08.06.2016

Monitorowanie pracowni laboratoryjnej - dokumentacja

Autorzy:

Adrian Kaczmarek

Krzysztof Rózga

Jakub Plebaniak

1. Temat

Naszym zadaniem było stworzenie aplikacji służącej do monitorowania przez prowadzącego zajęcia, pracy studentów na ich komputerach. Miałby on udostępniać najważniejsze informacje z stacji studenta: zrzuty ekranu, listę aktualnie otwartych procesów, oraz listę otwartych stron przeglądarki.

Powodem dla którego wybraliśmy ten temat była chęć udoskonalenia umiejętności programowania wielowątkowego, a także bliższe przyjrzenie się działaniu gniazd w aplikacjach sieciowych. Interesowały ich możliwe konfiguracje oraz sposoby łączenia poszczególnych hostów.

Chcieliśmy również poszerzyć swoją wiedzę z zakresu tworzenia aplikacji desktopowych w języku Java, z wykorzystaniem biblioteki JavaFX. Jest bardzo wygodna i przejrzysta biblioteka, pozwala na szybkie i proste tworzenie interfejsu użytkownika.

2. Zadania systemu

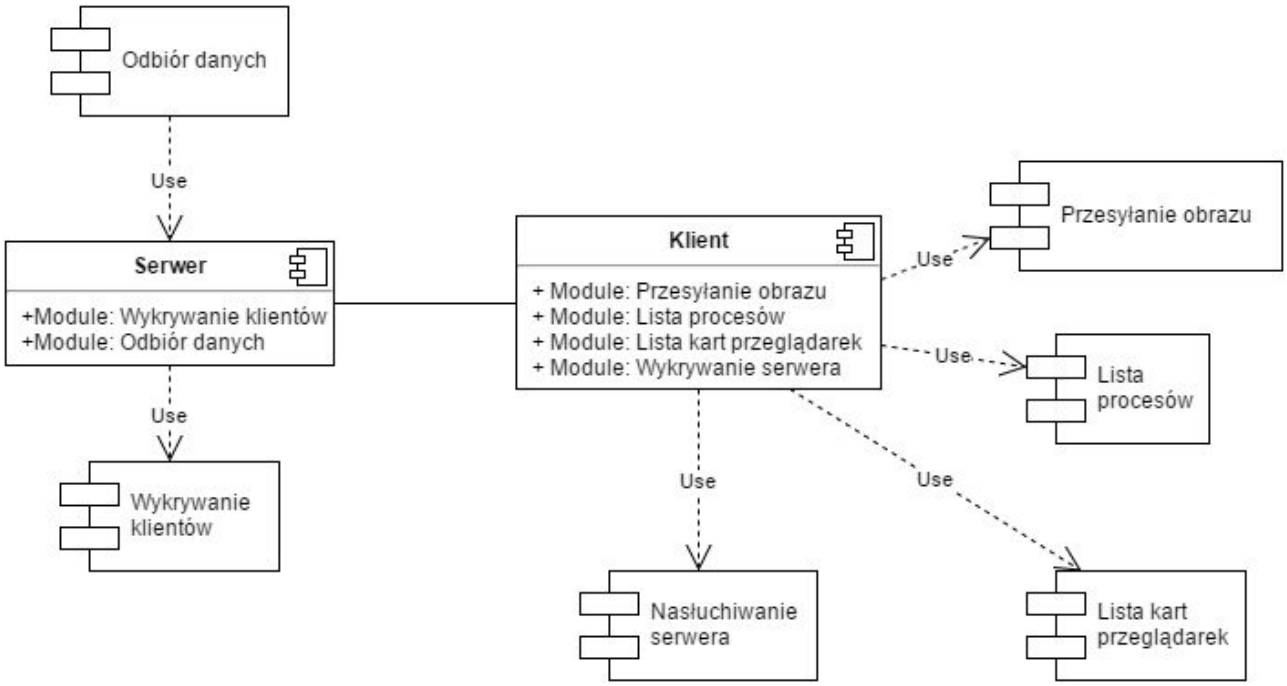
- Aplikacja prowadzącego:
    - dostarczenie przejrzystego i prostego interfejsu użytkownika,
    - wyświetlanie w oknie głównym, listy podłączonych stacji studentów w postaci miniatur zrzutów ekranu,
    - każda miniatura powinna dodatkowo wyświetlać adres IP węzła, z którego przesyłany jest obraz,
    - umożliwienie wyświetlenia szczegółowego okna, pokazującego obraz w wyższej rozdzielczości, dane o procesach w systemie, oraz opcjonalnie informacje o otwartych zakładkach w przeglądarce. Okno to zostaje uruchomione w momencie kliknięcia na daną miniaturę,
    - wysyłanie pakietów informujących o własnym IP na adres multicast, w celu ułatwienia stacjom studenckim wykrycia aplikacji z którą mają się łączyć

- Aplikacja studenta:
    - podłączenie się do aplikacji prowadzącego, po wykryciu jego adresu,
    - przesyłanie obrazu o niskiej rozdzielczości,
    - przesyłanie obrazu o wysokiej rozdzielczości
    - przesyłanie filtrowanej listy procesów,
    - przesyłanie listy otwartych kart przeglądarek.

3. Podział prac i harmonogram

Osoba	Zadania
Adrian Kaczmarek	● **realizacja modułu przesyłania obrazu przez klienta** ● **realizacja modułu odbioru obrazu przez serwer** ● **praca nad GUI**
Krzysztof Rózga	● **implementacja wykrywania serwera aplikacji w sieci** ● **realizacja modułu pobierania i przesyłania listy procesów** ● **praca nad GUI**
Jakub Plebaniak	● **realizacja modułu wykrywania otwartych stron przeglądarki oraz przesyłania ich w postaci listy,** ● **praca nad GUI**

4. Architektura systemu



System składa się dwóch części: aplikacji klienta (student) i serwera (prowadzącego).

5. Napotkane problemy

- **Obraz**

W przypadku przesyłania obrazu, najwięcej problemów sprawiło samo cykliczne jego przesyłanie. Pierwsze przyjęte rozwiązanie, wykorzystujące klasę *ImageIO*, pozwalało tylko na jednorazowe przesłanie obrazu. Wcześniej wspomniana klasa mogła nadal zostać wykorzystana, lecz wiązałoby się to z dużymi narzutami czasowymi operacji. Dlatego zastosowaliśmy rozwiązanie w postaci zamiany obrazu na tablice bajtów, która bez większych problemów może zostać przesłana.

Kolejnym problemem było opracowanie sposobu rozpoczęcia przesyłania obrazu o dużej rozdzielczości, w momencie kliknięcia przez prowadzącego danej miniatury. Przyjęte rozwiązanie zakładało stworzenie gniazda serwerowego, nasłuchującego na połączenie od aplikacji prowadzącego. Całość działała w

oddzielnym wątku. W momencie wykrycia połączenia, host rozpoczynał przesyłanie zrzutów ekranu o dużej rozdzielczości

- **Uruchomione procesy**

Głównym problemem przy wysyłaniu informacji o uruchomionych procesach było zarządzanie sesją, czyli zainicjowanie transmisji procesów wybranego klienta przez serwer. Problem został rozwiązany za pomocą implementacji klasy `ManageServer` (paczka `procManageLib`), która zarządza sesją, wysyła żądania transmisji.

Kolejnym problemem było filtrowanie listy procesów. Wystąpiła potrzeba filtrowania gdyż wiele procesów jest uruchomionych domyślnie oraz część z nich jest sesją konsolową. W rezultacie problem został rozwiązany poprzez usunięcie z listy procesów domyślnych i tych konsolowych.

- **Otwarte strony**

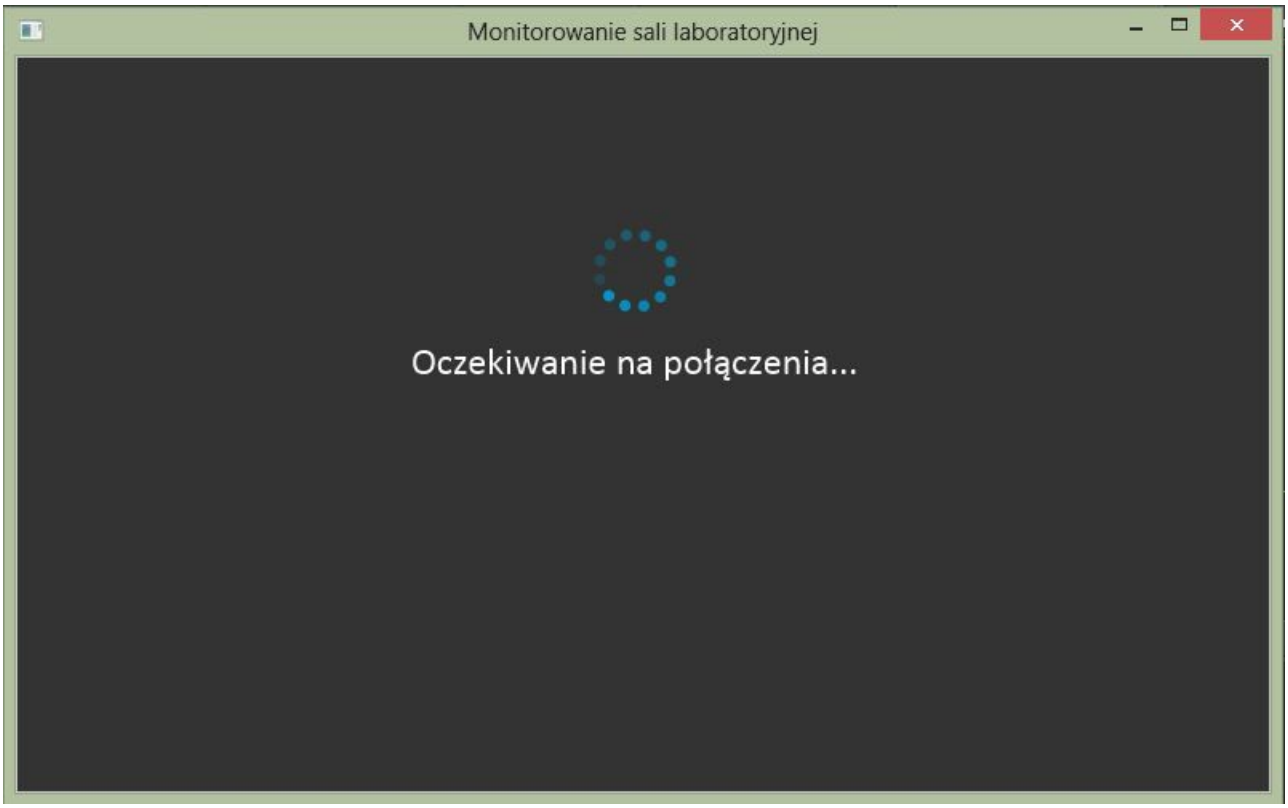
Podczas wykrywania odwiedzonych stron wystąpił problem z uzyskaniem adresu domenowego z pakietów https. Problem został rozwiązany poprzez ustalenie nazwy domenowej na podstawie adresu IP.

Kolejnym problemem była konieczność filtrowania wykrytych stron, co zostało zrealizowane wykorzystując następujące wyrażenie regularne: `"([a-zA-Z]+\\.)?[\\w]+\\.[a-zA-Z]+"`.

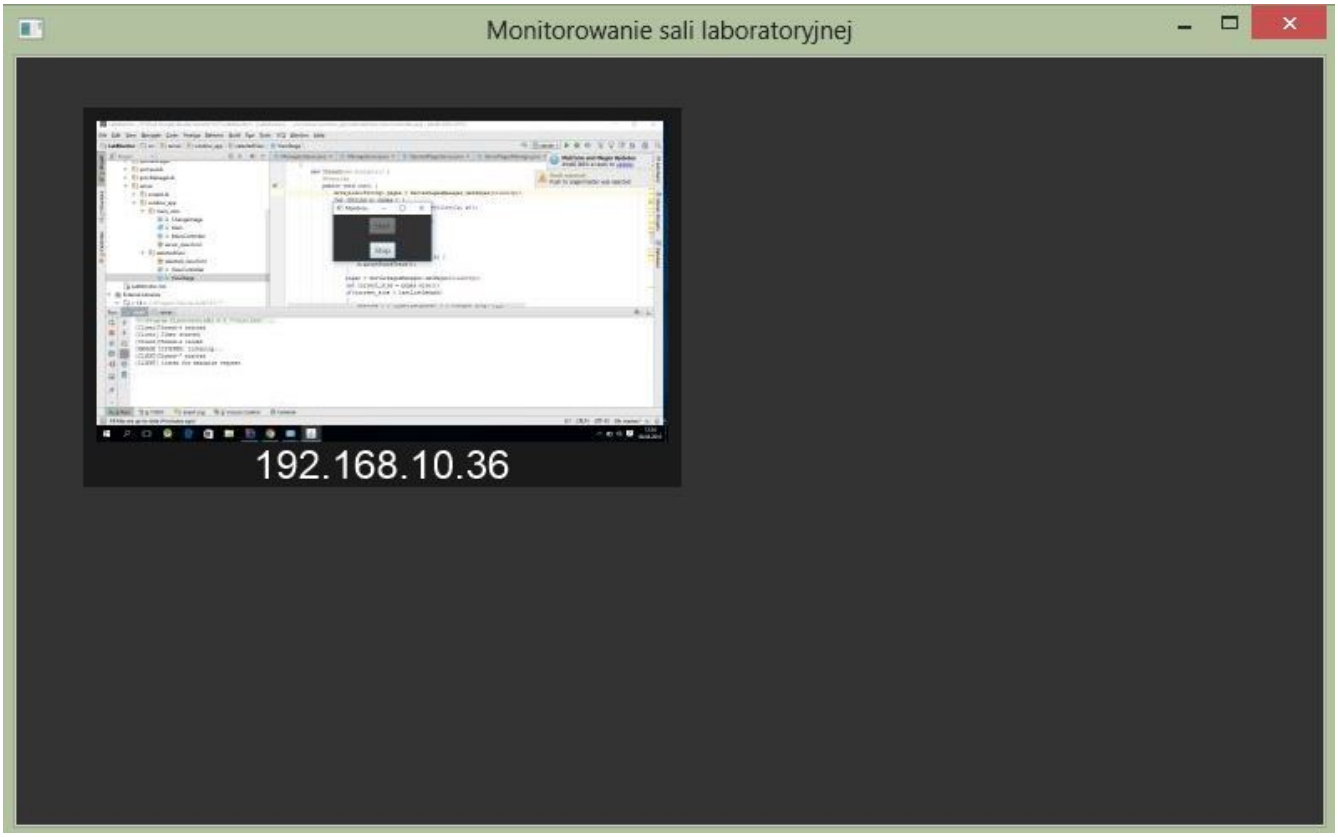
6. Środowiska i narzędzia

- Język programowania obiektowego Java,
- system kontroli wersji - GitHub,
- srodowisko programistyczne IntelliJ IDEA 2016.1,
- JavaFX– biblioteka graficzna używana w języku programowania Java,
- Wtyczka JavaFX Scene Builder - do budowania interfejsu użytkownika.
- JnetPcap - biblioteka do podsłuchiwania przesyłanych pakietów

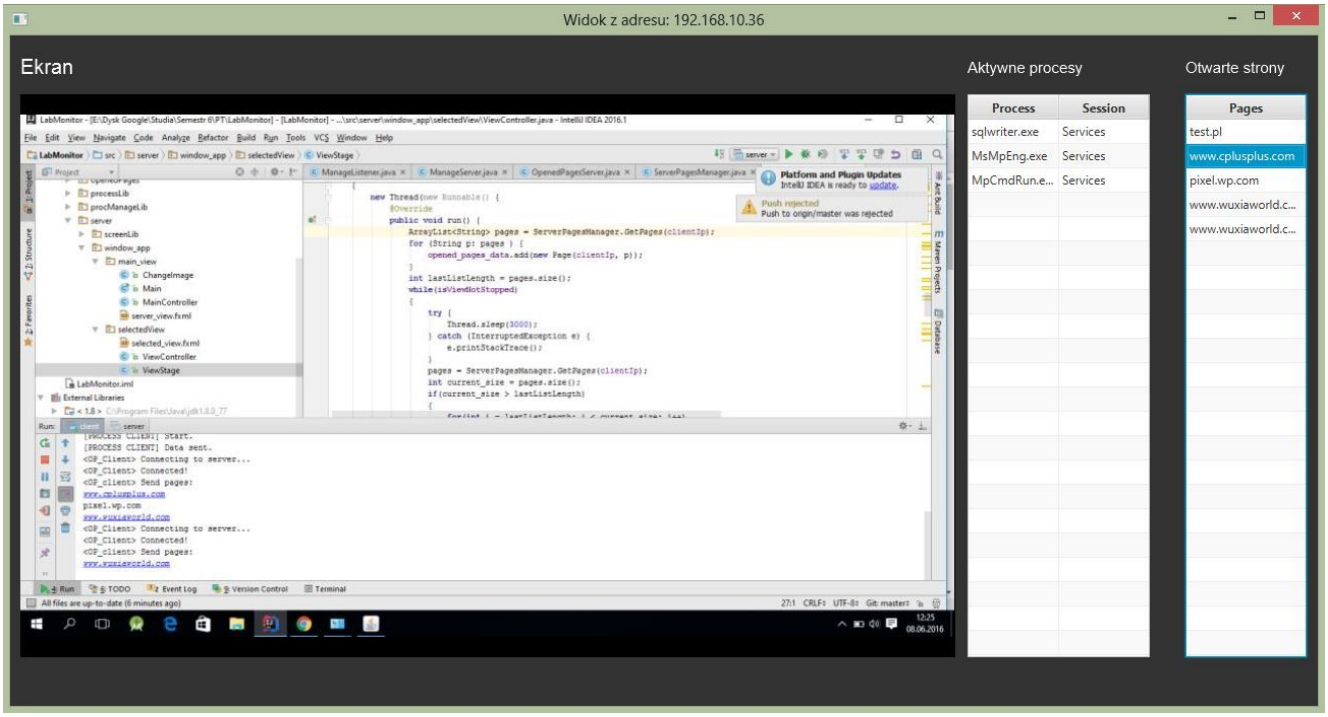
7. Użytkowanie



Prowadzący, po uruchomieniu aplikacji widzi powyższy ekran. Oznacza on, że do systemu nadal nie podłączył się żaden student.

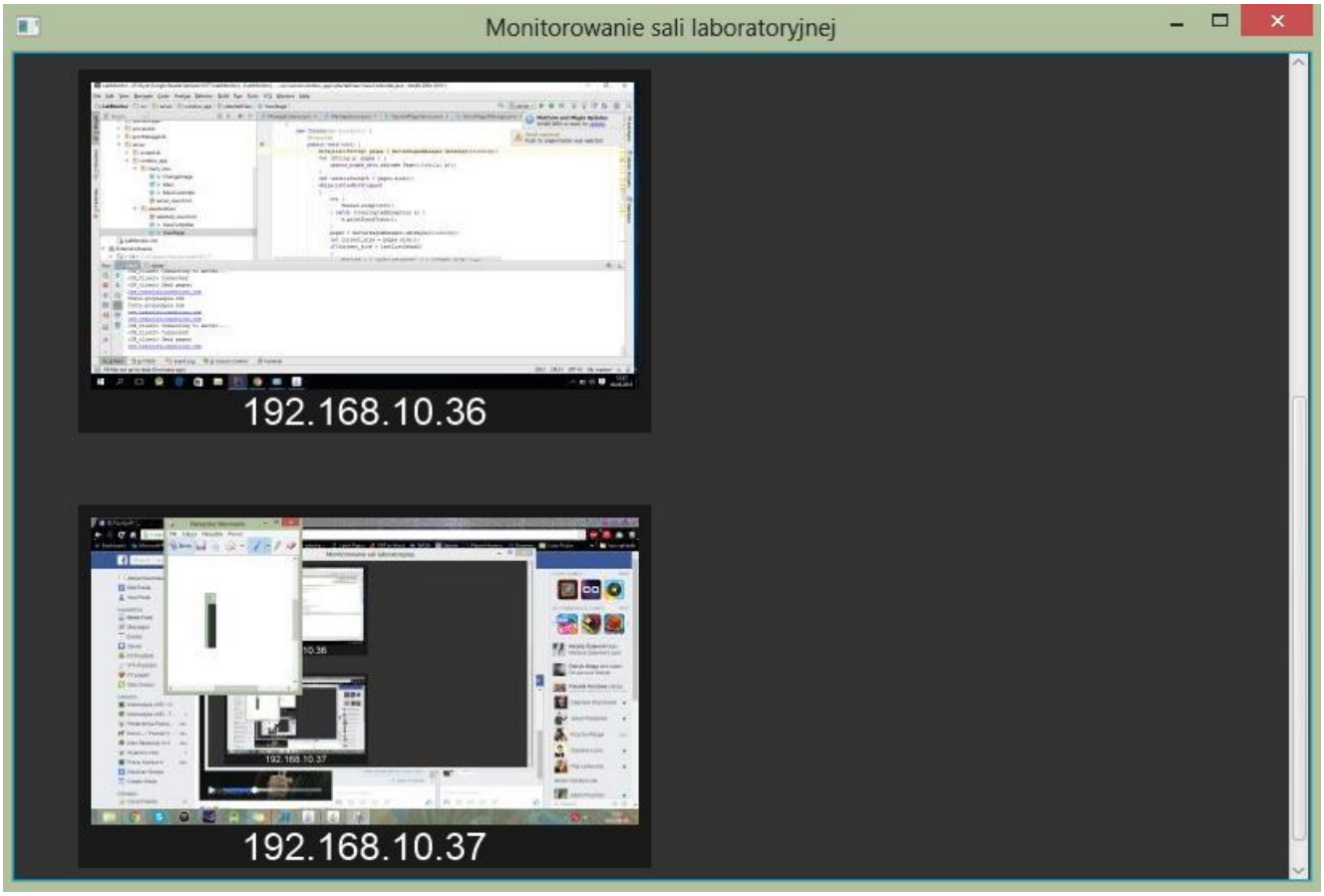


Po podłączeniu komputera studenta, na ekranie prowadzącego powinna pojawić się miniaturka. Przedstawia on zrzut ekranu danego komputera. Obraz został przeskalowany do takiego rozmiaru (więcej klatek na sekundę). Dodatkowo pod miniaturą wyświetlony zostaje adres IP podłączonej stacji. Na daną miniaturę można kliknąć, co spowoduje wyświetlenie szczegółowych informacji ze stacji w nowym oknie.



Powyższe okno otworzy się po kliknięciu miniatury odpowiadającej danej stacji. Elementami widoku są:

- Ekran - obraz przesłany ze stacji studenta. Jest on w wyższej rozdzielczości niż miniatura we wcześniejszym oknie (nie jest przeskalowany).
- Aktywne procesy - jest to lista w widoku tabeli, przedstawiająca listę otwartych procesów po stronie studenta. Warto zaznaczyć, że lista jest stale aktualizowana i przefiltrowana - usunięte procesy samego systemu operacyjnego, niezwiązane z działaniem studenta.
- Otwarte strony - podobnie jak aktywne procesy, lista jest przedstawiona w postaci tabeli jednokolumnowej. Przedstawia one aktualnie otwierane strony przez studenta, niezależnie od przeglądarki.



Powyższy widok przedstawia miniatury podłączonych stacji studenckich. Pokazane jest ich rozmieszczenie, wykonywane dynamicznie. Całość dodawana jest do widoku, który można przewinąć za pomocą scroll'a lub suwaka znajdującego się po prawo okna.

Interfejs aplikacji nie wymaga dużej ingerencji użytkownika. Okna z konkretnych stacji pojawiają i usuwają się automatycznie. Faktycznie, prowadzący może faktycznie jedynie kliknąć daną miniaturę, aby przejść do podglądu stacji.

8. Implementacja

8.1. Obraz

8.1.1. Przesyłanie do prowadzącego

Przesyłanie miniatur

```
private Socket refSock;
private DataOutputStream output;
private Dimension screenshotDimension;
private Timer timer;

public void closeConn(){
    try {
        getOutput().close();
        refSock.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public DataOutputStream getOutput(){
    return this.output;
}

@Override
public void run() {
    try{
        BufferedImage image = Screen.getScreenshot(screenshotDimension);
        byte[] a = Conversions.imageToByteArray(image);
        getOutput().writeInt(a.length);
        getOutput().write(a);

    }catch(Exception e){
        e.printStackTrace();
        timer.cancel();
        if(!refSock.isClosed()){
            closeConn();
        }
    }
}
```

Powyższy kod jest fragmentem klasy **`SendScreenTask`**. Metoda `run` odpowiada za przesłanie obrazu, jest wykonywana w cyklicznie w obiekcie klasy **`Timer`**. Najpierw następuje pobranie obrazu ze stacji (metoda ***`getScreenshot`***). Następnie obraz zapisywany jest jako tablica bajtów, za pomocą funkcji ***`imageToByteArray`***. Do przesłania obrazu używamy ***`write`***, jako argument podajemy tablicę bajtów. Wcześniej jednak funkcją ***`writeInt`***, przesłana jest liczba całkowita, będąca długością tej tablicy.

W przypadku wystąpienia wyjątków, działanie obiektu klasy **`Timer`** jest przerywane. Zamykane są również: strumień wyjściowy (**`output`**), oraz gniazdo (**`refSock`**).

Nasłuchiwanie na wybór miniaturki po stronie prowadzącego

```
private Timer timer;
private Socket socket;
private ServerSocket litener;

public void listenForMaximized(){
   new Thread(new Runnable() {

       @Override
       public void run() {
           try{
               socket = litener.accept();
               timer = new Timer();
               SendScreenTask task = new SendScreenTask(socket,timer,1.5);

               timer.schedule(task, 0, 10);
               litener.close();
           }catch (Exception e){}
       }
   }).start();
}
```

Jest to fragment klasy **`ScreenViewClient`**, która jest odpowiedzialna za operacje przesyłania obrazu, wykonywane przez klienta. W funkcji ***`listenForMaximized`***, w oddzielnym wątku aplikacja nasłuchuje na połączenie od prowadzącego, przy pomocy obiektu **`litener`**. Po wywołaniu funkcji ***`accept`***, inicjalizowany jest obiekt klasy **`Timer`**, który będzie wykorzystany do planowania operacji wysyłania obrazu. W kolejnej linii jest wykorzystana klasa **`SendScreenTask`** (jej główna funkcjonalność została opisana w przesyłaniu miniatur), jej obiekt jest odpowiedzialny za przesył obrazu o wysokiej rozdzielczości.

Ważną elementem tego kodu jest wywołanie metody ***`close`*** na obiekcie **`litener`**, gdyż obiekt ten zostanie powtórnie wykorzystany w chwili gdy przesyłanie obrazu o dużej rozdzielczości zostanie przerwane, i znów będzie trzeba nasłuchiwać na kolejne żądanie.

7.1.2 Odbieranie

Tworzenie połączenie z kolejnymi stacjami studentów

```
private static ServerSocket server;
private OnAcceptInterface acceptEvent;
public static boolean isRunning = true;

 public static void closeServer(){
     try{
         server.close();
     }catch (Exception e){}
 }


 @Override
 public void run() {
     while(isRunning) {
         try {
             Socket socket = this.server.accept();
             new ScreenView(socket, this.acceptEvent,
                                    new ImageView()).start();
         } catch (Exception e) {
             closeServer();
         }
     }
 }
```

Powyższy fragment kodu pochodzi z klasy **`ScreenViewServer`**, odpowiadającej za zarządzanie połączeniami oraz operacje odbierania obrazu. Metoda ***`run`*** (wykonywana w oddzielnym wątku) działa dopóki zmienna **`isRunning`** typu **`boolean`** jest równa **`true`** - jest ona zmieniana na **`false`** w momencie wyłączenia aplikacji. Wewnątrz pętli, progam czeka na nowe połączenie od innej stacji za pomocą metody ***`accept`***. W chwili nawiązania połączenia, tworzone jest nowe gniazdo, które następnie przekazywane jest do konstruktora obiektu klasy **`ScreenView`** - wyświetlającego obraz z danej stacji w osobnym wątku.

Wyświetlanie obrazu

```
private Socket socket;
private OnAcceptInterface acceptEvent;
private Object view;
private DataInputStream inputStream;
@Override
public void run() {
     int length;
     try {
          inputStream = IOOperations.initInput(socket);
          length = inputStream.readInt();
          view = acceptEvent.createView(
            Conversions.byteArrayToImage(length, inputStream), socket );

          while ((length=inputStream.readInt())!=-1 && isRunning){
               acceptEvent.onReceive(
                         Conversions.byteArrayToImage(length, inputStream) , view
                  );
          }
          closeSocket();
       }catch (Exception e) {
          closeSocket();
          acceptEvent.deleteView(view);
       }
}
```

Powyższy kod to fragment klasy **`ScreenView`**, dziedziczącej po **`Thread`**. Metoda ***`run`*** inicjalizuje strumień wejściowy do odbierania danych. Następnie, ze strumienia, odczytywana jest wartość całkowitoliczbowa, będą długością danych które zostaną otrzymane. Otrzymana wartość, oraz referencja **`inputStream`** są przekazywane do metody ***`byteArrayToImage`***, odczytującej odpowiednie dane ze strumienia i konwertująca je do obrazu. Obraz ten jest następnie przekazywany do metody ***`createView`***, która wraz z metodami ***`onReceive`*** i ***`deleteView`*** należą do interfejsu **`OnAcceptInterface`**. Każdy kontroler, który chce zarządzać wyświetlaniem obrazów, musi zaimplementować ten interfejs.

Metoda:

- ***`createView`*** tworzy widok z obrazem będącym argumentem,
- ***`onReceive`*** aktualizuje obraz w stworzonym widoku,

- ***deleteView*** usuwa widok podany jako argument.

Ważnym elementem jest obiekt **`view`**. Przechowuje on referencję do widoku w którym znajduje się obraz, co jest przydatne w przypadku gdy mamy stworzonych już wiele widoków. Dzięki temu możemy łatwo zaktualizować wszystkie obrazy równolegle.

8.2. Wykrywanie serwera aplikacji w sieci

Podstawą działania projektu jest moduł wykrywania serwera aplikacji w sieci, za pomocą którego uzyskujemy adres IP stacji prowadzącego na która aplikacja kliencka będzie wysyłała transmisję video, listę uruchomionych procesów oraz otwartych stron. Wykrywanie serwera w sieci odbywa się wg schematu:

- serwer nadaje komunikat, określający jego rolę, na adres multicast,
- komunikat jest wysyłany co 5 sekund,
- przy uruchomieniu aplikacja kliencka nasłuchuje na komunikat.

8.2.1 Rozgłaszanie

Klasa DiscoverServer

Klasa `DiscoverServer` służy do rozgłaszania adresu ip serwera w sieci. Komunikaty rozsyłane są w trybie multicast.

Pola wykorzystywane do utworzenia multicast'u:

```
final static String multicastAddress = "224.0.0.3";
final static int port = 8888;
```

Komunikaty są rozsyłane co 5 sekund przez cały czas działania aplikacji. Transmisja odbywa się z wykorzystaniem protokołu UDP (realizacja przez klasę `DatagramSocket`). Przesyłany datagram zawiera komunikat "`myDISCOVER`" po odebraniu którego klienci aplikacji mogą ustalić adres serwera.

Rozsyłanie komunikatu serwera:

```
InetAddress inetAddress = InetAddress.getByName(multicastAddress);
try (DatagramSocket serverSocket = new DatagramSocket()) {
   String msg = "myDISCOVER";
```

```
    while(!finished) {
        DatagramPacket discoverPacket = new DatagramPacket(msg.getBytes(),
msg.getBytes().length, inetAddress, port);
        serverSocket.send(discoverPacket);
        Thread.sleep(5000);
    }
```

8.2.2. Odbieranie

Klasa DiscoverClient

Klasa `DiscoverClient` służy klientowi do ustalenia adresu ip serwera aplikacji. Jej rola ogranicza się do nasłuchiwania na komunikat serwera oraz zwrócenia jego adresu.

Funkcja zwracająca adres ip serwera (typu `String`)

```
public String getServerAddress() {
    try {
        InetAddress inetAddress = InetAddress.getByName(multicastAddress);

        try (MulticastSocket clientSocket = new MulticastSocket(port)) {
            clientSocket.joinGroup(inetAddress);
            byte[] buf = new byte[10];

            DatagramPacket discoverPacket = new DatagramPacket(buf, buf.length);
            clientSocket.receive(discoverPacket);

            String msg = new String(buf, 0, buf.length);
            if(msg.startsWith("myDISCOVER")) {
                return (discoverPacket.getAddress().toString());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    } catch (UnknownHostException e) {
        e.printStackTrace();
    }

    return("");
}
```

8.3. Uruchomione procesy klienta aplikacji

Moduł pozwala aplikacji klienta na przesłanie danych dotyczących uruchomionych procesów do serwera aplikacji. Działanie modułu można przedstawić wg schematu:

- klienci nasłuchują na żądanie serwera,
- serwer wysyła żądanie (które zawiera adres ip klienta) rozpoczęcia transmisji na adres multicast,

- klienci po odebraniu żądania sprawdzają czy adres zawarty w żądaniu zgadza się z ich własnym,
- jeśli adres się zgadza, klient rozpoczyna transmisję.

8.3.1. Wysyłanie

Klasa ProcessClient

Klasa `ProcessClient` służy do przesyłania informacji o uruchomionych procesach. Konstruktor klasy, jako argumenty, przyjmuje adres ip (typu `String`) oraz port (typu `int`) serwera aplikacji.

Konstruktor:

```
public ProcessClient(String ipAdd, int port) {
    this.ipAdd = ipAdd;
    this.port = port;
}
```

Uruchomione procesy są uzyskiwane za pomocą wykonania komendy wykonywanej w środowisku w którym uruchomiona jest aplikacja.

Fragment kodu przedstawiający uzyskiwanie uruchomionych procesów:

```
String osName = System.getProperty("os.name");

if (osName.startsWith("Windows")) {
    try {
        connectAndSend(Runtime.getRuntime().exec(System.getenv("windir") + "\\system32\\" +
"tasklist.exe"));
    } catch (IOException e) {
        //e.printStackTrace();
    }
} else if (osName.startsWith("Linux")) {
    try {
        connectAndSend(Runtime.getRuntime().exec("ps -e"));
    } catch (IOException e) {
        //e.printStackTrace();
    }
}
```

Po uzyskaniu procesów przekazywane są one jako argument do metody, odpowiadającej za przesłanie ich do serwera aplikacji. Ich przesłanie odbywa się za pomocą klasy `Socket`. Procesy reprezentowane za pomocą klasy `Process` są odczytywane jako strumień i wysyłane linia po linii do serwera.

Metoda przesyłająca procesy:

```
private void connectAndSend(Process p) {
   try {
       System.out.println("[PROCESS CLIENT] Start.");
       Socket client = new Socket(ipAdd, port);
       OutputStream outToServer = client.getOutputStream();
       DataOutputStream output = new DataOutputStream(outToServer);
       BufferedReader reader = new BufferedReader(new
InputStreamReader(p.getInputStream()));
       String line;

       while ((line = reader.readLine()) != null) {
           output.writeUTF(line);
       }

       System.out.println("[PROCESS CLIENT] Data sent.");

       reader.close();
       client.close();
   } catch (Exception e) {
       System.out.println("[PROCESS CLIENT] Unable to connect with server.");
   }
}
```

Klasa ManageListener

Klasa `ManageListener` służy do zarządzania połączeniem, którego zadaniem jest przesyłanie informacji o uruchomionych procesach. Nasłuchuje ona na żądania serwera aplikacji odnośnie rozpoczęcia lub zaprzestania transmisji listy procesów. Komunikacja odbywa sie za pomocą metody multicast, której adres grupy oraz port jest zawarty w polach klasy.

Pola klasy, adres multicast oraz port

```
final static String multicastAddress = "224.0.0.4";
final static int port = 7777;
```

Nasłuchiwanie odbywa się stale i w przypadku otrzymania żądania rozpoczęcia transmisji sprawdzany jest adres zawarty w żądaniu i jeśli odpowiada on adresowi stacji klienta rozpoczyna się przesyłanie informacji o procesach z wykorzystaniem klasy `ProcessClient`.

Nasłuchiwanie na rządania serwera

```
while (!finished) {
   try {
       InetAddress inetAddress = InetAddress.getByName(multicastAddress);

       try (MulticastSocket clientSocket = new MulticastSocket(port)) {
           clientSocket.joinGroup(inetAddress);
           byte[] buf = new byte[21];
```

```
            DatagramPacket discoverPacket = new DatagramPacket(buf, buf.length);
            System.out.println("[MANAGE LISTENER] Listening...");
            clientSocket.receive(discoverPacket);

            String msg = new String(buf, 0, buf.length);
            System.out.println("[MANAGE LISTENER] Message received - " + msg);
            System.out.println("[TEST] SEND:/" +
InetAddress.getLocalHost().getHostAddress());

            if (msg.startsWith("SEND:/" + InetAddress.getLocalHost().getHostAddress()))
{
                System.out.println("Is started");
                Thread.sleep(1);
                new Thread(new ProcessClient(serverAddr, serverPort)).start();
                startReceived = true;
            } else if (startReceived == true && msg.startsWith("STOP:" +
InetAddress.getLocalHost().getHostAddress())) {
                System.out.println("Is stopped");
                isActive = false;
            }
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    } catch (UnknownHostException e) {
        e.printStackTrace();
    }
}
```

8.3.2 Odbieranie

Klasa ProcessServer

Klasa `ProcessServer` służy do odbierania informacji o procesach uruchomionych na stacji klienta. Serwer uruchamiany jest w metodzie `startServer` za pomocą klasy `ServerSocket`. Port (typu `int`), na którym serwer ma nasłuchiwać, jest argumentem konstruktora.

Metoda `startServer`:

```
private void startServer() throws IOException, InterruptedException {
    serverSocket = new ServerSocket(port);
    socket = serverSocket.accept();
}
```

Po uruchomieniu metody `startServer` wykorzystujemy metodę `readData`, która ze strumienia linia po linii dodaje informacje o procesach do listy `data` (elementy typu `String`).

Metoda `readData`:

```
private void readData() {
    try {
        DataInputStream in = new DataInputStream(socket.getInputStream());
        String input = null;
        do {
            input = in.readUTF();
            if(input != null)
                data.add(input);
        } while (input != null);

    } catch (IOException e) {
    } finally {
        try {
            socket.close();
            serverSocket.close();
        } catch (IOException e) {
            System.out.println("[SERVER] Unable to close socket.");
        }
    }
}
```

Klasa ManageServer

Klasa `ManageServer` służy do zarządzania połączeniem, którego zadaniem jest przesyłanie informacji o uruchomionych procesach. Wysyła ona żądanie, za pomocą metody `sendSendMessage`, w którego treści zawiera adres ip klienta, który ma rozpocząć wysyłanie (przykładowa forma żądania - `SEND:172.22.107.103`).

Metoda `sendSendMessage`:

```
public void sendSendMessage(String clientIpAddr) {
    try {
        InetAddress inetAddress = InetAddress.getByName(multicastAddress);
        try (DatagramSocket serverSocket = new DatagramSocket()) {
            String msg = "SEND:" + clientIpAddr;

            DatagramPacket discoverPacket = new DatagramPacket(msg.getBytes(),
msg.getBytes().length, inetAddress, port);
            serverSocket.send(discoverPacket);
        } catch (SocketException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    } catch (UnknownHostException e) {
        e.printStackTrace();
    }
}
```

Żądanie wstrzymania transmisji odbywa się za pomocą metody `sendStopMessage`.

Metoda `sendStopMessage`:

```
public void sendStopMessage(String clientIpAddr) {
    try {
        InetAddress inetAddress = InetAddress.getByName(multicastAddress);
        try (DatagramSocket serverSocket = new DatagramSocket()) {
            String msg = "STOPS:" + clientIpAddr;

            DatagramPacket discoverPacket = new DatagramPacket(msg.getBytes(),
msg.getBytes().length, inetAddress, port);
            serverSocket.send(discoverPacket);
        } catch (SocketException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    } catch (UnknownHostException e) {
        e.printStackTrace();
    }
}
```

8.3.3. Dodawanie procesów do listy oraz ich filtrowanie

Klasa ProcessParse

Klasa `ProcessParse` służy do przepisania przesłanych informacji o procesach. Cześć procesów jest uruchomionych domyślnie oraz część z nich jest sesją konsolową.

Filtrowanie tablicy zawierającej dane procesów oraz dodanie do listy

```
if(words[2].startsWith("Services") &&
!Arrays.asList(defaultProcesses).contains(words[0]))
    list.add(new ProcessModel(words[0], words[2]));
```

Tablica procesów domyślnych

```
private String[] defaultProcesses = {
        "audiodg.exe",
        "conhost.exe",
        "csrss.exe",
        "lsass.exe",
        "lsm.exe",
        "MSCamS64.exe",
        "naPrdMgr.exe",
        "OSPPSVC.EXE",
        "PresentationFontCache.exe",
        "SearchIndexer.exe",
        "services.exe",
        "smss.exe",
        "spoolsv.exe",
        "svchost.exe",
        "svchost.exe",
        "svchost.exe ",
        "System",
```

```
        "UNS.exe",
        "wininit.exe",
        "WmiApSrv.exe",
        "WmiPrvSE.exe",
        "wmpnetwk.exe",
        "WUDFHost.exe",
};
```

8.4. Otwarte strony

8.4.1. Wykrywanie odwiedzonych stron

Wykrywaniem otwartych stron zajmuje się klasa `OpenedPages`, wykorzystująca bibliotekę `JnetPcap`. Wyłapuje ona pakiety z nagłówkiem HTTP i zapisuje nazwy domenowe występujące w tych nagłówkach. W przypadku pakietów HTTPS nazwy domenowe uzyskuje się na podstawie adresów IP (metoda getHostName() klasy InetAddress).

Zapisywanie adresów URL ze złapanych pakietów HTTP

```
PcapPacketHandler<String> jpacketHandler = new PcapPacketHandler<String>() {
   public void nextPacket(PcapPacket packet, String user) {
        Http h = new Http();
        Ip4 ip = new Ip4();

        if(packet.hasHeader(h)){
            String a =  h.fieldValue((Http.Request.Host.Host));
            if(a!= null) {
                Pattern p = Pattern.compile("([a-zA-Z]+\\.)?[\\w]+\\.[a-zA-Z]+");
                Matcher m = p.matcher(a);
                boolean b = m.matches();

                if (!a.equals(prev) && b == true) {
                    stream.append(a + "\n");
                    prev = a;
                } else {//
                    if (packet.hasHeader(ip)) {
                        String destinationIP =
org.jnetpcap.packet.format.FormatUtils.ip(packet.getHeader(ip).destination());
                        InetAddress addr = null;
                        try {
                            addr = InetAddress.getByName(destinationIP);
                        } catch (UnknownHostException e) {
                            e.printStackTrace();
                        }
                        String host = addr.getHostName();

                        if(!host.equals(prevs) ) {
                            stream.append(a + "\n");
                            prevs = host;
                        }

                    }
                }
            }
```

```
        }
    }
};
```

8.4.2. Wysyłanie

Klasa `OpenedPagesClient` odpowiada za nawiązywanie i kończenie połączenia z serwerem, oraz przesyłanie danych do serwera.

Klasa OpenedPagesClient

```
public class OpenedPagesClient{
   private Socket socket;
   private String name;
   private int portNmb;
   public OpenedPagesClient(String name, int portNmb)
   {
       this.name = name;
       this.portNmb = portNmb;
   }

   public PrintWriter GetOpenedPagesStream() throws IOException {
       return new PrintWriter(socket.getOutputStream(), true);
   }


   public void Connect() throws IOException {
       System.out.println("<OP_Client> Connecting to server...");
       socket = new Socket(name, portNmb);
       System.out.println("<OP_Client> Connected!");

   }
   public void CloseConnetion()
   {
       try {
           socket.close();
       } catch (IOException e) {
           e.printStackTrace();
       }
   }


}
```

Klasa ClientPagesManager zarządza modułem otwartych stron po stronie klienta. Działa według następującego algorytmu:

1. Rozpocznij łapanie pakietów
2. Czekaj 5 sekund
3. Jeżeli odwiedzono nowe strony

a. Usuń poprzednio wysłane strony, jeżeli istnieją
b. Nawiąż połączenie z serwerem
c. Prześlij nowe odwiedzone strony
d. Zamknij połączenie z serwerem

4. Jeżeli nie odwiedzono nowych stron, wróć do punktu 2.

Klasa ClientPagesManager

```
public class ClientPagesManager {

   public static void StartClient(String serverIp)
   {

       new Thread(new Runnable() {
           @Override
           public void run() {
               StringBuilder captured_pages = new StringBuilder("");
               OpenedPages opened_pages = new OpenedPages(captured_pages);
               int captured_pages_last_length = 0;
               opened_pages.StartCapturing();

               while (true)
               {
                   try {
                       OpenedPagesClient client = new OpenedPagesClient(serverIp,
11938);
                       while (true) {
                           Thread.sleep(5000);
                           if(captured_pages.length() > captured_pages_last_length) {
                               for(int i = 0; i < captured_pages_last_length; i++)
                               {
                                   captured_pages.deleteCharAt(0);
                               }
                               client.Connect();
                               PrintWriter out = client.GetOpenedPagesStream();
                               String pagesToSent = captured_pages.toString();
                               out.printf(pagesToSent);
                               /*test:*/System.out.printf("<OP_client> Send pages: \n"
+ pagesToSent);
                               client.CloseConnetion();
                               captured_pages_last_length = captured_pages.length();
                           }
                       }
                   } catch (IOException e) {
                       e.printStackTrace();
                   } catch (InterruptedException e) {
                       e.printStackTrace();
                   }
               }
           }
       }).start();

   }
}
```

8.4.3. Odbieranie

Klasa `OpenedPagesServer` zawiera metody do łączenia z klientami, zamykania połączenia, oraz odbierania danych.

Klasa OpenedPagesServer

```
public class OpenedPagesServer {
   private ServerSocket serverSocket;
   private Socket clientSocket;

   public OpenedPagesServer(int portNmb)
   {
       try {
           serverSocket = new ServerSocket(portNmb);
           clientSocket = serverSocket.accept();
           System.out.println("Connected!");

       } catch (IOException e) {
           e.printStackTrace();
       }
   }
   public OpenedPagesServer(){}

   public String GetClientIp()
   {
       InetAddress ip = clientSocket.getInetAddress();
       return ip.getHostAddress();
   }

   public BufferedReader GetOpenedPagesStream () throws IOException {

       BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
       return in;

   }

   public void Connect(int portNmb)
   {
       System.out.println("<OP_Server> Connecting to client...");
       try {
           serverSocket = new ServerSocket(portNmb);
           clientSocket = serverSocket.accept();
           System.out.println("<OP_Server> Server connected to " +
clientSocket.getInetAddress().getHostAddress());

       } catch (IOException e) {
           e.printStackTrace();
       }
   }

   public void StopServer()
   {
       try {
           clientSocket.close();
```

```
            serverSocket.close();
            System.out.println("<OP_server> Server disconnected");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Klasa ServerPagesManager zarządza modułem otwartych stron po stronie serwera. Działa według następującego algorytmu:

1. Czekaj na połączenie od klienta
2. Odbierz i zapisz dane od klienta
3. Zamknij połączenie
4. Wróć do punktu 1.

KlasaServerPagesManager

```
public class ServerPagesManager {
   public static ArrayList<ClientPages> clientPagesList = new ArrayList<>();

   public static ArrayList<String> GetPages(String clientIp)
   {
       for (ClientPages cp : clientPagesList) {
           if(cp.getClientIp().equals(clientIp))
               return cp.getPages();
       }
       return new ArrayList<String>();
   }
   public static int getListLength()
   {
       return clientPagesList.size();
   }
   public static void AddPage(String clientIp, String page)
   {
       boolean found = false;
       for (ClientPages cp : clientPagesList) {
           if(cp.getClientIp().equals(clientIp) ) {
               cp.AddPage(page);
               found = true;
           }
       }
       if(found == false){
           clientPagesList.add(new ClientPages(clientIp, page));
       }
   }

   public static void StartServer()
   {
       new Thread(new Runnable() {
           @Override
```

```
            public void run() {
                while (true)
                {
                    try {
                        OpenedPagesServer server = new OpenedPagesServer();
                        while (true) {
                            server.Connect(19938);
                            BufferedReader in = server.GetOpenedPagesStream();
                            String inputLine;
                            while ((inputLine = in.readLine()) != null) {
                                AddPage(server.GetClientIp(), inputLine);
                                System.out.println("Server; Received line: " +
inputLine);
                            }
                            server.StopServer();
                        }
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }
        }).start();

    }
}
```