ĐẠI HỌC QUỐC GIA HÀ NỘI

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

BÁO CÁO BÀI TẬP LỚN MÔN HỌC

LẬP TRÌNH MẠNG

Chủ đề:

Basic Publish Subscribe Protocol

Giảng viên: Nguyễn Hoài Sơn

Nhóm 1

Mã sinh viên	Họ và tên
18020445	Đào Minh Hải
18020529	Hà Văn Hoài
18020243	Đào Đình Công
18020456	Phạm Xuân Hanh

1 Giới thiệu giao thức

1.1 Giới thiệu nhu cầu

IoT là một lĩnh vực vô cũng nóng ở thời điểm hiện tại do nhu cầu về tính tự động hóa cũng như mang lại sự tiện nghi cho cuộc sống hiện đại của con người. Trong đó Smarthome chính là một trong số những thứ đem lại cuộc sống tiện nghi cho con người. Để xây dựng được các mô hình nhà thông minh như vậy thì việc trao đổi thông tin giữa các thiết bị trong nhà là một vấn đề vô cùng quan trọng, việc cần thiết của một giao thức nhanh và tin cậy dùng để trao đổi thông tin giữa các thiết bị với nhau nên nhóm đã quyết định thiết kế, xây dựng một giao thức dạng publish/subscribe.

1.2 Giới thiệu giao thức

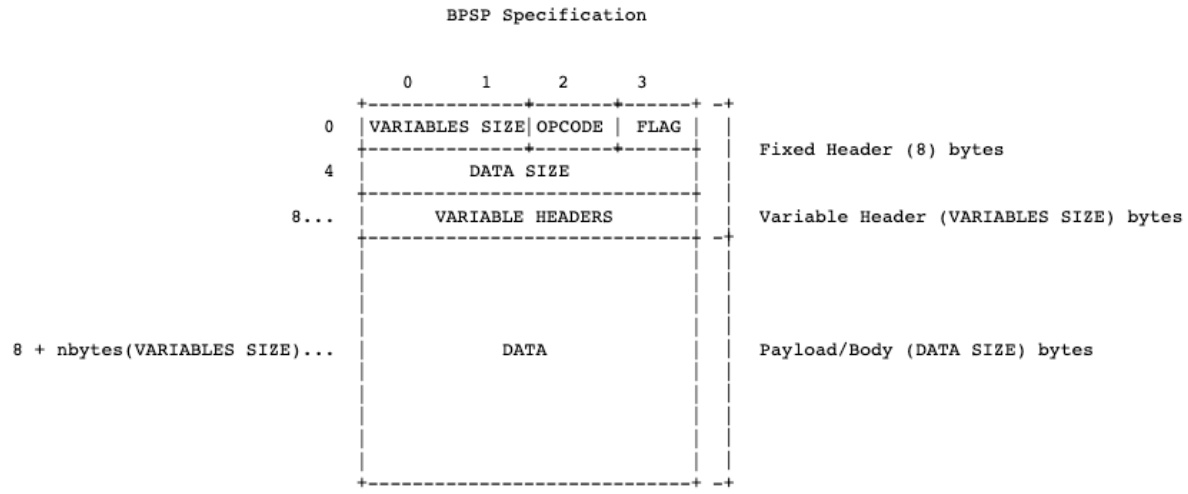
Basic Publish Subscribe Protocol (BPSP) là một giao thức trao đổi thông điệp dạng publish/subscribe giữa các client và server. Hướng thiết kế mong muốn nó trở thành một giao thức nhanh, nhẹ, tin cậy, dễ dàng để thiết kế và áp dụng, mã nguồn mở. Các đặc điểm này phụ hợp sử dụng trong các hệ thống giao tiếp máy với máy (M2M) và Internet of Things (IoT) nơi mà cần tốc độ xử lý nhanh và băng thông mạng không quá lớn.

Là một giao thức của tầng ứng dụng hoạt động trên giao thức TCP/IP vì vậy nó được cung cấp các chức năng cơ bản của TCP/IP bao gồm: đảm bảo đúng thứ tự, tin cậy, mất mát dữ liệu thấp và kết nối 2 chiều. Ngoài ra thì chức năng chính của giao thức là:

- Trao đổi thông điệp dựa trên mẫu thiết kế publish/subscribe, cung cấp khả năng phân tán thông điệp một-nhiều (one-to-many).
- Cung cấp chất lượng phân tán thông điệp “nhiều nhất một” (At most once) tức là thông điệp sẽ được cố gắng gửi đi chỉ duy nhất 1 lần, trong quá trình truyền tin nếu có mất mát sẽ không được gửi lại nữa, điều này cung cấp khả năng truyền tinh nhanh chóng nhưng lại không tin cậy vì sẽ có khả năng mất mát dữ liệu, phù hợp với các ứng dụng thời gian thực cao.
- Chi phí truyền tin nhỏ và tối ưu nhờ vào sự đơn giản của giao thức.
- Khả năng mở rộng dễ dàng.

2 Thiết kế giao thức

2.1 Tổng quan gói tin



2.2 Kiểu dữ liệu

2.2.1 Bits

- 2 giá trị 0 và 1

2.2.2 One Byte Integer

- Kiểu số nguyên không dấu 1 byte, 8 bits (**uint8_t**). Miền giá trị [0, 255].

2.2.3 Two Byte Integer

- Kiểu số nguyên không dấu 2 bytes, 16 bits (**uint16_t**), được biểu diễn dưới dạng Big Endian. Miền giá trị [0, 2^{16} - 1].

2.2.4 Four Byte Integer

- Kiểu số nguyên không dấu 4 bytes, 32 bits (**uint32_t**), được biểu diễn dưới dạng Big Endian. Miền giá trị [0, 2^{32} - 1].

2.2.5 Character

- Kiểu ký tự đơn, 8 bits (**char**). Miền giá trị [0, 255].

2.2.6 String

- Kiểu chuỗi ký tự, mảng các ký tự (**char[]**).

2.3 Chi tiết định dạng gói tin

2.3.1 **Fixed Header**

Mỗi gói tin của **BPSP** luôn chứa phần Header cố định gồm 8 bytes. Bao gồm các thành phần:

- Variables Size: Độ dài phần header linh động (có thể coi là phần header mở rộng).
- Opcode: Operation Code hay còn coi là các lệnh điều khiển (Command).
- FLAG: Giá trị cờ.
- Data Size: Độ dài của phần dữ liệu (payload/data của gói tin).

2.3.1.1 Variables Size

Chỉ định độ dài của phần **Variable Header** (phần header mở rộng). Được biểu diễn bằng kiểu dữ liệu Two Byte Integer, giá trị biểu thị độ dài trên đơn vị byte của phần **Variable Header**.

2.3.1.2 Opcode

Lệnh điều khiển của gói tin (Command). Được biểu diễn bằng kiểu dữ liệu One Byte Integer. Giá trị tương ứng với lệnh điều khiển được biểu diễn ở bảng dưới đây :

Opcode	Giá trị	Gửi từ	Mô tả
INFO	1	Server	Phản hồi của gói tin **CONNECT** nhằm trả về thông tin, thuộc tính của Broker.
CONNECT	2	Client	Gửi tới Broker sau khi thực hiện TCP Handshaking, bao gồm các thông tin nhận dạng, thuộc tính của Client **INFO**.
PUB	3	Client	Publish thông điệp vào một topic.
SUB	4	Client	Subscribe vào một topic.
UNSUB	5	Client	Unsubscribe từ một topic.
MSG	6	Server	Chuyển tiếp thông điệp được gửi từ Publisher tới Subscriber tương ứng.
+OK	7	Server	Phản hồi thành công.
-ERR	8	Server	Phản hồi thất bại, có thể dẫn tới việc kết nối bị ngắt.
….			

Hiện tại với phiên bản đầu tiên của giao thức chỉ gồm 8 lệnh điều khiển trên, vì miền giá trị của Opcode là [0, 255] vì vậy có thể mở rộng tối đa tới 256 lệnh điều khiển. Trong phiên bản này dù không đề cập tới 2 lệnh điều khiển là **PING** và **PONG** nhưng dự kiến sẽ được sử dụng để kiểm tra tốc độ mạng, tính toán RTT và kiểm tra duy trì kết nối.

2.3.1.3 FLAG

Các giá trị cờ, được biểu diễn bằng kiểu giá trị One Byte Integer bao gồm 8 bits cờ, mỗi bit tương ứng với một cờ riêng và các loại cờ ở các Opcode khác nhau sẽ khác nhau.

Flag bit	Opcode	Tên	Mô tả
1	ALL	ACK	Giá trị cờ quy định các gói tin được gửi từ Client tới Broker có cần phản hồi lại hay không.
2	PUB	ECHO	Giá trị cờ dành cho lệnh **PUB** quy định thông điệp được gửi từ một client có được Broker chuyển quay lại với Subscriber tương ứng trên chính Client đó hay không.
	Other	(Chưa dùng tới)	(Chưa dùng tới)
3	ALL	(Chưa dùng tới)	(Chưa dùng tới)
4	ALL	(Chưa dùng tới)	(Chưa dùng tới)
5	ALL	(Chưa dùng tới)	(Chưa dùng tới)
6	ALL	(Chưa dùng tới)	(Chưa dùng tới)
7	ALL	(Chưa dùng tới)	(Chưa dùng tới)
8	ALL	(Chưa dùng tới)	(Chưa dùng tới)

Ở phiên bản hiện tài thì các chức năng của **BPSP** chưa có nhiều vì vậy mô tả FLAG cũng khá ít, mỗi Opcode sẽ có một bộ cờ riêng, đem tới khả năng mở rộng sau này.

2.3.1.4 Data Size

Độ dài dữ liệu được biểu diễn bằng kiểu giá trị Four Byte Integer biểu diễn độ dài của phần dữ liệu trong gói tin trên đơn bị byte, khi đó dung lượng dữ liệu tối đa của gói tin có thể lên tới 4Gb. Ngoài ra do phần Data Size nằm ở phần cuối của **Fixed Header** vì vậy mà trong các phiên bản tiếp theo nếu vấn đề về dung lượng của một gói tin là trở ngại thì việc nâng Data Size lên 8 bytes là vô cùng dễ dàng khi đó gần như dung lượng của một gói tin là vô cùng lớn không có giới hạn 2^{64} bytes (18.446 exabytes).

2.3.2 Variable Header

Trong phần **Header** của gói tin thường chứa 2 phần là **Fixed Header** và **Variable Header**, trong đó **Fixed Header** là phần cố định luôn tồn tại ở mỗi gói tin **BPSP**, còn **Variable Header** như là phần mở rộng để thêm các thông tin, thuộc tính cho gói tin. Mỗi phần tử của **Variable Header** gọi là **var_header** là một chuỗi ký dạng cặp key-value dùng để cung cấp thêm thông tin cho gói tin. Giống với HTTP Headers thì **Variable Header** trong **BPSP** cũng được sử dụng với chức năng tương tự.
Định dạng của **var_header** sẽ có dạng "**key**""**value**"**;** trong đó:

- **Key**: là tên thuộc tính kiểu chuỗi ký tự String
- **Value**: là giá trị thuộc tính kiểu chuỗi ký tự String

Kết thúc của mỗi cặp **var_header** sẽ là dấu chấm phẩy (**;**). Đối với các ký tự đặc biệt ví dụ như (**"**) cần được escapse bằng gạch chéo (\) = **"** . Theo chuẩn **BPSP** thì với các **var_header** có key chứa tiền tố "**x-**" thì thường sẽ là **var_header** không phải do người dùng chỉ định mà là của hệ thống. Ví dụ như để Publish thông điệp vào một topic nào đó thì tên topic sẽ được gắn vào phần **Variable Header** với một **var_header** có key là: "**x-topic**". Một vài ví dụ về **var_header**:

- Authenticate Header: "**x-username**""**admin**";"**x-password**""**12345**";
- Topic: "**x-topic**""**locationA/sensorA**";

2.3.3 Data

Dữ liệu của gói tin là các bytes nằm ở phần cuối của một gói tin **BPSP**, đơn vị dữ liệu nhỏ nhất của gói tin là byte, độ dài của gói tin được xác định dựa vào phần Data Size nằm trong **Fixed Header**. Việc kết hợp với **Variable Header** để xác định định dạng của dữ liệu vô cùng tiện lợi, theo tiêu chuẩn thì phần **Variable Header** sẽ có một **var_header** để cho biết kiểu dữ liệu của phần data ví dụ: "**content-type**""**text**"**;** dùng để định dạng phần data là kiểu text. Phần định dạng này có thể tuân theo các chuẩn MIME Types có cấu trúc type/subtype ví dụ:

- text/plain
- text/html
- application/json
- image/png
- image/jpeg
- image/gif

2.4 Hành vi của giao thức

2.4.1 Các thành phần trong giao thức

2.4.1.1 Broker

Máy chủ xử lý toàn bộ các hành vi của giao thức, giao tiếp với các máy khách để cung cấp dịch vụ tới các máy khách, quản lý các máy khách, lữu trữ thông tin, …

2.4.1.2 Client

Máy khách là các ứng dụng, thiết bị sử dụng dịch vụ, tại đây thực hiện subscribe tới các topic và publish thông điệp tới các topic, gửi dữ liệu cho broker để truyền cho các client khác.

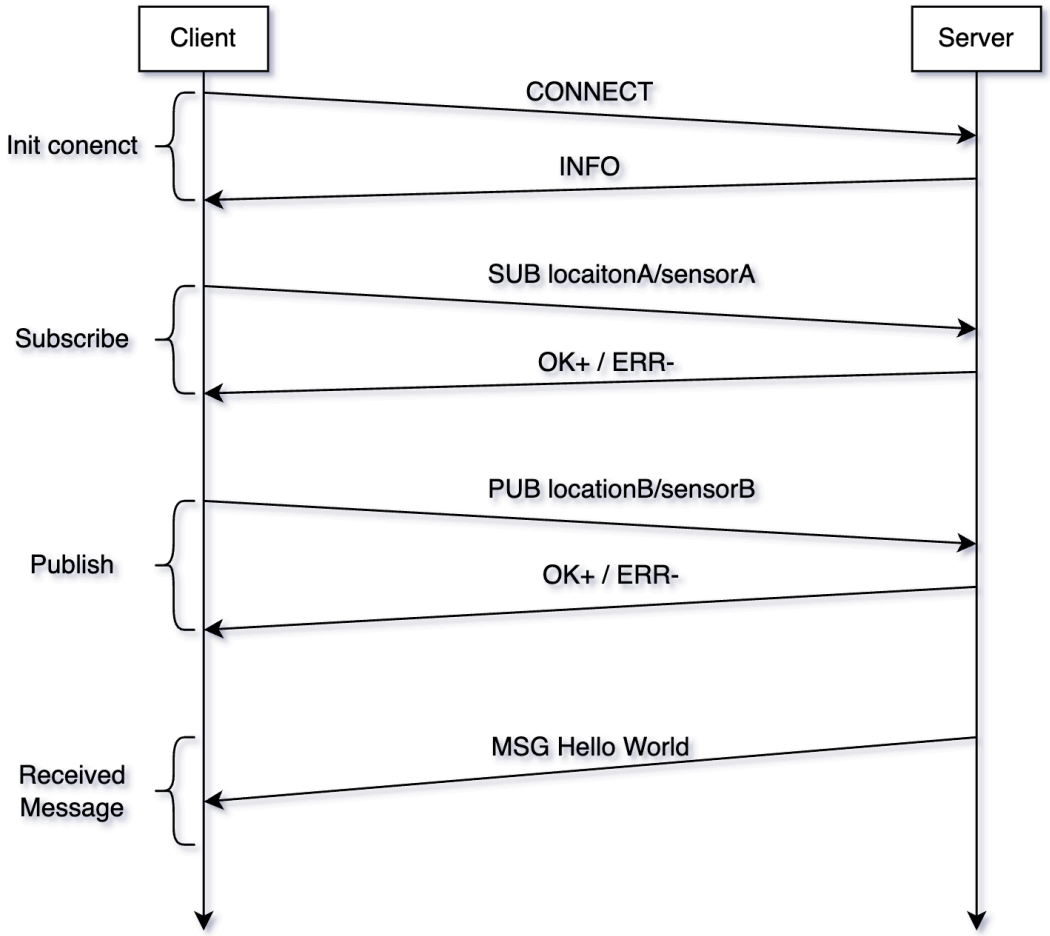
2.4.1.3 Subscriber

Là một đối tượng nhận dữ liệu trên một topic mà nó đăng ký với broker.

2.4.1.4 Publisher

Là một đối tượng thực hiện xuất bản thông điệp vào một topic tới broker để broker thực hiện phát tán tới các Subscriber có topic phù hợp.

2.4.2 Mô tả giao tiếp cơ bản



2.4.2.1 Init connect

Là bước khởi đầu của một phiên làm việc, sau khi thực hiện bắt tay 3 bước TCP xong thì **Client** sẽ thực hiện gửi gói tin **CONNECT** tới **Broker** để cung cấp các thông tin về **Client** như version, định danh, …
Sau khi nhận gói tin **CONNECT** từ **Client**, **Broker** sẽ thực hiện xác thực kết nối, lưu thông tin **Client** và gửi lại gói tin **INFO** cung cấp thông tin về **Broker** như version, định danh, …

2.4.2.2 Subscribe

Để thực hiện đăng ký nhận tin từ một topic, quá trình thực hiện gọi là **Subscribe** sẽ được thực hiện từ phía **Client**. Khi đó **Client** sẽ thực hiện gửi gói tin **SUB** kèm theo topic và các thông tin đi kèm tới **Broker** để đăng ký nhận tin từ topic đó. **Broker** sau khi nhận được gói tin **SUB** sẽ tiến hành xử lý và phản hồi lại kết quả cho **Client** nếu gói tin **SUB** gửi đi từ **Client** có bit ACK được bật.

2.4.2.3 Publish

Để thực hiện xuất bản thông điệp tới một topic, quá trình thực hiện gọi là **Publish** sẽ được thực hiện từ phía **Client**. Khi đó **Client** sẽ thực hiện gửi gói tin **PUB** kèm theo topic, thông điệp và

các thông tin đi kèm tới **Broker** để thực hiện xuất bản thông điệp. **Broker** sau khi nhận được gói tin **PUB** này sẽ tiến hành xử lý chuyển tiếp gói tin tới các **Subscriber** tương ứng với topic được gửi lên, sau khi hoàn tất toàn bộ quá trình chuyển tiếp **Broker** sẽ phản hồi lại kết quả cho **Client** nếu gói tin **SUB** gửi đi từ **Client** có bit ACK được bật.

2.4.2.4 Unsubscribe

Để thực hiện hủy nhận tin từ một topic mà trước đó đã đăng ký, quá trình thực hiện gọi là **Unsubscribe** sẽ được thực hiện từ phía **Client**. Khi đó **Client** sẽ thực hiện gửi gói tin **UNSUB** kèm theo topic và các thông tin đi kèm tới **Broker** để hủy nhận tin từ topic đó. **Broker** sau khi nhận được gói tin **UNSUB** sẽ tiến hành xử lý và phản hồi lại kết quả cho **Client** nếu gói tin **UNSUB** gửi đi từ **Client** có bit ACK được bật.

2.4.2.5 Received Message

Là quá trình **Subscriber** tại **Client** nhận được gói tin **MSG** từ **Broker** là thông điệp được xuất bản từ topic mà nó đăng ký.

2.5 Xử lý phân tán thông điệp, thuật toán

Phần này mô tả cách thức mà **BPSP** thực hiện để thực hiện phân tán các thông điệp được gửi từ **Publisher** tới các **Subscriber**.

2.5.1 Topic

Trong **BPSP** thuật ngữ **Topic** dùng để chỉ tới một chuỗi ký tự mà được sử dụng để **Publisher** thực hiện gửi thông điệp tới các **Subscriber**, **Broker** sử dụng topic để thực hiện tìm kiếm, lọc các thông điệp và phân tán nó tới các **Subscriber** phù hợp.

2.5.1.1 Định dạng Topic

Một chuỗi **Topic** được chia nhỏ thành một hoặc nhiều cấp, mỗi cấp trong một **Topic** được phân cách bởi dấu gạch chéo "/". Mỗi **Topic** phải chứa ít nhất một ký tự và các chủ đề có phân biệt chữ hoa chữ thường. Ví dụ: "**locationA/sensorA**" có 2 cấp là ["**locationA**", "**sensorA**"] và nó sẽ khác so với topic "**LOCATIONa/SENSORa**" do phân biệt chữ hoa chữ thường.

2.5.1.2 Ký tự đại diện (Wildcards)

Khi **Subscriber** subscribe tới một **Topic**, nó có thể subscribe tới một chuỗi ký tự chính xác, hoặc có thể sử dụng ký tự đại diện (wildcards) để subscribe tới nhiều topic. Wildcards chỉ được dùng để subscribe chứ không được dùng để publish. Có 2 loại ký tự đại diện: đơn cấp và đa cấp.

1. Đơn cấp "+" (Single-level)

Ký tự đại diện đơn cấp được dùng để thay thế một cấp trong **Topic**. Ví dụ: "**locationA/+**" sẽ được khớp với tất cả các **Topic** có 2 cấp và cấp đầu chứa "**locationA**", như là

"**locationA/sensorA**", "**locationA/sensorB**", … nhưng không khớp với:
"**locationA/sensorA/uptime**" vì có 3 cấp.

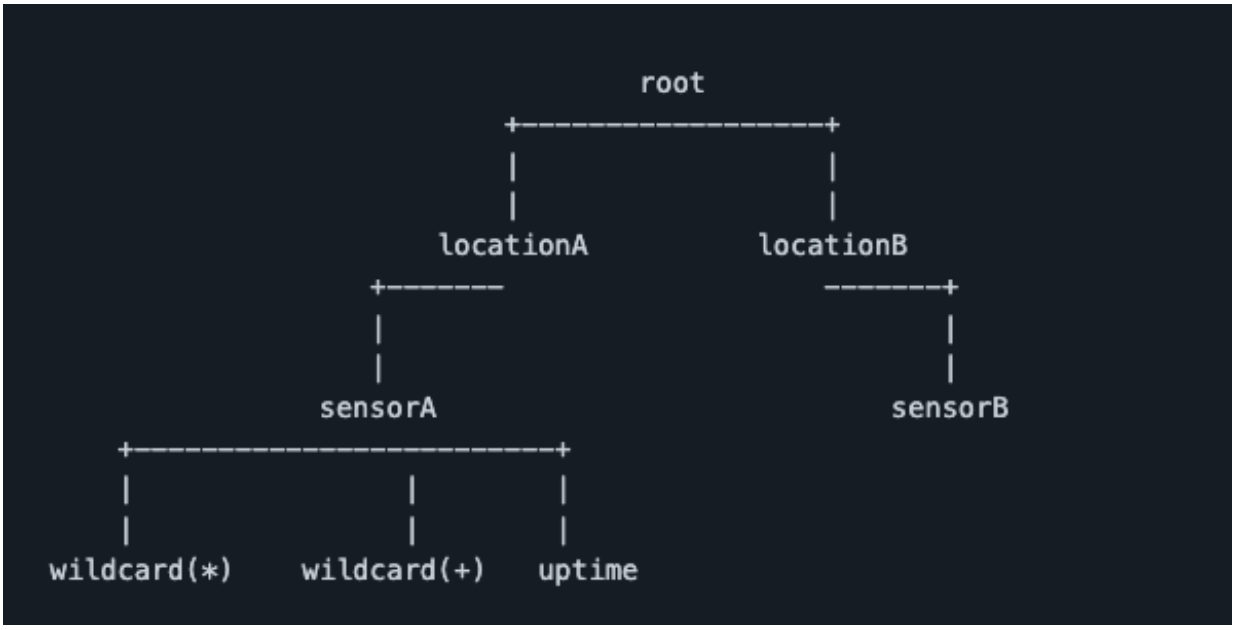
2. Đa cấp "*****" (Multi-level)

Ký tự đại diện đa cấp được dùng để thay thế một hoặc nhiều cấp trong **Topic**. Ví dụ:
"**locationA/***" sẽ khớp với tất cả các **Topic** có ít nhất 2 cấp và cấp đầu chứa "**locationA**", như là: "**locationA/sensorA**", "**locationA/sensor/uptime**", "**locationA/abc/zxy/123**", … nhưng sẽ không khớp với "**locationA**" vì chỉ có 1 cấp.

2.5.2 Cách lưu trữ và xử lý Topic

2.5.2.1 Cấu trúc dữ liệu

Do việc chia **Topic** thành nhiều cấp nên việc sử dụng cấu trúc dữ liệu dạng cây là thuận tiện nhất, khi đó mỗi một cấp sẽ là các nút trong cây, và tất cả các **Topic** tồn tại trong hệ thống sẽ tạo thành một cây với gốc là cấp 0 (nghĩa là root) và các topic sẽ rải đều từ cấp 1 cho tới cuối. Có thể tính toán đơn giản việc tìm kiếm một **Topic** thì sẽ chỉ phải duyệt qua **n** lần trong đó **n** là số cấp của **Topic** đó. Ví dụ về hình ảnh về cây **Topic** ở **Broker** tại 1 thời điểm.



Như đã nói mỗi nút trong cây biểu thị cho 1 cấp của **Topic** vậy mỗi nút này sẽ lưu những gì? Dưới đây là mã giả của mỗi node:

```
/**
 *  Node topic in Tree
 *
 */
struct topic__node {
    bpsp__subscriber[] subs;     // all subscribers match until end current token

    topic__node sl_node;  // single-level node
    topic__node ml_node;  // multi-level node

    /** node hash table **/
    topic__hash_node [string]nodes;
};

struct topic__node root;
```

Có 4 thành phần chính trong node:

- subs: Chứa các subscribe kết thúc tại node này (tức cấp cuối cùng là node này)
- sl_node: Trỏ tới node single-level wildcard
- ml_node: Trỏ tới node Multi-level wildcard
- nodes: Sử dụng cấu trúc dữ liệu HashTable dùng để ánh xạ sang các node tiếp theo đi từ node hiện tại.

Mô phỏng hình dáng của cây sau khi được tạo ra từ các topic:

2.5.2.2 Xử lý Topic

Mục tiêu chính của **Broker** là làm sao để nhanh nhất tìm kiếm được các **Subcriber** tương ứng với published message, vì vậy việc chọn Tree + HashTable làm cấu trúc dữ liệu cho tập **Topic** làm tăng tốc độ tìm kiếm với độ phức tạp thấp hơn rất nhiều so với cách lưu trữ thông thường. Với cách thông thường nếu ta thực hiện lưu các **Topic** vào một mảng thì mỗi khi có một message tới ta sẽ phải lặp qua cả mảng này để lọc các **Subscriber** tương ứng, khi đó độ phức tạp là **O(n)**.Trong khi nếu sử dụng cấu trúc Tree + HashTable như trên thì việc tìm kiếm các **Subscriber** sẽ chỉ đơn giản là sử dụng thuật toán duyệt **DFS** (Depth-First Search) đi từ root và

duyệt qua các cấp trên **Topic** cộng thêm việc duyệt các node của single-level wildcard (nếu có) khi đó số lần duyệt bằng tổng số cấp của topic + số các lần duyệt qua các single-level node (nếu có) và độ phức tạp chỉ đơn giản là **O(1)**.
Ngoài ra khi tổ chức dưới dạng Tree như trên thì việc đẩy subscriber vào cây và xóa subscriber đi cũng sẽ phức tạp hơn so với việc lưu thành mảng như thông thường.

3 Thiết kế chương trình

Mã nguồn của toàn bộ dự án nằm ở repository github sau: https://github.com/barrydevp/bpsp

3.1 Thiết kế Broker

3.1.1 Ngôn ngữ, công cụ

Nhóm sử dụng ngôn ngữ lập trình C để phát triển mã nguồn của **Broker**, do tính chất thô sơ của ngôn ngữ C vì vậy mà các thư viện hỗ trợ xử lý cho các cấu trúc dữ liệu nâng cao như dynamic array, linklist, hashtable, json là không có vì vậy nhóm có sử dụng thêm các thư viện ngoài để xử lý array, hashtable và json.

- uthash / utlist
- pthreads
- cJSON

Công cụ để phát triển mã nguồn sử dụng Text Editor Neovim.
Công cụ Build, kiểm thử CMake, Valgrind.
Môi trường phát triển Linux (Manjaro Arch Distro).

3.1.2 Cấu trúc

Broker sử dụng mô hình Multithreading để xử lý song song yêu cầu từ các **Client**, trong đó một main thread lắng nghe nhận yêu cầu tạo kết nối từ các **Client** sau khi bắt tay 3 bước TCP, main thread này sẽ tạo ra một thread mới cho mỗi client và quản lý các thread này qua một cấu trúc HashTable, mỗi client được tạo sẽ được định danh bởi một dãy ký tự không trùng lặp gọi là _id sẽ được **Broker** sinh ra. Dưới đây mà mã C về cấu trúc của **Broker**:

```
struct bpsp__broker {
    broker__info* info;
    bpsp__connection* listener;
    uint8_t is_close;

    pthread_mutex_t mutex;
    pthread_rwlock_t cli_rw_lock;
    bpsp__client* clients;
    /* UT_array* clients; */

    bpsp__topic_tree* topic_tree;
};
```

Trong đó:

- info: Chứa thông tin của broker
- listener: Socket lắng nghe, nhận kết nối
- is_close: Xác định trạng thái đóng mở của broker
- mutex: Lock mutex dành cho việc kiểm soát truy cập song song các thread
- cli_rw_lock: Để xử lý truy cập đồng thời vào các dữ liệu có tính đọc ghi rõ ràng
- clients: HashTable với key là client_id value là con trỏ tới struct Client
- topic_tree: Cấu trúc của cây Topic như đã giới thiệu ở phần **2.5.2**

Trong cấu trúc này ta sử dụng một loại Lock là Read-Write Lock, khóa này giúp ta cho phép đồng thời nhiều thread cùng đọc nhưng chỉ duy nhất 1 thread được phép ghi tại một thời điểm, điều này giúp tăng tốc độ xử lý bởi vì việc Read chỉ đọc chứ không thay đổi data vậy nên việc sử dụng Mutex lock sẽ làm giảm mức độ concurrency xuống. Áp dụng RW-Lock vào việc truy cập đồng thời trường dữ liệu HashTable clients, mục đích là dùng kết hợp với topic_tree mỗi khi cần phân tán dữ liệu từ publisher gửi lên thì ta sẽ tìm các subscriber thích hợp và cần kiểm tra client của subscriber đó, đây là thao tác read và khi sử dụng RW-Lock thì nhiều client được phép đồng thời đọc nên sẽ không làm chúng bị block nhau, còn khi có một yêu cầu cần loại bỏ một client thì đó sẽ là một thao tác ghi và nó sẽ block chỉ cho phép một thread được ghi tại 1 thời điểm.
Ngoài ra Lock Mutex được sử dụng khi cần đóng **Broker**, và các thao tác xử lý đọc ghi làm thay đổi dữ liệu chung của **Broker**.
Đối với môi client thì sẽ được lưu trữ dưới cấu trúc dưới đây, mã C về cấu trúc của **Client** mà **Broker** sẽ lưu:

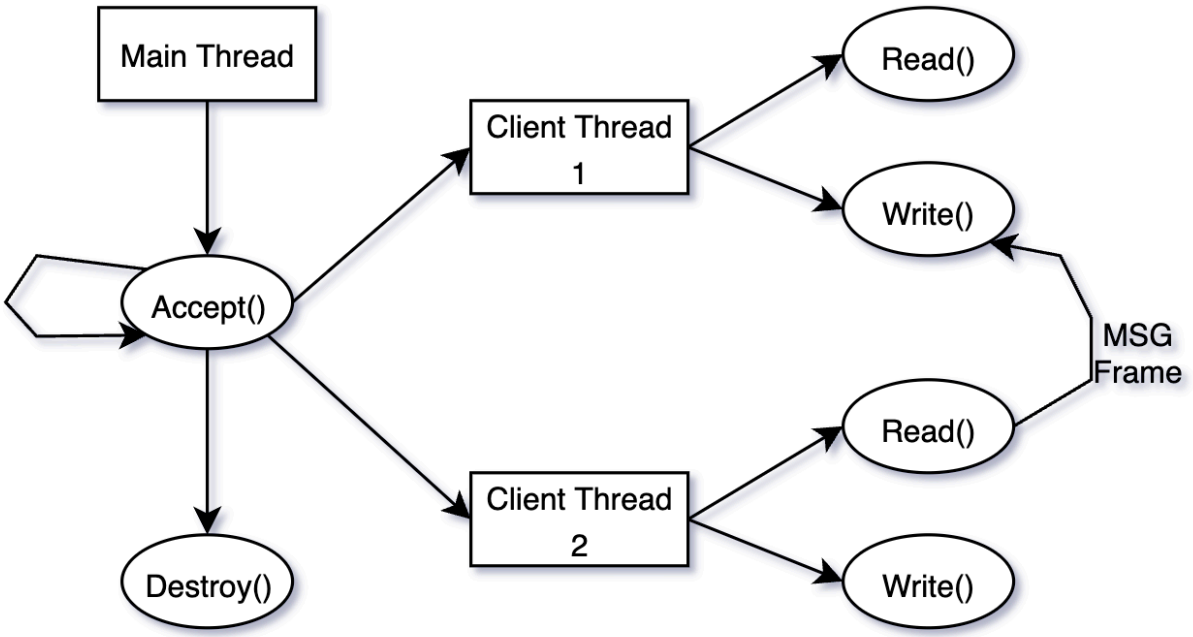
```
struct bpsp__client {
    // core
    char _id[BPSP_CLIENT_ID_LEN + 1];
    bpsp__broker* broker;
    /* UT_array* subs; */
    subscriber__hash* subs;
    bpsp__connection* conn;
    uint8_t is_close;

    // synchonization
    bpsp__uint16 ref_count;
    pthread_cond_t ref_cond;
    pthread_mutex_t mutex;
    pthread_rwlock_t rw_lock;  // multiple thread may write same time but we assume only
                               // so we use this mutex to lock write only

    // frame
    bpsp__frame* in_frame;
    bpsp__frame* out_frame;

    /** uthash.h **/
    UT_hash_handle hh;
};
```

Trong đó:

- _id: Là chuỗi ký tự định danh cho **Client** được sinh ra mỗi khi có một **Client** yêu cầu kết nối tới **Broker**
- broker: Con trỏ trỏ tới đối tượng **Broker**
- subs: Là một HashTable ánh xạ topic và subscriber tương ứng với nó, dùng để quản lý các subscriber trên **Client** hiện tại
- conn: Socket kết nối giữa **Broker** và **Client**
- is_close: Xác định trạng thái đóng mở của **Client**
- các thành phần để đồng bộ hóa đọc ghi nhiều thread
- in_frame: Buffer Frame tới
- out_frame: Buffer Frame ra ngoài

Với mỗi **Client** sẽ quản lý một tập các subscriber của chính nó, mỗi khi có một yêu cầu **SUB** tới sẽ được tạo ra một subscriber và đẩy vào danh sách này cũng như đẩy vào topic_tree và mỗi khi yêu cầu **UNSUB** sẽ thực hiện xóa subscriber trong topic_tree và trong danh sách này, khi một **Client** kết thúc phiên của nó đóng kết nối với **Broker** thì toàn bộ các subscriber của **Client** đó sẽ được loại bỏ khỏi topic_tree.

3.1.3 Luồng xử lý



Như đã đề cập, **Broker** sử dụng multithreading model để xử lý đồng thời các yêu cầu từ **Client**, trong đó mỗi **Client** sẽ tương ứng với một Thread riêng, có một Main Thread thực hiện việc xử lý nhận kết nối từ **Client** và tạo ra các **Client Thread**. Mỗi **Client Thread** sẽ có một read_loop, để nhận các yêu cầu từ **Client** hiện tại thông qua việc read(), vì vậy sẽ chỉ có duy nhất 1 thread được thực hiện read() trên cùng một **Client** sau khi read thì sẽ phụ thuộc vào từng loại Frame và các bit cờ để xác định yêu cầu hiện tại có cần phải phản hồi lại hay không, nếu có thì tại thread này sẽ xảy ra quá trình write(). Ngoài ra quá trình write() còn có thể được thực hiện bởi client thread khác trong trường hợp là gói tin yêu cầu từ client là PUB, khi đó broker sẽ cần xử lý phân tán các gói tin đó tới các subscriber tương ứng và đó cũng được coi là một loại phản hồi nhưng loại này đặc biệt là nó sẽ không thực hiện ghi vào socket của nó mà sẽ ghi vào socket của client khác, do đó write() sẽ là xử lý đồng thời song song vì vậy cần có các cơ chế đồng bộ, tại mỗi client sẽ có một rw_lock riêng giúp xử lý việc đồng bộ này, đảm bảo tại một thời điểm chỉ có duy nhất một thread được phép ghi lên socket đó.

3.1.4 Build và kiểm thử

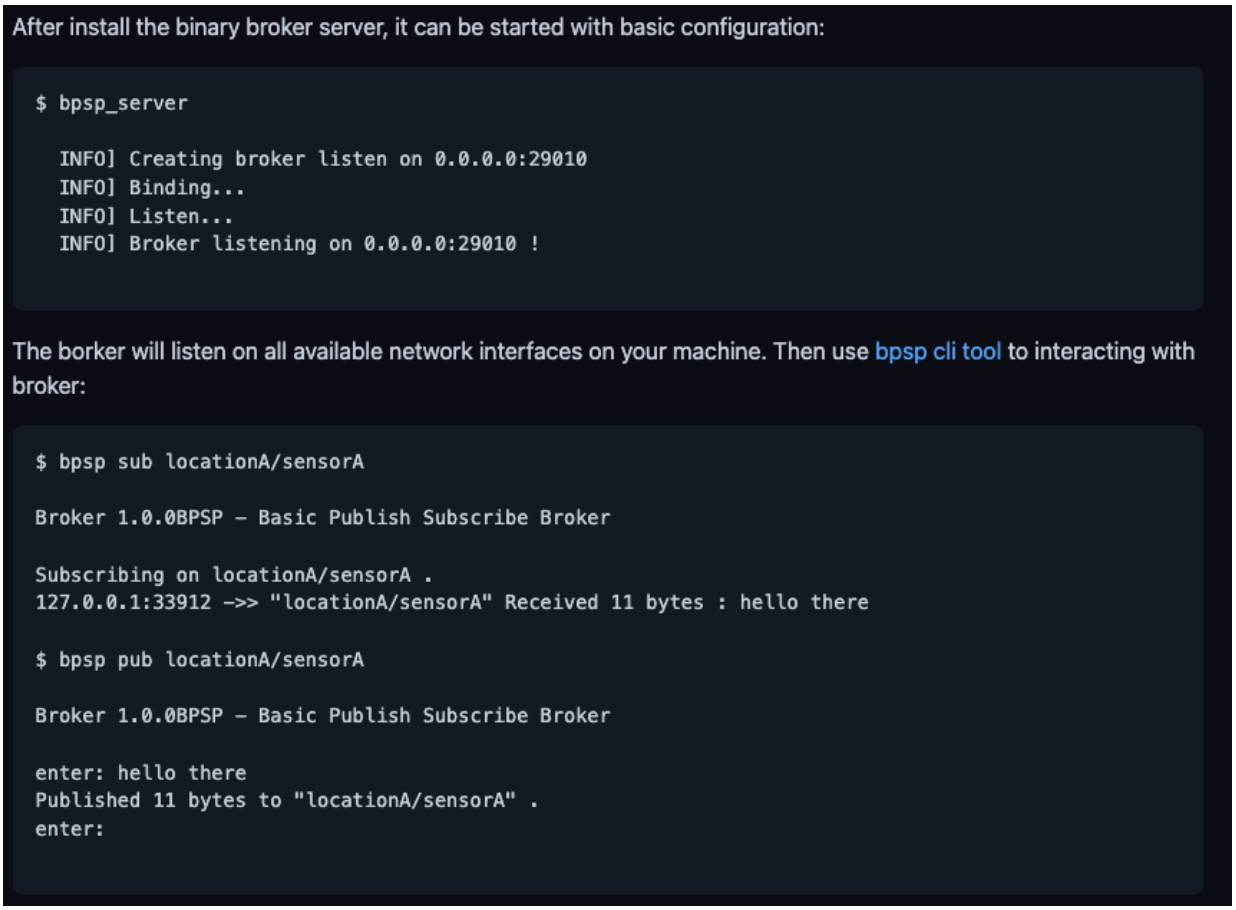
1. Tải mã nguồn của BPSP sử dụng git

   $ git clone https://github.com/barrydevp/bpsp .

2. Sử dụng CMake để build

```
$ mkdir build && cd build          # tạo folder chứa chương trình
$ cmake ..                         # chạy cmake gen Makefile
$ cmake --build . [--config Release]   # chạy cmake build từ Makefile
```

3. Sau khi build xong, chương trình sẽ có tên **bpsp_server** nằm ở thư mục build
4. Kiểm thử chạy
   $ ctest
5. Ngoài ra team cũng thực hiện viết bộ tool dùng để thao tác với BPSP Broker, bộ tool này cũng được build cùng với việc build broker server. Chương trình tool sau khi build nằm ở thự mục build/tools có tên là **bpsp**. Khi đó ta có thể khởi chạy broker và thao tác với broker sử dụng cli tool trên.



3.2 Thiết kế Client

3.2.1 Ngôn ngữ, công cụ

Vì Java là ngôn ngữ lập trình hướng đối tượng, có phần “high level" hơn so với C, nhờ tính chất của mình Java vốn rất mạnh mẽ trong việc lập trình ứng dụng có tính chặt chẽ và khả năng mở rộng cao. Java cũng có nhiều thư viện, framework hoạt động tốt và thân thiện đặc biệt là

các thư viện hỗ trợ xây dựng UI. Hơn nữa, để tách biệt cũng như phong phú hơn về mặt sử dụng ngôn ngữ lập trình trong bài tập lớn này, nhóm quyết định sử dụng Java để xây dựng Client cho BPSP.

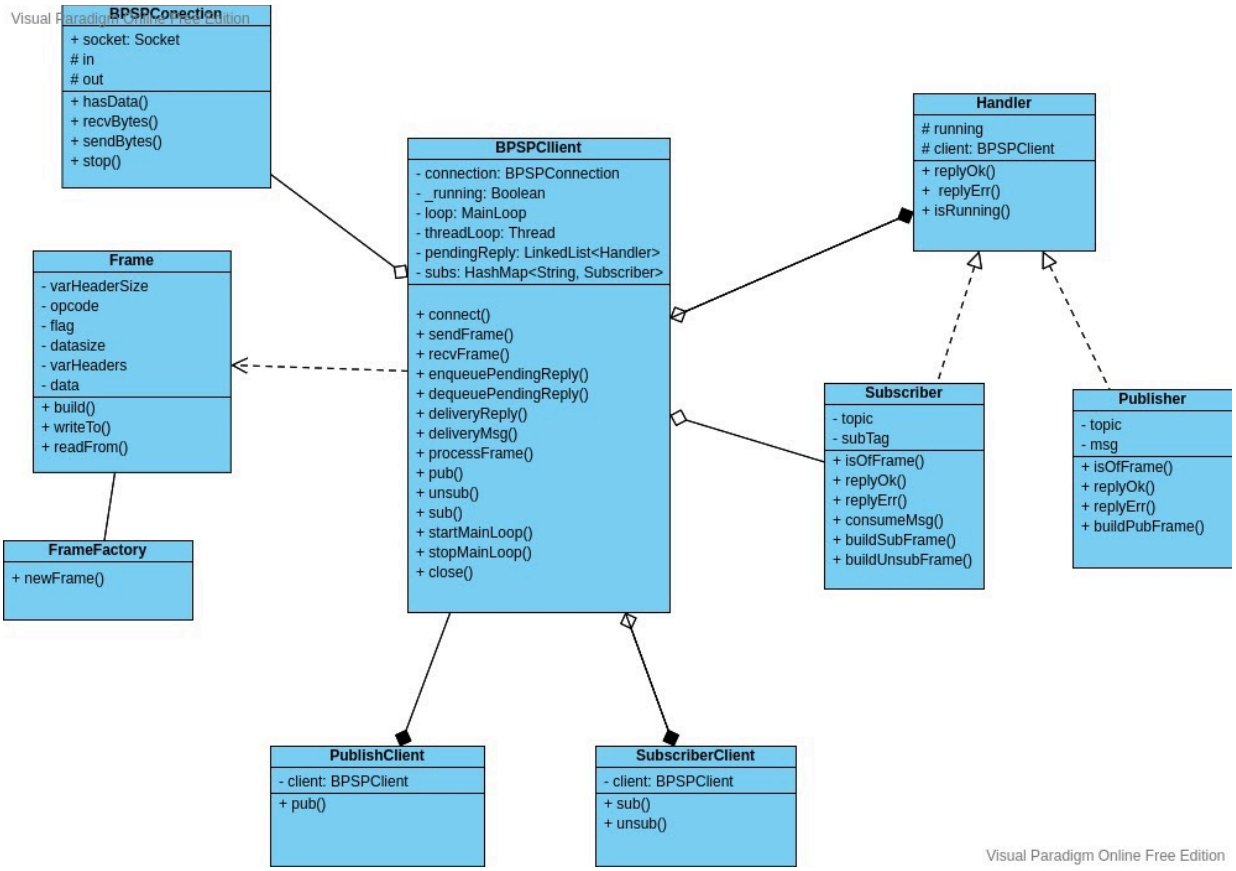
- Môi trường phát triển: Linux
- Ngôn ngữ: Java
- Công cụ build, quản lý thư viện: Maven
- Một số thư viện sử dụng:
  - Java Swing (bộ công cụ GUI mà Sun Microsystems phát triển để xây dựng các ứng dụng tối ưu dùng cho window)
  - Apache Log4j (Logger)

3.2.2 Thiết kế chương trình

3.2.2.1 Các lớp cốt lõi

- Frame: Đây là lớp hiện thực hóa cho gói tin BPSP, trong đó bao gồm các thuộc tính như:
  - Kích thước header, opcode, flag, datasize (phần fixed header) .
  - Các variable header: là một HashMap của VariableHeader (lớp tương ứng cho variable header với thuộc tính key và value) để có thể dễ dàng lưu và truy xuất.
  - Dữ liệu (payload) của frame, dưới dạng một byte array.
- FrameFactory: Lớp này để quản lý và trả về các đối tượng Frame theo yêu cầu, giúp cho việc khởi tạo Frame một cách linh hoạt hơn.
- BPSPConnection: Đây là lớp mà đối tượng của nó có vai trò trong việc trực tiếp tương tác với socket, chẳng hạn như khởi tạo kết nối, đưa dữ liệu lên socket, nhận dữ liệu từ socket, đóng kết nối socket.
- BPSPClient: Lớp này nằm bên trên lớp BPSPConnection, là lớp xử lý chính của client. BPSPClient thông qua một BPSPConnection để tương tác với socket, từ đó nó có thể gửi/nhận các Frame đến/từ server → thực hiện các lệnh (Operation) của giao thức mà nhóm đã thiết kế như tiến hành kết nối đến server, public, subscribe, unsubscribe,.... Để client có thể hoạt động được chính xác và hiệu quả trong việc vừa có thể thực hiện các lệnh vừa đọc dữ liệu cảm biến server gửi tới thì ở đây, ngoài luồng xử lý chính BPSPClient sẽ tạo ra một luồng nữa, chuyên biệt để đọc các Frame nhận được. Chi tiết về cách xây dựng và xử lý các luồng này sẽ được trình bày ở phần sau.
- PublisherClient, SubscriberClient: Các lớp cụ thể hơn của BPSPClient tương ứng cho hai loại client là Publisher Client và Subscriber Client.
- Subscriber, Publisher: Đây là các lớp được sử dụng như một trình đại diện, và hỗ trợ xử lý (Handler) cho các hành động subscribe, unsubscribe, publish. Subscriber/Publisher là tham số truyền vào cho các phương thức tương ứng của BPSPClient.

3.2.2.2 Sơ đồ lớp



3.2.3 Luồng xử lý

Client cũng sử dụng mô hình Multithreading để xử lý song song việc gửi request và nhận response. Đầu tiên, một **BPSPClient** được tạo ra với địa chỉ (IP address + port) của **Server** và thực hiện kết nối bắt tay ba bước TCP với **Server**. Từ đối tượng **BPSPClient**, ta có thể tạo ra các đối tượng SubscriberClient cũng như PublisherClient và sử dụng các method (sub,unsub,pub) mà SubscriberClient/PublisherClient cung cấp.

Ngay sau khi **BPSPClient** được khởi tạo, kết nối và được gọi hàm khởi chạy, **BPSPClient** sẽ tạo ra một luồng mới (gọi là MainLoop). Hoạt động của MainLoop đơn giản chỉ là qúa trình thực hiện vòng lặp vô hạn (đương nhiên sẽ dừng cho tới khi **BPSPClient** bị đóng) để chờ nhận các Frame mà server gửi đến. Nếu nhận được Frame, MainLoop sẽ chuyển Frame này đến cho **BPSPClient** (hoạt động ở luồng chính) để xử lý.

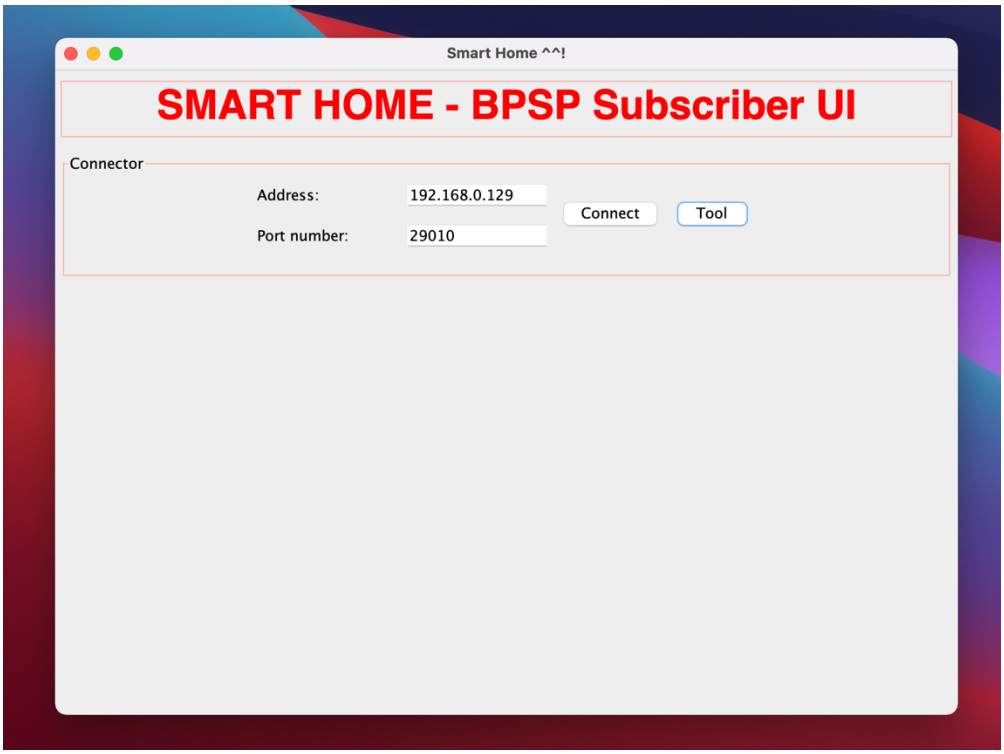
Việc thực hiện lệnh của giao thức BPSP (connect, sub, unsub, pub) được cài đặt với cơ chế callback, diễn ra theo trình tự như sau:

- **BPSPClient** Tạo ra đối đượng Subscriber/Publisher (Handler) với đầu vào dữ liệu cho lệnh (Ví dụ như pub thì có topic và nội dung cần pub). Trong đối tượng Handler sẽ chứa dữ liệu tương ứng cùng với các hàm callback được chạy khi có response/gói tin **Server** gửi đến. Hiện tại có 3 loại callback chính, trong 3 trường hợp:
  - replyOk: response xác nhận lệnh thực hiện thành công (OK)
  - replyError: response xác nhận thực hiện lệnh lỗi (ERR)
  - consumeMsg: server gửi gói tin MSG

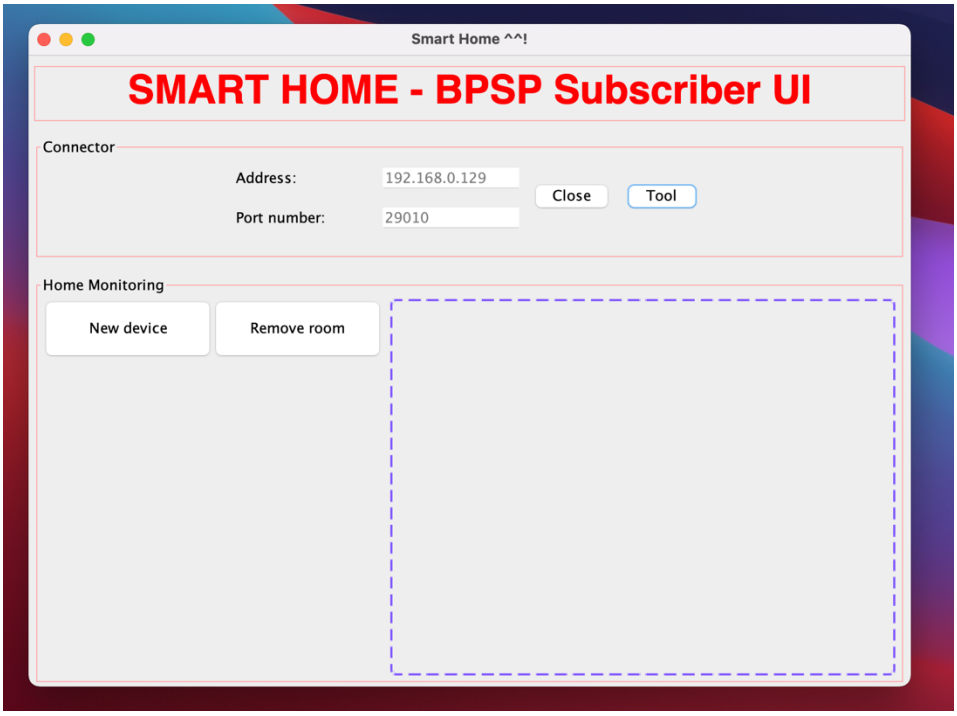
- **BPSPClient** Lấy Frame được build từ Handler và gửi cho **Server**. Sau đó thêm Handler này vào pendingReply (một hàng đợi các Handler). Trong trường hợp lệnh thực hiện là sub thì Subsciber sẽ được thêm vào subs (danh sách các Subscriber)
- Trong hoạt động của MainLoop, nếu có Frame mà **Server** gửi đến, MainLoop sẽ bắt được Frame này ngay và gửi đến cho **BPSPClient**. Tại đây **BPSPClient** cần xử lý Frame nhận được theo 3 trường hợp:
    - Nếu Frame thuộc loại Frame xác nhận (OK/ERR: lấy ra (dequeue) một Handler từ hàng đợi pendingReply và thực thi callback tương ứng (replyOk() / replyError()).
    - Nếu Frame thuộc lại Frame Message: đọc frame để lấy ra định danh của Subscriber, lấy ra Subscriber trong subs bằng định danh của nó sau đó thực thi callback consumeMsg().
    - Nếu Frame không thuộc loại nào mà BPSP quy định thì cảnh báo lỗi.

Lý do cho việc áp dụng cơ chế callback và tách việc xử lý Frame thành các trường hợp như trên là bởi vì ta không thể đảm bảo được rằng **Client** sau khi gửi một Frame thì Frame nhận được ngay sau đó luôn là Frame phản hồi của nó, mặc dù hoạt động của phía **Server** luôn luôn đảm bảo rằng thứ tự của các Frame gửi đi luôn luôn cùng thứ tự với các Frame nhận được, một **Client** vẫn có thể nhận về một số Frame "bất ngờ". Ví dụ như trong trường hợp **Client** đã subscribe một topic nào đó, sau đó nó gửi tiếp một lệnh cho **Server**, lúc này **Client** mong muốn nhận được Frame xác nhận từ **Server** tuy nhiên nó có thể nhận về một Frame MSG của topic đã subscribe nếu một **Publisher Client** nào đó khác gửi Frame PUB lên server trước khi Frame của **Client** kịp đến **Server.**

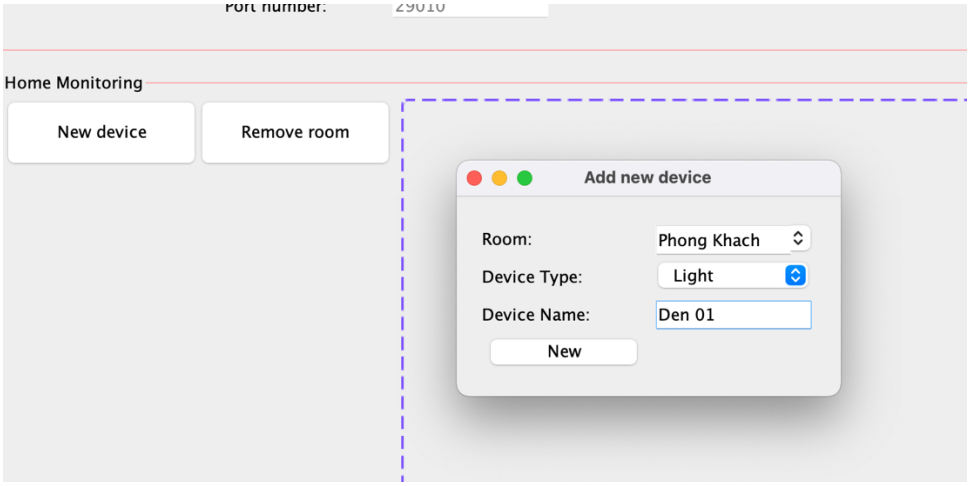
3.2.4 Trực quan hóa bằng giao diện

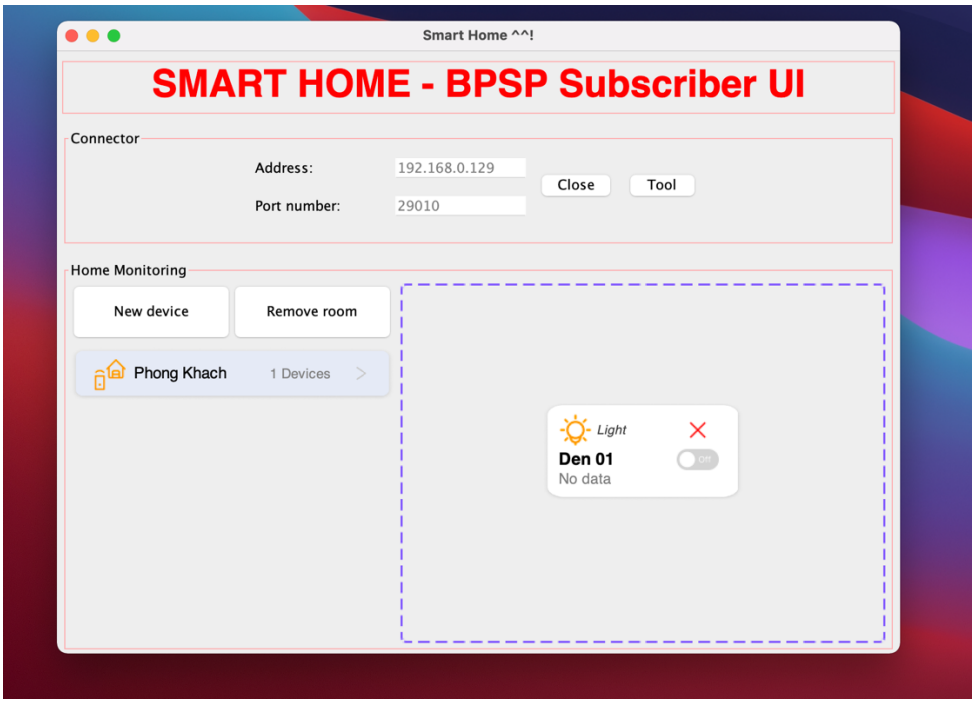


Kết nối tới **Broker**

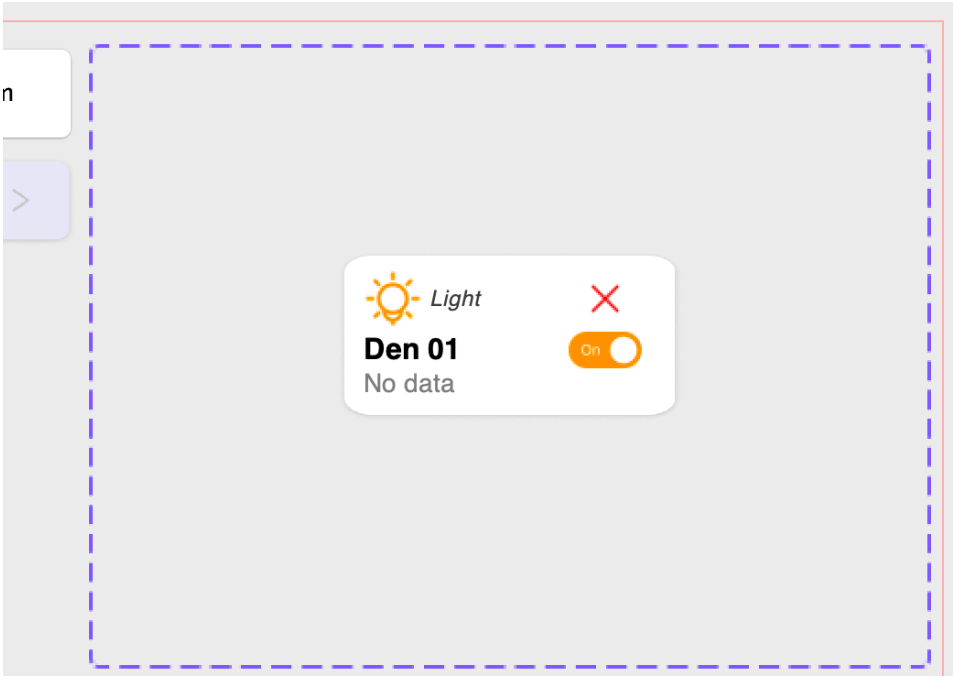


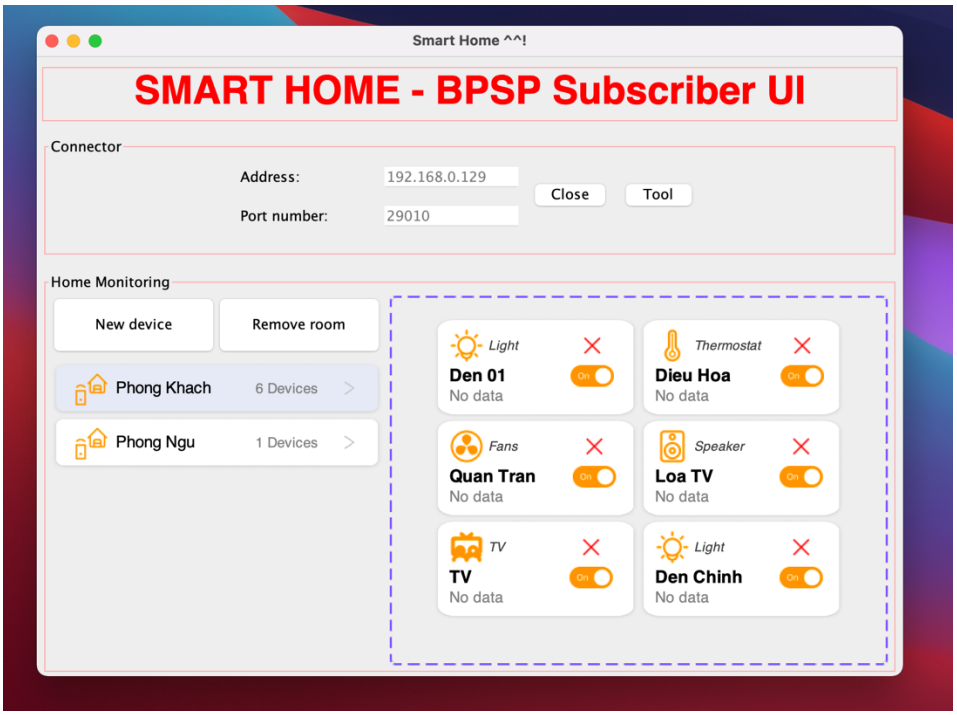
Thêm mới thiết bị



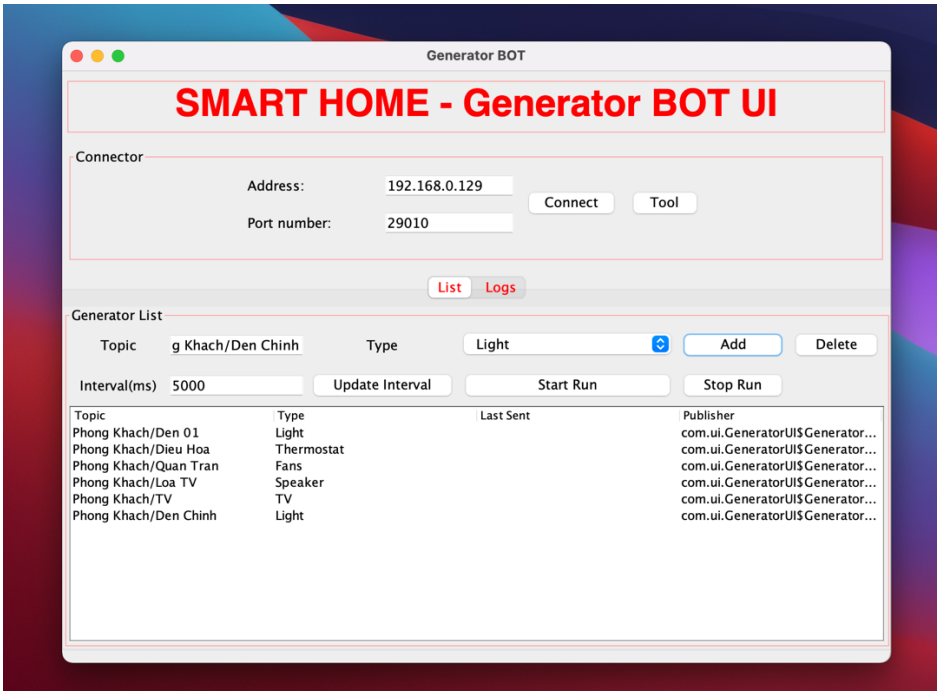


Bật theo dõi thiết bị

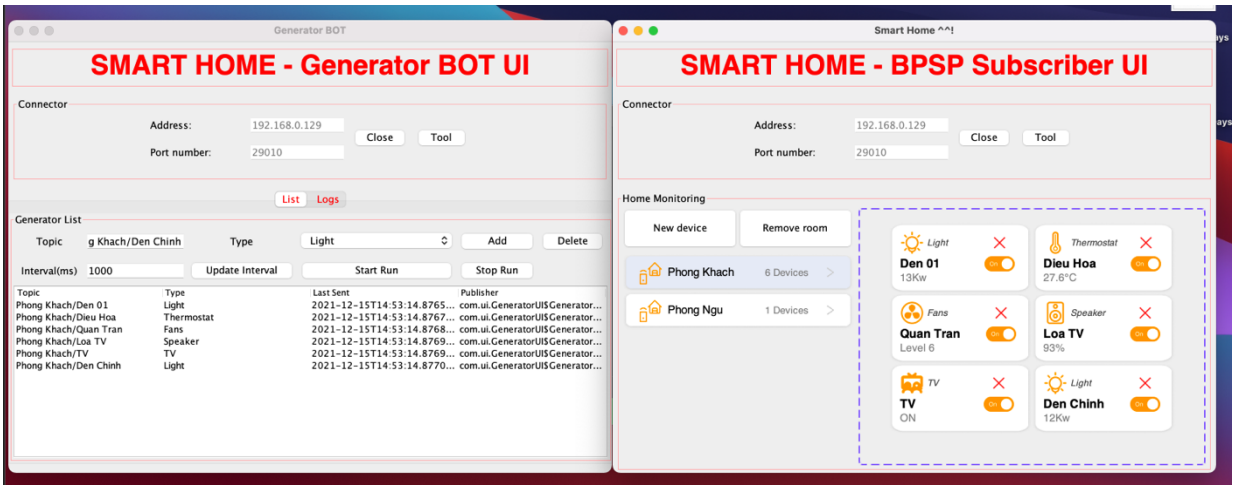


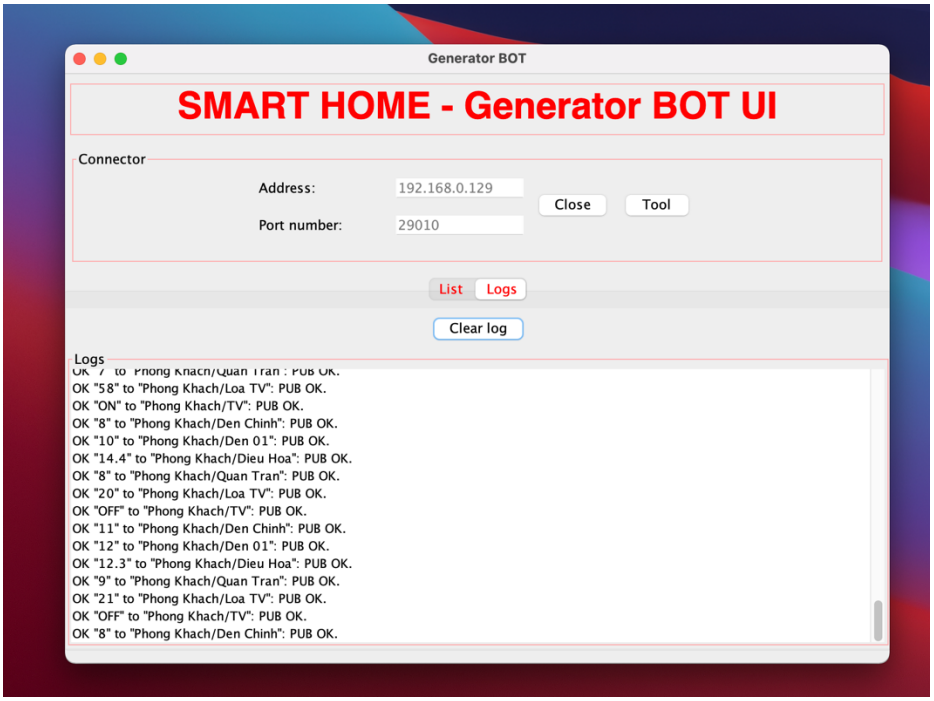


Khởi động phần mềm giả lập sinh dữ liệu cho các thiết bị



Tiến hành sinh gửi dữ liệu





3.2.5 Build và chạy chương trình

1. Cài đặt Java 11 nếu chưa có, có thể cài OpenJDK 11 trên linux với apt như sau:

   $ sudo apt update
   $ sudo apt install openjdk-11-jdk

2. Cài đặt Apache Maven,có thể cài OpenJDK 11 trên linux với apt như sau:

   $ sudo apt update
   $ sudo apt install maven

3. Tải mã nguồn của BPSP sử dụng git

   $ git clone https://github.com/barrydevp/bpsp .

4. Sau đó cd tới thư mục *client/java-client*, chạy lệnh để build:

   $ cd client/java-client
   $ mvn compile

5. Sau khi build xong, chạy lệnh sau để khởi động ứng dụng:

   $ mvn exec:java -Dexec.mainClass=com.AppUI

4 Kết thúc

Mặc dù trong thiết kế còn nhiều những thiếu sót và chưa chính xác nhưng dưới phạm vi và thời gian cho phép của môn học chúng em đã hoàn thành thiết kế và xây dựng chương trình.
Định hướng phát triển giao thức trong tương lai:

- Thay đổi lại thiết kế mô hình sử dụng Nonblocking IO, multiplexing và thread pool cho broker nhằm hạn chế khả năng giới hạn số lượng client, bởi với mô hình hiện tại mỗi connection từ client sẽ cần khởi tạo một thread điều này sẽ làm cạn kiệt bộ nhớ hơn do lượng tài nguyên cấp cho 1 thread là khá lớn. Vì vậy khi chuyển qua thread pool và multiplexing sẽ không còn phụ thuộc vào tài nguyên nhiều nữa.
- Hỗ trợ thêm các client driver cho các ngôn ngữ phổ biến khác như: Javascript, Golang, C++ (tùy chỉnh từ C), Rust, …
- Hỗ trợ thêm các tính năng về Quality Of Service (QoS), message model thay vì chỉ support At Most Once thì sẽ support thêm At Least One.
- Thêm tùy chọn hỗ trợ Cluster Mode cho Broker, để đảm bảo High availability(HA) giúp tăng tính sẵn sàng của dịch vụ cũng như phân tán gánh nặng cho 1 server bằng cách chạy nhiều server tạo thành 1 cụm.

Phân công công việc

Họ và tên	Công việc
Đào Minh Hải	Thiết kế giao thức + Broker + Base Subscriber UI java
Hà Văn Hoài	Core Client Java(xử lý gửi nhận Frame) + Subscriber
Đào Đình Công	Auto gen data publisher C client + Publisher java
Phạm Xuân Hanh	Publisher UI java + docs