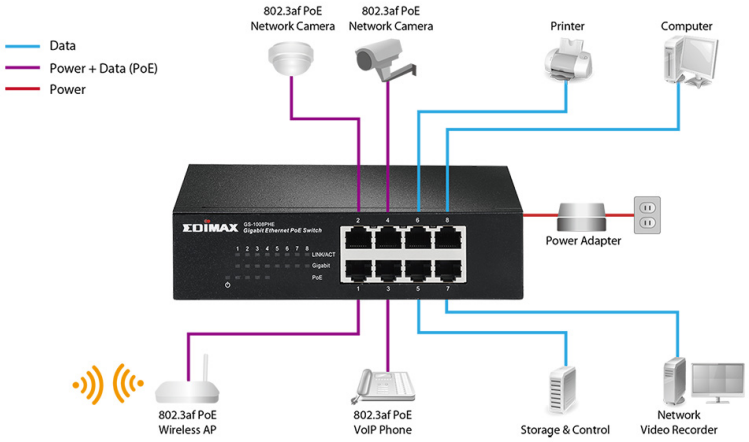
گزارش کار

Ethernet Switch



شبکه‌های کامپیوتری

غزل کلهر (۸۱۰۱۹۶۶۷۵)

محمدامین باقرشاهی (۸۱۰۱۹۷۴۶۴)

فهرست مطالب

مقدمه

در این پروژه به طراحی و پیاده‌سازی یک شبکه به کمک اترنت سوئیچ‌ها پرداختیم. به این منظور از پایپ‌های نام‌دار برای تعریف موجودیت‌های سیستم در قالب پراسس‌های جداگانه استفاده کردیم. در برنامه خود تعدادی سیستم تعریف کردیم و با ایجاد توپولوژی‌های مختلف به رد و بدل پیام بین سیستم‌ها پرداختیم. به منظوری پیاده‌سازی آن از مفاهیمی که در درس فراگرفته بودیم استفاده کردیم و به پیاده‌سازی الگوریتم‌های مربوط به این شبکه مانند spanning tree protocol پرداختیم.

ساختار پروژه

این پروژه از دو پوشه‌ی src و include تشکیل شده است. در پوشه‌ی include هدر فایل‌های مربوط به هر کلاس قرار گرفته است که در آن‌ها تعریف متدها و فیلدها و نیز کتابخانه‌ها قرار گرفته است. در پوشه‌ی src نیز فایل‌های cpp مربوط به کلاس‌ها آمده است که در آن‌ها بدنه‌ی تمامی متدها قرار گرفته است.

همچنین در پوشه‌ی اصلی پروژه یک MakeFile قرار دارد که از آن برای کامپایل و ساخت فایل‌های اجرایی پروژه بهره می‌گیریم.

نحوه اجرا

برای اجرای پروژه کافی است که با دستور cd ethernet-switch به پوشه اصلی پروژه وارد شویم. سپس با دستور make فایل‌ها را کامپایل می‌کنیم. در این مرحله یک از فایل‌هایی که ایجاد می‌شود Network.out است که با اجرای آن در خط فرمان اجرای برنامه آغاز می‌شود و می‌توانیم دستورات تعریف شده در گام دوم را اجرا کنیم.

گام اول

در این گام موجودیت های System، Switch و EthernetFrame و Network را پیاده سازی می‌کنیم. کلاس های مربوط به هریک از این 4 موجودیت در ادامه توضیح داده شده اند.

کلاس Network

این کلاس اصلی برنامه است که وظیفه دریافت دستو ها از command line و انجام کار های لازم برای انجام آن دستور ها را به عهده دارد. (البته برای مثال در دستور send این کلاس تنها شروع کننده ارسال پیام است و ارسال پیام از طریق node های موجود در شبکه انجام می‌شود.)

فیلد ها

map<int, Pid> switches: این فیلد یک مپ از شناسه سوییچ ها به pid پردازه مربوط به آن سوییچ است.

map<int, Pid> systems: این فیلد یک مپ از شناسه سیستم ها به pid پردازه مربوط به آن سیستم است.

متد ها

- void handle_command(string command);

این متد که پس دریافت دستور ورودی صدا زده می شود پس از parse کردن آن handler متناظر با آن دستور را صدا میزند:

```
void Network::handle_command(string command) {
    vector<string> command_parts = split(command, SPACE);

    if (command_parts[COMMAND] == ADD_SWITCH_COMMAND)
        add_switch(stoi(command_parts[ARG1]), stoi(command_parts[ARG2]));

    else if (command_parts[COMMAND] == ADD_SYSTEM_COMMAND)
        add_system(stoi(command_parts[ARG1]));

    else if (command_parts[COMMAND] == CONNECT_COMMAND)
        connect(stoi(command_parts[ARG1]), stoi(command_parts[ARG2]), stoi(command_parts[ARG3]));

    else if (command_parts[COMMAND] == SEND_COMMAND)
        send(stoi(command_parts[ARG1]), stoi(command_parts[ARG2]), command_parts[ARG3]);

    else if (command_parts[COMMAND] == RECEIVE_COMMAND)
        receive(stoi(command_parts[ARG1]));

    else
        cout << "Invalid command!" << NEW_LINE;
}
```

- int add_switch(int number_of_ports, int switch_number);

در این متد دستور MySwitch که برای اضافه کردن سوییچ است هندل می شود. در ابتدا یک pipe برای ارتباط سوییچ با شبکه ساخته می شود. در ادامه پردازه جدید برای سوییچ ساخته می شود و مسیر pipe ساخته شده به آن داده می شود.

```
int Network::add_switch(int number_of_ports, int switch_number) {
    string network_pipe_path = SWITCH_PREFIX + to_string(switch_number);
    unlink(network_pipe_path.c_str());
    mkfifo(network_pipe_path.c_str(), READ_WRITE);

    Pid p;

    p = fork();

    if (p < ZERO) {
        cout << FORK_FAILED_MESSAGE << NEW_LINE;
        return ONE;
    }
    else if (p > ZERO) {
        this->switches.insert({switch_number, p});
        return ZERO;
    }
    else {
        string switch_message = make_switch_message(number_of_ports, switch_number);
        char *args[] = {(Message)SWITCH_DIR, (Message)switch_message.c_str(), NULL};
        execv(args[ZERO], args);
        return ZERO;
    }
}
```

- int add_system(int system_number);

در این متد دستور MySystem که برای اضافه کردن سیستم است هندل می شود. در ابتدا یک pipe برای ارتباط سیستم با شبکه ساخته می شود. در ادامه پردازه جدید برای سیستم ساخته می شود و مسیر pipe ساخته شده به آن داده می شود.

```
int Network::add_system(int system_number) {
    string path_name_path = SYSTEM_PREFIX + to_string(system_number);
    unlink(path_name_path.c_str());
    mkfifo(path_name_path.c_str(), READ_WRITE);

    Pid p;

    p = fork();

    if (p < ZERO) {
        cout << FORK_FAILED_MESSAGE << NEW_LINE;
        return ONE;
    }
    else if (p > ZERO) {
        this->systems.insert({system_number, p});
        return ZERO;
    }
    else {
        string system_message = to_string(system_number);
        char *args[] = {(Message)SYSTEM_DIR, (Message)system_message.c_str(), NULL};
        execv(args[ZERO], args);
        return ZERO;
    }
}
```

- int send(int sender_number, int receiver_number, string file_path);

در این متد دستور Send که برای فرستادن پیام است هندل می شود. در ابتدا فایلی که قرار است ارسال شود خوانده می شود و در ادامه بر اساس حداکثر اندازه فایلی که میتواند ارسال شود به قسمت های کوچک تر تقسیم می شود و به سیستم فرستنده دستور ارسال داده می شود و آن هم فریم اترنت را به پورتش می فرستد و درصورت وجود سوییچ پیام فرستاده می شود.

```
int Network::send(int sender_number, int receiver_number, string file_path) {
    string content = read_file_to_string(file_path);
    vector<string> partitions = partition_content(content, MAX_FILE_PARTITION_SIZE);

    string system_pipe_path = SYSTEM_PREFIX + to_string(sender_number);
    int fds_system = open(system_pipe_path.c_str(), O_WRONLY);
    for (string partition : partitions) {
        string message = string(SEND_COMMAND)
                + string(COMMAND_SEPARATOR)
                + EthernetFrame::encode(EthernetFrame(sender_number, receiver_number, partition));
        write(fds_system, (Message) message.c_str(), message.size() + ONE);
    }

    close(fds_system);
    return ZERO;
}
```

- int receive(int system_number);

در این متد دستور Receive که برای دریافت پیام است هندل می شود. در این متد یک پیام به سیستم مورد نظر که باید پیامی را از صف پیامش دریافت کند فرستاده می شود.

```
int Network::receive(int system_number) {

    string system_pipe_path = SYSTEM_PREFIX + to_string(system_number);
    int fds_system = open(system_pipe_path.c_str(), O_WRONLY);

    string message = RECEIVE_COMMAND;
    write(fds_system, (Message) message.c_str(), message.size() + ONE);

    close(fds_system);
    return ZERO;
}
```

- int connect(int system_number, int switch_number, int port_number);

در این متد دستور Connect که برای اتصال یک سیستم و سوییچ است هندل می شود. در این متد ابتدا برای هر کدام از دو طرف اتصال یک pipe ساخته می‌شود. سپس اسم این پایپ‌ها برای آن‌ها فرستاده می‌شود.

- string make_switch_message(int number_of_ports, int switch_number);

از این متد در حین ایجاد یک سوئیچ استفاده می‌کنیم. به این ترتیب که تعداد پورت و شماره سوئیچ را در ورودی خود دریافت می‌کند سپس آرگومان ورودی پراسس مربوط به سوئیچ را در قالب یک رشته می‌سازد. به این منظور شماره سوئیچ و تعداد پورت را به کمک یک $ در کنار هم قرار می‌دهد و رشته حاصل را در خروجی خود برمی‌گرداند.

```
string Network::make_switch_message(int number_of_ports, int switch_number) {
    string message = to_string(switch_number);
    message += PROPS_SEPARATOR;
    message += to_string(number_of_ports);
    return message;
}
```

- string make_connect_pipe_path(int system_number, int switch_number, int port_number);

این متد در اصل برای ساخت پسوند نام پایپ‌هایی که برای اتصال یک سوئیچ و سیستم به کار می‌رود استفاده می‌شود. در ورودی خود شماره سیستم و شماره سوییچ و شماره پورت را دریافت می‌کند و با استفاده از _ این اعداد را در قالب یک رشته به هم متصل می‌کند.

```
string Network::make_connect_pipe_path(int system_number, int switch_number, int port_number) {
    string path = to_string(system_number) + PATH_SEPARATOR;
    path += to_string(switch_number) + PATH_SEPARATOR;
    path += to_string(port_number);
    return path;
}
```

- string make_connect_message(string switch_connection_pipe_path, string system_connection_pipe_path);

این متد برای ساخت پیام مربوط به دستور اتصال برای المان‌های شبکه (سیستم و سوییچ) استفاده می‌شود. به این ترتیب که در ابتدای پیام Connect نوشته می‌شود تا المان مربوطه متوجه دستور آمده از سوی شبکه بشود. در ادامه اسم پایپ‌های متناظر با سیستم و سوییچ برای این اتصال اضافه می‌شود.برای جدا کردن اجزای پیام از @ استفاده کردیم.

```
string Network::make_connect_message(string switch_connection_pipe_path,
    string system_connection_pipe_path) {
    string message = CONNECT_COMMAND;
    message += COMMAND_SEPARATOR;
    message += system_connection_pipe_path;
    message += COMMAND_SEPARATOR;
    message += switch_connection_pipe_path;
    return message;
}
```

- vector<string> partition_content(string content, int partition_size);

در این متد رشته ورودی براساس حداکثر سایز که در ورودی می آید به قسمت های کوچک تر تقسیم می شود و برگردانده می شود. (این متد برای دستور Send استفاده می شود.)

```
vector<string> Network::partition_content(string content, int partition_size) {
    int number_of_partitions = content.size() / partition_size + 1;
    vector<string> partitions;
    for (int i = 0; i < number_of_partitions - 1; i++) {
        partitions.push_back(content.substr(i * partition_size, (i + 1) * partition_size));
    }
    partitions.push_back(content.substr((number_of_partitions - 1) * partition_size));
    return partitions;
}
```

- Int stp();

در این قسمت متدی را می‌بینیم که برای شروع الگوریتم spanning tree در شبکه نوشته شده است. به این ترتیب که به همه‌ی سوییچ‌های موجود در سیستم دستور می‌دهد که فرستادن پیام‌های حاوی ریشه و فاصله از ریشه و آیدی خودشان را آغاز کنند. محتوای این دستور تنها یک Stp است.

```
int Network::stp() {
    map<int, Pid>::iterator it;
    for (it = switches.begin(); it != switches.end(); it++) {
        int switch_number = it->first;
        string switch_pipe_path = SWITCH_PREFIX + to_string(switch_number);
        string message = STP_COMMAND;
        int fd = open(switch_pipe_path.c_str(), O_RDWR);
        write(fd, (Message) message.c_str(), strlen((Message) message.c_str()) + ONE);
        close(fd);
    }
    return ZERO;
}
```

کلاس Switch

این کلاس در برنامه ما بیانگر اترنت سوئیچ است که امر پیام‌رسانی در شبکه کمک می‌کند.

فیلد ها

- int id: این فیلد نشان دهنده شناسه سوییچ است.
- int number_of_ports: این فیلد نشان دهنده تعداد پورت هایی است که سوییچ دارد.
- string network_pipe_path: این فیلد مشخص کننده مسیر pipe ای است که سوییچ از طریق آن می تواند با کلاس Network ارتباط داشته باشد.
- map<string, int> lookup: این فیلد یک مپ از شناسه سیستم (یا آدرس سیستم ها که در اینجا یکسان با شناسه آن ها است) به شماره پورت است که مشخص کننده lookup table سوییچ است.
- map<int, pair<string, string>> connection_pipe_paths: این فیلد یک مپ از شماره پورت به یک pair است که مشخص کننده مسیر pipe های متناظر با آن پورت است. عنصر اول آن نشان دهنده pipe برای دریافت و عنصر دوم آن نشان دهنده pipe برای ارسال پیام است.

متد ها

- void start(const char* args);

در این متد در ابتدا متد set_props فراخوانی می شود و مقادیر اولیه و مسیر pipe برای ارتباط با Network مقداردهی اولیه می شوند. در ادامه وارد حلقه می شود و با استفاده از تابع select به تمامی پورت های سوییچ و همچنین Network گوش داده می شود. هنگامی که پیامی دریافت شود براساس این که از طرف Network آمده است یا از پورت ها متد های handle_network_message و handle_ports_message فراخوانی می شود. و در انتها file descriptor هایی که باز شده اند بسته می شوند.

```
void Switch::start(const char* args) {
    set_props(args);

    char received_message[MAX_LINE] = {0};
    fd_set fds;
    int maxfd, activity;
    while (true) {
        int network_pipe_fd = open(this->network_pipe_path.c_str(), O_RDWR);
        maxfd = network_pipe_fd;

        FD_ZERO(&fds);
        FD_SET(network_pipe_fd, &fds);

        vector<int> connection_pipe_fds;
        vector<int> connection_ports;
        map<int, pair<string, string>>::iterator it;
        for (it = connection_pipe_paths.begin(); it != connection_pipe_paths.end(); it++)
        {
            int connection_pipe_fd = open(it->second.first.c_str(), O_RDWR);
            connection_pipe_fds.push_back(connection_pipe_fd);
            connection_ports.push_back(it->first);
            maxfd = connection_pipe_fd > maxfd ? connection_pipe_fd : maxfd;
            FD_SET(connection_pipe_fd, &fds);
        }

        activity = select(maxfd + 1, &fds, NULL, NULL, NULL);
        if (activity < 0)
            return;

        memset(received_message, 0, sizeof received_message);

        if (FD_ISSET(network_pipe_fd, &fds)) {
            read(network_pipe_fd, received_message, MAX_LINE);
            handle_network_command(received_message);
        }
        else {
            int incomming_message_port;
            for (size_t i = 0; i < connection_pipe_fds.size(); i++) {
                if (FD_ISSET(connection_pipe_fds[i], &fds)) {
                    incomming_message_port = connection_ports[i];
                    read(connection_pipe_fds[i], received_message, MAX_LINE);
                    break;
                }
            }

            for (int connection_pipe_fd : connection_pipe_fds)
                close(connection_pipe_fd);

            handle_ports_message(received_message, incomming_message_port);
        }

        close(network_pipe_fd);
    }
}
```

- void set_props(string data);

در این متد با استفاده از داده‌هایی که این پراسس در آرگومان تابع main از شبکه دریافت کرده است به مقداردهی فیلدهای آن (شامل شماره سوییچ و تعداد پورت) می‌پردازیم. از طرفی دیگر با توجه به قراردادی که در طراحی این شبکه در نظر گرفتیم نام پایپ ارتباط یک سوئیچ با شبکه به صورت switch_i خواهد بود که i بیانگر شماره این سوییچ است. به این ترتیب ویژگی network_pipe_path را نیز تنظیم می‌کنیم.

```
void Switch::set_props(string data) {
    vector<string> info = split(data, PROPS_SEPARATOR);
    this->id = stoi(info[ID]);
    this->network_pipe_path = PATH_PREFIX + info[ID];
    this->number_of_ports = stoi(info[NUMBER_OF_PORTS]);
}
```

- void handle_network_command(char* message);

این متد در اصل دستوراتی که از سوی شبکه به سوئیچ می‌رسد را مدیریت می‌کند. به این ترتیب که با استفاده از کاراکتر @ دستورات دریافت شده از شبکه را به اجزای آن تجزیه می‌کند. سپس با توجه به اولین بخش این دستور آن را تشخیص داده و سایر آرگومان‌ها را به تابع متناظر آن دستور می‌فرستد تا مدیریت شود.

```
void Switch::handle_network_command(char* message) {
    vector<string> info = split(message, COMMAND_SEPARATOR);

    if (info[COMMAND] == CONNECT_COMMAND || info[COMMAND] == CONNECT_SWITCH_COMMAND)
        connect(info[ARG2], info[ARG1]);
}
```

- void connect(string read_path, string write_path);

این متد اتصال بین یک سوییچ و سیستم یا اتصال بین یک سوییچ و سوییچ دیگر را مدیریت می‌کند. در ورودی خود دو رشته دریافت می‌کند که به ترتیب بیانگر نام پایپ مربوط به خواندن و نام پایپ مربوط به نوشتن هستند. سوییچ در ادامه برای پورتی که برای آن درخواست connect آمده است pipe های خواندن و نوشتن متناظر با آن را اضافه می کند.
(سوییچ برای فرستادن و دریافت پیام به پورت مربوطه از این دو pipe می تواند استفاده کند)

```
void Switch::connect(string read_path, string write_path) {
    vector<string> parts = split(read_path, PATH_SEPARATOR);
    if (connection_pipe_paths.find(stoi(parts[PORT_NUMBER])) == connection_pipe_paths.end())
        connection_pipe_paths.insert({stoi(parts[PORT_NUMBER]), make_pair(read_path, write_path)});
}
```

- void handle_ethernet_message(char* message, int port);

از این متد برای مدیریت پیام اترنت استفاده می‌کنیم.به این ترتیب که در ورودی خود پیام و پورت را دریافت می‌کند. سپس با استفاده از کاراکتر % پیام را به اجزای آن تجزیه می‌کند. حال بررسی می‌کند که آیا آدرس مبدا استخراج شده از پیام در جدول lookup آن وجود دارد یا خیر. در صورتی که وجود نداشت آن را به همراه شماره پورت در جدول خود درج می‌کند.

```
void Switch::handle_ethernet_message(char* message, int port) {
    vector<string> info = split(message, ETHERNET_SEPERATOR);
    if (lookup.find(info[SRC_ADDR_IDX]) == lookup.end())
        lookup.insert({info[SRC_ADDR_IDX], port});
```

```
if (lookup.find(info[DST_ADDR_IDX]) != lookup.end()) {
    int port = lookup[info[DST_ADDR_IDX]];
    int connection_pipe_fd = open(connection_pipe_paths[port].second.c_str(), O_WRONLY);
    write(connection_pipe_fd, message, strlen(message) + ONE);
    close(connection_pipe_fd);
}
```

- int get_id();

این متد getter ساده‌ای است که شماره سوییچ موردنظر را برمی‌گرداند.

- string get_path();

این متد نیز getter ای است که نام پایپ بین شبکه و سوییچ را برمی‌گرداند.

- std::string make_stp_message();

این متد به منظور ساخت پیام stp که توسط سوییچ برای الگوریتم spanning tree به پورت‌ها فرستاده می‌شود نوشته شده است.

```
string Switch::make_stp_message() {
    string message = "Stp%";
    message += to_string(id) + ETHERNET_SEPERATOR;
    message += to_string(root_id) + ETHERNET_SEPERATOR;
    message += to_string(root_distance);
    return message;
}
```

- void handle_stp_message(char* message, int port);

این متد برای اجرای الگوریتم spanning tree توسط سوییچ که در آن به انتخاب پورت‌هایی که باید خاموش شوند می‌پردازد نوشته شده است. در قسمت اول کد بررسی می‌شود که آیا پیامی‌که دریافت شده است در incoming از خود سوییچ بهتر است یا خیر.

```
void Switch::handle_stp_message(char* message, int port) {
    vector<string> info = split(message, ETHERNET_SEPERATOR);
    int incoming_id = stoi(info[ARG1]);
    int incoming_root_id = stoi(info[ARG2]);
    int incoming_root_distance = stoi(info[ARG3]);
    bool is_better = false;
    if (incoming_root_id < this->root_id) {
        if (incoming_root_distance == this->root_distance)
            is_better = (incoming_id <= this->sender_id);
        else
            is_better = (incoming_root_distance < this->root_distance);
    }
    else
        is_better = (incoming_root_id < this->root_id);
```

در قسمت دوم کد در صورتی که incoming از خود سوییچ بهتر بود پورت روت و آیدی روت و آیدی فرستنده و فاصله از روت آپدیت می‌شود. در غیر این صورت بررسی می‌شود که آیا incoming در اصل designated هست یا خیر.

```
if (is_better) {
    this->root_port = port;
    this->root_id = incoming_root_id;
    this->sender_id = incoming_id;
    this->root_distance = incoming_root_distance + 1;
}
else {
    bool is_designated = false;
    if (incoming_root_id > root_id)
        is_designated = false;

    else if (root_distance + 1 == incoming_root_distance)
        is_designated = (this->sender_id > incoming_id);

    else
        is_designated = (root_distance + 1 > incoming_root_distance);
```

در قسمت آخر کد در صورتی که incoming در اصل designated بود پورت موردنظر حذف می‌شود. در انتها نیز عمل همه‌پخشی روی همه‌ی پورت‌ها به جز روت صورت می‌گیرد.

```
    if (is_designated) {
        cout << "port " << port << " in switch " << id << " removed" << endl;
        cout << "> ";
        connection_pipe_paths.erase(port);
    }
}

map<int, pair<string, string>>::iterator it;
for (it = connection_pipe_paths.begin(); it != connection_pipe_paths.end(); it++) {
    if (it->first == port)
        continue;
    int connection_pipe_fd = open(it->second.second.c_str(), O_RDWR);
    string message = make_stp_message();
    write(connection_pipe_fd, message.c_str(), strlen(message.c_str()) + ONE);
    close(connection_pipe_fd);
}
```

کلاس System

این کلاس در برنامه ما بیانگر یک سیستم موجود در شبکه است که می‌خواهد با سایر سیستم‌ها به رد و بدل کردن پیام‌ها بپردازد.

فیلد ها

- int id: این فیلد نشان دهنده شناسه سیستم است.
- string network_pipe_path: این فیلد مشخص کننده مسیر pipe ای است که سیستم از طریق آن می تواند با کلاس Network ارتباط داشته باشد.
- pair<string, string> connection_pipe_path: این فیلد یک pair است که عنصر اول و دوم آن به ترتیب نشان دهنده مسیر pipe برای دریافت و ارسال پیام از طریق تنها پورت سیستم است.
- queue<EthernetFrame> message_queue: این فیلد یک صف از EthernetFrame است که پیام های دریافتی سیستم در آن ذخیره می شوند. (با فراخوانی دستور Receive پیام ها از این صف خارج می شوند.)

متد ها

- void start(const char* args);

در این متد در ابتدا با فراخوانی متد set_props مقادیر اولیه و مسیر pipe برای ارتباط با شبکه مقداردهی اولیه می شود. در ادامه سیستم وارد یک حلقه می شود که در آن با استفاده از تابع select به پورت سیستم و همچنین شبکه گوش داده می شود تا زمانی که از یکی از این دو پورت پیامی برسد. در ادامه بر اساس اینکه پیام از Network آمده است یا از پورت سیستم متد های handle_network_message و handle_ethernet_message فراخوانی می شوند و پیام دریافتی پردازش می شود. و در انتها file descriptor هایی که باز شده اند بسته می شوند.

```
void System::start(const char* args) {
    set_props(args);

    char received_message[MAX_LINE] = {0};
    fd_set fds;
    int maxfd, activity;
    while (true) {
        int network_pipe_fd = open(this->network_pipe_path.c_str(), O_RDWR);
        maxfd = network_pipe_fd;
        FD_ZERO(&fds);
        FD_SET(network_pipe_fd, &fds);
        int connection_pipe_fd;
        if (connection_pipe_path.first != "") {
            connection_pipe_fd = open(this->connection_pipe_path.first.c_str(), O_RDWR);
            maxfd = connection_pipe_fd > network_pipe_fd ? connection_pipe_fd : network_pipe_fd;
            FD_SET(connection_pipe_fd, &fds);
        }

        activity = select(maxfd + 1, &fds, NULL, NULL, NULL);
        if (activity < 0)
            return;

        memset(received_message, 0, sizeof received_message);

        if (FD_ISSET(network_pipe_fd, &fds)) {
            if (connection_pipe_path.first != "")
                close(connection_pipe_fd);
            read(network_pipe_fd, received_message, MAX_LINE);
            handle_network_command(received_message);
        }

        else if (FD_ISSET(connection_pipe_fd, &fds)) {
            read(connection_pipe_fd, received_message, MAX_LINE);
            if (connection_pipe_path.first != "")
                close(connection_pipe_fd);
            handle_ethernet_message(received_message);
        }

        close(network_pipe_fd);
    }
}
```

- void set_props(std::string data);

در این متد با استفاده از داده‌هایی که این پراسس در آرگومان تابع main از شبکه دریافت کرده است به مقداردهی فیلد آن (شماره سیستم) می‌پردازیم. از طرفی دیگر با توجه به قراردادی که در طراحی این شبکه در نظر گرفتیم نام پایپ ارتباط یک سیستم با شبکه به صورت system_i خواهد بود که i بیانگر شماره این سیستم است. به این ترتیب ویژگی network_pipe_path را نیز تنظیم می‌کنیم.

```
void System::set_props(string data) {
    this->id = stoi(data);
    this->network_pipe_path = PATH_PREFIX + data;
}
```

- void handle_network_command(char* message);

این متد در اصل دستوراتی که از سوی شبکه به سیستم می‌رسد را مدیریت می‌کند. به این ترتیب که با استفاده از کاراکتر @ دستورات دریافت شده از شبکه را به اجزای آن تجزیه می‌کند. سپس با توجه به اولین بخش این دستور آن را تشخیص داده و سایر آرگومان‌ها را به تابع متناظر آن دستور می‌فرستد تا مدیریت شود.

```
void System::handle_network_command(char* message) {
    vector<string> info = split(message, COMMAND_SEPARATOR);

    if (info[COMMAND] == CONNECT_COMMAND)
        connect(info[ARG1], info[ARG2]);

    if (info[COMMAND] == SEND_COMMAND)
        network_send(info[ARG1]);

    if (info[COMMAND] == RECEIVE_COMMAND)
        network_receive();
}
```

- void connect(std::string read_path, std::string write_path);

این متد اتصال بین سیستم و یک سوییچ و سوییچ را مدیریت می‌کند. در ورودی خود دو رشته دریافت می‌کند که به ترتیب بیانگر نام پایپ مربوط به خواندن و نام پایپ مربوط به نوشتن هستند. سپس این دو را در قالب یک pair در فیلد connection_pipe_path خود قرار می‌دهد.

```
void System::connect(string read_path, string write_path) {
    this->connection_pipe_path = make_pair(read_path, write_path);
}
```

- void network_send(std::string ethernet_message);

این متد در ورودی خود یک پیام اترنت را در قالب یک رشته دریافت می‌کند. سپس در صورتی که نام پایپ مربوط به اتصال آن تنظیم شده بود این پایپ را در حالت نوشتن باز می‌کند و پیام موجود در آرگومان خود را در آن می‌نویسد و در نهایت پایپ را می‌بندد.

```
void System::network_send(string ethernet_message) {
    if (connection_pipe_path.second != "") {
        int connection_pipe_fd = open(this->connection_pipe_path.second.c_str(), O_WRONLY);
        write(connection_pipe_fd, ethernet_message.c_str(), ethernet_message.size() + ONE);
        close(connection_pipe_fd);
    }
}
```

- void network_receive();

همانطور که پیش‌تر نیز گفته شده از یک صف برای ذخیره پیام‌های دریافتی استفاده می‌کنیم. هرگاه که send صورت می‌گیرد پیام موردنظر به تعدادی اترنت فریم شکسته می‌شود و به ترتیب در صف مربوط به سیستم مقصد قرار می‌گیرد. بنابراین در متد دریافت تنها کافی است که اولین عنصری که در صف وارد شده است را از صف خارج کنیم و آدرس مبدا و محتوای آن را چاپ کنیم.

```
void System::network_receive() {
    if (!message_queue.empty()) {
        EthernetFrame frame = message_queue.front();
        cout << "Source Address: " << frame.getSourceAddress() << NEW_LINE;
        cout << "Content: " << frame.getContent() << NEW_LINE;
        message_queue.pop();
    }
    else {
        cout << "No messeage has been received yet!" << NEW_LINE;
    }
    cout << "> ";
}
```

- void handle_ethernet_message(char* message);

این متد در ورودی خود یک پیام دریافت می‌کند. سپس آن را با استفاده از % به اجزای آن تجزیه می‌کند. در صورتی که آدرس مقصد این پیام با شماره این سیستم یکسان بود آن را به کمک متد decode از کلاس EthernetFrame به یک اترنت فریم تبدیل می‌کند و به انتهای صف پیام‌های دریافتی خود اضافه می‌کند.

- int get_id();

این متد getter ساده‌ای است که شماره سیستم موردنظر را برمی‌گرداند.

کلاس EthernetFrame

این کلاس در اصل یک شبیه‌سازی از اترنت فریم است و در آن متدهایی از قبیل encode و decode تعریف کرده‌ایم تا به نوعی فشرده‌سازی پیام‌ها در قالب یک رشته توسط خود این کلاس encapsulate شود و قراردادها مختص خود آن باشد و المان‌های دیگر برنامه درگیر آن نشوند.

فیلد ها

- int src_address: این فیلد نشان دهنده آدرس مبدا فریم است که در اصل همان شماره سیستم فرستنده است.
- int dst_address: این فیلد نشان دهنده آدرس مقصد فریم است که در اصل همان شماره سیستم گیرنده است.
- string content: این فیلد بیانگر محتوای پیام موجود در فریم است.

متد ها

- static EthernetFrame decode(const std::string ethernet_message)

این متد در ورودی خود یک پیام اترنت را در قالب یک رشته دریافت می‌کند. سپس اجزای این پیام شامل آدرس مبدا و مقصد و محتوا را تجزیه می‌کند و آن‌ها را به کانستراکتور کلاس داده و یک آبجکت اترنت فریم برمی‌گرداند.

```
EthernetFrame EthernetFrame::decode(const std::string ethernet_message) {
    vector<string> frame_parts = split(ethernet_message, ETHERNET_SEPERATOR);
    return EthernetFrame(stoi(frame_parts[SRC_ADDR_IDX]), stoi(frame_parts[DST_ADDR_IDX]),
        frame_parts[CONTENT_IDX]);
}
```

- static std::string encode(const EthernetFrame ethernet_frame)

این متد در ورودی خود یک آبجکت اترنت فریم دریافت می‌کند و سپس اجزای تشکیل دهنده آن را در قالب یک رشته encode می‌کند و در نهایت این رشته را برمی‌گرداند.

```
std::string EthernetFrame::encode(const EthernetFrame ethernet_frame) {
    return to_string(ethernet_frame.src_address)
           + ETHERNET_SEPERATOR
           + to_string(ethernet_frame.dst_address)
           + ETHERNET_SEPERATOR
           + ethernet_frame.content;
}
```

- std::string getContent()

این متد getter ساده‌ای است که محتوای پیام فریم را برمی‌گرداند.

- int getSourceAddress()

این متد getter ای است که آدرس مبدا پیام فریم را برمی‌گرداند.

گام دوم

توجه داریم که توضیح تمامی این دستورات در کدهای گام اول آورده شده است.

اضافه کردن سوئیچ

با استفاده از دستور زیر یک سوئیچ به شبکه اضافه می‌کنیم. در آرگومان اول دستور تعداد پورت‌ها و در آرگومان بعدی شماره سوئیچ را می‌نویسیم.

ورودی

```
> MySwitch <number_of_ports> <switch_number>
```

ورودی نمونه

```
> MySwitch 3 1
```

اضافه کردن سیستم

با استفاده از دستور زیر یک سیستم به شبکه اضافه می‌کنیم. در تنها آرگومان آن شماره سیستم را می‌نویسیم. همانطور که در گام اول توضیح داده شد کلیت اضافه کردن یک سیستم به این ترتیب است که

ورودی

```
> MySystem <system_number>
```

ورودی نمونه

```
> MySystem 1
```

اتصال سیستم و سوئیچ

با استفاده از دستور زیر یک سیستم و سوئیچ را به هم وصل می‌کنیم. در آرگومان اول آن شماره سیستم و در آرگومان دوم شماره سوئیچ و در آرگومان سوم آن نیز پورت موردنظر سوئیچ برای اتصال می‌آید.

ورودی

```
> Connect <system_number> <switch_number> <port_number>
```

ورودی نمونه
`> Connect 1 2 1`

اتصال سوئیچ و سوئیچ

با استفاده از دستور زیر دو سوئیچ را به هم وصل می‌کنیم. در آرگومان اول و دوم آن به ترتیب شماره‌های دو سوئیچ موردنظر می‌آید. سپس در آرگومان‌های سوم و چهارم پورت‌های متناظر با این اتصال از سوئیچ‌های اول و دوم می‌آید.

ورودی
`> ConnectSwitch <switch_number1> <switch_number2> <port_number1> <port_number1>`

ورودی نمونه
`> ConnectSwitch 1 2 2 3`

ارسال فایل

با استفاده از دستور زیر به ارسال یک فایل از یک سیستم به سیستم دیگر می‌پردازیم. در آرگومان اول و دوم آن به ترتیب شماره‌های سیستم‌های فرستنده و گیرنده می‌آید. آرگومان سوم آن نیز بیانگر آدرس نسبی فایل موردنظر است.

ورودی
`> Send <sender_number> <receiver_number> <file_path>`

ورودی نمونه
`> Send 1 2 testfile2`

دریافت

با استفاده از دستور زیر یک دریافت توسط سیستمی که شماره آن در آرگومان اول آمده است صورت می‌گیرد. به این ترتیب که این سیستم یک فریم از سر صف خود برمی‌دارد و آدرس مبدا و محتوای آن را در ترمینال چاپ می‌کند.

ورودی

```
> Receive <system_number>
```

ورودی نمونه

```
> Receive 1
```

الگوریتم Spanning Tree Protocol

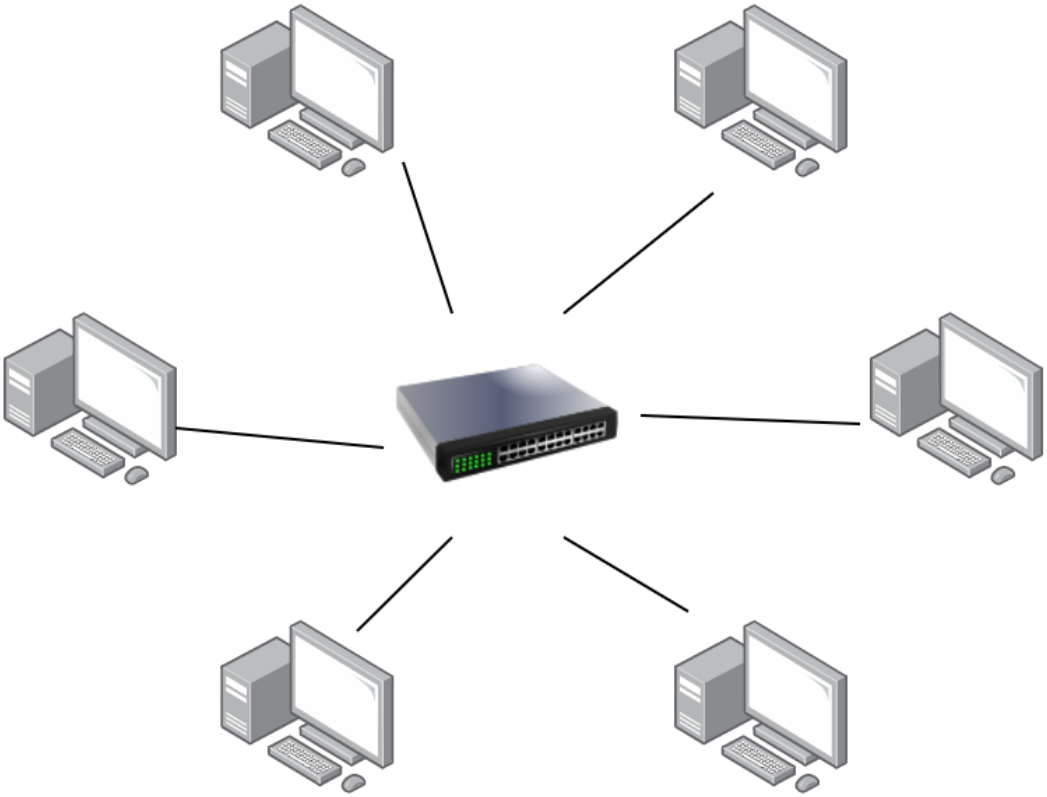
با استفاده از دستور زیر به اجرای الگوریتم پروتکل درخت پوششی می‌پردازیم. به این ترتیب که در صورت وجود دور پورت‌های موردنیاز را خاموش می‌کند تا همچنان بتوانیم به رد و بدل فایل بین سیستم‌ها بپردازیم.

ورودی

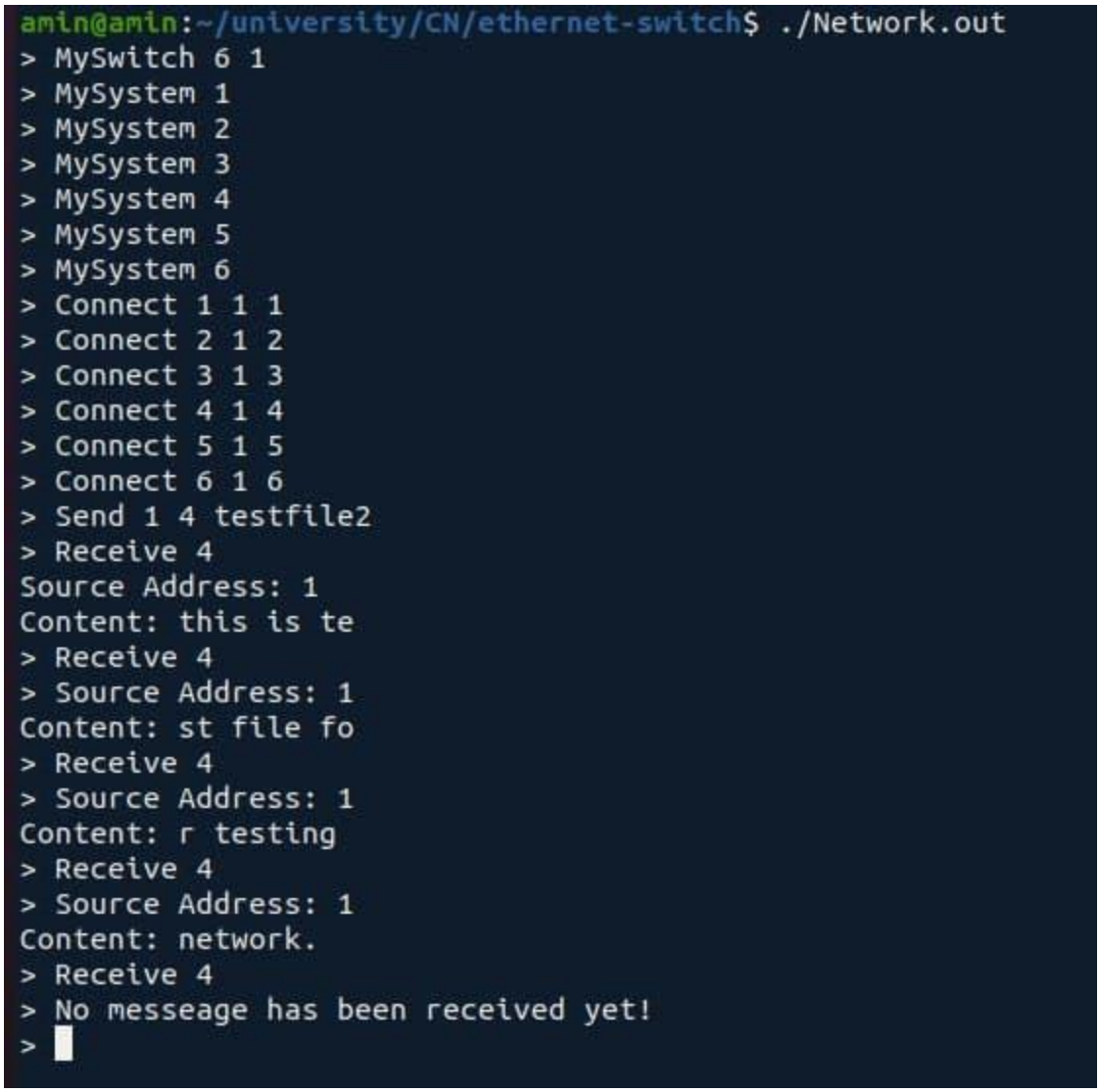
```
> Stp
```

گام سوم

توپولوژی که برای این گام تعرف کردیم به صورت زیر است که از ۶ سیستم و ۱ سوئیچ تشکیل شده است.

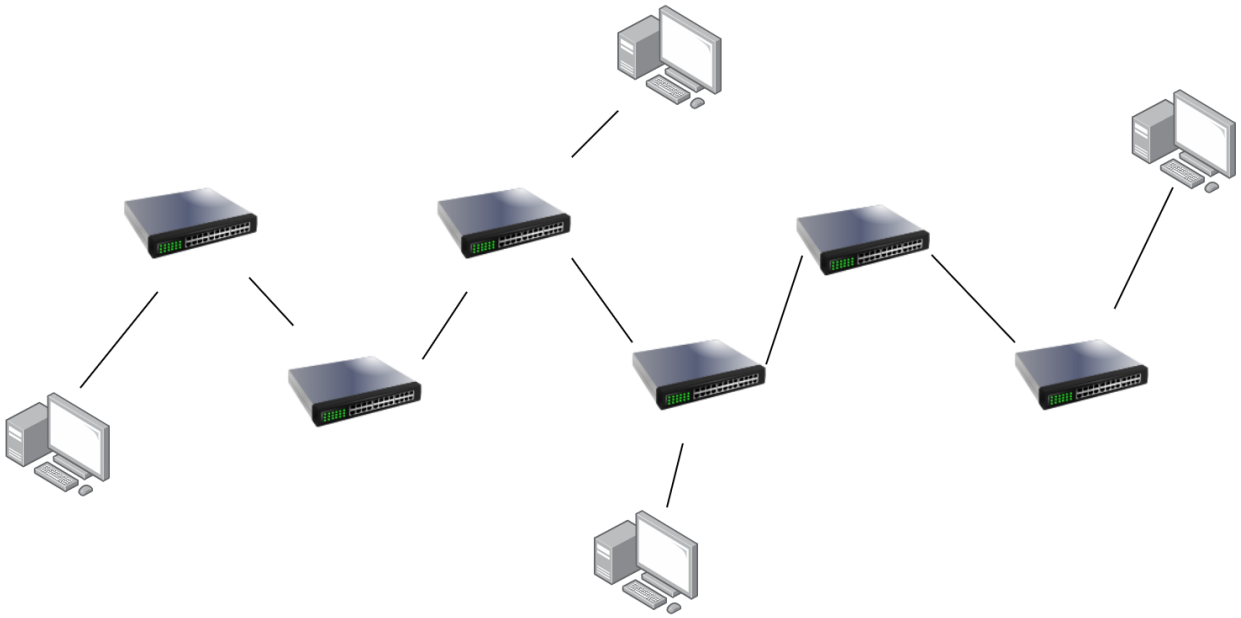


در این قسمت دستورات موردنیاز برای ساخت توپولوژی را در ترمینال مشاهده می‌کنیم. سپس به مبادله فایل testfile2 بین سیستم‌های 1 و 4 پرداختیم که در طی 4 فریم به دست سیستم 4 رسیده است.

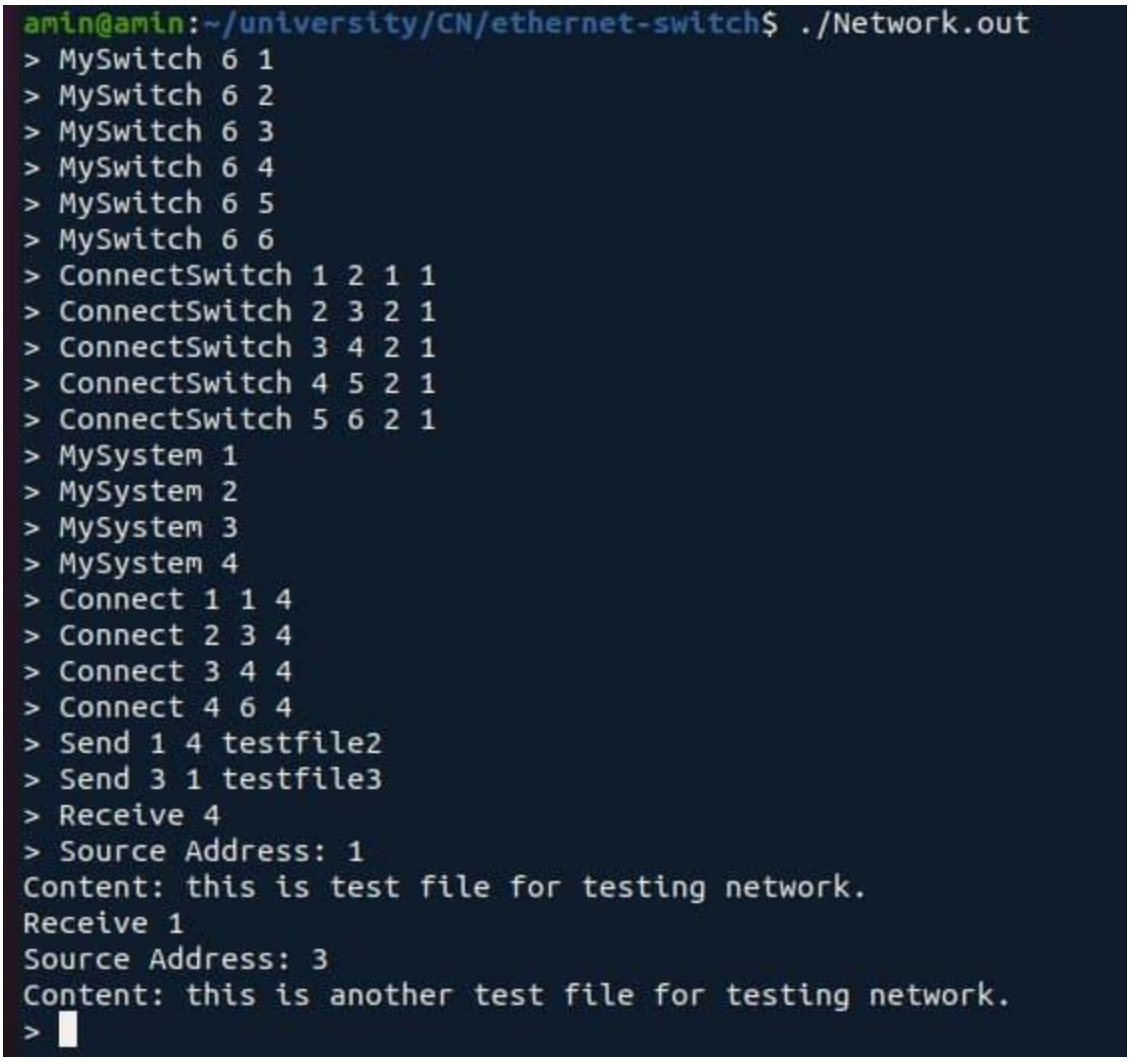

amin@amin:~/university/CN/ethernet-switch$ ./Network.out
> MySwitch 6 1
> MySystem 1
> MySystem 2
> MySystem 3
> MySystem 4
> MySystem 5
> MySystem 6
> Connect 1 1 1
> Connect 2 1 2
> Connect 3 1 3
> Connect 4 1 4
> Connect 5 1 5
> Connect 6 1 6
> Send 1 4 testfile2
> Receive 4
Source Address: 1
Content: this is te
> Receive 4
> Source Address: 1
Content: st file fo
> Receive 4
> Source Address: 1
Content: r testing
> Receive 4
> Source Address: 1
Content: network.
> Receive 4
> No messeage has been received yet!
>

گام چهارم

توپولوژی که برای این گام تعرف کردیم به صورت زیر است که از ۴ سیستم و ۶ سوئیچ تشکیل شده است.

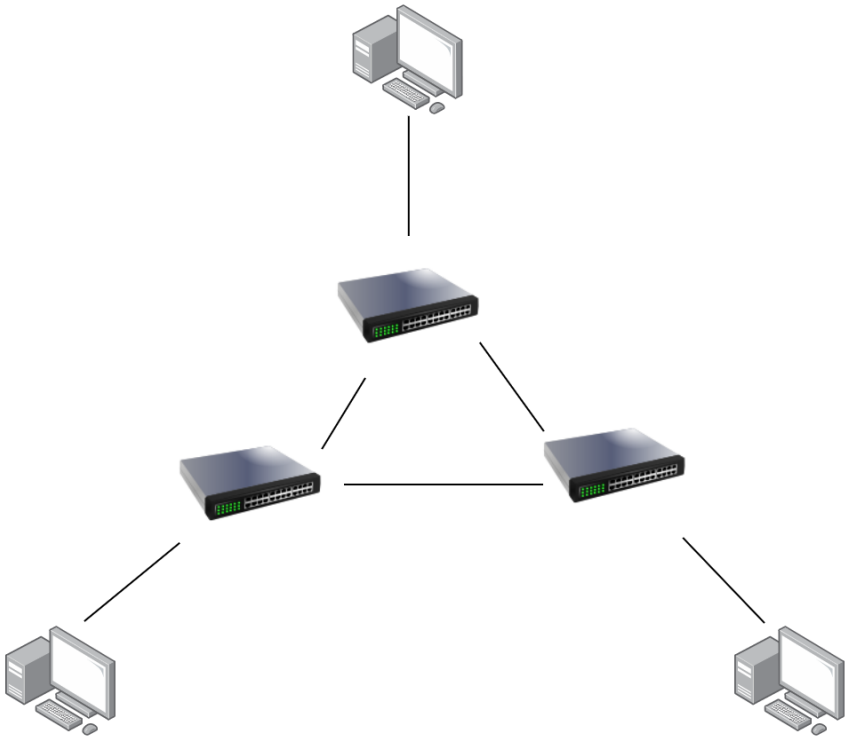


در این قسمت دستورات موردنیاز برای ساخت توپولوژی را در ترمینال مشاهده می‌کنیم. سپس به مبادله فایل testfile2 بین سیستم‌های 1 و 4 پرداختیم. سپس به مبادله testfile3 بین سیستم‌های 3 و 1 پرداختیم که همگی به درستی به گیرنده رسیده‌اند.


amin@amin:~/university/CN/ethernet-switch$ ./Network.out
> MySwitch 6 1
> MySwitch 6 2
> MySwitch 6 3
> MySwitch 6 4
> MySwitch 6 5
> MySwitch 6 6
> ConnectSwitch 1 2 1 1
> ConnectSwitch 2 3 2 1
> ConnectSwitch 3 4 2 1
> ConnectSwitch 4 5 2 1
> ConnectSwitch 5 6 2 1
> MySystem 1
> MySystem 2
> MySystem 3
> MySystem 4
> Connect 1 1 4
> Connect 2 3 4
> Connect 3 4 4
> Connect 4 6 4
> Send 1 4 testfile2
> Send 3 1 testfile3
> Receive 4
> Source Address: 1
Content: this is test file for testing network.
Receive 1
Source Address: 3
Content: this is another test file for testing network.
>

گام پنجم

توپولوژی که برای این گام تعرف کردیم به صورت زیر است که از ۳ سیستم و ۳ سوئیچ تشکیل شده است.



در این قسمت دستورات موردنیاز برای ساخت توپولوژی را در ترمینال مشاهده می‌کنیم. توجه داریم که با اجرای دستور Stp پورت موزدنظر که باید خاموش شود در ترمینال چاپ شده است.

```
ghazal@ghazal-System-Product-Name:~/Desktop/CN-CA2$ ./Network.out
> MySwitch 3 1
> MySwitch 3 2
> MySwitch 3 3
> MySystem 1
> MySystem 2
> MySystem 3
> Connect 1 1 1
> Connect 2 2 2
> Connect 3 3 3
> ConnectSwitch 1 2 2 1
> ConnectSwitch 1 3 3 1
> ConnectSwitch 2 3 3 2
> Stp
> port 1 in switch 2 removed
```

در این قسمت نیز ادامه‌ی دستورات برای رد و بدل فایل بین سیستم‌های 1 و 2 در همان توپولوژی قبلی را مشاهده می‌کنیم. که به درستی صورت گرفته است.

نتیجه‌گیری

در این پروژه یک شبکه متشکل از اترنت سوئیچ‌ها را شبیه‌سازی نمودیم و به طور دقیق با ویژگی‌های آن آشنا شدیم. همچنین به طور گسترده با نحوه‌ی کار با پایپ‌های با نام در زبان C++ برای ایجاد یک برنامه multi-processing آشنا شدیم.