MultiProcessing Programming

Домашнее задание № 5

Симуляция и анализ TokenRing

https://github.com/IgorOzherelev/token-ring

Ожерельев Игорь

29 мая 2022 г.

Условие

Задача состоит в построении простой модели сетевого протокола в распределенной сети с топологией "кольцо"под названием TokenRing и исследовании его свойств.

1. Система состоит из N пронумерованных от 0 до N-1 узлов (в модели - потоков). Узлы упорядочены по порядковому номеру. После состояния N-1 следует узел 0, т.е. узлы формируют кольцо.

2. Соседние в кольце потоки могут обмениваться пакетами. При этом обмен возможен только по часовой стрелке.

3. Каждый поток, получив пакет от предыдущего, отдает его следующему.

4. Пакеты не могут обгонять друг друга.

Необходимо исследовать пропускную способность сети (throughput) и характерное время задержки (latency) в зависимости от количества узлов N и количества пакетов P (1...N), находящихся в транзите одновременно. Для простоты - считайте, что сообщения "выпускаются"в кольцо на старте и не имеют конечной точки.

Дополнительно нужно попытаться оптимизировать (улучшить) throughput или latency как в целом так и для отдельно взятых конкретных режимов (недогруженная сеть, перегруженная сеть) и исследовать влияние оптимизаций для одного режима на весь спектр режимов. Описывете историю оптимизации.

Рисуете красивые графики, формулируете свои мысли в виде мини-статьи. Сдаете рабочий код, статью.

Оценка будет зависить от:

1. корректности кода

2. качества анализа результатов (что наблюдаем, почему, какие могли ли повлиять какие-то сайдэффекты и т.д.)

3. решения доп.задач / ответов на вопросы

4. оформления (кратко, четко, корректно, научный стиль)

Над чем стоит задуматься(+**Краткие ответы**):

1. в каком случае считать сеть недогруженной/перегруженной?:
   Ответ: возможно при $P/N < 1$ недогруженная, при $P >> N$ перегруженная, но это очень условно

2. какую информацию нужно хранить в пакетах и на узлах?:
   Ответ: см. статью

3. как провести замеры производительности?:
   Ответ: System.nano() + набор временных меток, см. статью

4. как смоделировать тесты с учетом имеющихся в вашем распоряжении мощностей?:
   Ответ: правильность работы можно проверить junit'ом

5. что будет отвечать за передачу сообщений? (от самого простого варианта, например с максимумом в 1 сообщение за раз - к более сложным):
   Ответ: одна нода, создает фиксированное число пакетов

6. как сделать результаты наглядными?:
   Ответ: см. статью

Реализация

2.1 Узлы

Каждый узел сети **RingNode** реализуется от интерфейса **Node**. Между узлами сети передаются фреймы **Frame**. Разберем на примере листинга, содержащего поля класса:

```
1  public class RingNode implements Node {
2      private static final int NODE_BUFFER_CAPACITY = 100;
3      private static final int SLEEP_HANDLING = 5;
4      private static final int POLLING_TIME = 2;
5
6      private long framesToGenerate = 1;
7      private final long nodeId;
8      private final String nodeInfo;
9      private Node nextNode;
10     private final TokenRingObserver observer;
11     // private final BlockingQueue<Frame> framesToSend = new ArrayBlockingQueue<>(
           NODE_BUFFER_CAPACITY);
12     private final ConcurrentLinkedQueue<Frame> framesToSend = new ConcurrentLinkedQueue
           <>();
13
14     public RingNode(long nodeId, TokenRingObserver observer, AtomicBoolean aliveRingFlag,
15                     long framesToGenerate, boolean generate) {
16         this.nodeId = nodeId;
17         this.observer = observer;
18         this.aliveRingFlag = aliveRingFlag;
19         this.nodeInfo = "Node[" + nodeId + "]";
20         this.framesToGenerate = framesToGenerate;
21         if (generate) {
22             initFrames();
23         }
24     }
25 }
```

```
1  public interface Node extends Runnable {
2      void handleFrame(Frame frame);
3      void forward(Frame frame);
4      void receive(Frame frame);
5      void setNext(Node node);
6  }
```

В упрощенной модели, предложенной в условии, можно считать, что в сети только один узел, генерирующий фреймы. Тем не менее реализация не ограничена этим предположением: для генерации данных используется флаг generated. Однако, при снятии метрик была только одна вершина, создающая сообщения.

Для фиксирования временных меток проходящих сообщений используется сущность **TokenRingObserver**. В работе протестированно два способа реализации узлов: при помощи блокирующей очереди(**ArrayBlockingQueue**) и неблокирующей(**ConcurrentLinkedQueue**).

Во время работы топологии(флаг **aliveRingFlag** поднят) все узлы проверяют на наличие сообщений в своей очереди, метод **doWork()**. В зависимости от соблюдения условий, наложенных на полученный фрейм происходит обработка в методе **handleFrame(Frame frame)**, далее узел пересылает соседнему сообщение из своей очереди в его. Если очередь соседа оказалась заполнена, то сосед пересылает пакет следующему, и так далее.

Передача сообщений прекращается, когда **TokenRingObserver** зарегистрирует временные метки для всех фреймов в сети, затем выставит флаг **aliveRingFlag** в **false**.

2.2 Фреймы

Фреймы, которые дошли до адресата, из сети не удаляются: выставляется статус **RECEIVED** и временная метка, после, направляется соседу.

Если фрейм совершил круг(вернулся к тому, кто его создал) и пакет дошел до адресата(со статусом **RECEIVED**), то прописывается статус **RETURNED** и время, затем временные метки фиксирует **TokenRingObserver**.

В момент отправки сообщения после его рождения записывается статус **FLYING** и метка времени, далее отправляется соседу. **Отметим, что пакеты из сети не удаляются вне зависимости от статуса фрейма.**

Опишем состав полей фрейма листингами ниже:

```
@Getter
public class Frame {
    private String UUID;
    private long fromId;
    private long toId;
    private String message;

    private TimeMarks timeMarks;
    private Status status;
}
```

```
public enum Status {
    RETURNED,
    RECEIVED,
    FLYING;
}
```

```
@Getter
public class TimeMarks {
    private long sent = 0;
    private long received = 0;
    private long returned = 0;

    public void setSent() {
        this.sent = System.nanoTime();
    }

    public void setReceived() {
        this.received = System.nanoTime();
    }

    public void setReturned() {
        this.returned = System.nanoTime();
    }
}
```

Сбор данных и результаты

3.1 Параметры и порядок сбора метрик

Сбор метрик осуществлялся на машине со следующими характеристиками(результат **lscpu**):

1. Версия ОС: Ubuntu 20.04.4 LTS

2. Архитектура: x86-64
   CPU op-mode(s): 32-bit, 64-bit
   Порядок байт: Little Endian
   Address sizes: 39 bits physical, 48 bits virtual

3. CPU(s): 8
   On-line CPU(s) list: 0-7
   Потоков на ядро: 1
   Ядер на сокет: 4
   Сокетов: 1
   NUMA node(s): 1
   ID прроизводителя: GenuineIntel
   Семейство ЦПУ: 6
   Модель: 158
   Имя модели: Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz

4. Есть поддержка hyper-threading; **при сборе данных был отключен**, что соответствует одному потоку на ядро, в случае гипертредининга 2 потока на ядро.
   Отключал подобным образом как здесь

Получение временных меток осуществлялось не сразу: сначала делаем **4 серии по 10 запусков** для **прогрева кэшей**, затем производим еще одну серию по **10 запусков** для каждой конфигурации сети со сбором данных.

Серии выполнялись для каждой конфигурации сети(число узлов и количество пакетов), то есть по 10 запусков для каждого варианта топологии и количества сообщений.

3.2 Результаты

В рамках задачи в качестве throughput удобно рассматривать число токенов, деленное на усреднённое время циркуляции(время, за которое пакет вернется в тому, кто его создал) сообщения в сети. Действительно, эта величина пропорциональна количеству информации, которую можно передать через сеть.

$$Throughput = \frac{n_{frames}}{Time_{Returned} - Time_{Sent}}$$

В качестве latency предлагается рассматривать время, прошедшее между генерацией сообщения и его достижением адресата. Действительно, эта величина характеризует задержки при передаче информации посредством нашего протокола.

$$Latency = Time_{Received} - Time_{Sent}$$

В ходе работы собираемся проверить следующие предположения:

1. throughput растёт с увеличением количества фреймов

2. throughput падает с увеличением количества узлов

3. latency падает с увеличением количества фреймов

4. latency растёт с увеличением количества узлов

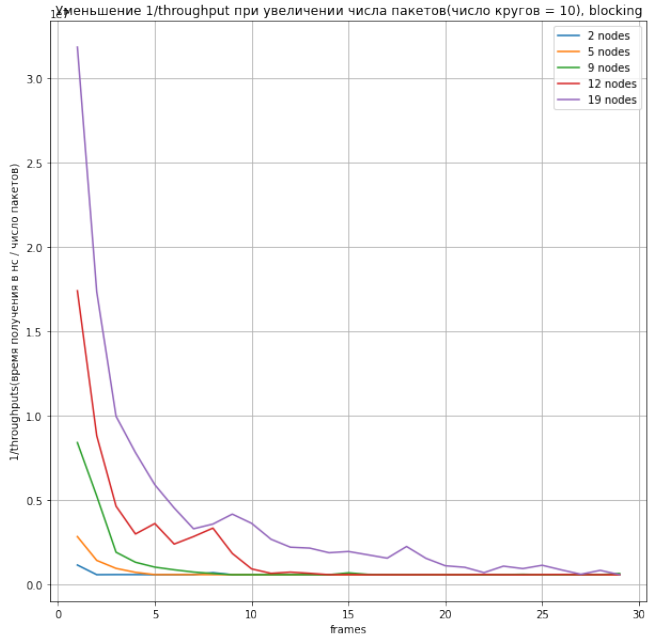

Рис. 1: Падение 1/Throughput для blocking queue при росте фреймов

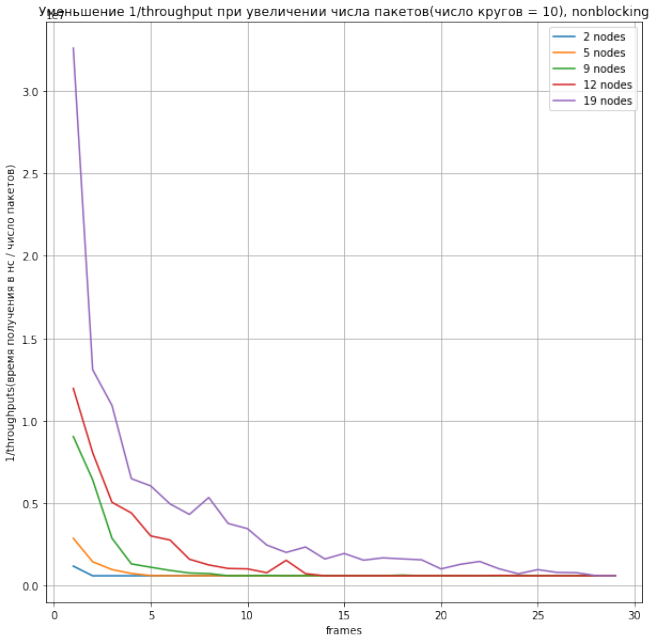

Рис. 2: Падение 1/Throughput для non-blocking queue при росте фреймов

Видно, что throughput растёт с увеличением количества фреймов, так как 1/throughput падает при увеличением числа фреймов.

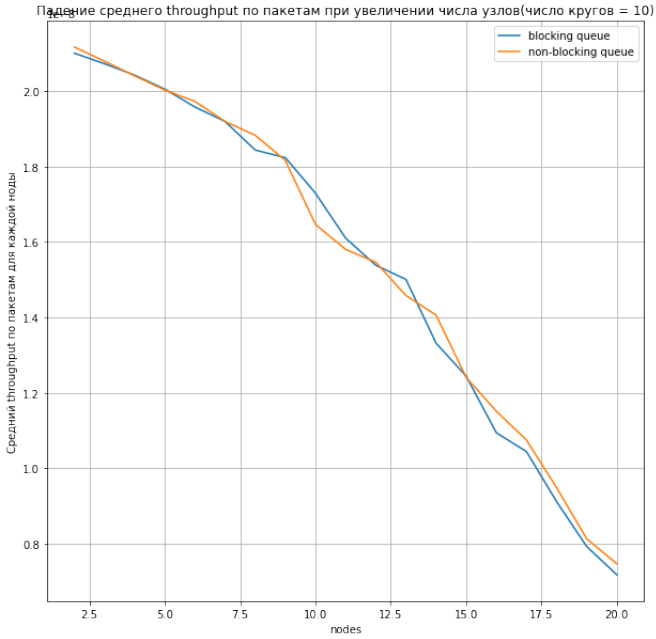

Рис. 3: Падение Throughput для non-blocking и blocking queue в зависимости от числа узлов (среднее по пакетам)

Отметим, что throughput падает с увеличением количества узлов. Неблокирующая очередь обеспечивает чуть лучше throughput для большого числа узлов в среднем по пакетам.

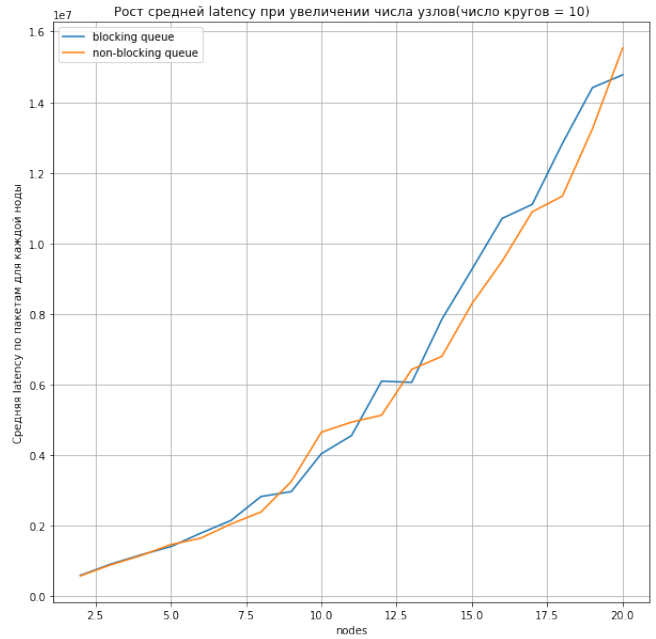

Рис. 4: Рост latency non-blocking и blocking queue в зависимости от числа узлов (среднее по пакетам)

Latency растёт с увеличением количества узлов. В среднем блокирующая и неблокирующая очереди ведут себя примерно в среднем одинаково в отношении latency. Хотя при большом числе узлов, latency для non-blocking была лучше.

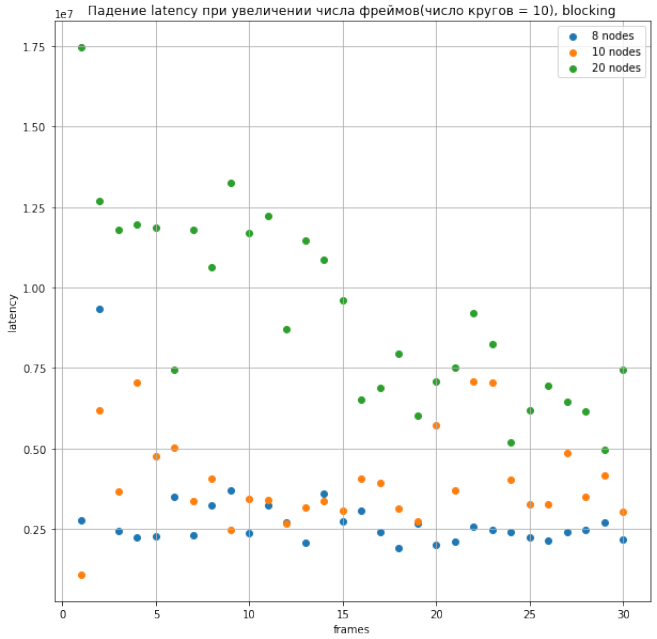

Рис. 5: Падение latency падает с увеличением количества фреймов для blocking

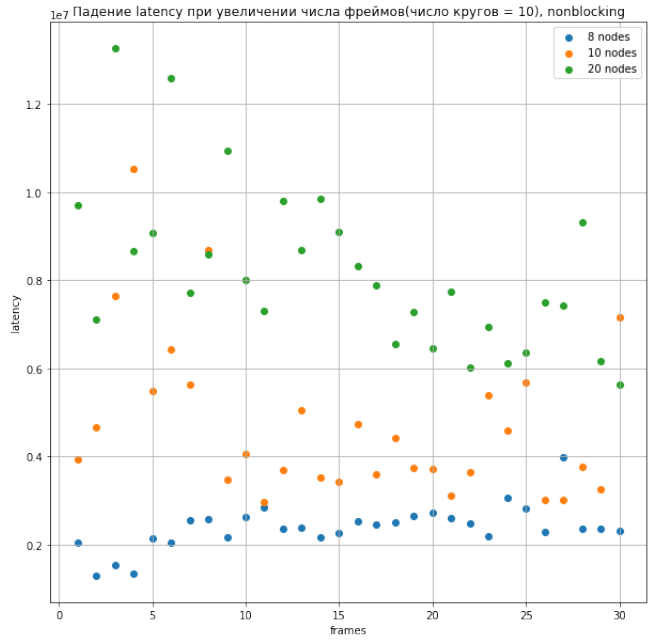

Рис. 6: Падение latency падает с увеличением количества фреймов для non-blocking

Latency для неблокирующей очереди сильно колеблется в среднем по пакетам, но для блокирующей и неблокирующей наблюдается тренд к падению latency при росте числа пакетов в сети. Более детальные картинки для latency в приложении, в конце работы.

Вывод

В результате работы был просимулирован протокол TokenRing. Сравнены реализации узлов, основанные на блокирующей и неблокирующей очередях. Было проверено:

1. throughput растёт с увеличением количества фреймов
2. throughput падает с увеличением количества узлов

3. latency падает с увеличением количества фреймов

4. latency растёт с увеличением количества узлов

Также отметим, что разница между блокирующей и неблокирующей очередью сотоит в следующем:

1. Latency для неблокирующей очереди сильно колеблется в среднем по пакетам, но для блокирующей и неблокирующей наблюдается тренд к падению latency при росте числа пакетов в сети.

2. Блокирующая и неблокирующая очереди ведут себя примерно в среднем одинаково в отношении latency при росте числа узлов. Хотя для большего числа узлов, latency для non-blocking была лучше.

3. Неблокирующая очередь обеспечивает чуть лучше throughput для большого числа узлов в среднем по пакетам.

Приложение

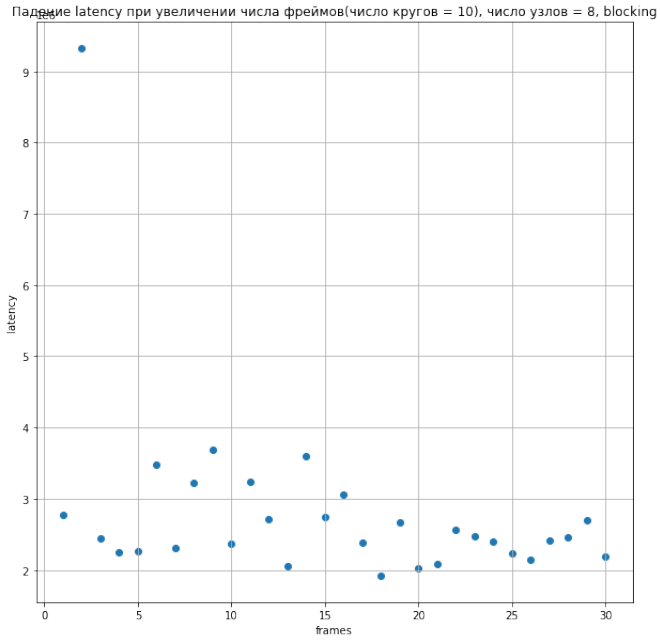

Рис. 7: Падение latency падает с увеличением количества фреймов для blocking, nodes = 8

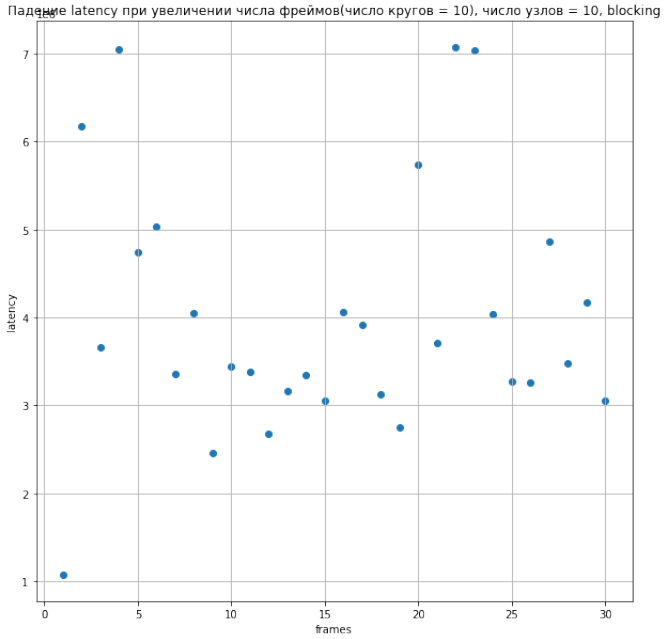

Рис. 8: Падение latency падает с увеличением количества фреймов для blocking, nodes = 10

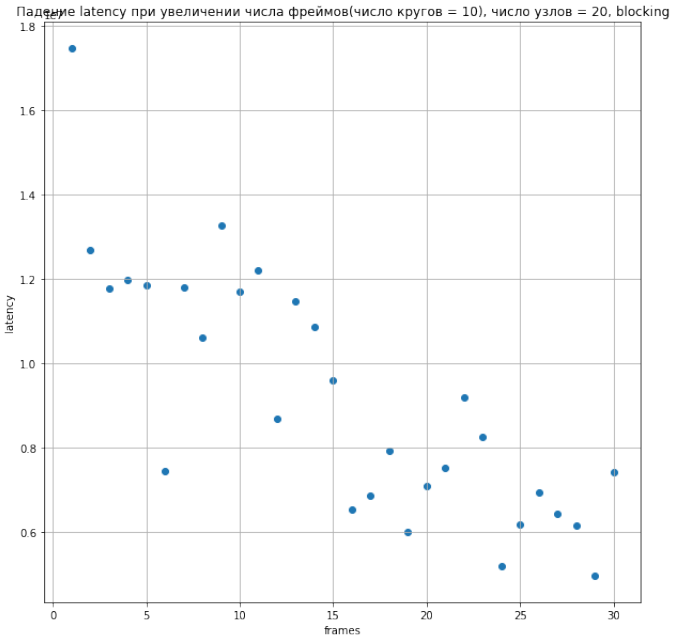

Рис. 9: Падение latency падает с увеличением количества фреймов для blocking, nodes = 20

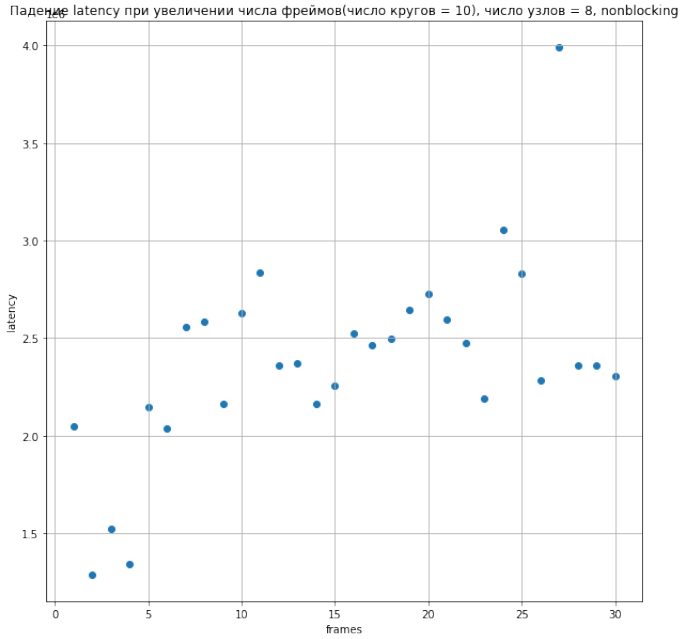

Рис. 10: Падение latency падает с увеличением количества фреймов для non-blocking, nodes = 8

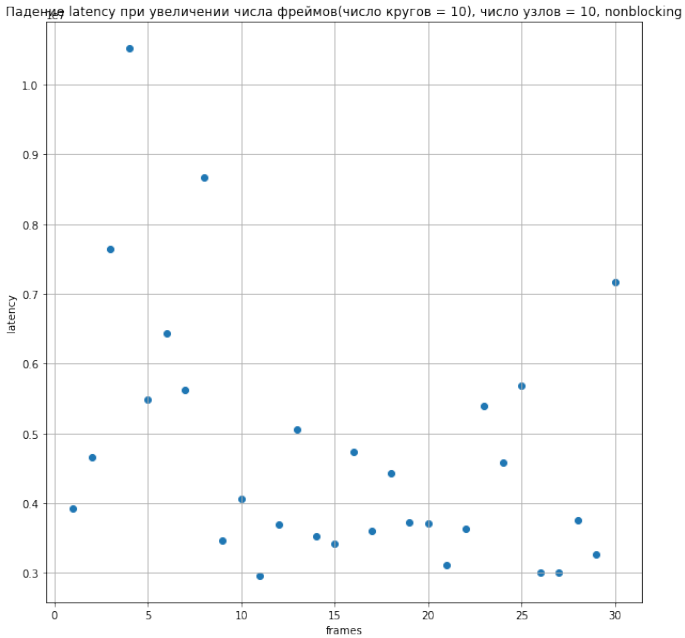

Рис. 11: Падение latency падает с увеличением количества фреймов для non-blocking, nodes = 10

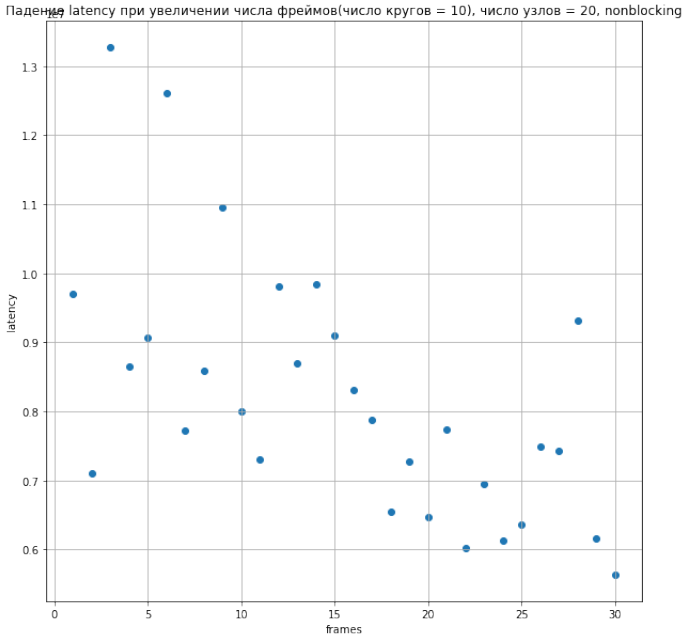

Рис. 12: Падение latency падает с увеличением количества фреймов для non-blocking, nodes = 20