

Projekt

Systemy odporne na błędy

Wydział Elektrotechniki Automatyki i Informatyki

Politechnika Świętokrzyska

Studia: **Niestacjonarne II stopnia**

Kierunek: **Informatyka**

Grupa: **1IZ21**

Wykonanie: **Bartosz Rokita, Adam Markowski**

Temat projektu:

Paxos głosowanie

Spis treści

1.	CEL PROJEKTU	3
2.	TECHNOLOGIE.....	BŁĄD! NIE ZDEFINIOWANO ZAKŁADKI.
3.	ALGORYTMY.....	BŁĄD! NIE ZDEFINIOWANO ZAKŁADKI.
4.	DIAGRAMY KLAS.....	BŁĄD! NIE ZDEFINIOWANO ZAKŁADKI.
5.	DIAGRAMY PRZYPADKÓW UŻYCIA.....	BŁĄD! NIE ZDEFINIOWANO ZAKŁADKI.
6.	DZIAŁANIE APLIKACJI.....	7
7.	WNIOSKI.....	BŁĄD! NIE ZDEFINIOWANO ZAKŁADKI.

1. Cel projektu

Celem projektu było zaimplementowanie protokołu paxos na którym zostało oparte głosowanie na wartością liczbową i symulacja awarii części systemu.

2. Paxos

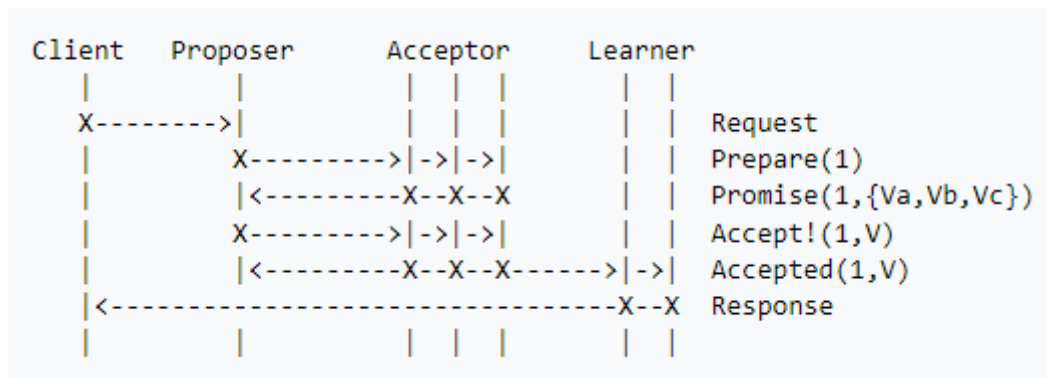
Paxos to protokół rozwiązywania konsensusu w systemach rozproszonych. Pozwala na uzgodnienie(konsensus) jednego wyniku wśród wielu uczestników. Takie uzgodnienie staje się problemem, gdy uczestnicy lub komunikacja między nimi ulega awarii.

W tym protokole można wyróżnić kilka ról:

- Klient – wysyła żądanie do systemu.
- Akceptor – odbiera komunikaty od wnioskodawcy i zwraca wynik.
- Proposer – odbiera od klienta żądanie i koordynuje cały proces ustalania wyniku z akceptantami.

Pełna komunikacja w protokole odbywa się w następujący sposób:

1. Proposer wysyła wiadomość „Prepare” zawierającą numer identyfikacyjny do kworum akceptorów.
2. Akceptorzy odbierają wiadomość od Proposera. Po odebraniu wiadomości porównuje jej numer identyfikacyjny z zapisanym w pamięci. Jeśli nadesłany numer jest większy od zapisanego to wysyła do Proposera wiadomość „Promise” informującą o tym, że zignoruje przyszłe wiadomości, jeśli będą miały mniejszy numer od przesłanego. Dodatkowo, jeśli zaakceptował już jakąś poprzednią wiadomość to odsyła ją w raz z wiadomością „Promise”.
3. Proposer oczekuje na wiadomości „Promise” z kworum akceptorów. Jeśli odebrał wiadomości od akceptorów to rozsyła do kworum wiadomość „Accept” z numerem identyfikacyjnym i wartością, która ustalana jest w następujący sposób:
 - Jeśli część akceptorów już wcześniej zaakceptowała jakąś wiadomość to wartość jest ustalana na podstawie przesłanych.
 - Jeśli żaden z akceptorów nie zaakceptował wcześniej wiadomości to Proposer zostaje przy swojej wartości.
4. Akceptor odbiera wiadomość „Accept” od Proposera jeśli obiecał mu to wcześniej i wysyła do niego wiadomość „Accepted”.



Rysunek 1 – graficzna prezentacja komunikacji protokołu paxos

3. Symulacja awarii w protokole Paxos

W systemie zostały zaimplementowane 3 symulacje awarii w protokole Paxos. Wszystkie błędy zostały opisane poniżej.

Szalony akceptor

Symuluje błędne działanie akceptora. Awaria polega na zwracaniu losowego numeru sekwencyjnego wiadomości i nieprzyjmowaniu wiadomości.

port	Lider?	Aktualne ID	Ustalona wartość	Wyłączony?	Szalony?	Problemy z połączeniem?
4440	true	2	2	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4441	false	7941440918840103471	1	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
4442	false	2	2	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4443	false	2	2	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4444	false	2	2	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4445	false	2	2	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4446	false	2	2	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4447	false	2	2	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Rysunek 2 – losowy numer sekwencyjny

Wyłączony akceptor

Symuluje błędne działanie akceptora. Awaria polega na braku połączenia akceptora do systemu. Nie dostaje żadnych wiadomości przez co nie bierze udziału w głosowaniu.

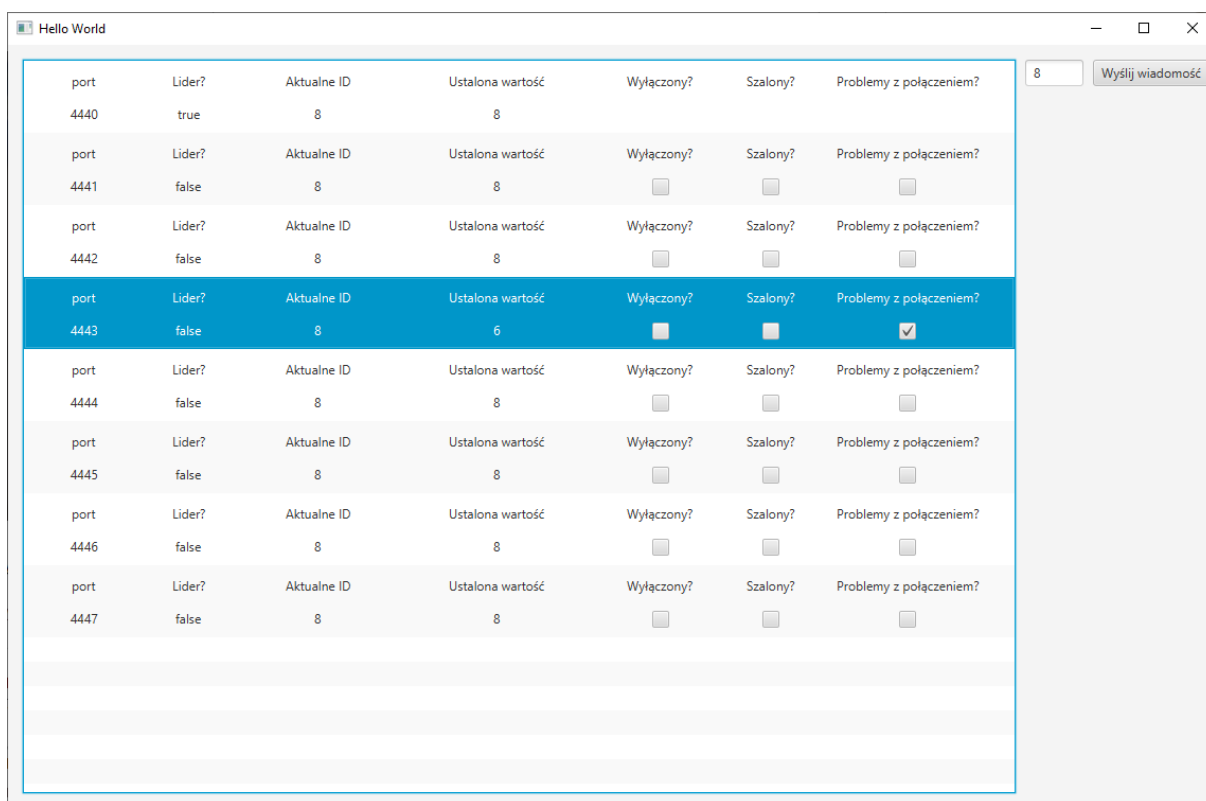


port	Lider?	Aktualne ID	Ustalona wartość	Wyłączony?	Szalony?	Problemy z połączeniem?
4440	true	3	3			
4441	false	3	3	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4442	false	2	2	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4443	false	3	3	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4444	false	3	3	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4445	false	3	3	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4446	false	3	3	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4447	false	3	3	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Rysunek 3 – Wyłączony akceptor

Problem z połączeniem akceptora

Symuluje błędne działanie akceptora. Awaria polega na utracie wiadomości. Gdy symulacja jest włączona akceptor ma szanse 50% na utratę przysłanej wiadomości.



The screenshot shows a window titled "Hello World" with a table of connection data and a message input field. The table has 7 columns: "port", "Lider?", "Aktualne ID", "Ustalona wartość", "Wyłączony?", "Szalony?", and "Problemy z połączeniem?". The rows represent different ports (4440 to 4447). The "Wyłączony?" and "Szalony?" columns contain checkboxes. The "Problemy z połączeniem?" column contains checkboxes, with the row for port 4443 having a checked checkbox. To the right of the table is a text input field with the value "8" and a button labeled "Wyślij wiadomość".

port	Lider?	Aktualne ID	Ustalona wartość	Wyłączony?	Szalony?	Problemy z połączeniem?
4440	true	8	8			
4441	false	8	8	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4442	false	8	8	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4443	false	8	6	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
4444	false	8	8	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4445	false	8	8	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4446	false	8	8	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4447	false	8	8	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Rysunek 4 – Problem z połączeniem akceptora

4. Działanie aplikacji

W tym rozdziale zostaną przedstawione fragmenty kodu odpowiedzialne za działanie systemu.

Serwer

Instancje klasy *Server* odwzorowują fizyczne serwery połączone w sieci. Każdy z serwerów posiada swój port oraz mechanizmy do wysyłania i odbierania pakietów. Wewnątrz obiektu zapisywany jest stan awarii, aktualny numer sekwencyjny oraz ustalona wartość. *Server* steruje wysyłaniem wiadomości i decyduje co zrobić z wiadomościami przychodzącymi.

```
public class Server {  
  
    private Connection connection;  
    private List<Server> pools;  
    private boolean leader;  
    private Long currentId = Long.valueOf(0);  
    private Long backupCurrentId;  
    private Long currentValue;  
    private boolean isShutdown = false;  
    private boolean isCrazyAcceptor = false;  
    private boolean hasConnectionProblem = false;  
    private Timer timer;  
    private HashMap<Long, List<Long>> receivedPromiseValues = new HashMap<>();  
  
    public Server(int port, List<Server> pools) throws SocketException {  
        this(port, pools, leader: false);  
    }  
  
    public Server(int port, List<Server> pools, boolean leader) throws SocketException {  
        this.leader = leader;  
        this.pools = pools;  
        connection = new Connection(port, new MultiDispatcher(server: this));  
        this.pools.add(this);  
    }  
  
    public void sendPrepareMessage(Long currentValue) {  
        if(this.isShutdown()) {  
            return;  
        }  
  
        if(!this.leader){  
            return;  
        }  
  
        this.currentValue = currentValue;  
        this.currentId += 1;  
        for (Server server: pools) {  
            if (server.equals(this)) {  
                continue;  
            }  
  
            this.receivedPromiseValues.put(this.currentId, new ArrayList<>());  
            this.connection.sendTo(  
                server.getConnection(),  
                MessageSerializer.serialize(new Prepare(this.currentId))  
            );  
        }  
    }  
}
```

Rysunek 5 – Część klasy *Server*

Odbieranie i wysyłanie wiadomości

Każda wiadomość zaimplementowana jest jako osobna klasa rozszerzona o interfejs *Message*. Wysyłanie wiadomości odbywa się poprzez klasę *Connection*, która posiada w sobie obsługę UDP.

```
public class Promise implements Message, Serializable {

    private Long id;
    private Long AcceptedId;
    private Long acceptedValue;

    public Promise(Long id) { this.id = id; }

    public Promise(Long id, Long acceptedId, Long acceptedValue) {
        this.id = id;
        AcceptedId = acceptedId;
        this.acceptedValue = acceptedValue;
    }

    public Long getId() { return id; }

    public Long getAcceptedId() { return AcceptedId; }

    public Long getAcceptedValue() { return acceptedValue; }

    public boolean hasAcceptedValue() { return acceptedValue != null; }
}
```

Rysunek 6 – Klasa wiadomości Promise

```
public class Connection {
    static int BUFF_SIZE = 256;
    private DatagramSocket socket;

    public Connection(int port, Dispatcher dispatcher) throws SocketException {
        socket = new DatagramSocket(port);
        socket.setReuseAddress(true);
        LinkedBlockingQueue<byte[]> receiveQueue = new LinkedBlockingQueue<>();
        PacketReceiver pr = new PacketReceiver(socket, receiveQueue);
        pr.setDaemon(true);
        pr.start();
        PacketDispatcher pd = new PacketDispatcher(socket, receiveQueue, dispatcher);
        pd.setDaemon(true);
        pd.start();
    }

    public void sendTo(Connection connection, byte[] message) {
        try {
            DatagramPacket packet = new DatagramPacket(message, message.length);
            packet.setAddress(InetAddress.getByName("localhost"));
            packet.setPort(connection.socket.getLocalPort());
            synchronized (this) {
                socket.send(packet);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public int getPort() { return socket.getLocalPort(); }
}
```

Rysunek 7 – Klasa *Connection*

Każdy pakiet, który trafia do nadawcy odbierany jest przez *PacketReceiver* w celu wrzucenia go na kolejkę, która obsługiwana jest przez klasę *PacketDispatcher*. To w niej następuje deserializacja wiadomości poprzez dispatchery i przekazanie jej do klasy *Server*.

```
public class PacketReceiver extends Thread {

    private DatagramSocket socket;
    private BlockingQueue<byte[]> queue;
    private DatagramPacket receivePacket;

    public PacketReceiver(DatagramSocket socket, BlockingQueue<byte[]> queue) {
        this.socket = socket;
        this.queue = queue;
        receivePacket = new DatagramPacket(new byte[Connection.BUFF_SIZE], Connection.BUFF_SIZE);
    }

    @Override
    public void run() {
        while (true) {
            try {
                socket.receive(receivePacket);
                if (receivePacket.getLength() > Connection.BUFF_SIZE) {
                    throw new IOException("message too big " + receivePacket.getLength());
                }
                queue.put(receivePacket.getData().clone());
            } catch (IOException e) {
                e.printStackTrace();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Rysunek 8 – Klasa *PacketReceiver*

```

public class PacketDispatcher extends Thread {

    private DatagramSocket socket;
    private BlockingQueue<byte[]> queue;
    private Dispatcher dispatcher;

    public PacketDispatcher(
        DatagramSocket socket,
        BlockingQueue<byte[]> queue,
        Dispatcher dispatcher
    ) {
        this.socket = socket;
        this.queue = queue;
        this.dispatcher = dispatcher;
    }

    @Override
    public void run() {
        while (true) {
            try {
                byte[] message = queue.take();
                dispatcher.dispatch(message);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

Rysunek 9 – Klasa *PacketDispatcher*

```

public class PaxosDispatcher implements Dispatcher {

    private Server server;

    public PaxosDispatcher(Server server) { this.server = server; }

    @Override
    public void dispatch(byte[] messageData) {
        if(server.hasConnectionProblem() && new Random().nextInt()%2 == 0) {
            return;
        }
        Object messageObject = MessageSerializator.deserialize(messageData);
        if(messageObject instanceof Message) {
            Message message = (Message) messageObject;
            if(message instanceof Prepare) {
                server.trySendPromiseMessage(((Prepare) message).getId());
            } else if (message instanceof Promise) {
                if(((Promise) message).hasAcceptedValue()) {
                    server.trySendAcceptMessage(
                        ((Promise) message).getId(),
                        ((Promise) message).getAcceptedValue()
                    );
                } else {
                    server.trySendAcceptMessage(((Promise) message).getId());
                }
            } else if (message instanceof Accept) {
                server.trySendAcceptedMessage(
                    ((Accept) message).getId(),
                    ((Accept) message).getValue()
                );
            }
        }
    }
}

```

Rysunek 10 – Klasa *PaxosDispatcher*

Logika każdego z etapów jest rozbita na osobną metodę w klasie *Server*.

```
public void trySendPromiseMessage(Long sentId) {
    if(this.isShutdown()) {
        return;
    }
    if(this.currentId == null || sentId > this.currentId) {
        for (Server server: this.pools) {
            if (server.equals(this) || !server.isLeader()) {
                continue;
            }
            if(this.currentValue != null) {
                this.connection.sendTo(
                    server.getConnection(),
                    MessageSerializator.serialize(new Promise(sentId, this.currentId, this.currentValue))
                );
            } else {
                this.connection.sendTo(
                    server.getConnection(),
                    MessageSerializator.serialize(new Promise(sentId))
                );
            }
        }
        this.currentId = sentId;
    }
}

public void trySendAcceptMessage(Long sentId) {
    if(this.isShutdown()) {
        return;
    }
    if(!this.leader || !sentId.equals(this.currentId)){
        return;
    }
    if(this.receivedPromiseValues.containsKey(sentId)) {
        List<Long> raValues = this.receivedPromiseValues.get(sentId);
        raValues.add(null);
    } else {
        List<Long> raValues = new ArrayList<>();
        raValues.add(null);
        this.receivedPromiseValues.put(sentId, raValues);
    }
    sendAcceptMessage(sentId);
}

public void trySendAcceptedMessage(Long sentId, Long sentValue) {
    if(this.isShutdown()) {
        return;
    }
    if(sentId.equals(this.currentId)) {
        for (Server server : this.pools) {
            if (server.equals(this) || !server.isLeader()) {
                continue;
            }
            this.currentValue = sentValue;
            this.connection.sendTo(
                server.getConnection(),
                MessageSerializator.serialize(new Accepted(sentId, sentValue))
            );
        }
    }
}
```

Rysunek 11 – Metody do wysyłania wiadomości Promise, Accept i Accepted

Interfejs

Interfejs został stworzony przy użyciu *SceneBuilder*. W okienku aplikacji znajduje się lista serwerów z informacjami o ich aktualnym stanie oraz checkboxy, które informują o tym czy dana symulacja awarii została aktywowana. Obok listy znajduje się pole z przyciskiem, którym wysyłana jest wartość do Proposera.

The screenshot shows a window titled "Hello World" with a table of server status and a button to send a message. The table has 7 columns: port, Lider?, Aktualne ID, Ustalona wartość, Wyłączony?, Szalony?, and Problemy z połączeniem?. The data is as follows:

port	Lider?	Aktualne ID	Ustalona wartość	Wyłączony?	Szalony?	Problemy z połączeniem?
4440	true	0	null			
4441	false	0	null	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4442	false	0	null	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4443	false	0	null	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4444	false	0	null	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4445	false	0	null	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4446	false	0	null	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4447	false	0	null	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Below the table, there are three empty rows. To the right of the table, there is a text input field and a button labeled "Wyślij wiadomość".

Rysunek 12 – Interfejs

Obsługę pola tekstowego i przycisku „Wyślij wiadomość” zapewnia klasa *Controller*. Znajduje się w niej również konfiguracja i tworzenie serwerów.

```
public class Controller implements Initializable {

    @FXML
    private TextField votingValueField;

    @FXML
    private ListView<Server> serverList;

    ObservableList<Server> serversData;

    private Servers servers;

    private List<Server> connections = new ArrayList<>();

    public Controller() {
        try {
            this.servers = new Servers(
                new Server( port: 4440, connections, leader: true),
                new Server( port: 4441, connections),
                new Server( port: 4442, connections),
                new Server( port: 4443, connections),
                new Server( port: 4444, connections),
                new Server( port: 4445, connections),
                new Server( port: 4446, connections),
                new Server( port: 4447, connections)
            );
        } catch (SocketException e) {
            e.printStackTrace();
        }
        serversData = FXCollections.observableArrayList(this.connections);
    }

    public void sendToVoting(MouseEvent mouseEvent) {
        servers.sendPrepareMessage(Long.valueOf(votingValueField.getText()));
    }

    @Override
    public void initialize(URL url, ResourceBundle resourceBundle) {
        serverList.setItems(serversData);
        serverList.setCellFactory(serverListView -> new ServerListViewCell());
        new Timer().schedule(() -> { serverList.refresh(); }, delay: 2000, period: 2000);
    }
}
```

Rysunek 12 – Klasa *Controller*