УДК 004.94

ТЕХНОЛОГИЯ ИМИТАЦИОННОГО МОДЕЛИРОВАНИЯ И МЕТОДИКА СРАВНИТЕЛЬНОГО АНАЛИЗА АЛГОРИТМОВ СЕТЕВОГО ПЛАНИРОВАНИЯ

И.И. Труб (Москва)

I. Введение

Сетевые алгоритмы планирования (*network scheduler algorithms*) развиваются и активно исследуются на протяжении нескольких десятилетий. Их традиционной областью применения является маршрутизация пакетов в сетях, а целью – обеспечение задержки прохождения, не превышающей ограничений по уровню качества (*quality of service*). Но на самом деле эта область гораздо шире, и в качестве таковой может выступать любое адаптивное управление ресурсами (*adaptive admission control*), примером чего является организация очередей запросов и выборки из них в пуле потоков СУБД [14]. Первичное представление о таксономии таких алгоритмов можно получить из обзора [18].

Практически все алгоритмы так или иначе стремятся приблизить идеальную дисциплину обслуживания GPS (*Generalized Processor Sharing*). Эта дисциплина известна еще с конца 60-х годов, но ее исчерпывающий математический анализ, включая вычисление распределений числа заявок, периода занятости и времени пребывания, дал советский и российский математик С.Ф. Яшков в классической монографии [2]. Приведем данное им определение: «*Система обслуживает каждое из присутствующих требований с одинаковой скоростью, которая зависит от общего числа требований и равна 1/n, если число требований в текущий момент равно n. … Поступившее в систему требование сразу начинает обслуживаться (очередь в традиционном ее понимании отсутствует) и обслуживается с переменной скоростью до тех пор, пока его длина не станет равной нулю. В моменты поступлений новых требований или ухода обслуженных происходят скачки скорости обслуживания*». GPS еще трактуют как предельный случай дисциплины *Round Robbin* при размере кванта, стремящемся к нулю.

К настоящему времени предложено множество алгоритмов, приближающих в той или иной степени GPS, для каждого из которых построена математическая модель, дающая теоретические оценки этого приближения. Улучшения и модификации продолжают появляться даже непосредственно для *Round Robbin* [16], но мы перечислим те алгоритмы, которые используются в современных системах и реализованы в ядре Linux. Типичным атрибутом этих алгоритмов является виртуальное время (*virtual time*). Это моделируемое время, которое протекало бы в случае обслуживания заявок в соответствии с идеальной дисциплиной GPS с учетом их приоритетов. Каждый логический процесс в моделируемой системе имеет свое виртуальное время, которое увеличивается при обработке событий. Подробнее мы обсудим его использование при моделировании в секциях 2 и 3. Простейшим алгоритмом является *SFQ* (*Start Time Fair Queueing*), предложенный в [7], согласно которому на обслуживание из очереди выбирается заявка с минимальным виртуальным временем начала обслуживания. Программная реализация *SFQ* описана в [12]. Давно и хорошо изучен алгоритм *WFQ* (*Weighted Fair Queueing*), применение которого описано в [10] и [11]. В [4] был предложен WF^2Q (*Worst-case Fair Weighted Fair Queueing*), а его модификация - WF^2Q+ в [3] и [6]. Реализация WF^2Q+ описана в [8], а его адаптивное расширение для смешанного трафика – в [15]. В [5] предложен наиболее быстрый из всех алгоритмов - *QFQ* (*Quick Fair Queueing*), в котором

операция поиска минимума, так или иначе присутствующая во всех алгоритмах, заменена с некоторым ущербом в точности быстрыми операциями на битовых строках. Наконец, в [13] описан алгоритм *EEVDF* (*Earliest Eligible Virtual Deadline First*). [9] рассматривает аспекты практической реализации этих алгоритмов в современном ядре Linux.

Такое разнообразие алгоритмов сетевого планирования не может не вызывать естественного вопроса – как сравнивать их эффективность и какой же алгоритм лучше всех прочих? Перечисленные работы содержат достаточно глубокий аналитический анализ, но авторы «своего» алгоритма, как правило, приходят к выводу, что их алгоритм лучше, и это интересно было бы проверить каким-либо общим методом моделирования, в котором конкретный алгоритм явился бы в некотором смысле подставляемой величиной. Заметим, что существует множество работ (например, [1]), в которых алгоритмы сравниваются на частных сетях с конкретным трафиком. Данная работа представляет попытку создать и проверить общий метод имитационного моделирования алгоритмов.

II. Среда моделирования

Предложенная модель максимально абстрагируется от каких-либо технических особенностей системы, в которой применяется алгоритм сетевого планирования, и сосредоточена на моделировании самого алгоритма. Идея заключается в размещении заявок в очередях узлов-листов многоуровневого дерева, в котором каждый внутренний узел (не являющийся листом) управляет своим поддеревом, реализуя для него моделируемый алгоритм, и на его основе вычисляет, из какого дочернего узла должна быть выбрана на обслуживание следующая заявка. Этот дочерний узел на основании анализа состояния уже своего поддерева выбирает свой дочерний узел до тех пор, пока процесс не дойдет до узла-листа, из обычной очереди в котором уже непосредственно выбирается стоящая первой реальная заявка. Похожая модель была рассмотрена в [3], где для моделирования алгоритма WF^2Q+ использовалась иерархическая система узлов, в которой траффик распределяется в соответствии с весом узла. В данной работе предлагается ее обобщение, формализация и реализация на основе современных средств программирования и измерения производительности. Иллюстрация модели представлена на рис. 1.

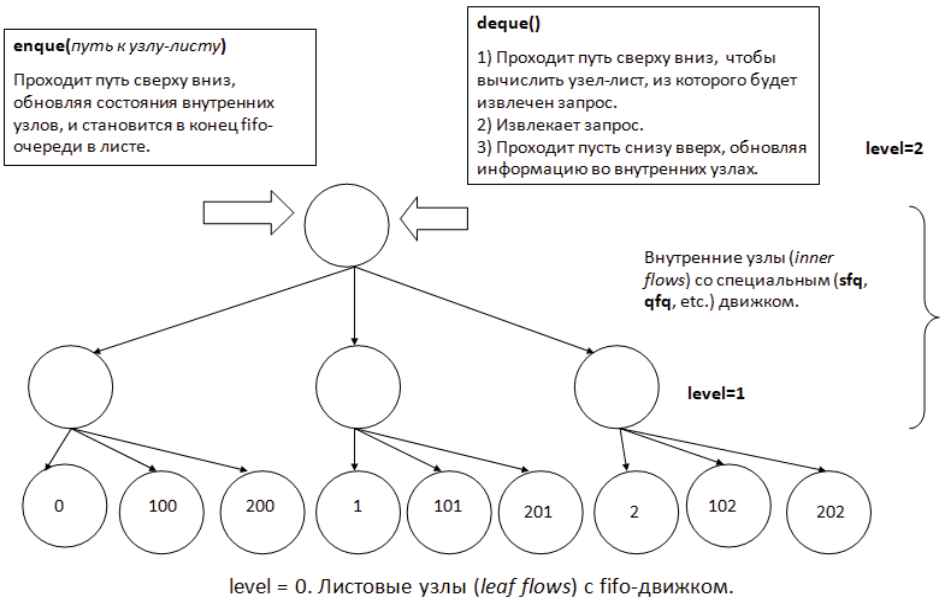

Рис. 1. Схема среды моделирования

Основными параметрами модели являются: *N* – количество уровней иерархии, *K* – количество потомков каждого внутреннего узла, функции распределения длин и весов заявок и, разумеется, используемый алгоритм выборки заявок из очередей. Модель легко распространяется на переменное значение *K*, здесь мы рассматриваем постоянное для упрощения изложения. Количество узлов-листьев, в которых расположены реальные очереди заявок, равно K^N. Каждому же из внутренних узлов придается свой объект (подробнее об этом – в секции III), который реализует используемый алгоритм, перевычисляя после каждого события прихода/ухода заявки значение виртуального времени и контекст алгоритма. Возникает вопрос: если мы хотим поставить заявку в *i*-ый узел-лист, как вычислять ее путь сверху вниз, чтобы «попасть» в нужный лист? Для этого каждому значению $i= 0.. K^N -1$ сопоставляется некое числовое значение *path*, вычисляемое по следующему алгоритму :

Листинг 1.

```
long int getPath(int i) {
  long int path = 0; int ostatok = i;
  for (int j = 0; j < N; j++) {
    path += (ostatok / K^(N-j-1)) * m^j; //m – параметр модели, m > K
    if (j < N-1) { ostatok = ostatok % K^(N-j-1) ; }
  }
  return path; }
```

Далее, каждому значению пути сопоставляются две функции:

head(path) = path % m; tail(path) = path / m;

Для схемы на рис. 1 *N=2, K=3, m=100*, а в узлах-листьях подписаны их значения *path*. Возьмем последний узел с индексом 8, для которого рассчитанный *path*=202. *head*(202)=2, следовательно, в соответствии данными поступившей в узел 8 заявки обновляется информация в третьем по счету потомке начального узла, а *path* меняет свое значение на *path=tail(path)=2*. Далее *head(path)=2*, следовательно, дальше информация обновляется в третьем по счету потомке предыдущего узла первого уровня, которым и является лист с индексом 8. Так это работает для произвольного числа уровней модели.

На примере алгоритма *SFQ* рассмотрим логику работы ключевых методов *enque* и *deque*, т.к. состояние этого алгоритма наиболее просто и описывается только одним значением – минимальным для узла виртуальным временем *S* начала обслуживания следующей заявки. Оба эти метода реализуют рекурсию, сводясь в листовых узлах к простейшим операциям *push* и *pop* для обычной *fifo*-очереди.

void enque(int path, T *request) {

- выбираем у текущего узла потомок flow с индексом head(path);
- пересчитываем для узла flow значение S и обновляем виртуальное время;
- для узла flow делаем вызов метода push(tail(path), request), который реализован как вызов enque уже для объекта, реализующего алгоритм сетевого планирования в узле flow;

}

T *deque() {

- если узел пуст – возвращаем nullptr;
- выбираем дочерний узел min_flow с минимальным значением S;
- извлекаем заявкуT *rmin из этого узла, чье виртуальное время начала обслуживания равно S,

с помощью операции T *rmin = min_flow->pop(); этот вызов реализован как вызов deque уже для объекта, реализующего алгоритм сетевого планирования в узле min_flow;

- пересчитываем для узла min_flow значение S с учетом длины и веса извлеченной заявки;
- пересчитываем виртуальное время в узле;
- возвращаем rmin в качестве результата;

}

III. Особенности реализации

Описанная в секции II среда моделирования реализована на языке C++. Классы, реализующие алгоритмы сетевого планирования, наследуют абстрактный класс *BaseQueue*, который описывает интерфейс, единый как для этих классов, так и для обычных очередей в узлах-листьях. Класс *Flow* реализует узлы дерева. Все эти классы представляют собой шаблоны, параметризуемые именем класса *T*, описывающим заявки. Объектный дизайн реализации моделирующей среды изображен на рис. 2. Класс с условным названием *EngineQueue* реализует алгоритм и носит соответствующее название, например *SFQ* или *WF2Q_plus*.

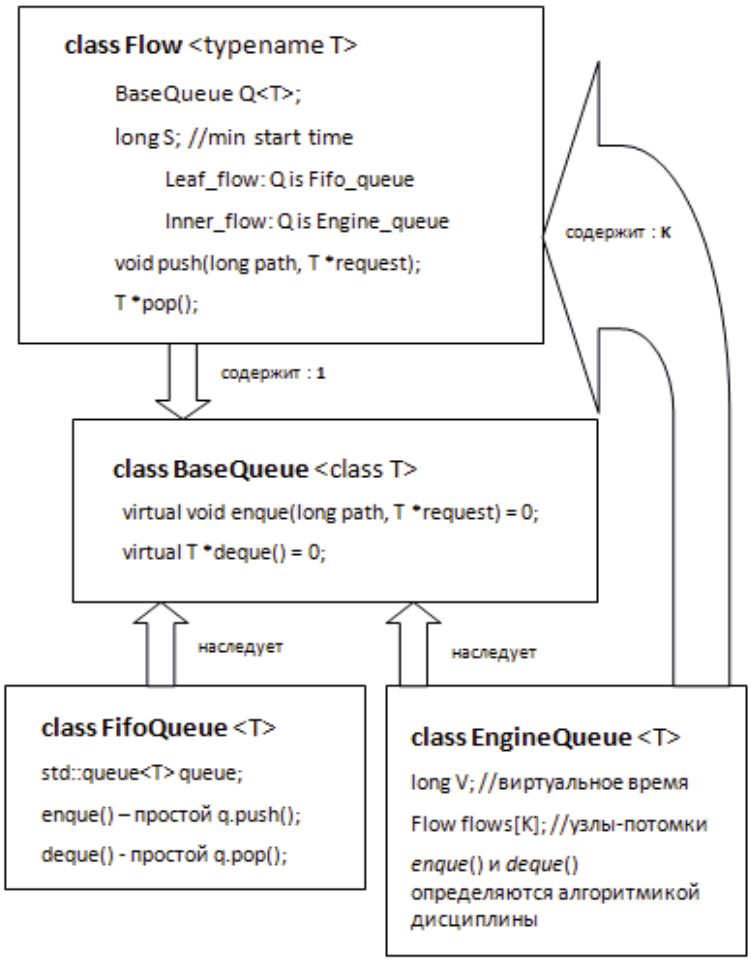

Рис.2. Объектный дизайн среды моделирования

Остановимся на важных функциях фреймворка, не изображенных на схеме, в частности, на технологии измерений. Упрощенный код непосредственно профилируемой функции показан в листинге 2.

Листинг 2.

```
long long int build_function() {
  long long int ops = 0; //счетчик операций
  for (int i = 0; i < iterations; i++) {
    for (int z = 0; z < M; z++) { //сколько заявок размещаем в очереди каждого листа
      for (int j = 0; j < leaves; j++) { //количеств листьев, K^N
         q->enque(j, requests[j]); //занесение заявки в очередь j-го листа, проход сверху вниз
            ops++; }  }
    for (int j = 0; j < M * leaves; j++) { //извлечение всех заявок из очередей узлов-листьев
      q->deque();
      ops++; }  }
  return ops; //возвращаем количество операций
}
```

Здесь *q* – корневая очередь *EngineClass* дерева, *iterations* – параметр модели, управляющий числом повторений при одних и тех же исходных данных для получения статистически устойчивого результата.

Листинг 3 содержит главную функцию *Google Benchmark*, управляющую измерениями.

Листинг 3.

```
    static void BM_Sfq(benchmark::State& state) {
      long long int total_items,  ops;
      N = state.range(0); //число уровней
      K = state.range(1); //число потомков внутреннего узла
      M = state.range(2); //начальная длина очереди в каждом листе
      q = build_init_recursive(0, "0"); //инициализация дерева
      build_init_other(); //предварительное вычисление всех постоянных данных, выделение
памяти
      for (auto _ : state) { //профилирование…
         ops = build_function();
        total_items += ops; }
      build_deinit(); //обнуление данных, освобождение памяти
      q.reset(); //освобождение памяти, выделенной под дерево
      state.SetItemsProcessed(total_itetms); //обработка собранной статистики
    }
```

Опишем, каким образом модель управляет весами и длинами заявок, т.к. этого нет на рис.2. Как у узлов, так и у очередей есть два свойства: *lmax* (максимальная длина заявки) и *weight* (вес). У объекта *Flow*, который является листом, они вычисляются так:

lmax = 1000 +getUniform(1000);

weight = 1+getUniform(512);

что соответствует равномерному распределению *lmax* от 1000 до 2000 и *weight* от 1 до 512. Разумеется, таким образом можно вычислять *lmax* и *weight* согласно любому закону распределения вероятностей, который является параметром модели. Если же *Flow* – внутренний узел, то его параметры берутся из соответствующих параметров связанной с ним очереди *EngineClass*. В свою очередь, эти параметры у очереди вычисляются как максимальные значения по связанному с ней массиву *flows*. Наконец, длина конкретной заявки вычисляется как равномерно распределенная случайная величина в диапазоне от 1 до *lmax* узла-листа, к которому она стала в очередь.

К реализации алгоритмов предъявляется требование предельной оптимизации, в противном случае один алгоритм может показать лучший результат за счет того, что другой неудачно реализован, а не за счет того, что он лучше сам по себе. В связи с этим при реализации применяются следующие приемы:

- все данные, которые являются постоянными для всех экспериментов, вычисляются заранее;
- минимумы вычисляются не вручную, а с помощью высокооптимизированных библиотечных контейнеров C++, например, *std::priority_queue* или *boost::heap::d_ary_heap*, которые сами всякий раз «выталкивают» наверх минимальный элемент;
- т.к. в случае пустой очереди в узле в методе *enque()* не требуется проход сверху вниз (заявка сразу ставится в *fifo*-очередь листа), адреса очередей в листах можно заранее записать в некоторый контейнер – *std::vector* или *std::unordered_map*, из которых можно сразу получить адрес *i*-го по счету листа;
- стороннее рецензирование кода экспертами высокой квалификации.

IV. Результаты

На данный момент в авторском фреймворке надежно отлажены два алгоритма: *SFQ* и *QFQ*. По своему контенту они не имеют практически ничего общего: *SFQ* наиболее прост, но наиболее медленен, т.к. требует постоянного пересчета минимумов; *QFQ* в реализации сложен, но в нем все вычисления основаны на операциях с битовыми строками, что делает его предельно быстрым, но и менее точным, т.к. объединение заявок в группы по признаку относительной близости характеристик стирает отличия между ними.

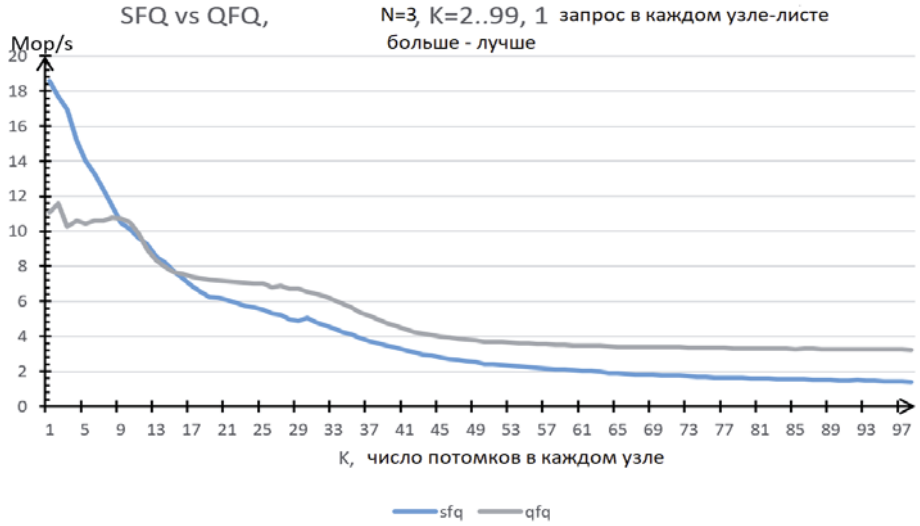

Рис. 3. Сравнение производительности SFQ и QFQ

Заметим, что в [17] предложена фактически предельная реализация этого подхода под названием *QFQ+*, но она применима не всегда, поэтому мы остановились на *QFQ*. За подробностями опять-таки отсылаем читателя к [5], а здесь рассмотрим результаты моделирования.

Из рис. 3 мы видим, что при малых *K SFQ* дает лучшие результаты, начиная же с какого-то значения *K* проявляется преимущество *QFQ*. Это логично, т.к. при малых *K* простота *SFQ* «перевешивает» издержки *QFQ* по формированию сложных структур данных.

Сравним теперь обе дисциплины не по количеству миллионов операций в секунду, а по критерию честности (*fairness*), когда заявки с бОльшим весом и меньшей длиной должны выбираться раньше и, соответственно, иметь меньшую задержку. Взвешенная задержка при моделировании вычисляется следующим образом:

$$L_w(r) = order * \frac{r.weight}{r.length}, \text{ где}$$

- *order* – номер итерации в *deque*()-цикле, на которой заявка *r* извлекается для исполнения в соответствии с используемой дисциплиной;
- *r.weight* и *r.length* – соответственно вес(приоритет) и длина заявки *r*.

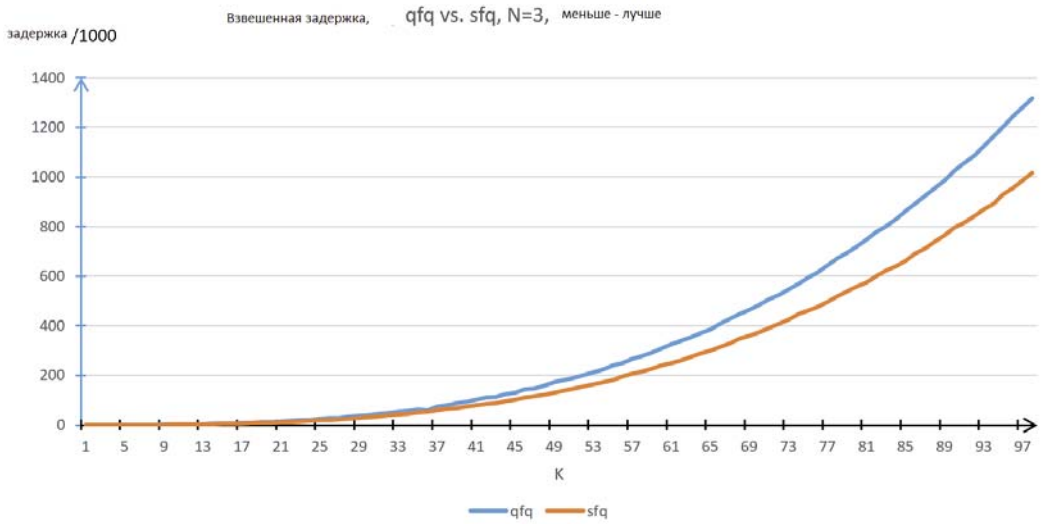

Рис. 4. Сравнение честности SFQ и QFQ

Рис.4 показывает лучшую честность алгоритма *SFQ*, т.к. ускорение *QFQ* достигается за счет потери точности – менее приоритетная, но близкая по весу и длине заявка, может попасть в одну группу с более приоритетной и опередить ее, т.к. в группе заявки становятся полностью равноправными. *QFQ* быстрее, но *SFQ* точнее. Кроме того, анализ результатов для *N*=2 и *N*=3 позволяет предположить следующую зависимость:

$$L_w \approx C(N) * K^N,$$

где *C* зависит от *N* и функций распределения веса и длины и может быть назван *коэффициент честности* (*fairness coefficient*). Чем он меньше, тем лучше. Рис. 5 говорит в пользу гипотезы независимости *C* от *K*. После некоторой области неустойчивости при малых *K* кривая ведет себя как случайная величина с очень небольшой дисперсией, практически как константа. В силу этой гипотезы коэффициент честности может

рассматриваться в качестве оценки честности алгоритма в целом. Возможно, что этот коэффициент не зависит и от *N*, что повышает его описательную силу.

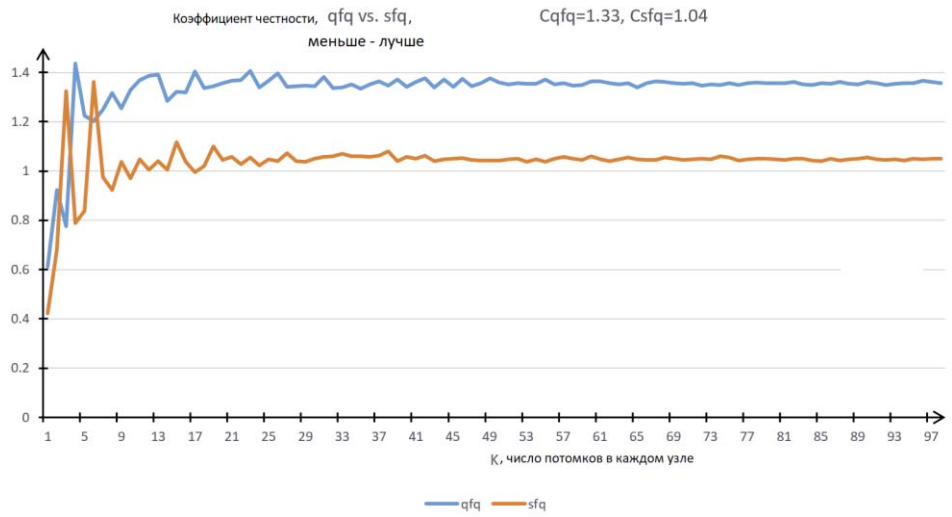

Рис.5. Сравнение коэффициента честности SFQ и QFQ, N=3

В качестве единой характеристики алгоритма можно предложить смешанный критерий – *perf(K) / C(K)*, и чем он выше – тем лучше (рис. 6 и 7). Для *N*=2 алгоритмы равноценны. При *N*=3 повторяется поведение для производительности: *SFQ* лучше при малых значениях *K*, а, начиная с некоторого значения, лучше становится *QFQ*.

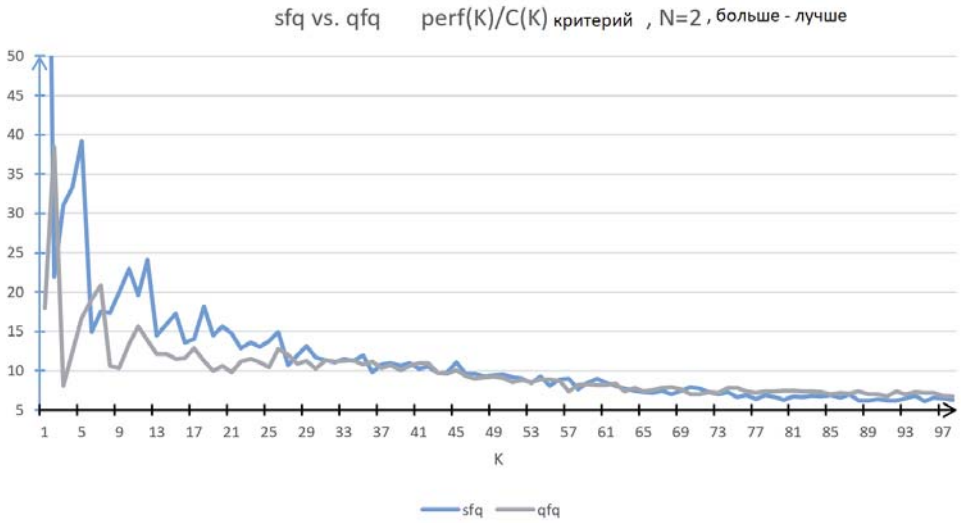

Рис.6. Смешанный критерий, N=2

V. Выводы

1) Предложенные в совокупности технология моделирования и критерии оценки эффективности позволяют в перспективе сравнить любые существующие и будущие алгоритмы сетевого планирования на широком множестве конфигураций.

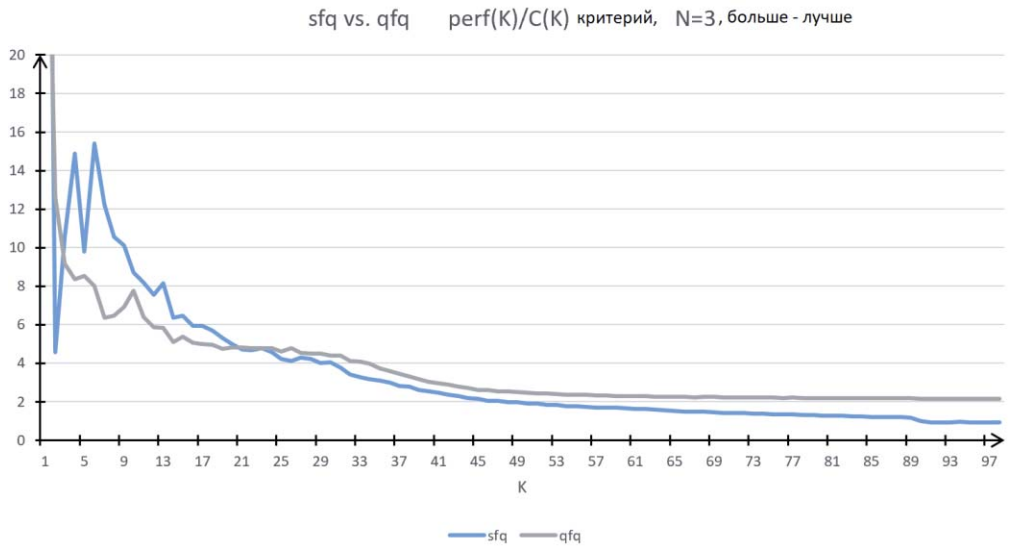

Рис.7. Смешанный критерий, N=3

2) Алгоритмически более простые дисциплины показывают худшую производительность, но превосходят более сложные по критерию честности.

3) Вместе с тем выигрыш в производительности для алгоритмически более сложных дисциплин проявляется только для конфигураций достаточно больших размерностей, т.к. при малых размерностях издержки реализации сложного алгоритма нивелируют его более высокие теоретические оценки производительности.

4) Введение смешанного критерия делает реализацию алгоритмов содержательной, т.к. позволяет поставить задачу о выборе наилучшей дисциплины. Вряд ли какой-то алгоритм сумеет превзойти *QFQ* по производительности, но мы ничего не можем сказать о наилучшей дисциплине по совокупному смешанному критерию производительности и честности. Однако интуиция подсказывает, что «наилучшей» во всех отношениях дисциплины, которая явным образом превосходила бы все остальные, не существует.

Благодарности

Автор выражает благодарность эксперту по СУБД MySQL Сергею Глущенко за консультации по реализации алгоритмов и рецензирование программного кода.

Литература

1. **Рыжих С.В., Марыкова Л.А., Марыков М.В., Лихтциндер Б.Я., Клитероу Ш.** Сравнение алгоритмов построения очередей при обеспечении качества обслуживания // T-Comm: Телекоммуникации и транспорт. 2017. №11. С. 74-79.
2. **Яшков С.Ф.** Анализ очередей в ЭВМ. – М. : Радио и связь, 1989. 215 с.
3. Bennett J., Zhang H. Hierarchical packet fair queueing algorithms. - IEEE/ACM Transactions on Networking, vol. 5, no. 5, pp. 675-689, Oct. 1997. https://www.cs.cmu.edu/~hzhang/papers/TON-97-Oct.pdf.
4. **Bennett J., Zhang H.** WF2Q : Worst-case Fair Weighted Fair Queueing. – INFOCOM'96: Proc. of the 15-th joint conference of the IEEE computer and communications societies. – Vol. 1, pp. 120-128. https://www.cs.cmu.edu/~hzhang/papers/INFOCOM96.pdf.
5. **Checconi F., Rizzo L., Valente P.** QFQ: Efficient Packet Scheduling With Tight Guarantees. – IEEE Transactions on Networking, vol. 21, no. 3, pp. 802-816, June 2013. https://docenti.ing.unipi.it/~a007834/papers/20121017-qfq-ton.pdf.

6. **Ciulli N., Giordano S.** Analysis and simulation of WF2Q+ based schedulers: comparisons, compliance with theoretical bounds and influence on end-to-end delay jitter. – Computer Networks, volume 37, issue 5, November 2001, pp. 579-599. https://dl.acm.org/doi/abs/10.5555/646463.693747.
7. **Goyal P., Harrick M., Cheng V.H.** Start-time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. – ACM SIGCOMM Comp. Comm. Rev., Volume 26, Issue 4, 1996, pp. 157-168. https://dl.acm.org/doi/10.1145/248157.248171.
8. **Dwekat Z., Rouskas G.N.** A Practical and Efficient Implementation of WF2Q+. – Computer Networks Volume 55, Issue 10, 14 July 2011, pp. 2392-2406. https://www.sciencedirect.com/science/article/abs/pii/S1389128611001320.
9. **Isstaif A.** Towards Latency-Aware Linux Scheduling for Serverless Workloads. – SESAME '23: Proceedings of the 1st Workshop on Serverless Systems, Applications and Methodologies, pp. 19-26. https://dl.acm.org/doi/10.1145/3592533.3592807.
10. **Mawlood M.A., Mahmood D.A.** Performance Analysis of Weighted Fair Queuing (WFQ) Scheduler Algorithm through Efficient Resource Allocation in Network Traffic Modeling. – Journal of communications software and systems, VOL. 20, NO. 3, SEPTEMBER 2024, pp.266-277. https://hrcak.srce.hr/file/464656.
11. **Mayer A., Teylo L., Boito F.** Implementation and Test of a Weighted Fair Queuing (WFQ) I/O Request Scheduler - [Research Report] RR-9480, Inria. 2022. https://inria.hal.science/hal-03758890v2/document.
12. **Mitchell D., Yeung J.** Implementation of Start-Time Fair Queuing Algorithm in Opnet. – Proceedings IEEE. SIGCOMM'96, August, 2002, 57 p. http://www.cs.sfu.ca/~daryn/personal/school/885/.
13. **Stoica I. Abdel-Wahab H.** Earliest Eligible Virtual Deadline First : A Flexible and Accurate Mechanism for Proportional Share Resource Allocation. – (Technical report). CS Dpt., Old Dominion Univ. TR-95-22, 1995.https://dl.acm.org/doi/10.5555/890606.
14. **Trub Ilya.** Adaptive Thread Pool: Improving MySQL Scalability With AI (December, 2021) https://mysqlperf.github.io/mysql/adaptive-thread-pool.
15. **Valente P.** Extending WF^2 Q+ to Support a Dynamic Traffic Mix. – First Int. Workshop on Adv. Architectures and Algorithms for Internet Delivery and Apps. (AAA-IDEA'05), Orlando, FL, USA, 2005, pp. 26-33. https://ieeexplore.ieee.org/document/1652333.
16. **Valente P.** Providing Near-Optimal Fair-Queueing Guarantees at Round-Robin Amortized Cost. – 2013 22nd Int. Conf. on Comp. Comm. and Networks (ICCCN), Nassau, Bahamas, 2013, pp. 1-7. https://ieeexplore.ieee.org/document/6614175.
17. **Valente P.** Reducing the Execution Time of Fair-Queueing Packet Schedulers. – Comp. Comm., Vol. 47, 1 July 2014, pp. 16-33. https://www.sciencedirect.com/science/article/abs/pii/S0140366414001455.
18. https://linkmeup.gitbook.io/sdsm/15.-qos.