U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Distributed Systems Project 1: Distributed Backup System

Pedro Hugo Noevo up201604725
Pedro Carmo Pereira up201708807
T4G07

Introduction

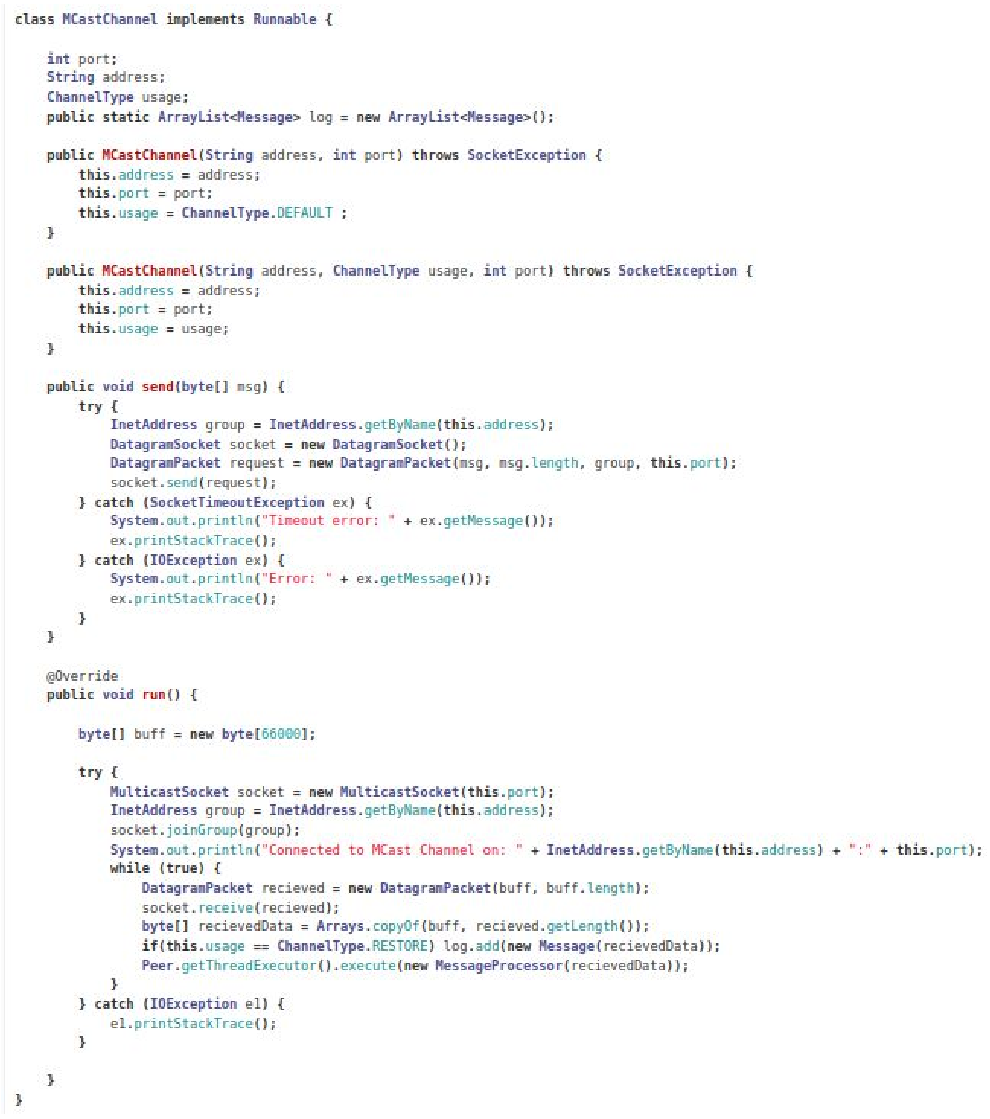
This report aims to shed light on the implementation of concurrency for the Distributed Systems Project 1: Distributed Backup Service. The level of concurrency achieved supported the processing of different messages received on the same channel simultaneously.

Implementation

As previously stated, our design allows for the processing of multiple messages received at the same time, in the same or different channels. This is possible due to the receiver thread creating a new thread for the processing of each received message.

The receiver threads (one for each channel) are defined in the MCastChannel class, and used in the Peer class, like so:

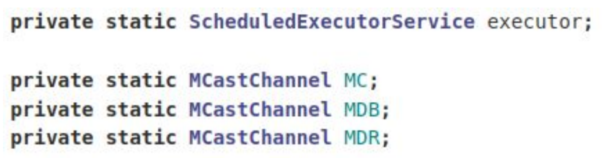
- MCastChannel class:



```java
class MCastChannel implements Runnable {

    int port;
    String address;
    ChannelType usage;
    public static ArrayList<Message> log = new ArrayList<Message>();

    public MCastChannel(String address, int port) throws SocketException {
        this.address = address;
        this.port = port;
        this.usage = ChannelType.DEFAULT ;
    }

    public MCastChannel(String address, ChannelType usage, int port) throws SocketException {
        this.address = address;
        this.port = port;
        this.usage = usage;
    }

    public void send(byte[] msg) {
        try {
            InetAddress group = InetAddress.getByName(this.address);
            DatagramSocket socket = new DatagramSocket();
            DatagramPacket request = new DatagramPacket(msg, msg.length, group, this.port);
            socket.send(request);
        } catch (SocketTimeoutException ex) {
            System.out.println("Timeout error: " + ex.getMessage());
            ex.printStackTrace();
        } catch (IOException ex) {
            System.out.println("Error: " + ex.getMessage());
            ex.printStackTrace();
        }
    }

    @Override
    public void run() {

        byte[] buff = new byte[66000];

        try {
            MulticastSocket socket = new MulticastSocket(this.port);
            InetAddress group = InetAddress.getByName(this.address);
            socket.joinGroup(group);
            System.out.println("Connected to MCast Channel on: " + InetAddress.getByName(this.address) + ":" + this.port);
            while (true) {
                DatagramPacket recieved = new DatagramPacket(buff, buff.length);
                socket.receive(recieved);
                byte[] recievedData = Arrays.copyOf(buff, recieved.getLength());
                if(this.usage == ChannelType.RESTORE) log.add(new Message(recievedData));
                Peer.getThreadExecutor().execute(new MessageProcessor(recievedData));
            }
        } catch (IOException el) {
            el.printStackTrace();
        }

    }
}
```

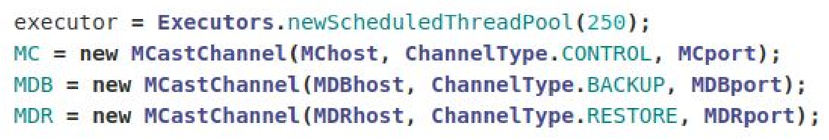
- Usage in the Peer class:

  Definition



```
private static ScheduledExecutorService executor;

private static MCastChannel MC;
private static MCastChannel MDB;
private static MCastChannel MDR;
```

  Instantiation



```
executor = Executors.newScheduledThreadPool(250);
MC = new MCastChannel(MChost, ChannelType.CONTROL, MCport);
MDB = new MCastChannel(MDBhost, ChannelType.BACKUP, MDBport);
MDR = new MCastChannel(MDRhost, ChannelType.RESTORE, MDRport);
```

Whenever a message is received, the receiver thread initiates a new thread to process the said message, the MessageProcessor class handles the processing of messages asynchronously.

Reasoning

We agreed on this design even before seeing the provided concurrency hints, because we felt it was the most intuitive way for the program to work. If this project was to be implemented and used as described it would only make sense for the application to be able to process simultaneous requests since the system is distributed and we can not control when the application is called.