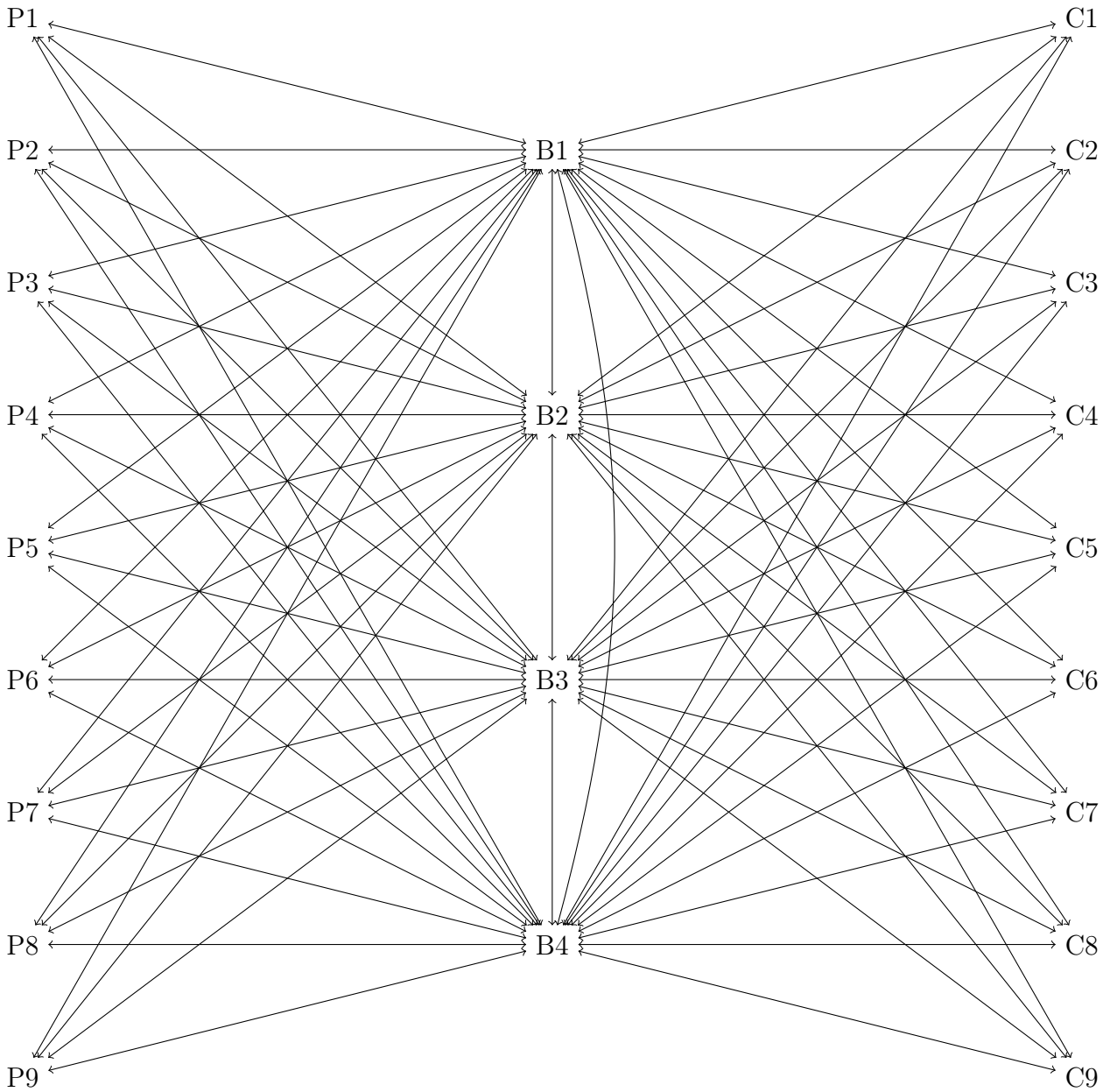
Rozproszony bufor w JCSP

09.12.2025

1 Schemat połączeń



2 Implementacja

2.1 Bufor

```
package org.example.buffer;
import org.jcsp.lang.*;

import java.util.Deque;
import java.util.ArrayDeque;

import org.example.utilities.*;

public class QueueBuffer implements CSProcess {
    final private Stats stats;
    final private One2OneChannelInt[] fromProducerChannels;
    final private One2OneChannelInt[] toProducerChannels;
    final private One2OneChannelInt[] fromConsumerChannels;
    final private One2OneChannelInt[] toConsumerChannels;
    final private One2OneChannel<PassPayload> passProducerChannel;
    final private One2OneChannel<PassPayload>
       receivePassedProducerChannel;
    final private One2OneChannel<PassPayload> passConsumerChannel;
    final private One2OneChannel<PassPayload>
       receivePassedConsumerChannel;
    final private int bufferSize;
    final private int Id;
    private int current;

    public QueueBuffer (One2OneChannelInt[] fromProducerChannels,
       One2OneChannelInt[] toProducerChannels,
                        One2OneChannelInt[] fromConsumerChannels
                           , One2OneChannelInt[]
                           toConsumerChannels,
                        One2OneChannel<PassPayload>
                           passProducerChannel, One2OneChannel<
                           PassPayload>
                           receivePassedProducerChannel,
                        One2OneChannel<PassPayload>
                           passConsumerChannel, One2OneChannel<
                           PassPayload>
                           receivePassedConsumerChannel,
                        int bufferSize, Stats stats, int Id) {
        this.stats = stats;
        this.fromProducerChannels = fromProducerChannels;
        this.toProducerChannels = toProducerChannels;
        this.fromConsumerChannels = fromConsumerChannels;
        this.toConsumerChannels = toConsumerChannels;
        this.passProducerChannel = passProducerChannel;
```

```
        this.receivePassedProducerChannel =
           receivePassedProducerChannel;
        this.passConsumerChannel = passConsumerChannel;
        this.receivePassedConsumerChannel =
           receivePassedConsumerChannel;
        this.bufferSize = bufferSize;
        this.Id = Id;
        this.current = bufferSize / 2;
    }

    public void run () {
        Guard[] guards = new Guard[fromProducerChannels.length +
           fromConsumerChannels.length + 2];
        for (int i = 0; i < fromProducerChannels.length; i++) {
            guards[i] = fromProducerChannels[i].in();
        }
        for (int i = 0; i < fromConsumerChannels.length; i++) {
            guards[fromProducerChannels.length + i] =
               fromConsumerChannels[i].in();
        }
        guards[guards.length - 2] = receivePassedProducerChannel.
           in();
        guards[guards.length - 1] = receivePassedConsumerChannel.
           in();

        Alternative alt = new Alternative(guards);

        Deque<Integer> producerQueue = new ArrayDeque<>();
        Deque<Integer> consumerQueue = new ArrayDeque<>();

        while (true) {
            int index = alt.select();
            if (index < fromProducerChannels.length) {
                int producerIndex = index;
                if (fromProducerChannels[producerIndex].in().read
                   () == Payload.WHERE.ordinal()) {
                    if (current < bufferSize) {
                        toProducerChannels[producerIndex].out().
                           write(Payload.HERE.ordinal());
                        if (fromProducerChannels[producerIndex].in
                           ().read() == Payload.PACKAGE.ordinal())
                            {
                            current++;
                            stats.recordProduced(producerIndex);
                            if (!consumerQueue.isEmpty()) {
                                int queuedConsumerIndex =
                                   consumerQueue.removeFirst();
                                toConsumerChannels[
                                   queuedConsumerIndex].out().
```

```
                            write(Payload.HERE.ordinal());
                        toConsumerChannels[
                            queuedConsumerIndex].out().
                            write(Payload.PACKAGE.ordinal()
                            );
                        current--;
                        stats.recordConsumed(
                            queuedConsumerIndex);
                    }
                }
            } else {
                passProducerChannel.out().write(new
                    PassPayload(Id, producerIndex));
                stats.recordProducerPass(producerIndex);
            }
        }

    } else if (index < fromProducerChannels.length +
        fromConsumerChannels.length) {
        int consumerIndex = index - fromProducerChannels.
            length;
        if (fromConsumerChannels[consumerIndex].in().read
            () == Payload.WHERE.ordinal()) {
            if (current > 0) {
                toConsumerChannels[consumerIndex].out().
                    write(Payload.HERE.ordinal());
                toConsumerChannels[consumerIndex].out().
                    write(Payload.PACKAGE.ordinal());
                current--;
                stats.recordConsumed(consumerIndex);
                if (!producerQueue.isEmpty()) {
                    int queuedProducerIndex =
                        producerQueue.removeFirst();
                    toProducerChannels[queuedProducerIndex
                        ].out().write(Payload.HERE.ordinal
                        ());
                    if (fromProducerChannels[
                        queuedProducerIndex].in().read() ==
                            Payload.PACKAGE.ordinal()) {
                        current++;
                        stats.recordProduced(
                            queuedProducerIndex);
                    }
                }
            } else {
                passConsumerChannel.out().write(new
                    PassPayload(Id, consumerIndex));
                stats.recordConsumerPass(consumerIndex);
            }
```

```
        }

    } else if (index == guards.length - 2) {
        PassPayload passPayload =
           receivePassedProducerChannel.in().read();
        int producerIndex = passPayload.pcIndex();
        int ogBufferIndex = passPayload.ogBufferIndex();
        if (current < bufferSize) {
            toProducerChannels[producerIndex].out().write(
               Payload.HERE.ordinal());
            if (fromProducerChannels[producerIndex].in().
               read() == Payload.PACKAGE.ordinal()) {
                current++;
                stats.recordProduced(producerIndex);
                if (!consumerQueue.isEmpty()) {
                    int queuedConsumerIndex =
                       consumerQueue.removeFirst();
                    toConsumerChannels[queuedConsumerIndex
                       ].out().write(Payload.HERE.ordinal
                       ());
                    toConsumerChannels[queuedConsumerIndex
                       ].out().write(Payload.PACKAGE.
                       ordinal());
                    current--;
                    stats.recordConsumed(
                       queuedConsumerIndex);
                }
            }
        } else if (ogBufferIndex != Id) {
            passProducerChannel.out().write(passPayload);
            stats.recordProducerPass(producerIndex);
        } else {
            producerQueue.addLast(producerIndex);
        }

    } else if (index == guards.length - 1) {
        PassPayload passPayload =
           receivePassedConsumerChannel.in().read();
        int consumerIndex = passPayload.pcIndex();
        int ogBufferIndex = passPayload.ogBufferIndex();
        if (current > 0) {
            toConsumerChannels[consumerIndex].out().write(
               Payload.HERE.ordinal());
            toConsumerChannels[consumerIndex].out().write(
               Payload.PACKAGE.ordinal());
            current--;
            stats.recordConsumed(consumerIndex);
            if (!producerQueue.isEmpty()) {
```

```
                        int queuedProducerIndex = producerQueue.
                            removeFirst();
                        toProducerChannels[queuedProducerIndex].
                            out().write(Payload.HERE.ordinal());
                        if (fromProducerChannels[
                            queuedProducerIndex].in().read() ==
                            Payload.PACKAGE.ordinal()) {
                            current++;
                            stats.recordProduced(
                                queuedProducerIndex);
                        }
                    }
                } else  if (ogBufferIndex != Id) {
                    passConsumerChannel.out().write(passPayload);
                    stats.recordConsumerPass(consumerIndex);
                } else {
                    consumerQueue.addLast(consumerIndex);
                }
            }
        }
    }

}
```

2.2 Producent

```
package org.example.producer;
import org.jcsp.lang.*;
import java.util.Random;

import org.example.utilities.Payload;

public class QueueProducer implements CSProcess {
    final private One2OneChannelInt[] toBufferChannels;
    final private One2OneChannelInt[] fromBufferChannels;
    final private int Id;

    public QueueProducer (One2OneChannelInt[] toBufferChannels,
       One2OneChannelInt[] fromBufferChannels, int Id) {
        this.toBufferChannels = toBufferChannels;
        this.fromBufferChannels = fromBufferChannels;
        this.Id = Id;
    }

    public void run () {
        Guard[] guards = new Guard[fromBufferChannels.length];
        for (int i = 0; i < fromBufferChannels.length; i++) {
            guards[i] = fromBufferChannels[i].in();
```

```
        }

        Alternative alt = new Alternative(guards);
        Random rand = new Random();

        while (true) {
            int queryBufferIndex = rand.nextInt(toBufferChannels.
               length);
            toBufferChannels[queryBufferIndex].out().write(Payload
               .WHERE.ordinal());

            int selectedBufferIndex = alt.select();

            int response = fromBufferChannels[selectedBufferIndex
               ].in().read();

            if (response == Payload.HERE.ordinal()) {
                toBufferChannels[selectedBufferIndex].out().write(
                   Payload.PACKAGE.ordinal());
                try {
                    Thread.sleep(100 + rand.nextInt(400));
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                    break;
                }
            }
        }
    }
}
```

2.3 Konsument

```
package org.example.consumer;
import org.jcsp.lang.*;
import java.util.Random;

import org.example.utilities.Payload;

public class QueueConsumer implements CSProcess {
    final private One2OneChannelInt[] toBufferChannels;
    final private One2OneChannelInt[] fromBufferChannels;
    final private int Id;

    public QueueConsumer (One2OneChannelInt[] toBufferChannels,
       One2OneChannelInt[] fromBufferChannels, int Id) {
        this.toBufferChannels = toBufferChannels;
        this.fromBufferChannels = fromBufferChannels;
        this.Id = Id;
```

```
    }

    public void run () {
        Guard[] guards = new Guard[fromBufferChannels.length];
        for (int i = 0; i < fromBufferChannels.length; i++) {
            guards[i] = fromBufferChannels[i].in();
        }

        Alternative alt = new Alternative(guards);
        Random rand = new Random();

        while (true) {
            int queryBufferIndex = rand.nextInt(toBufferChannels.
               length);
            toBufferChannels[queryBufferIndex].out().write(Payload
               .WHERE.ordinal());

            int selectedBufferIndex = alt.select();

            int response = fromBufferChannels[selectedBufferIndex
               ].in().read();

            if (response == Payload.HERE.ordinal()) {
                if (fromBufferChannels[selectedBufferIndex].in().
                   read() == Payload.PACKAGE.ordinal()) {
                    try {
                        Thread.sleep(100 + rand.nextInt(400));
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                        break;
                    }
                }
            }
        }
    }
}
```

3 Wyniki testu równoważenia obciążenia

3.1 10 producentów, 10 konsumentów, 10 buforów, 20 wielkość bufora

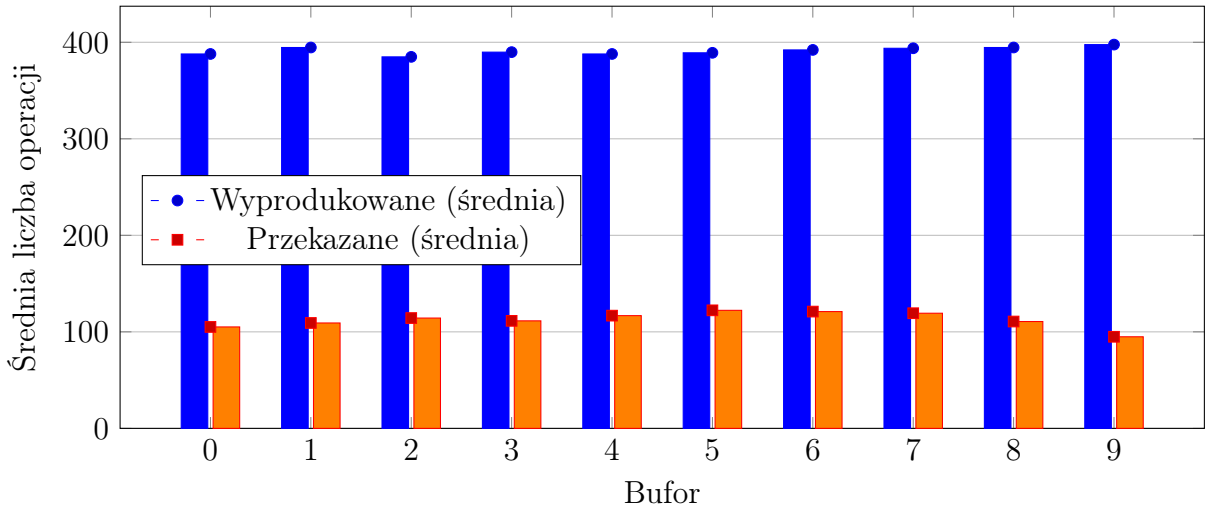


Figure 3.1.1: Producenci - wyprodukowe, a przekazane

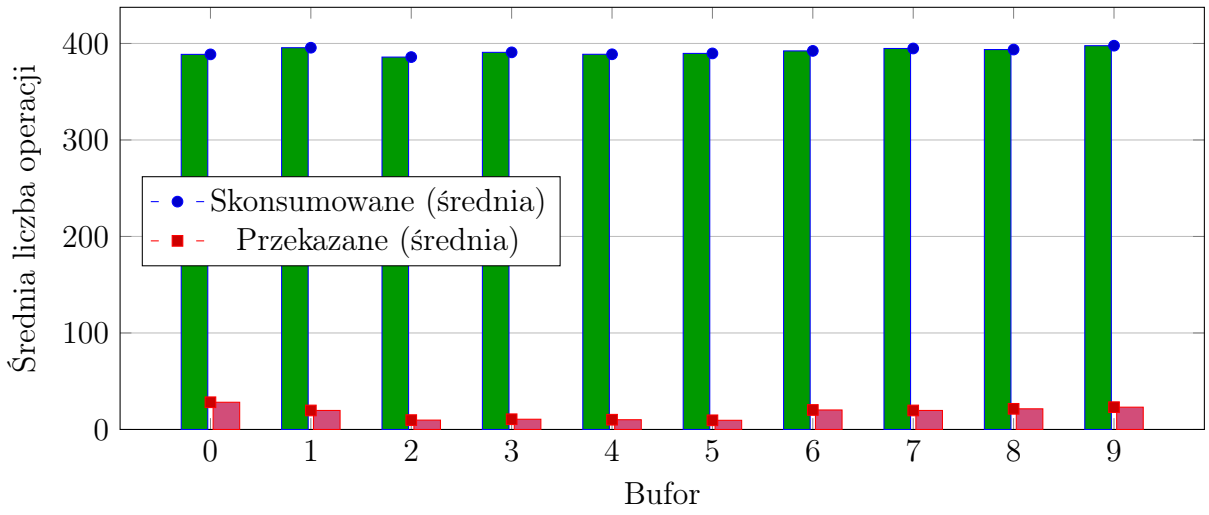


Figure 3.1.2: Konsumenci - skonsumowane, a przekazane

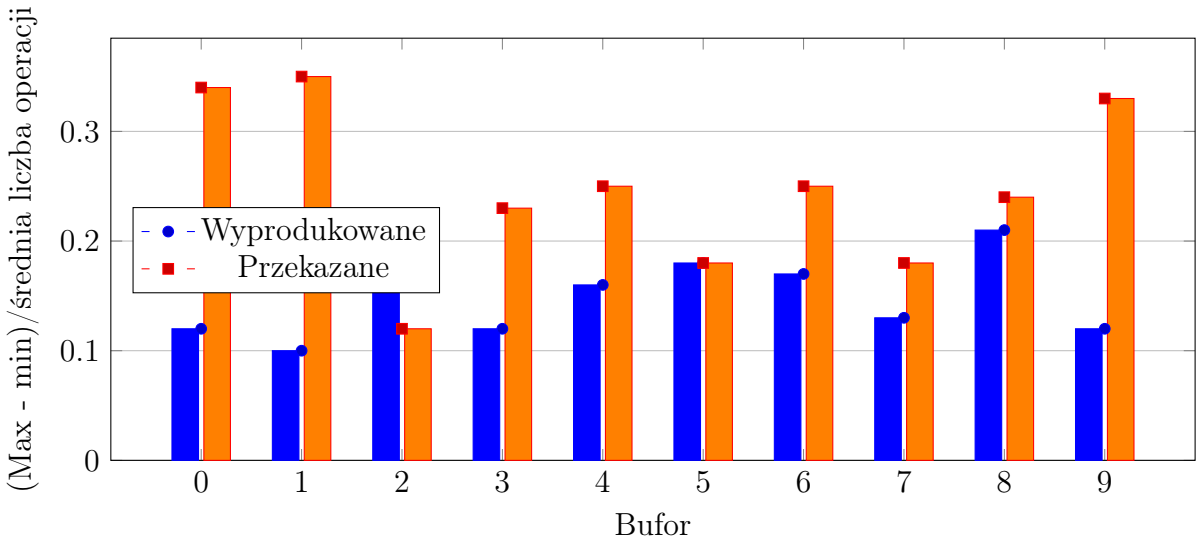


Figure 3.1.3: Producenci - wyprodukowe, a przekazane ((Max - min)/średnia)

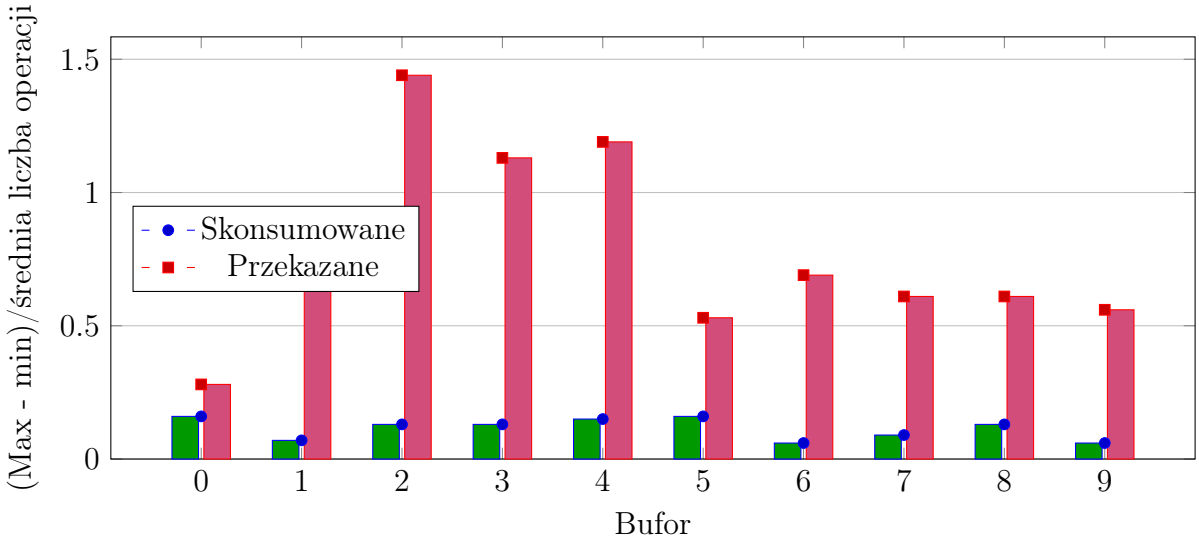


Figure 3.1.4: Konsumenci - skonsumowane, a przekazane (Max - min)/średnia

3.2 10 producentów, 10 konsumentów, 10 buforów, 5 wielkość bufora

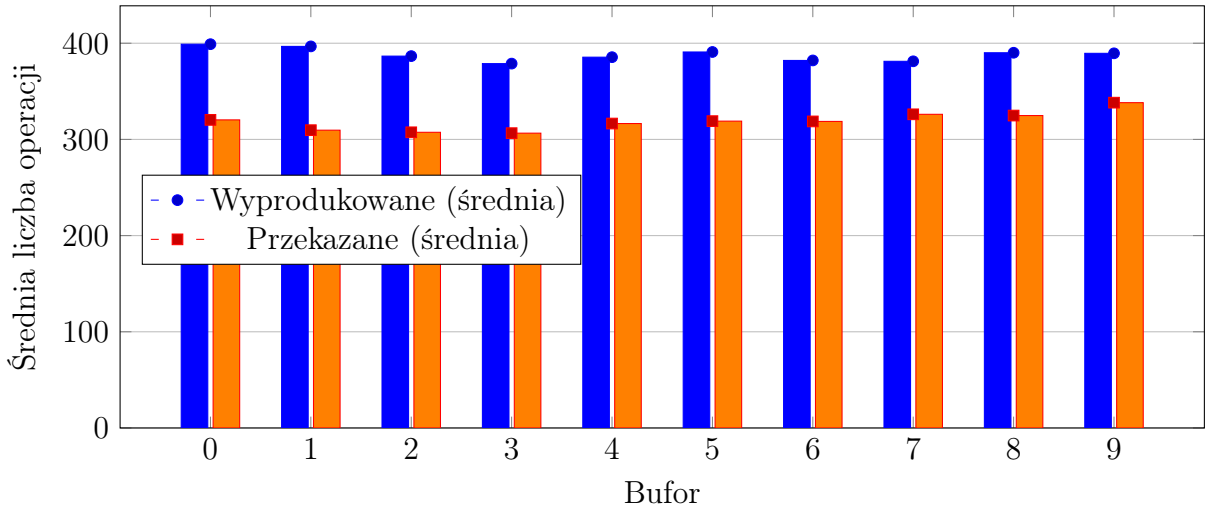


Figure 3.2.1: Producenci - wyprodukowe, a przekazane

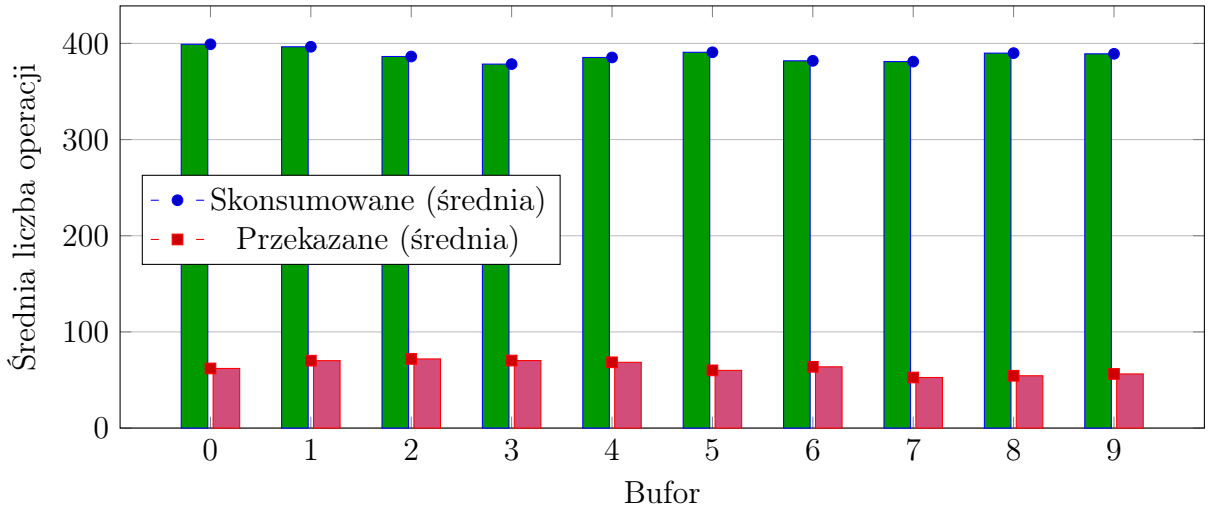


Figure 3.2.2: Konsumenci - skonsumowane, a przekazane

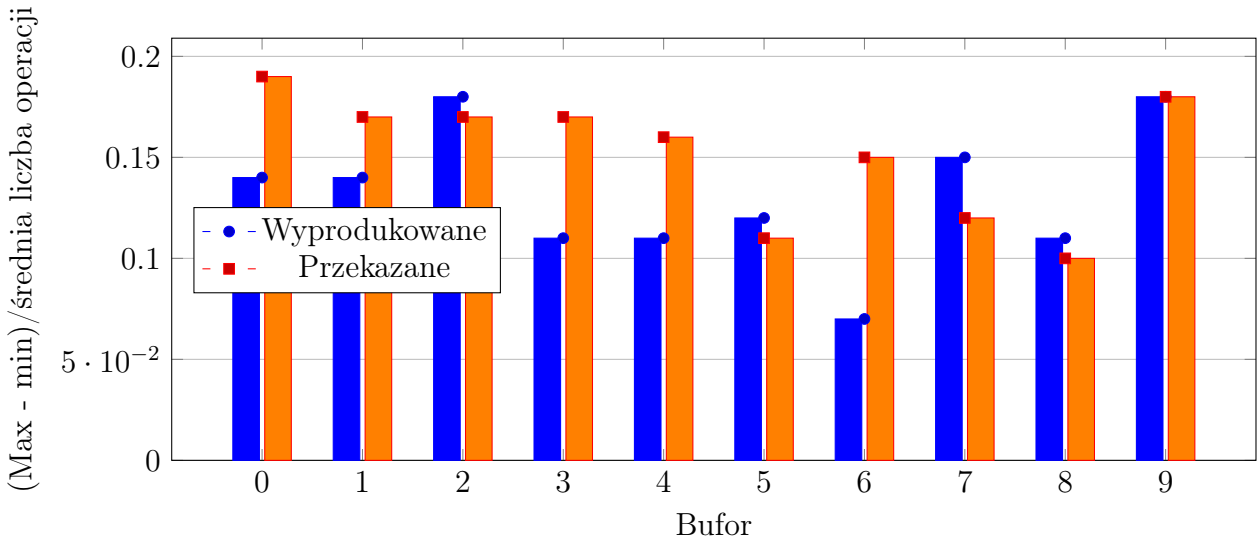


Figure 3.2.3: Producenci - wyprodukowe, a przekazane ((Max - min)/średnia)

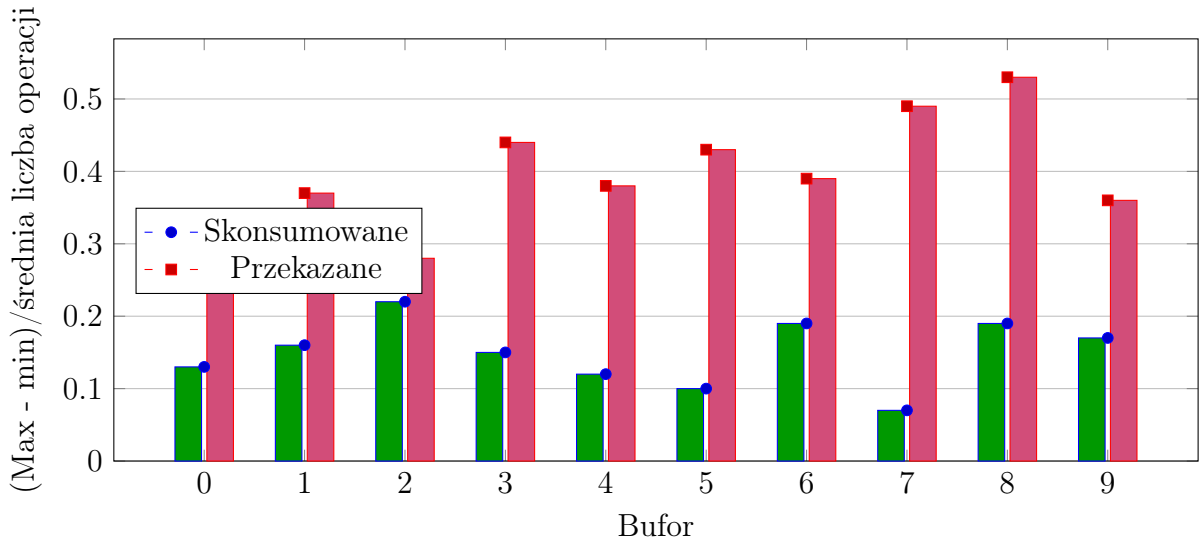


Figure 3.2.4: Konsumenci - skonsumowane, a przekazane (Max - min)/średnia

3.3 10 producentów, 10 konsumentów, 5 buforów, 20 wielkość bufora

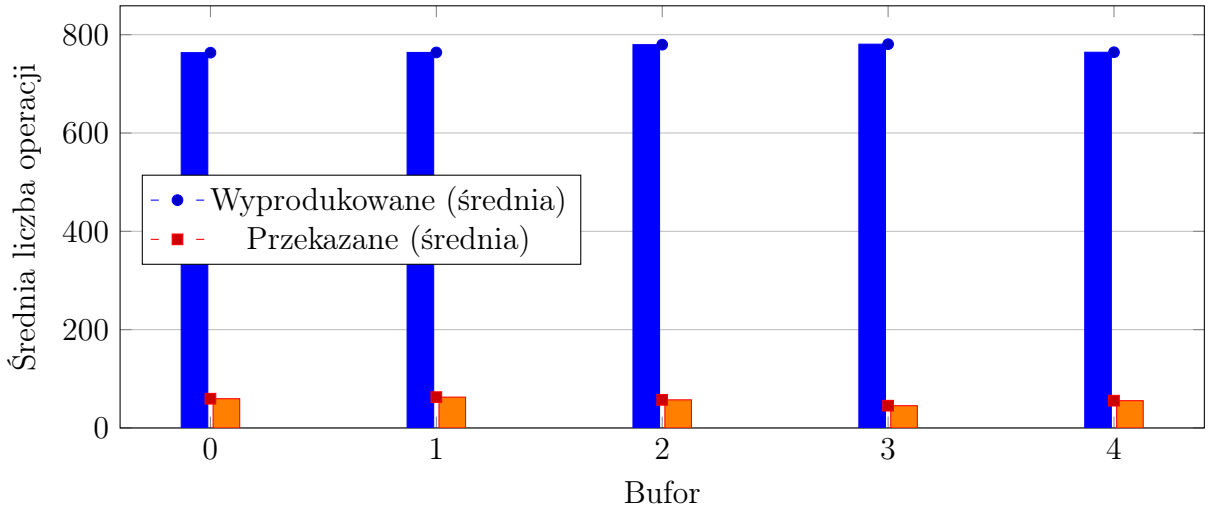


Figure 3.3.1: Producenci - wyprodukowe, a przekazane

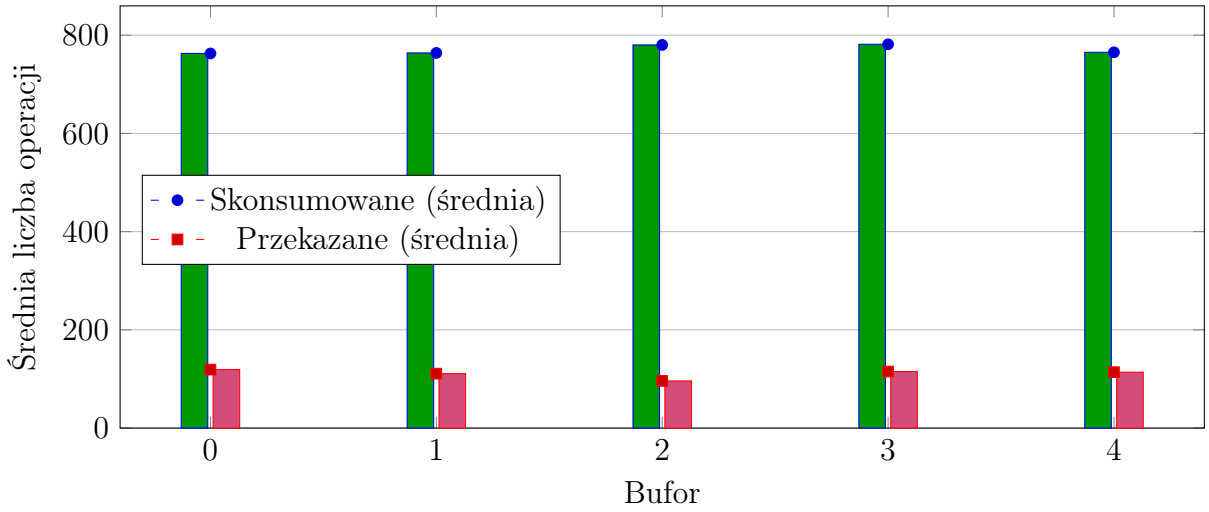


Figure 3.3.2: Konsumenci - skonsumowane, a przekazane

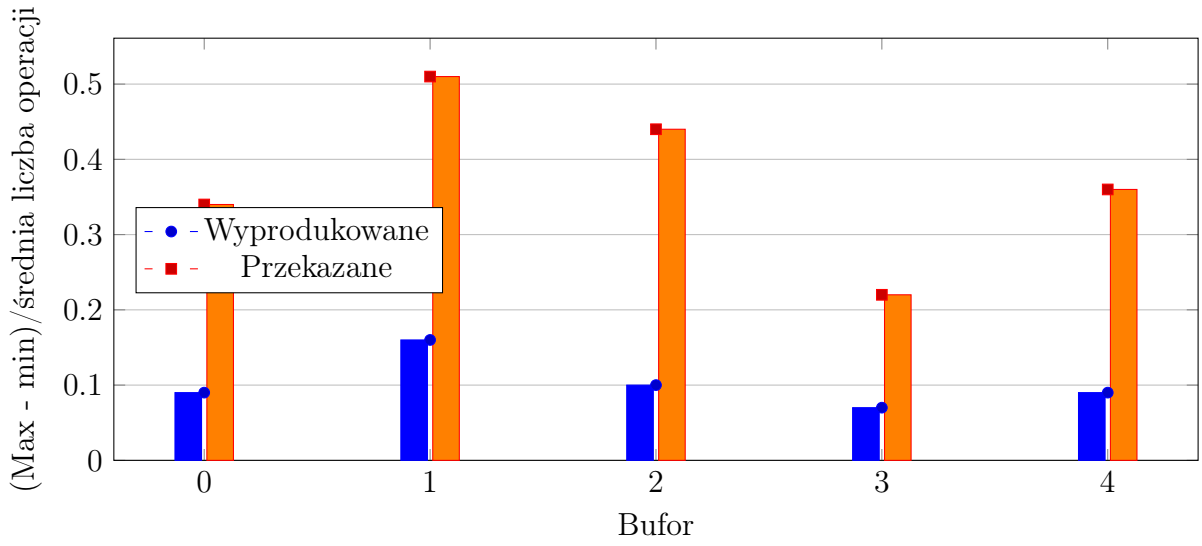


Figure 3.3.3: Producenci - wyprodukowe, a przekazane ((Max - min)/średnia)

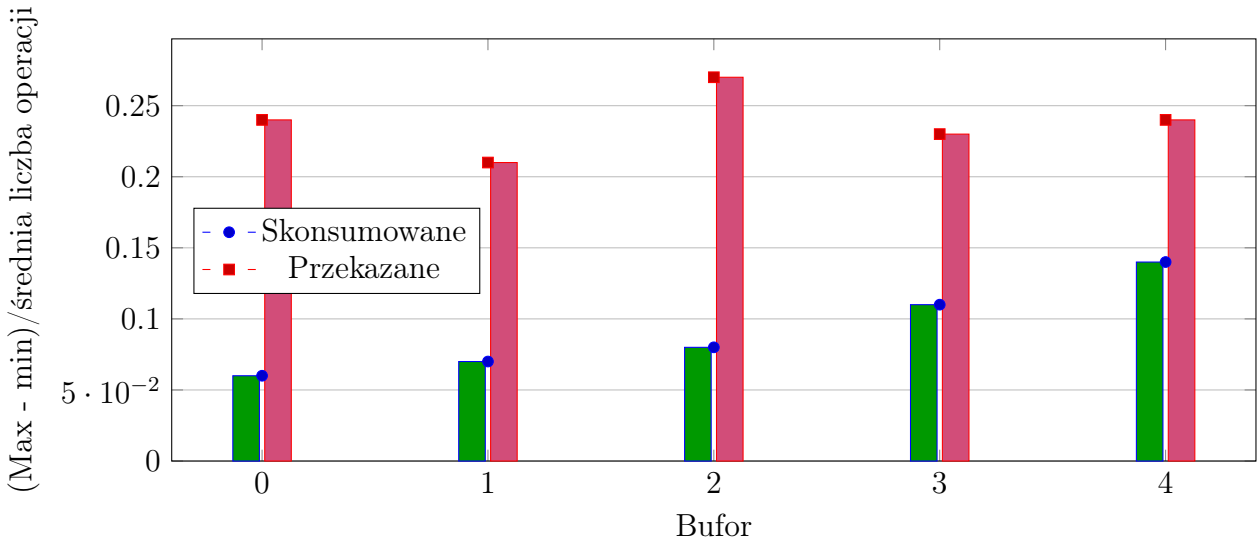


Figure 3.3.4: Konsumenci - skonsumowane, a przekazane (Max - min)/średnia

3.4 20 producentów, 10 konsumentów, 10 buforów, 20 wielkość bufora

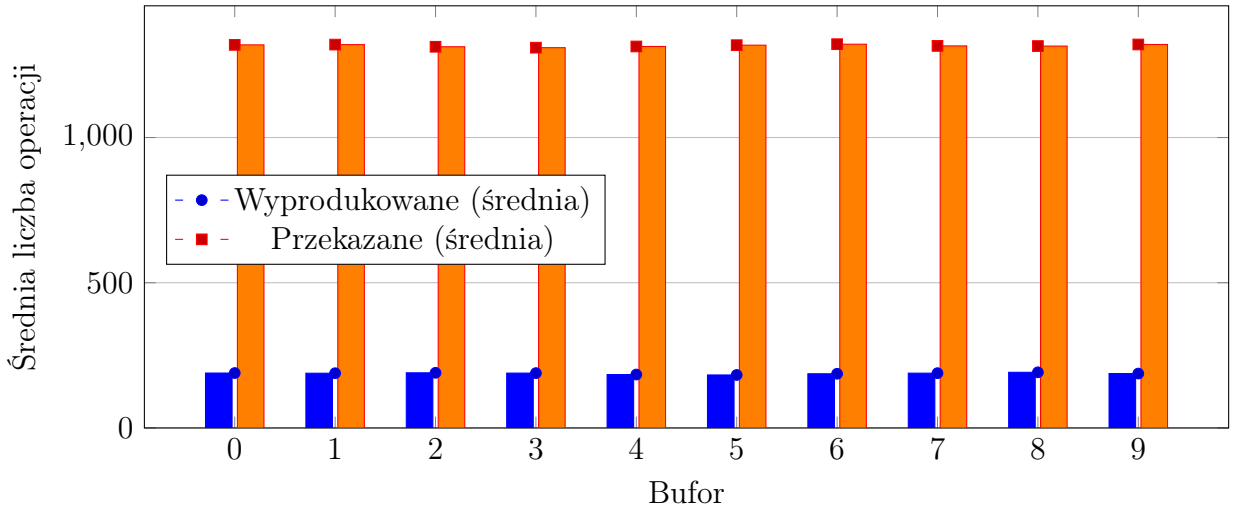


Figure 3.4.1: Producenci - wyprodukowe, a przekazane

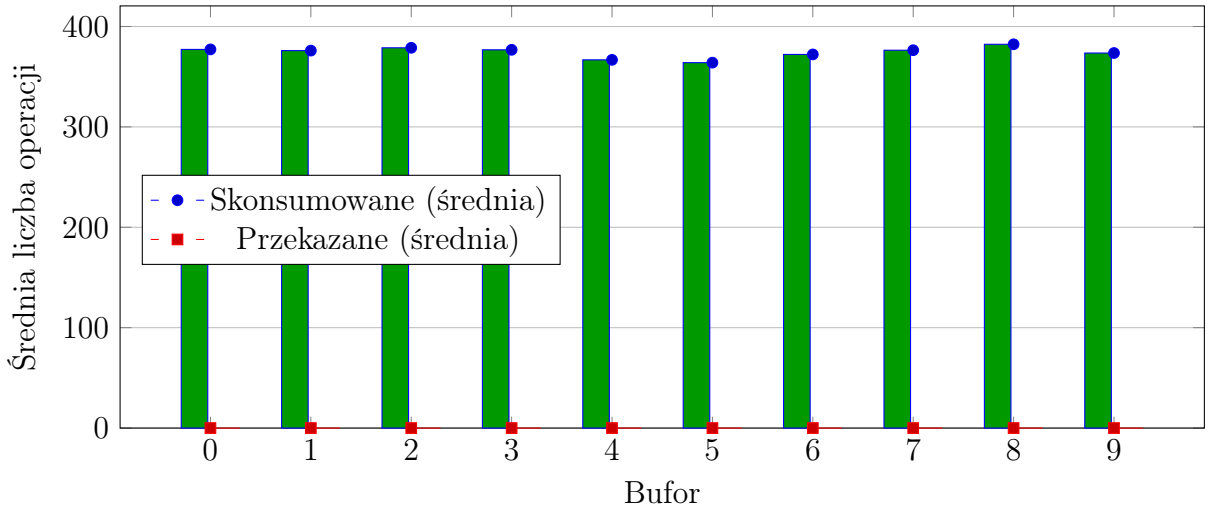


Figure 3.4.2: Konsumenci - skonsumowane, a przekazane

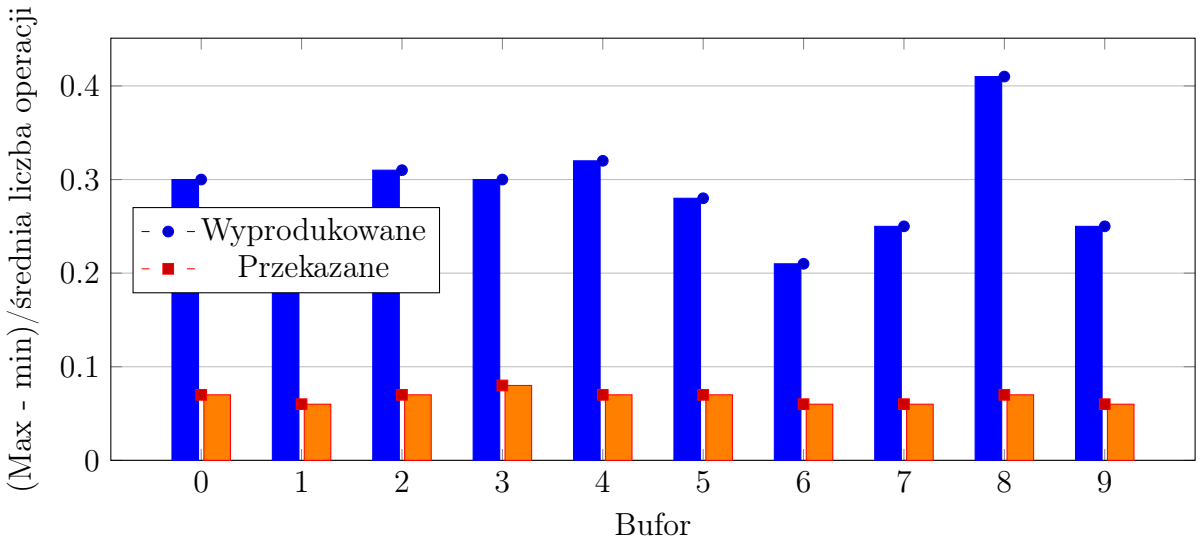


Figure 3.4.3: Producenci - wyprodukowe, a przekazane ((Max - min)/średnia)

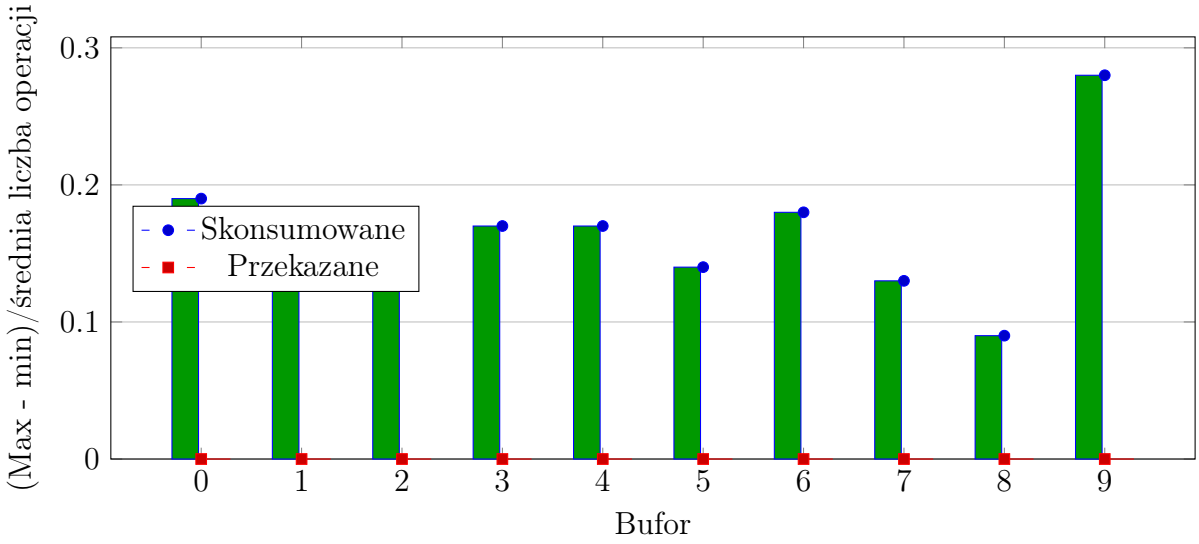


Figure 3.4.4: Konsumenci - skonsumowane, a przekazane (Max - min)/średnia

3.5 10 producentów, 20 konsumentów, 10 buforów, 20 wielkość bufora

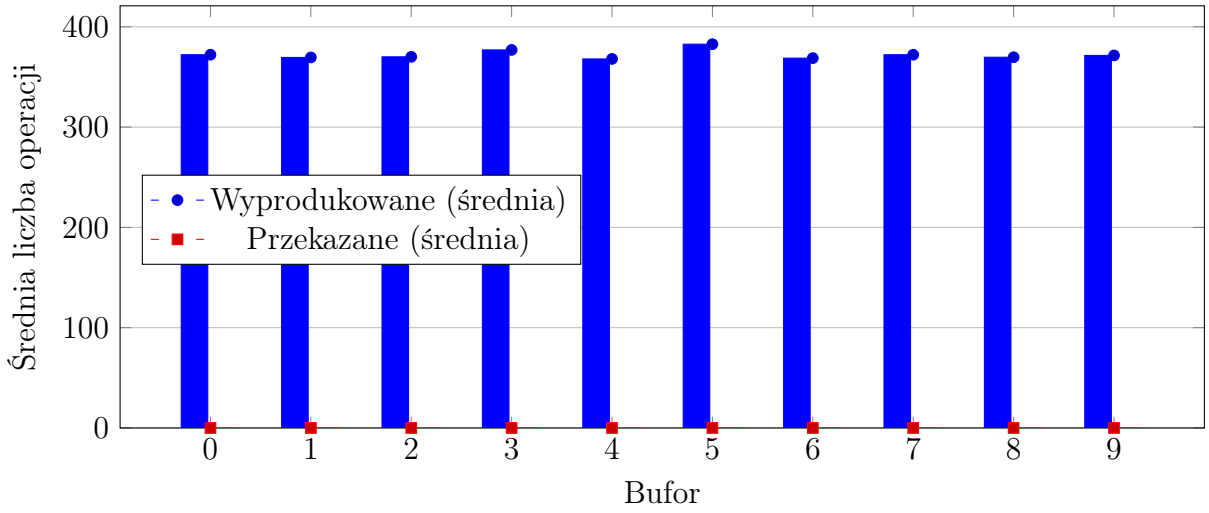


Figure 3.5.1: Producenci - wyprodukowe, a przekazane

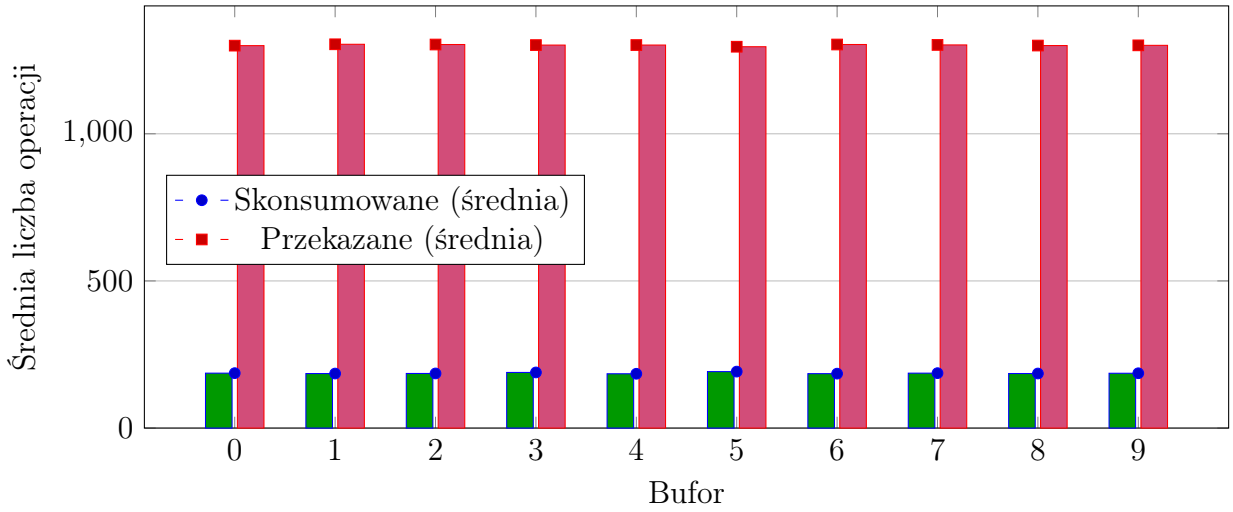


Figure 3.5.2: Konsumenci - skonsumowane, a przekazane

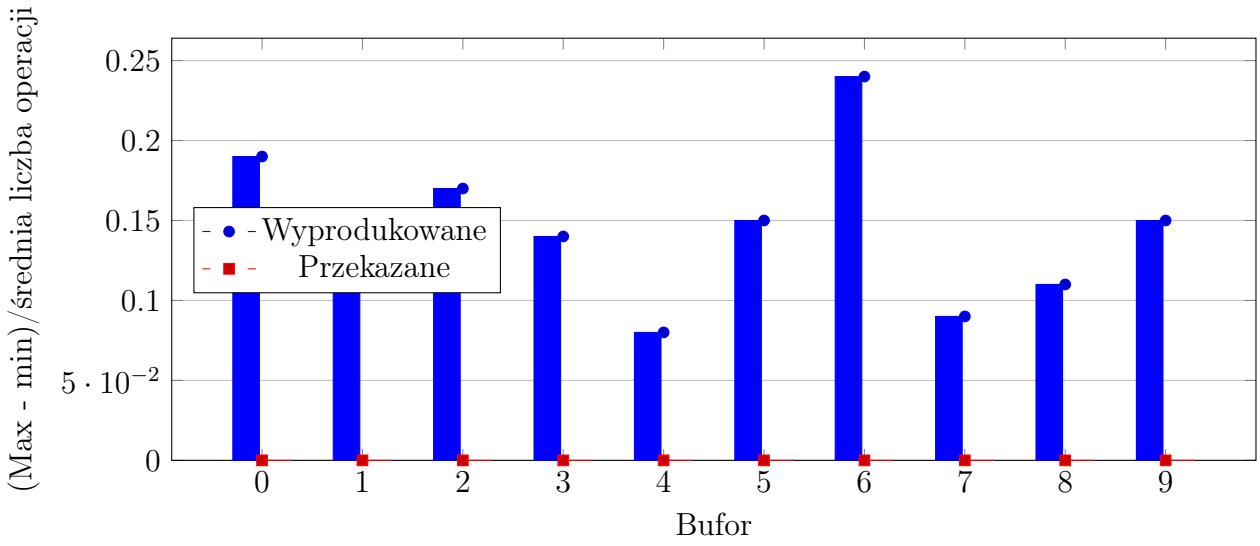


Figure 3.5.3: Producenci - wyprodukowe, a przekazane ((Max - min)/średnia)

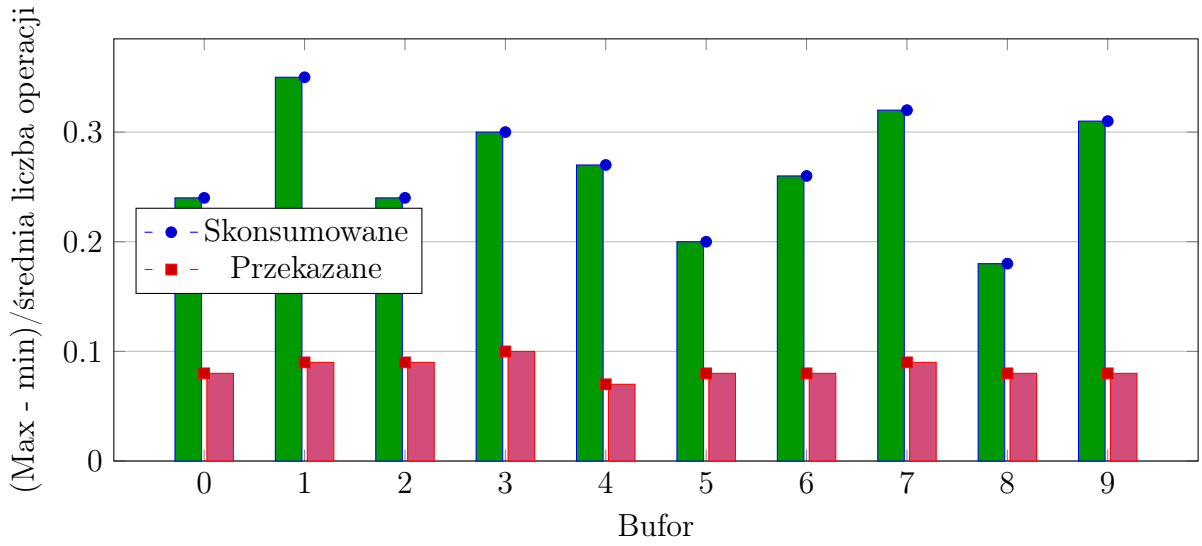


Figure 3.5.4: Konsumenci - skonsumowane, a przekazane (Max - min)/średnia

4 Wnioski

System dobrze radzi sobie z równoważeniem obciążenia w każdym testowanym przypadku. Zmniejszenie przestrzeni magazynowej nie zmniejsza przepustowości. Minusem jest skomplikowanie implementacji oraz duża liczba kanałów.