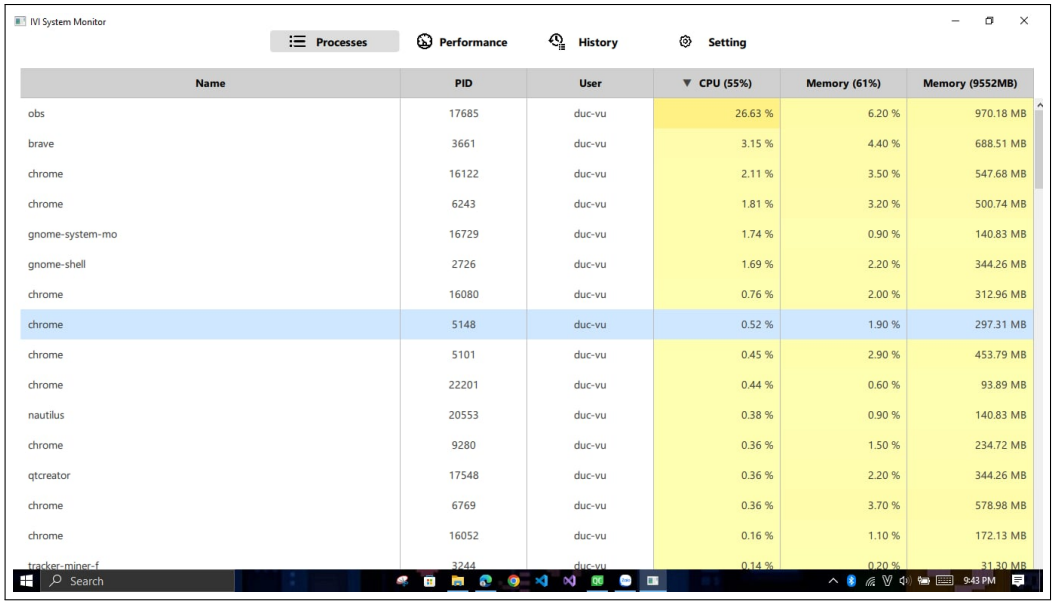
Lệnh trên đảm bảo toàn bộ tiến trình tiêu tốn tài nguyên giả lập được dừng lại ngay lập tức, trả lại trạng thái hoạt động bình thường cho hệ thống. Cơ chế này giúp đảm bảo quá trình kiểm thử không gây ảnh hưởng lâu dài đến hệ thống sau khi hoàn thành kiểm tra.

Phương pháp kiểm thử bằng Stress Test mà chúng tôi áp dụng là một công cụ quan trọng giúp đánh giá toàn diện độ tin cậy của phần mềm nói riêng và cả hệ thống IVI nói chung khi hoạt động trong điều kiện khắc nghiệt. Phần kết quả kiểm thử sẽ được trình bày tại §3.



Hình 12: Thiết lập tham số và điều khiển Stress Test tại Windows App.

3 Kết quả đạt được

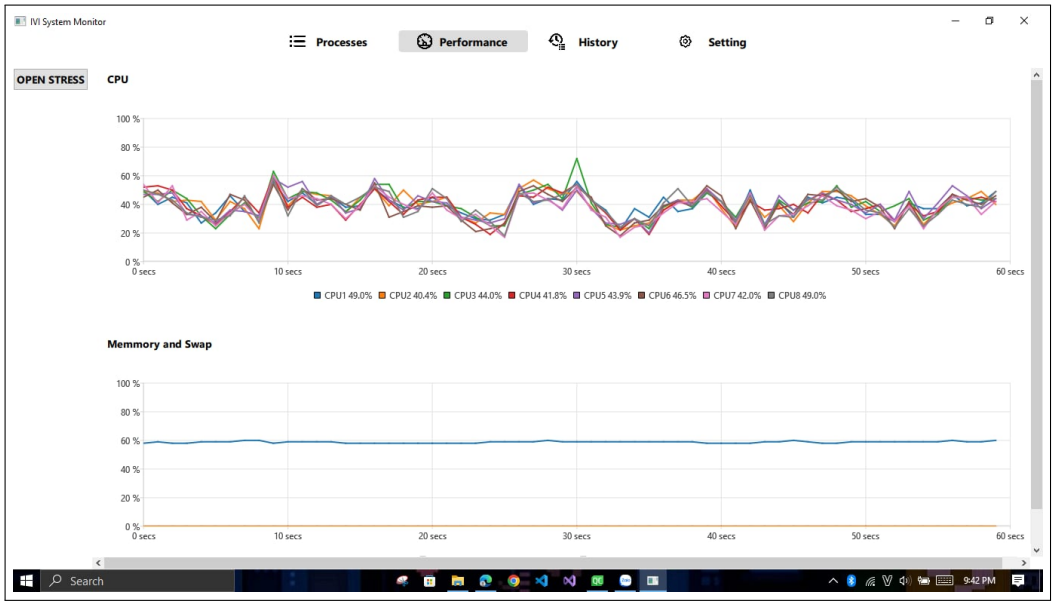


Hình 13: Màn hình Processes trên Windows App.

Chúng tôi sử dụng phần mềm System Monitor trên hệ điều hành Linux (như Hình 15 và Hình 17) để

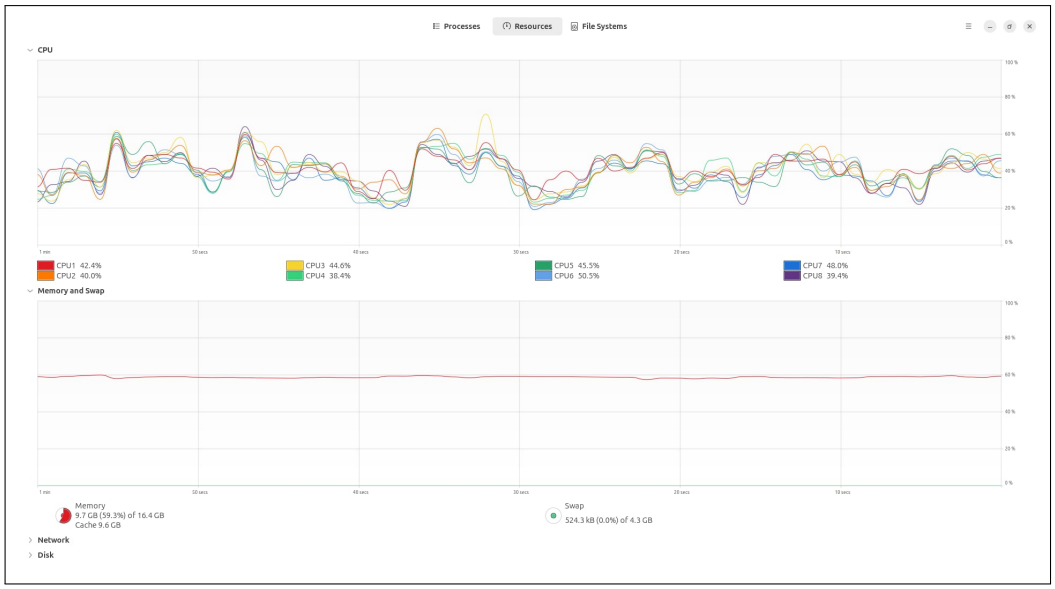
kiểm tra độ chính xác về dữ liệu mà chúng tôi đã thu thập và hiển thị lên giao diện Windows App.

Hình 13 là màn hình giao diện biểu diễn trực quan danh sách các tiến trình cùng với các giá trị thông số mà nó chiếm dụng.



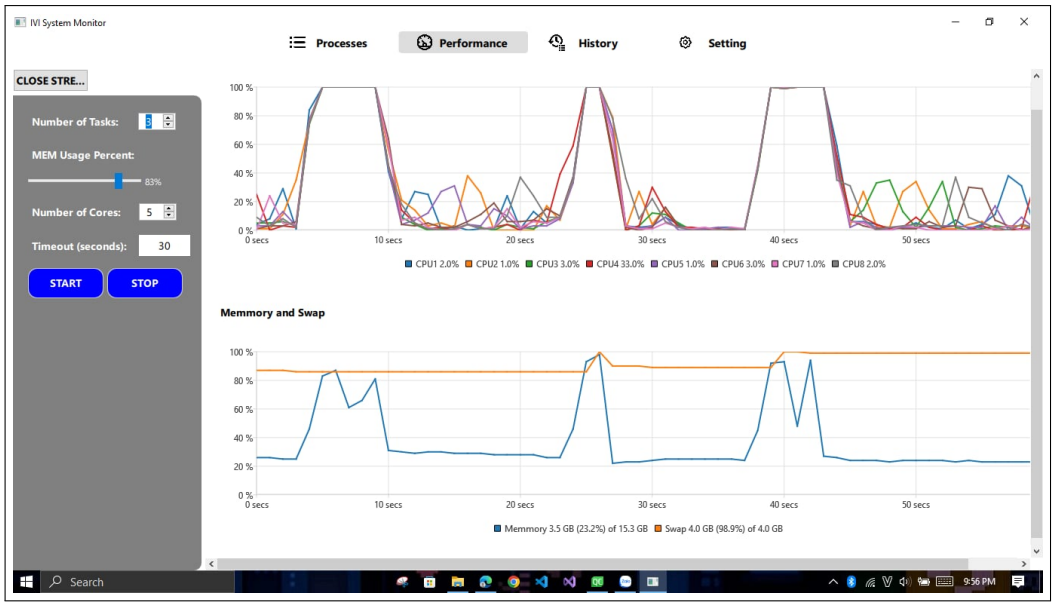
Hình 14: Màn hình Performance trên Windows App.

Hình 14 là màn hình biểu diễn các biểu đồ LineChart liên quan đến CPU của từng core và MEM gồm RAM và SWAP. Với giao diện trực quan giúp người dùng dễ dàng quan sát được sự thay đổi liên tục của dữ liệu theo thời gian. Trong màn hình Performance có một cửa sổ đóng/mở giao diện Stress Test với các thông số Stress Test và các nút bấm "START", "STOP".



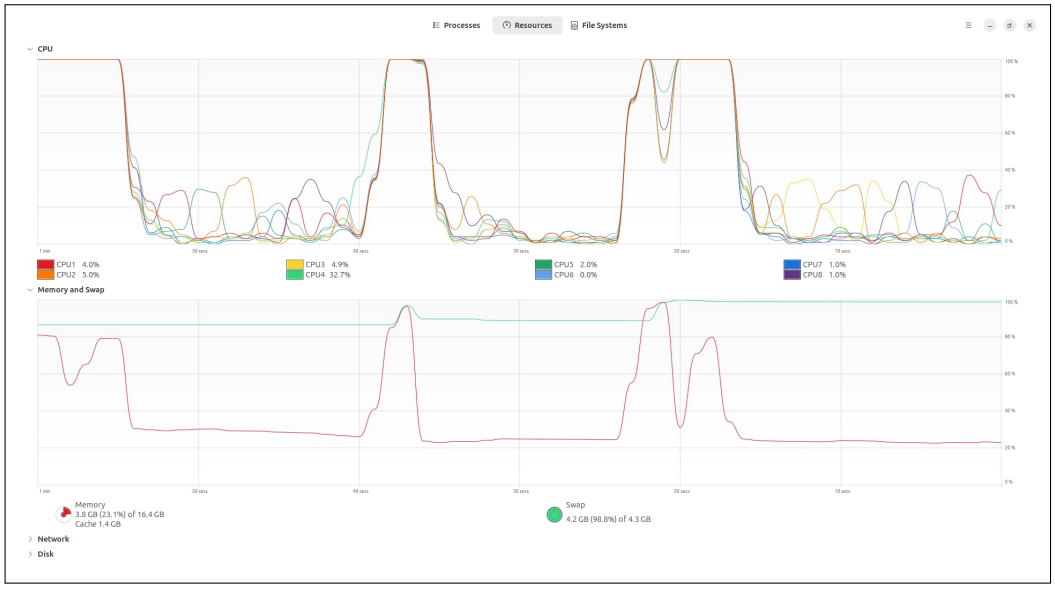
Hình 15: Màn hình Performance của System Monitor trên Linux
(so sánh với Hình 14).

Hình 16 là kết quả khi chạy Stress Test với các thông số như đã thiết lập trong cửa sổ Stress Test. So với phần mềm theo dõi có sẵn trên Linux (như Hình 17) ta thấy được xu hướng thay đổi của các thống



Hình 16: Màn hình Performance khi Stress Test trên Windows App.

số trong quá trình Stress Test là khá tương đồng với nhau. Và trong khi hệ thống hoạt động bình thường cũng tương tự vậy, ta có thể thấy rõ trong Hình 14 và Hình 15.



Hình 17: Màn hình Performance của System Monitor khi Stress Test trên linux (so sánh với Hình 16).

4 Kế hoạch

Bổ sung hiển thị phần trăm CPU tổng, MEM tổng, Current Time tại Performance bar.

Xây dựng thêm chức năng xem lại lịch sử tải đã sử dụng.

Bổ sung cơ chế SSL/TTL vào TCP/IP nhằm tăng bảo mật cho dữ liệu truyền/nhận giữa Linux App và Windows App.

Thuật toán đánh giá quá tải hệ thống chỉ mới là phiên bản thử nghiệm đầu tiên, chưa thể đánh giá được chính xác hệ thống quá tải hay không nên cần phải thử nghiệm và nghiên cứu thêm.

To do (Window App)	Component	From	To	Duration	Status	Assigment
Xây dựng giao diện hiển thị	Xây dựng UI tải chiếm dụng của từng tiến trình	20/3/2025	22/3/2025	2 tuần	Done	Nguyễn Minh Phương
	Xây dựng UI line chart với CPU	23/3/2025	25/4/2025		Done	
	Xây dựng UI line chart với RAM	25/4/2025	26/4/2025		Done	
	Sắp xếp ứng dụng theo mức sử dụng tài nguyên	11/4/2025	12/4/2025		Done	
	Xây dựng UI Stress Test	24/4/2025	24/4/2025		Done	
	Xây dụng UI setting các thông số cần thiết cho ứng dụng	25/4/2025	25/4/2025		Done	
	Xây dựng UI lịch sử tải sử dụng				In Progress	
	Xây dựng UI pop-up cảnh báo				In Progress	
Module xử lý dữ liệu nhận từ app linux	Xây dựng các module theo Class diagram đã tạo trước	27/4/2025	3/4/2025	1 tuần	Done	
	Xây dựng module kết nối với Linux app dùng SocketTCP	4/3/2025	5/4/2025	2 tuần	Done	
	Xây dựng module xử lý data nhận được	6/4/2025	7/4/2025		Done	
	Xây dựng ViewModel SystemStats nhận data rồi truyền cho UI	8/4/2025	9/4/2025		Done	
	Xây dựng ViewModel ProcessesList nhận data rồi truyền cho UI	10/4/2025	11/4/2025		Done	
Xử lý database	Xây dựng module lưu dữ liệu nhận được vào file .csv	12/4/2025	13/4/2025		Done	
	Xây dựng module lưu dữ liệu nhận được trong thời gian quá tải vào file .csv	13/4/2025	14/4/2025		Done	
Truyền dữ liệu	Xây dựng module tạo các command json để gửi cho linux app	15/4/2025	15/4/2025		Done	
	Xây dựng Json truyền từ Window App đế Linux App				Done	
Xử lý Thread	Xây dựng thread cho các chức năng như phát hiện quá tải, xác định ứng dụng cần đóng, lưu data vào file,...	16/4/2025	17/4/2025		Done	
	Module ghi LOG tại window app					
Module thông báo	Hiển thị Pop-up cảnh báo khi quá tải (chưa làm)				Not start yet	
Các chức năng khác	Tạo và gửi các thống số Stress Test từ UI (Điều khiển Stress Test)				Done	
	Tạo record dữ liệu theo ngày, tháng,... ()				Done	

To do (Linux App)	Component	From	To	Duration	Status	Assigment
Thu thập thông số tải hệ thống IVI	Module thu thập thông số CPU: Tổng hệ thống và của từng core	18/3/2025	19/3/2025	2 tuần	Done	Hồ Đức Vũ
	Module thu thập thông số MEM	20/3/2025	21/3/2025			
	Module thu thập thông số của các tiến trình: %CPU, MEM	22/3/2025	24/3/2025			
	Module lọc các tiến trình "phù hợp"	24/3/2025	26/3/2025			
	Tích hợp các module vào hệ thống lớn	27/3/2025	30/3/2025			
Module truyền dữ liệu đến window app	Xây dựng module truyền/nhận dữ liệu tại Linux App			1 tuần	Done	
	Xây dựng Json truyền từ Linux App đến Window App					
	Xây dựng module phân luồng truyền/nhận					
	Xây dựng module lọc gói tin và thực thi lệnh điều khiển tại Linux App					
Xây dựng các hành động xử lý quá tải tại Linux App	Module xử lý terminate các tiến trình khi bị quá tải			1 tuần	Done	
	Module cảnh báo khi chạm ngưỡng tải (phát âm thanh cảnh báo thông qua PC speaker)					
Xây dựng các trường hợp phát hiện quá tải hệ thống đơn giản	Module thiết lập rules cho các trường hợp quá tải toàn bộ hệ			1 tuần	Done	
	Module đánh giá tải toàn bộ hệ thống dựa trên các rules đã thiết lập					
	Module thiết lập rules cho các trường hợp ngưỡng tải của các tiến trình					
	Module đánh giá tải của các tiến trình dựa trên các rules đã thiết lập					
	Module phát hiện chạm ngưỡng quá tải hệ thống					
Stress Test	Xây dựng module tạo tải giả cho kiểm thử phần mềm			1 tuần	Done	
	Xây dựng module phân luồng cho Stress Test					
Kiểm tra và đánh giá hệ thống	Kiểm tra từng module			1 tuần	Done	
	Tích hợp toàn bộ module vào hệ thống lớn					
	Kiểm tra tổng thể hệ thống					
Đóng gói Linux App	Hoàn thành sản phẩm release (bản 1) Linux app				Done	

Công việc chung	Component	From	To	Duration	Status	Assigment
Build Documentation	Phân tích requirement	12/3/2025	13/3/2025	1 tuần	Done	Nguyễn Minh Phương, Hồ Đức Vũ
	Vẽ Usecase	14/3/2025	15/3/2025			
	Vẽ Class Diagram	16/3/2025	17/3/2025			
Xây dựng thuật toán phát hiện quá tải	Module lọc tiến trình hệ thống và tiến trình thường				Done	
	Module xử lý khi quá tải nghiêm trọng (xử lý riêng biệt các trường hợp quá tải)				Done	
	Module các trường hợp quá tải toàn hệ thống				In Progress	
	Build datasheet whilelist và priority				Done	
	Tích hợp các module vào hệ thống lớn				Done	
Phát triển thuật toán xử lý kết thúc các tiến trình trên	Module ngưỡng tải của các tiến trình			3 tuần	In Progress	
	Phát triển thuật toán kết thúc các tiến trình để tối ưu tải cho hệ thống và hạn chế ảnh hưởng đến trải nghiệm người dùng				In Progress	

hệ thống	Testing chức năng phát hiện và xử lý quả tải trong quá trình stress test				Done	
Quản lý và phân quyền tài khoản sử dụng dịch vụ				1,5 tuần	Not start yet	

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] Qt framework documentation: https://doc.qt.io/qt-6/index.html

Phụ lục

[1] Performance-Service App Repository:
https://github.com/HODUCVU/Capstone-Project_Linux-App.git

[2] Performance-Viewer App Repository:
https://github.com/Wb1305/Project_Capstone_WindownApp.git

[3] Understanding memory usage on Linux:
https://virtualthreads.blogspot.com/2006/02/understanding-memory-usage-on-linux.html