Домашнее задание 6 – Token Ring

Меньших Игорь
M05-014в

---

1 Постановка задачи

Задача состоит в построении простой модели сетевого протокола под названием TokenRing и исследовании его свойств.

1. Система состоит из *N* пронумерованных от *0* до *N-1* узлов (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. 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*, находящихся в транзите одновременно.

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.

2 Имплементация

Исходный код доступен в виде IDEA-проекта на github. Основные этапы реализации:

1. Фиксирование гиперпараметров *TokenRing*:
    - *N* – размер TokenRing, то есть количество потоков, которые участвуют в передаче токенов по кольцу и их доставки до пункта (узла) назначения
    - *P* – число пакетов, находящихся в транзите одновременно
    - *warmUpRuns* – количество прогревочных ранов (без записи результатов)
    - *normalRuns* – количество обычных ранов (с записью результатов)
2. Конструирование *TokenRing*:

- генерация экземпляров узлов-передатчиков (интерфейс *Transporter*), задачей которых является промежуточное хранение токенов при их движении по *TokenRing* и предоставление API операций *push* (один из потоков может положить токен на узел-передатчик) и *poll* (следующий по порядку часовой стрелки поток может взять этот токен с рассматриваемого узла-передатчикка) в соответствии со стратегией FIFO; таким образом, обеспечиватся требования 1, 4 постановки задачи;

- генерация экземпляров узлов-обработчиков (интерфейс *Node*), задачей которых является передача токенов от предыдущего узла-передатчика к следующему, а также поглощение токенов, которые прошли полный круг по TokenRing, с фиксацией временных отметок начала движения и его окончания; таким образом, обеспечиватся требования 2, 3 постановки задачи;

- генерация экземпляров токенов (интерфейс *Token*), задачей которых является продвижение по TokenRing от начального узла-передатчика, на котором они были сгенерированы, до него же, преодолев все остальные узлы по порядку часовой стрелки;

3. Подключение узлов друг к другу и запуск потоков, заключающийся в переключении узлов-обработчиков в состояние ожидания токенов на узле-передатчике, который находится ранее их по порядку часовой стрелки.

4. Ожидание основной программой (запустившей экземпляр *TokenRing*), пока все токены не будут доставлены на соотвествующие узлы, и затем завершение всех слушающих потоков.

5. Результаты в форме временных отметок порядка наносекунд фиксируются внутри каждого токена по его отправке и получении на соответствующем узле-обработчике.

3 Эксперименты

3.1 Тестовое окружение

Тестирование проводилось на стационарном ПК со следующими характеристиками:

- ОП – 16GB
- ЦП – Interl Core i5-9600K 3.70GHz
  - 9MB Cache
  - 6 Core
  - Hyperthreading disabled

3.2 Методология тестирования

Результаты были получены путем фиксирования временных меток (*timestamp*) прохождения пакетами определенного (одного) узла-обработчика (момент отправки и момент получения). При этом использовалась стандартная функция *System.nanoTime()*.

Эксперименты проводились над *5* реализациями узлов-передатчиков (*LockBasedLinkedList*, *LockBasedLinkedListMod, LockBasedArrayDeque*, *ArrayBlockingQueue*, *LinkedBlockingQueue*) на параметрической сетке *N* x *P.* Использовались следующие значения параметров:

N = {2 .. 8} P = {840, 1680, 2520} warmUpRuns = 10 normalRuns = 40

Значения параметра *N* выбирались так, чтобы можно было оценить изменение поведения *TokenRing* при разном количестве задействованных потоками ядер, при этом обязательно включить в диапазон число ядер установки (N=6). Значения *P* выбирались так, чтобы, с одной стороны, в минимальной конфигурации (N=2, P=840) было пройдено достаточно warmup итераций для применения оптимизаций на уровне JIT-компилятора языка Java (более 10000), и с другой стороны, чтобы подсчет в максимальной конфигурации не занимал излишне много времени (более 1 мин), а также результаты одного эксперимента (реализации *Transporter*) не занимали излишне много дискового пространства (более 100 мб). Рапределение по узлам полагалось как равномерное, поэтому значения параметра P выбирались кратными всевозможным значением параметра N.

Оценивая загруженность *TokenRing*, можно сказать, что меньшие значения *P* предполагают *недозагруженность* сети (то есть токены более свободно передвигаются по кольцу, потоки большую часть времени находятся в ожидании), в то время как б**о**льшие – ее *перегруженность* (то есть, токены достаточно плотно «упакованы» по кольцу, и их продвижение по нему происходит медленее).

Оценивая ресурс производительности *TokenRing*, можно ожидать, что она будет возрастать с увеличением *N* от 2 до 6, а затем спадать на значениях N от 6 до 8. В основе предположения находится количество ядер установки.

3.3 Результаты

В процессе экспериментов оценивались следующие величины:

- **L**atency
  – время, которое требуется токену, чтобы осуществить передвижение по полному кругу TokenRing и закончить свой путь на начальном узле-передатчике; размерность величины – мкс (графики), нс (эксперименты)

- **T**hroughput
  – количество токенов, которые проходят через узел-передатчик в единицу времени (пропускная способность сети); размерность величины – токен/мкс (графики), токен/нс (эксперименты)

*на некоторых рисунках далее отсутствуют доверительные интервалы (95%) оцениваемых величин вследствие их узости и, как следствие, малого визуального эффекта (показательности)

3.3.1 LockBased

Группа реализаций узла-передатчика под общим названием *LockBased* представляет собой две имплементации стандартных Java-коллекций (связный список *LinkedList* и двунаправленная очередь *ArrayDeque*), дополненные механизмом блокировки (*ReentrantLock*) для конкурентного доступа на чтение (*poll*) и запись (*push*) для нескольких процессов.

LockBasedLinkedListTransporter

Реализация на основе связного списка была взята в качестве baseline.

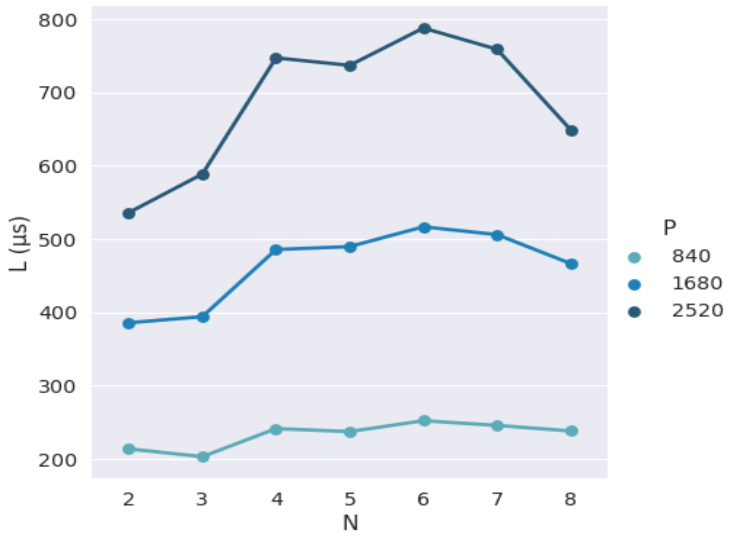

Рисунок 1a. С увеличением значений параметра P наблюдается закономерное увеличение задержки L, при этом есть явный пик на значении N=6. Скачок на значении N=4, вероятно, вызван поведением GC (Garbage Collector)

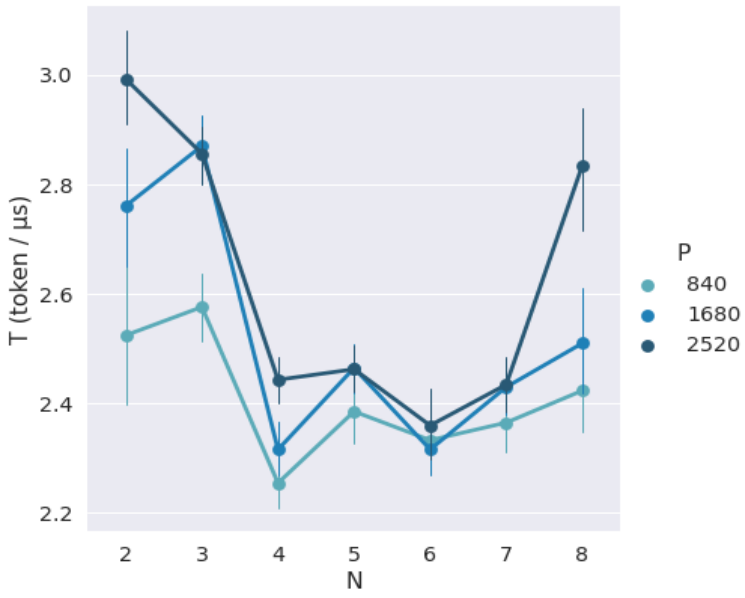

Рисунок 1b. С увеличением значений параметра P наблюдается увеличение пропускной способности сети T, при этом наименьшее увеличение соответствует значению N=6.

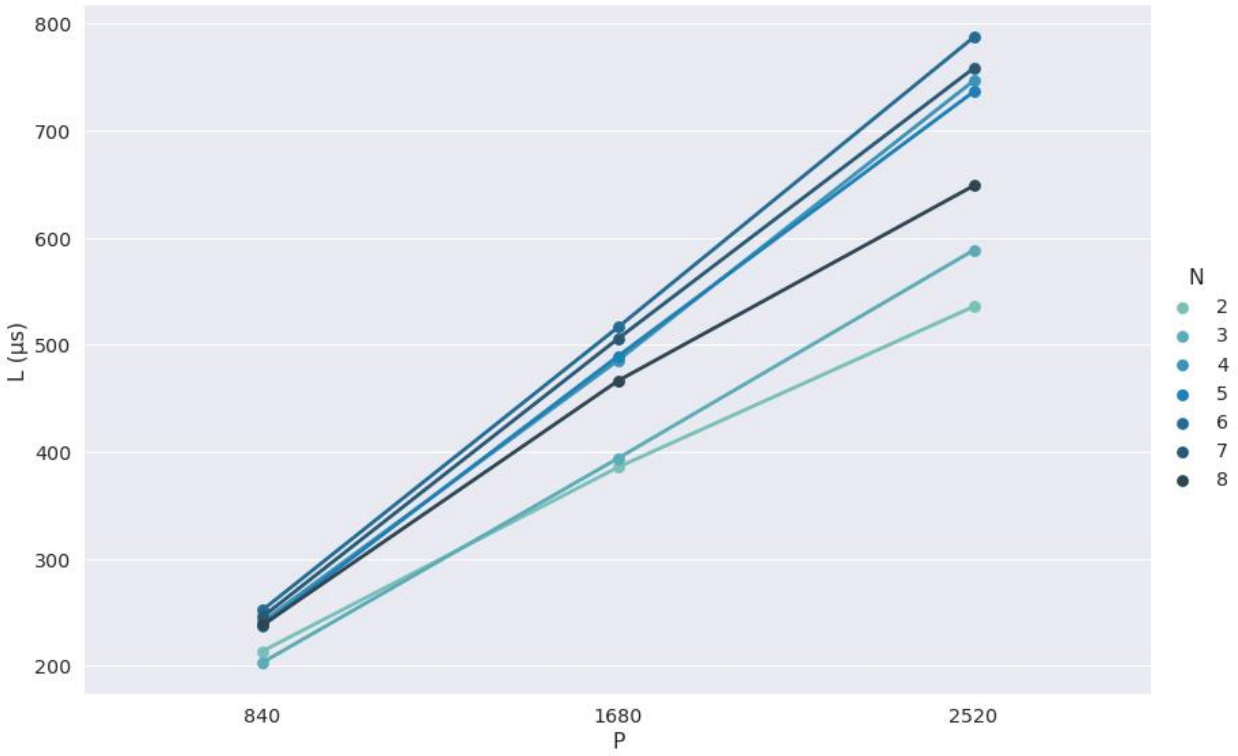

Рисунок 1c. С увеличением значений параметра N наблюдается увеличение задержки L, при этом зависимость L от P стремится к линейной с увеличением N от 2 до 6, далее становится заметен «изгиб» линий на значении P=1680.

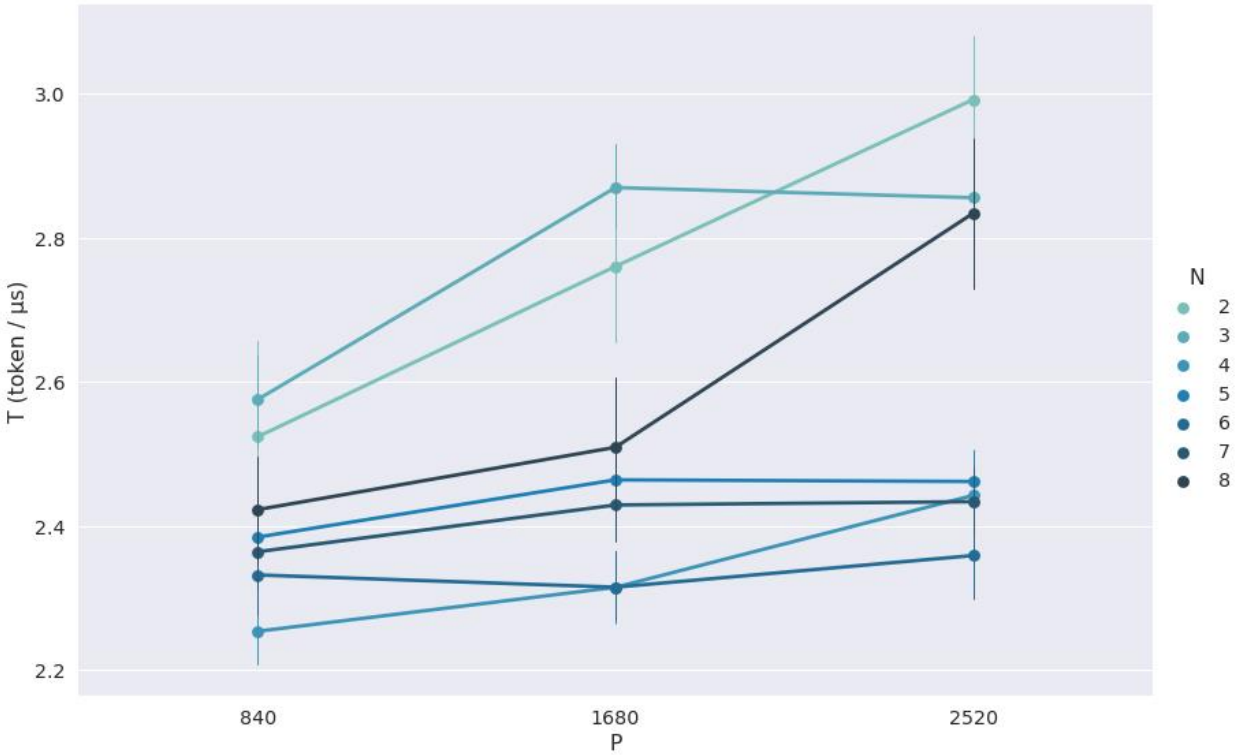

Рисунок 1d. Более высокие значения пропускной способности сети T и ее более стремительный рост по P соответствуют значениям N = {2,3,8}, в то время как значения N = {4,5,6,7} показывают более «стабильные» значения T.

LockBasedLinkedListTransporter + Modification

В этом варианте к предыдущей реализации добавляется небольшая модификация метода *push* у коллекции на узле-передатчике с целью снижения количества уведомлений (*signal*) потоков о наличии элемента в этой коллекции. Исследуется влияние данного фактора на рассматриваемые показатели качества сети.

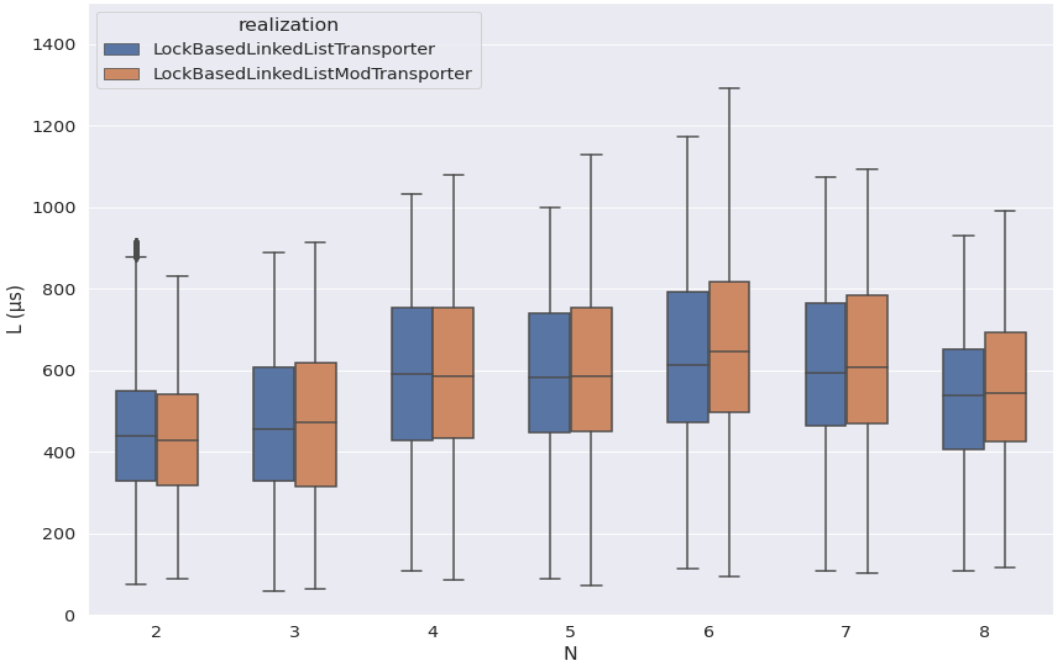

Рисунок 2а. Начиная с N=5 модификация показывает более высокую задержку по сравнению с оригиналом.

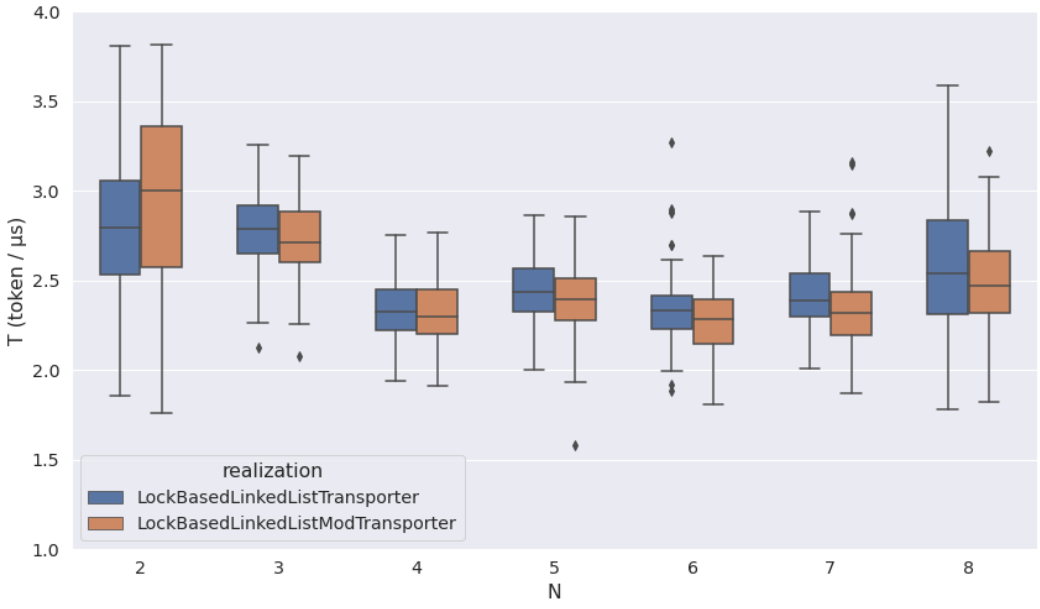

Рисунок 2b. На значении параметра N=2 модификация показывает заметно лучший результат, однако на значениях N > 2 постепенно деградирует по пропускной способности

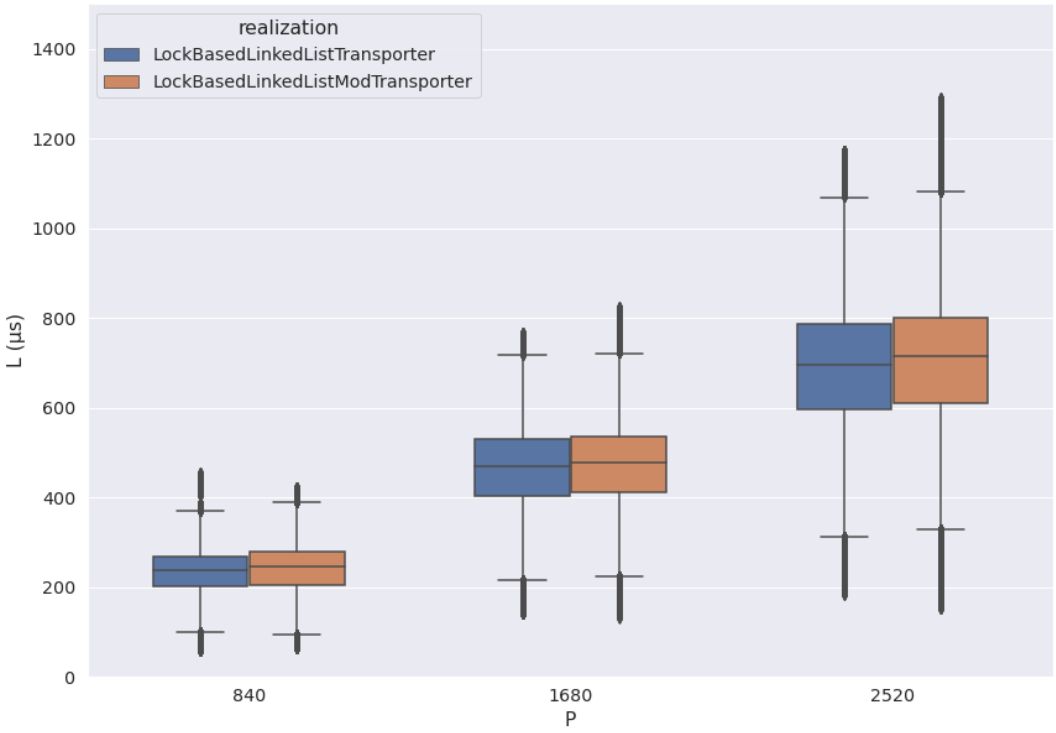

Рисунок 2c. Результаты реализаций практически не отличаются, на больших значениях параметра P модификация показывает чуть большую задержку L.

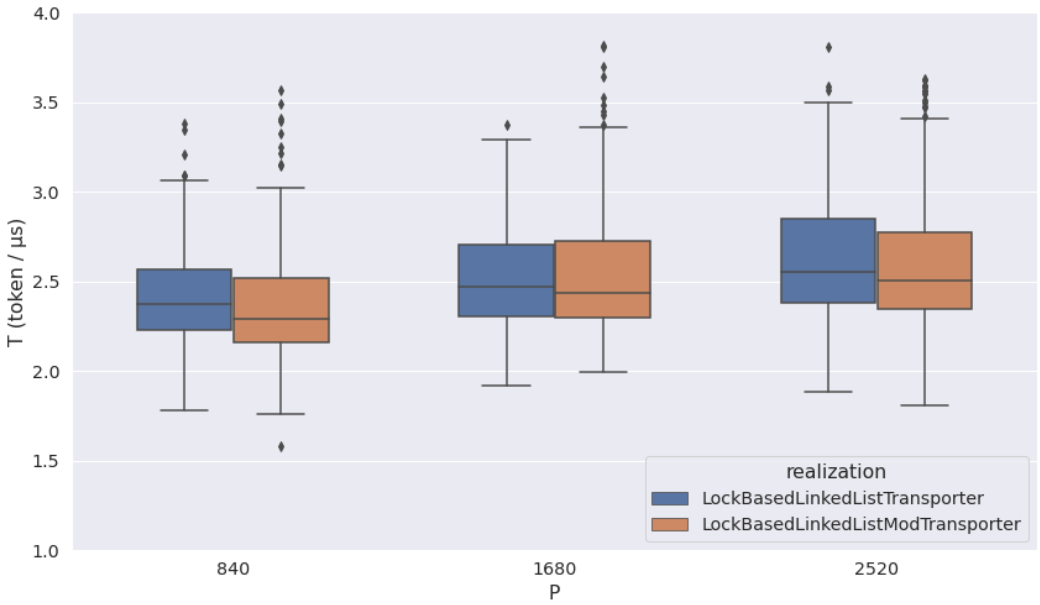

Рисунок 2d. У модификации наблюдается заметно более низкая пропускная способность T на N = {840,2520}.

Итог. Приведенные наблюдения на дают возможности утверждать, что модификация улучшает baseline по рассматриваемым параметрам L и T. Поэтому лучшей моделью остается *LockBasedLinkedListTransporter*.

LockBasedArrayDequeTransporter

Данная имплементация представляет собой неограниченную (с возможностью расширения) очередь на массиве. По документации она полагается более быстрой, чем связный список при использовании последнего в качестве очереди. Рассмотрим графики.

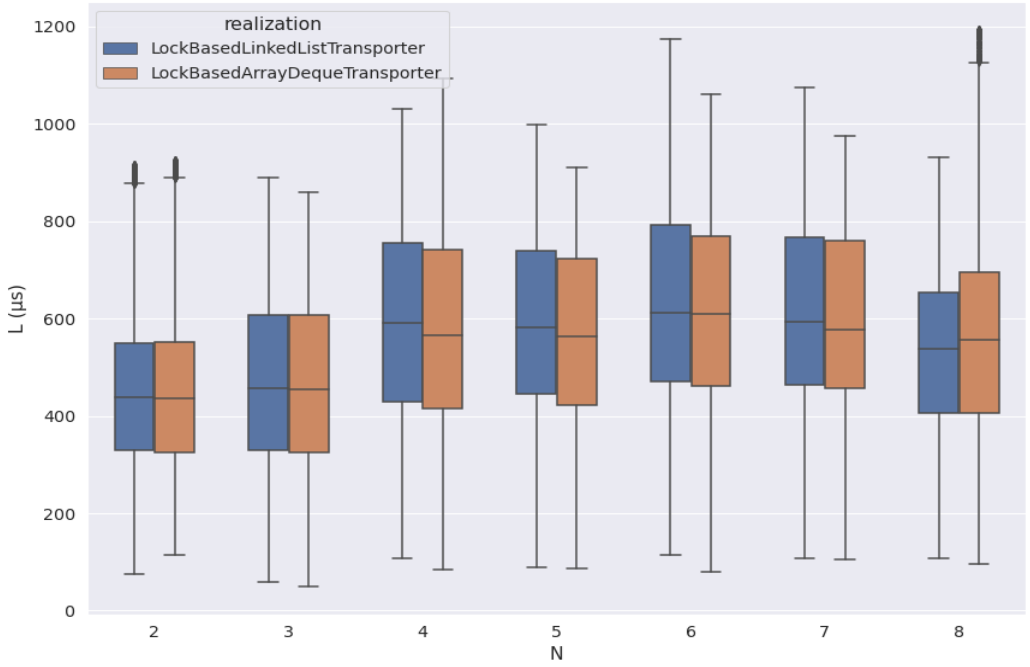

Рисунок 3а. На значениях параметра N = {4,5,6,7} реализация *ArrayDeque* показывает заметно более низкие значения задержки L.

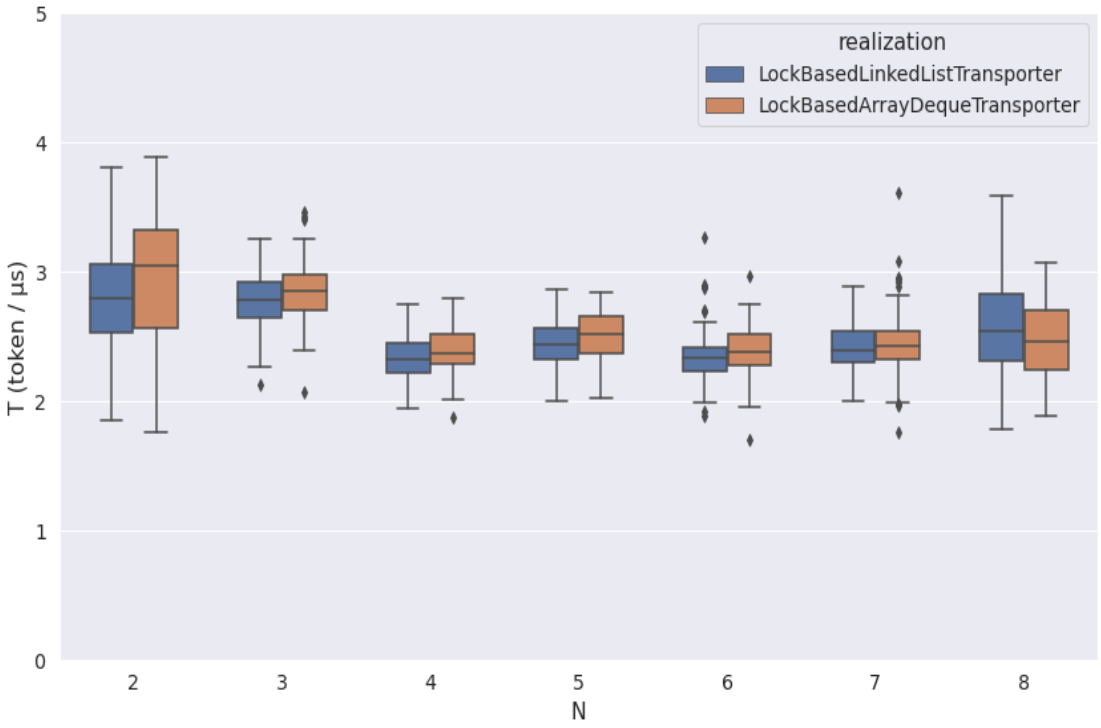

Рисунок 3b. Реализация *ArrayDeque* показывает стабильно более высокую пропускную способность T на всех значениях параметра N, за исключением N = {7,8}.

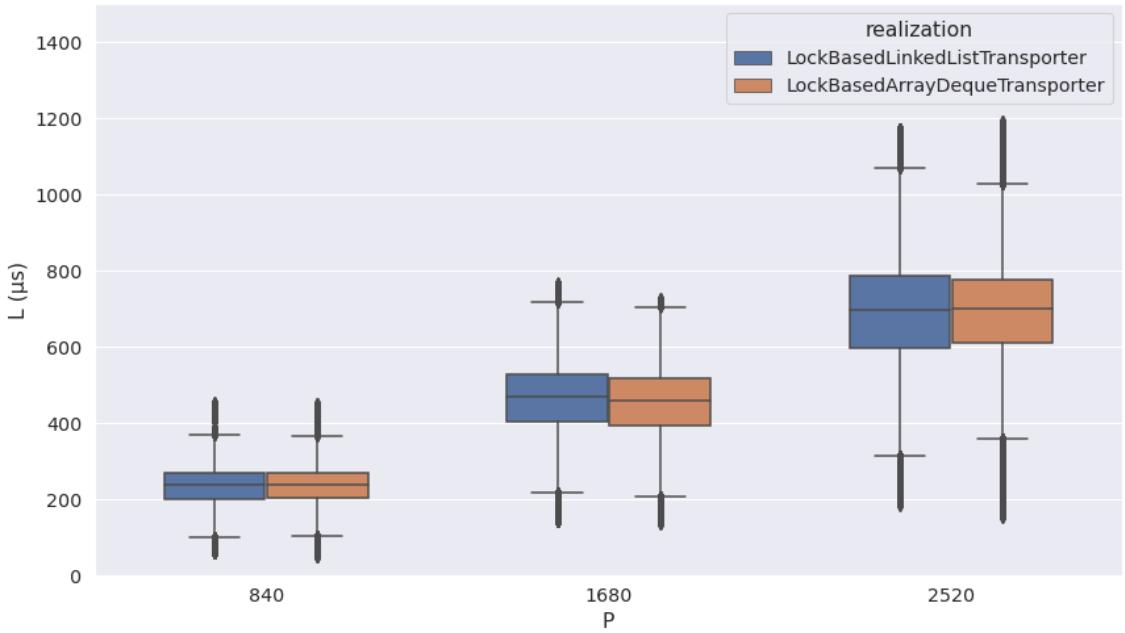

Рисунок 3c. Обе реализации показывают схожие результаты по параметру L в разрезе числа пакетов P.

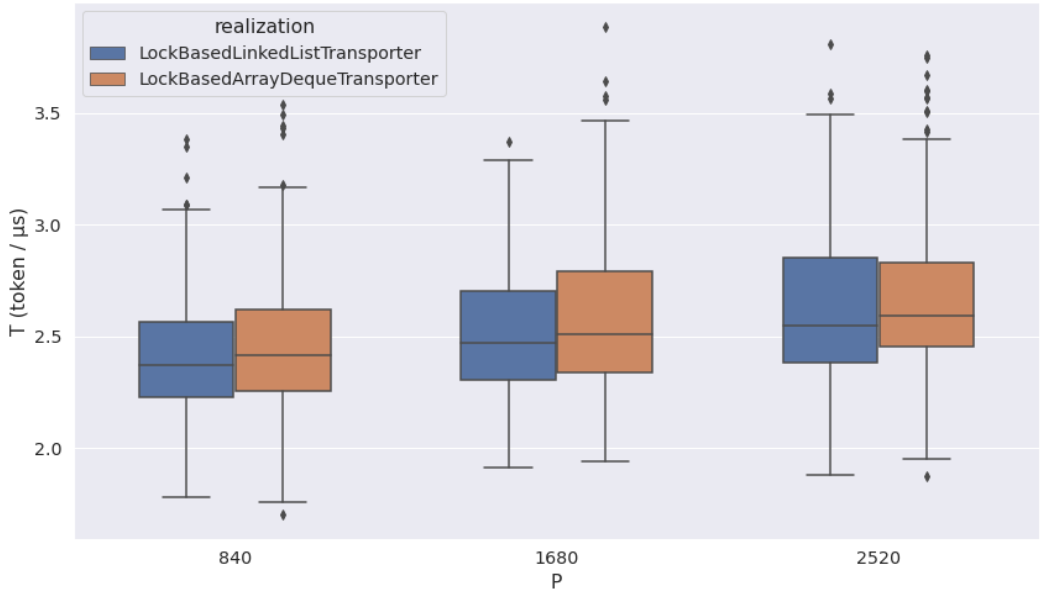

Рисунок 3d. Реализация *ArrayDeque* показывает более высокую пропускную способность T в разрезе числа пакетов P.

Итог. По приведенным наблюдениям можно сказать, что реализация *ArrayDeque* действительно оказывается лучше имплементации на связном списке по рассматриваемым параметрам N и T, улучшая результат в основном в контексте пропускной способности. Это связано с тем, что имплементация *LinkedList* имеет более ресурсоемкую операцию добавления нового элемента (расположение нового узла в не выделенном заранее месте на хипе), в то время как *ArrayDeque* имеет заранее выделенный массив с возможностью расширения при необходимости, что приводит к более компактному размещению элементов в памяти. Как следствие из архитектуры *LinkedList* стоит отметить закономерно большее количество cache miss и более активное поведение garbage collector (внутренние узлы становятся «eligible for collection» после совершения операции pop) при операции получения элемента.

Таким образом, лучшей моделью становится *LockBasedArrayDequeTransporter*.

3.3.2 Blocking

Группа реализаций узла-передатчика под общим названием *Blocking* представляет собой две имплементации Java-коллекций из пакета java.util.concurrent (блокирующая очередь на массиве *ArrayBlockingQueue* и блокирующая очередь на связном списке *LinkedBlockingQueue*).

Если две предыдущие реализации не имели ограничений на размер коллекции, то текущая имплементация *ArrayBlockingQueue* исследуется с параметром-размером, равным количеству пакетов *P* в сети. Реализация *LinkedBlockingQueue* используется в неограниченном варианте.

ArrayBlockingQueue

Данная имплементация очень похожа на рассмотренную ранее *ArrayDeque,* однако имеет ограничение на размер коллекции. Цель экспериментов – исследовать влияние расширения *ArrayDeque* на производительность *TokenRing* в сравнении с нерасширяемым вариантом очереди *ArrayBlockingQueue.*

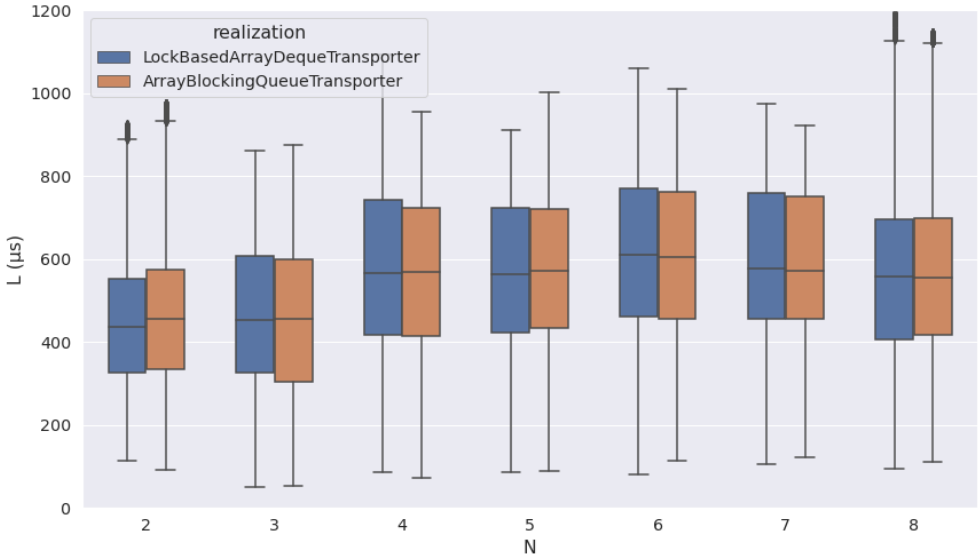

Рисунок 4a. Обе реализации показывают схожие результаты по параметру L в разрезе N.

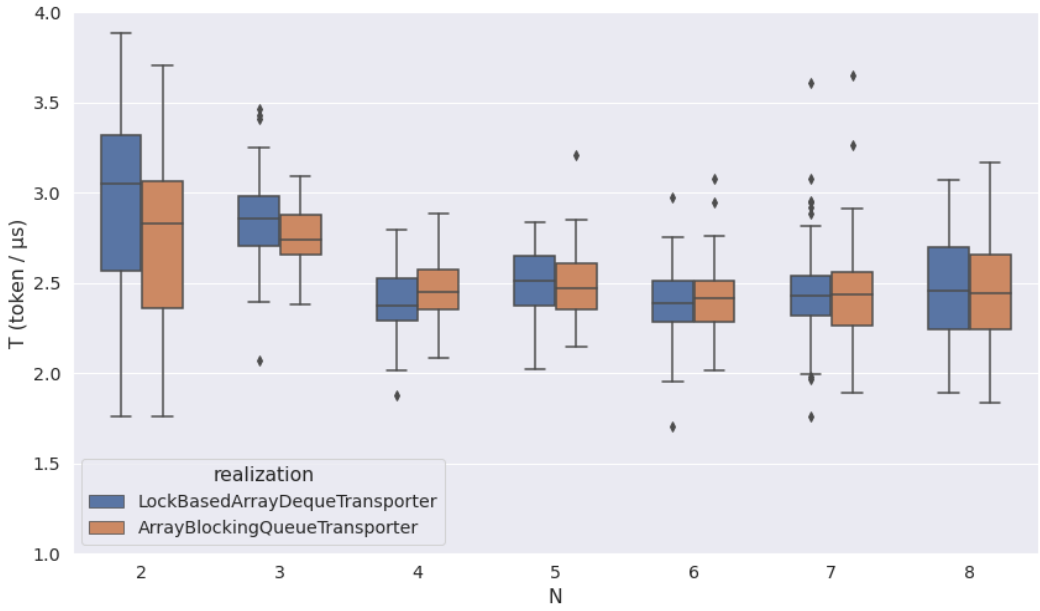

Рисунок 4b. При N = {2,3} реализация *ArrayBlockingQueue* заметно уступает *ArrayDeque* по пропускной способности T, при значениях N = 3 сокращает разрыв, при N = 4 оказывается лучше, но уже при N > 4 показатели реализаций примерно схожи.

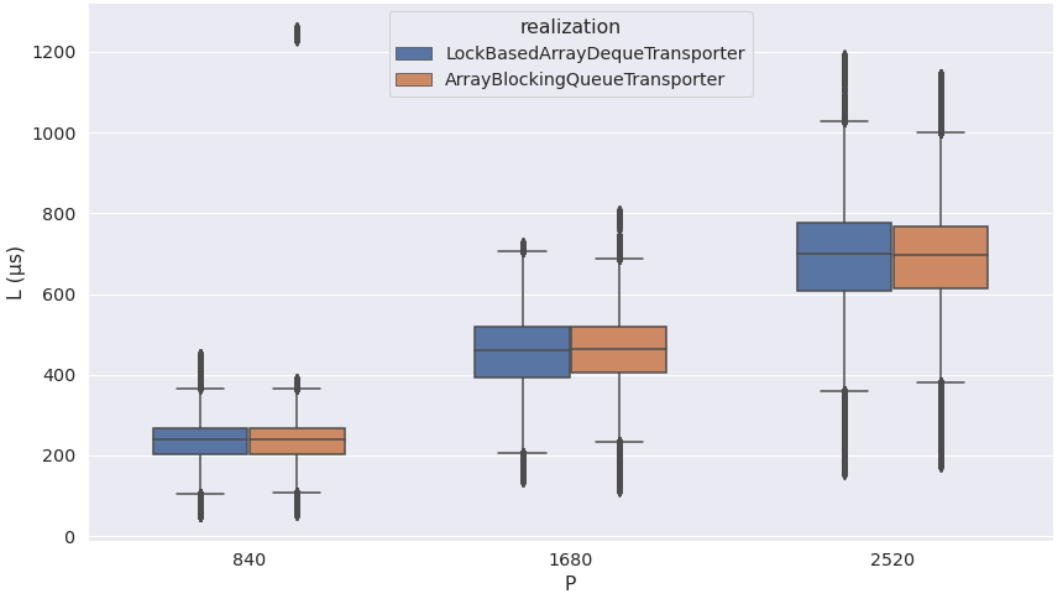

Рисунок 4c. Результаты двух реализаций практически не отличаются.

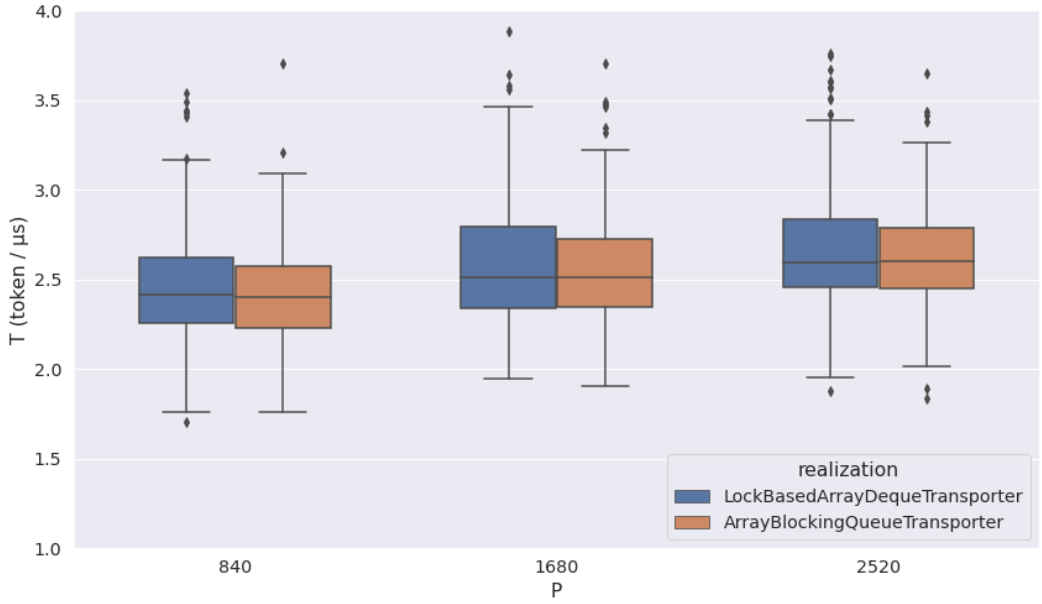

Рисунок 4d. Реализация *ArrayBlockingQueue* показывает чуть более стабильные результаты по пропускной способности (размер бока, «усов» боксплота), но медианы имеют примерно равные значения.

Итог. По приведенным наблюдениям можно сказать, что реализация *ArrayBlockingQueue* не улучшает в достаточной мере показатели предыдущей модели *LockBasedArrayDeque*. Поэтому лучшей моделью остается *LockBasedArrayDequeTransporter*.

LinkedBlockingQueue

Данная имплементация очень похожа на рассмотренную ранее *LinkedList,* однако в отличие от последнего имеет в основе односвязный список. Цель экспериментов – исследовать производительность реализации *LinkedBlockingQueue* на основе односвязного списка (без накладных расходов на излишнюю связность) относительно текущего лидера *ArrayDeque* – реализации очереди на массиве (с возможностью расширения).

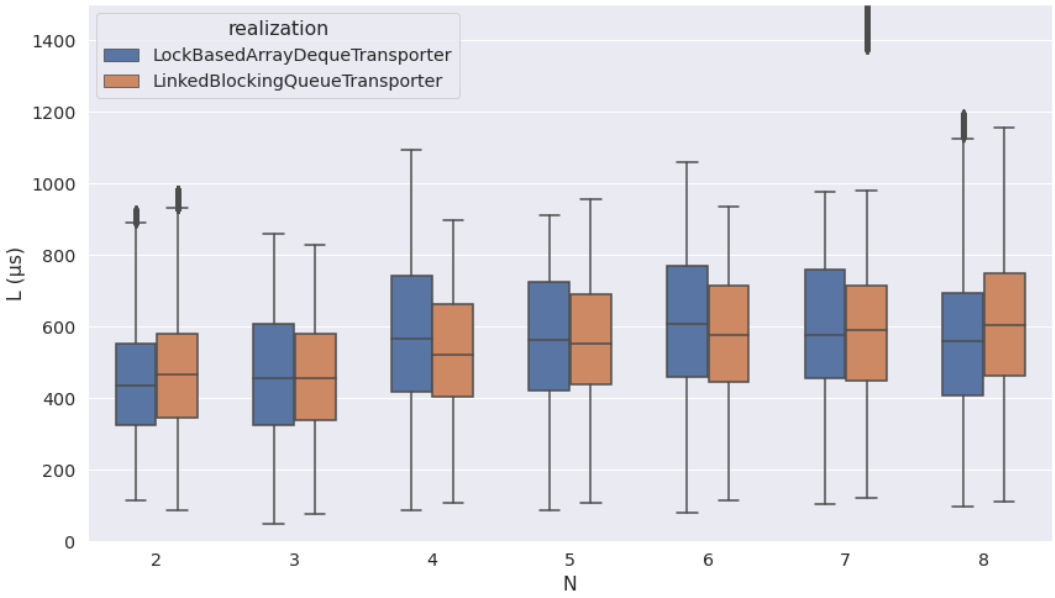

Рисунок 5а. Реализация *ArrayDeque* показывает меньшую задержку только на краевых значениях диапазона N = {2,8}, в остальных же случаях имплементация *LinkedBlockingQueue* достигает заметно более низкой задержки L.

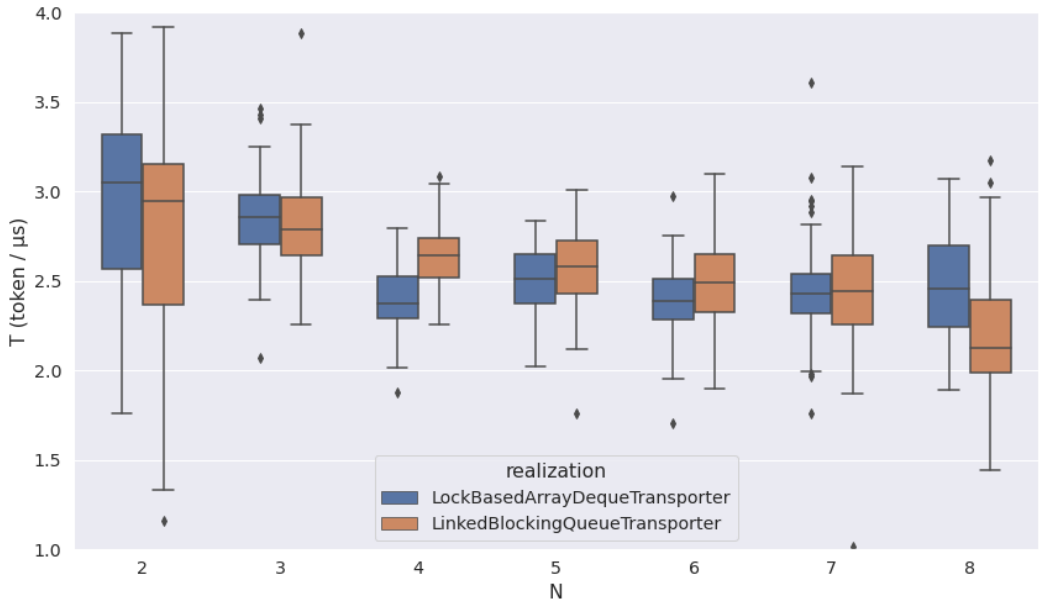

Рисунок 5b. Значения пропускной способности T на N = {2,3,8} демонстрируют лидерство модели *ArrayDeque*, в то время как остальные значения N = {4,5,6,7} отдают предпочтение реализации *LinkedBlockingQueue*.

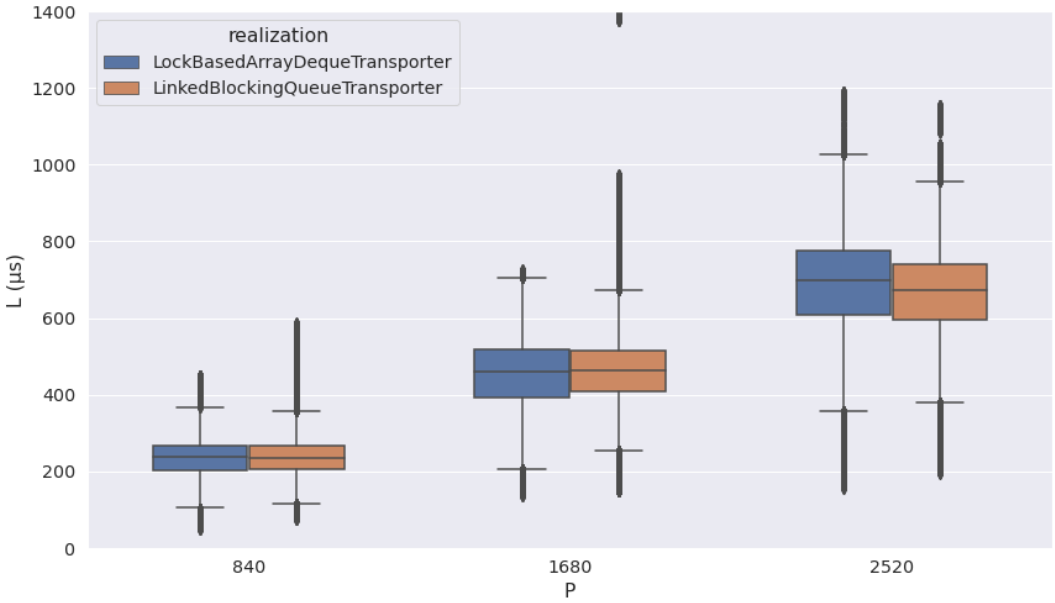

Рисунок 5c. Результаты моделей примерно одинаковы на P = {840, 1680}, при этом на большем числе пакетов P = 2520 реализация *LinkedBlockingQueue* показывает заметное снижение задержки L.

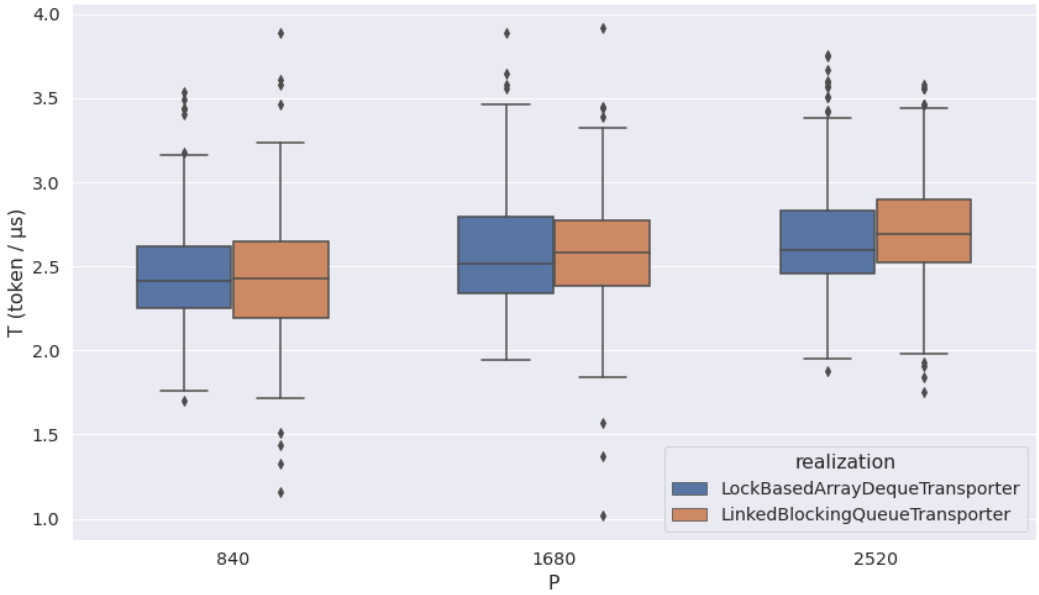

Рисунок 5d. Результаты моделей примерно одинаковы на P = {840, 1680}, при этом на большем числе пакетов P = 2520 реализация *LinkedBlockingQueue* показывает заметное увеличение пропускной способности T.

Итог. По приведенным наблюдениям можно сказать, что реализация *LinkedBlockingQueue* показывает лучшие результаты (по сравнению с *ArrayDeque*) по L и T на примерно середине диапозона N = {3 .. 7} и на большом числе пакетов P = 2520, то есть при достаточно высокой нагрузке. На остальном параметрическом пространстве, можно сказать, данная реализация не сильно уступает рассмотренной имплементации очереди на массиве, поэтому лучшей моделью становится *LinkedBlockingQueue*.

3.3.3 Сравнение реализаций

В целях составления более полной картины наблюдений стоит оценить все рассмотренные модели на одном графике относительно двух оцениваемых параметров L и T.

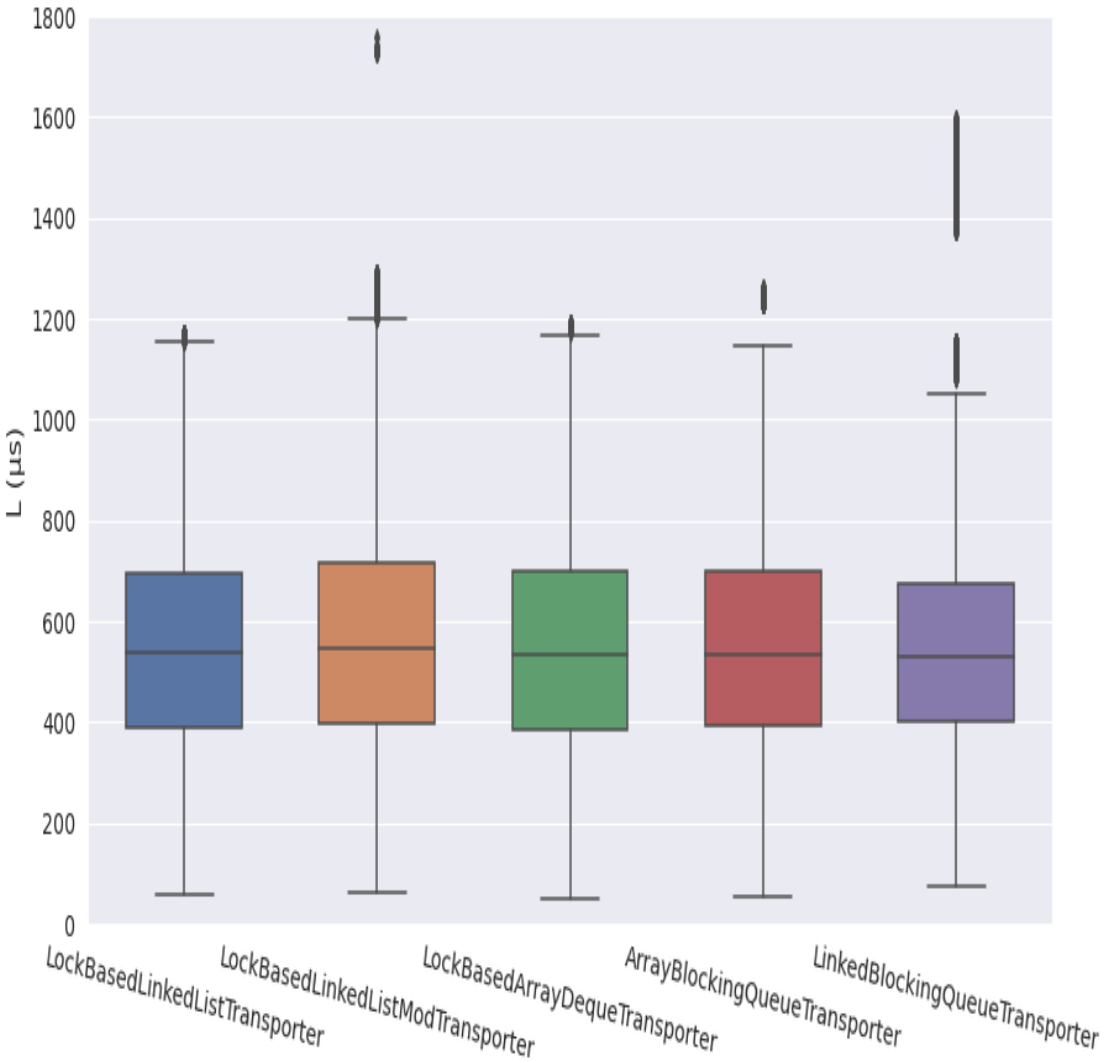

Рисунок 6а. Принимая во внимание значение медианы (чем ниже, тем лучше), следует выделить три последние реализации. Оценивая далее первую и третью квартиль (размер бокса), стоит признать лидером (по L) последнюю имплементацию.

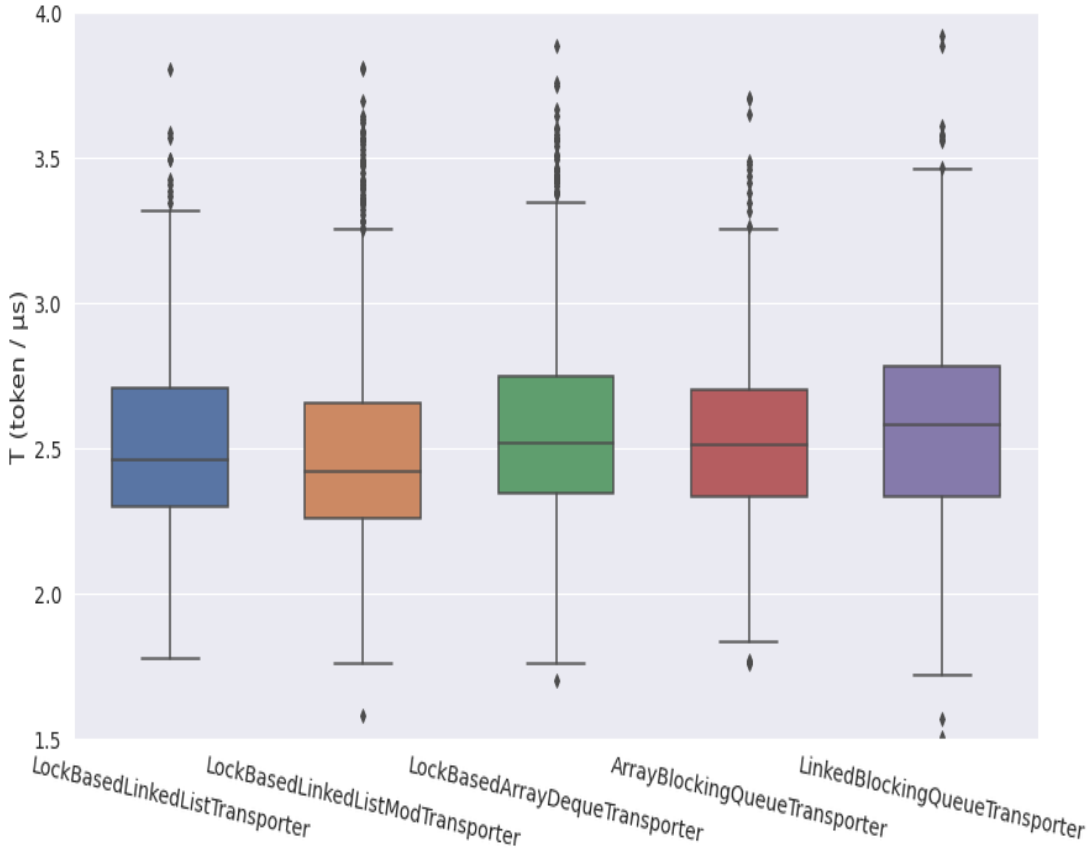

Рисунок 6b. Поступая аналогично предыдущему рисунку и принимая во внимание значение медианы, можно выделить в лидеры три последние реализации (медиана более T = 2.5 tok / µs). Далее, оценивая не только первую и третью квартиль (размер бокса), но и величину медианы вместе с размером интерквартильного размаха (IQR, «усы боксплота») стоит признать лидером (по T) последнюю имплементацию.

4 Заключение

Проделанную работу можно описать в следующих поинтах:

a. Была построена имплементация сетевого протокола TokenRing с узлами-обработчиками и узлами-передатчиками. Первые были предназначены для запуска на них потоков (общим количеством N), которые осуществляли передачу токенов в строго установленном порядке по часовой стрелке. Второй тип узлов был предназначен для хранения токенов (общим количеством P) и предоставления конкурентного доступа к ним посредством использования механизма блокировок (написанного в рамках данного проекта либо уже имплементированного в рамках используемой Java-коллекции).

b. Была подобрана и закреплена конфигурация для тестирования построенного TokenRing. Для количественной оценки качества реализации использовалась двумерная параметрическая сетка таких величин как Latency (**L**) и Throughput (**T**).

c. Было рассмотрено *5* реализаций узла-посредника (интерфейс *Transporter*): *3* имплементации с написанным в рамках данного проекта механизмом блокировки на доступ к коллекции токенов и *2* имплементации, где этот механизм уже есть. В список тестируемых реализаций вошли:

   - LockBasedLinkedListTransporter
   - LockBasedLinkedListModTransporter
   - LockBasedArrayDequeTransporter
   - ArrayBlockingQueueTransporter
   - LinkedBlockingQueueTransporter

d. Для всех рассматриваемых вариантов были построены графики, позволяющие выбрать лучшую имплементацию с точки зрения как минимизации задержки L, так и максимизации пропускной способности T.

e. В качестве лучшей реализацией в соответствии с описанными параметрами была выбрана последняя, LinkedBlockingQueueTransporter (блокирующая очередь на связанном списке).

Возможности для дальнейшего исследования:

- Расширение параметрической сетки N x P, увеличение числа ранов
- Параметризация размера коллекции (+ добавление ограничения на размер очереди в группе реализаций под общим названием LockBased)
- Отличное от равномерного начальное распределение токенов по узлам
- Использование имплементаций PriorityQueue и BlockingPriorityQueue, реализация интерфейса Comparable для класса токенов
- Использование (в каком-либо применимом варианте) неблокирующих коллекций (ConcurrentLinkedQueue, Disruptor)