МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М. В. ЛОМОНОСОВА

Факультет вычислительной математики и кибернетики

ПРОГРАММНЫЕ СИСТЕМЫ И ИНСТРУМЕНТЫ

Тематический сборник

№ 19

Под общей редакцией
чл.- корр. РАН, профессора Р. Л. Смелянского

МАКС ПРЕСС

Москва – 2019

УДК 519.6+517.958
ББК 22.19
П75

Печатается по решению Редакционно-издательского совета факультета вычислительной математики и кибернетики Московского государственного университета имени М. В. Ломоносова

Программные системы и инструменты : Тематический сборник / П75 Под ред. Р. Л. Смелянского. – Москва : Издательский отдел факультета ВМК МГУ имени М. В. Ломоносова (лицензия ИД № 05899 от 24.09. 2001 г.); МАКС Пресс, 2019. – № 19. – 136 с.

ISBN 978-5-89407-604-1 (ВМК МГУ имени М. В. Ломоносова)
ISBN 978-5-317-06324-5 (МАКС Пресс)

Данный выпуск сборника составлен по материалам работ студентов и аспирантов, получивших рекомендации научных семинаров кафедры Автоматизации систем вычислительных комплексов, которую Л. Н. Королев создал и которой бессменно руководил, кафедры Алгоритмических языков, а также государственных аттестационных комиссий выпускников бакалавриата и магистратуры. Редколлегия сборника продолжает традицию издания сборника в память об этом выдающимся человеке. В предлагаемом читателю тематическом сборнике публикуются статьи, посвященные проблемам современных компьютерных сетей, методам и средствам организации и управления облачными вычислениями, инструментальным средствам, обеспечивающим работу СРВ, средствам статического анализа. Статьи сборника будут интересны студентам, аспирантам и специалистам в области разработки прикладных программных систем с использованием новых информационных технологий.

Ключевые слова: информационно-телекоммуникационные технологии, программно-конфигурируемые сети, OpenFlow-коммутатор, централизованный контроллер, облачные вычисления, центр обработки данных, виртуальные ресурсы, виртуализация сетевых функций, облачная платформа, система реального времени, интегрированная модульная архитектура, построение расписаний, DDOS атаки, сетевой процессор, средства статического анализа, функциональные языки, архитектура компьютеров, история вычислительной техники, БЭСМ-6.

УДК 519.6+517.958
ББК 22.19

ISBN 978-5-89407-594-5
ISBN 978-5-317-06324-5

Оглавление

От редколлегии **5**

1 *Антипина А.В., Пашков В.Н.* **Метод предотвращения DDoS атак на контроллер в программно-конфигурируемых сетях** **6**

2 *Балашов В.В., Антипина Е.А.* **Итерационная схема планирования вычислений в модульных системах реального времени.** **18**

3 *Волканов Д.Ю., Маркобородов А.А.* **Исследование ячейки конвейера сетевого процессорного устройства на модели уровня регистровых передач.** **30**

4 *Галкина Е.В.* **Средства статического анализа программ на функциональных языках с динамической типизацией.** **42**

5 *Зайцева О.А., Антоненко В.А.* **Разработка и реализация системы холодного старта функции для бессерверных вычислений.** **53**

6 *Пантюхин Л.К., Антоненко В.А.* **Разработка алгоритма распределения трафика для SD-WAN решения.** **66**

7 *Синякова М.А., Степанов Е.П.* **Оценка задержки потоков виртуального пласта.** **80**

8 *Степанов Е.П., Войнов Н.А.* **Динамическое сегментирование транспортных соединений.** **97**

9 *Титов Н.И., Антоненко В.А.* **Разработка системы первичной настройки клиентских устройств для доступа к облачным сервисам.** **109**

10 *Королев Л.Н., Мельников В.А.* **Об ЭВМ БЭСМ-6.** **120**

Аннотации **130**

СБОРНИК

«Программные системы и инструменты»

От редколлегии:

Тематический сборник “Программные системы и инструменты” был инициирован Львом Николаевичем Королевым в 2000 году, как площадка где студенты и молодые ученые могли бы публиковать свои достижения. Все эти годы Лев Николаевич неустанно вел его, отбирал публикации, редактировал. Редколлегия сборника продолжает эту традицию в память об этом выдающимся человеке.

Этот выпуск сборника составлен по материалам работ студентов, получивших рекомендации научных семинаров кафедры Автоматизации Систем Вычислительных Комплексов, которую Л.Н. Королев создал и бессменно руководил, кафедры Алгоритмических Языков а также государственных аттестационных комиссий выпускников бакалавриата и магистратуры. В предлагаемом читателю тематическом сборнике публикуются статьи, посвященные проблемам создания информационно вычислительной инфраструктуры для научных исследований, проблемам современных компьютерных сетей, методам и средствам организации и управления облачными вычислениями, инструментальным средствам, обеспечивающим работу систем управления в реальном времени (СРВ), средствам статического анализа..

Также в этом сборнике представлена одна из работ Льва Николаевича Королёва «ОБ ЭВМ БЭСМ-6». В публикуемом ниже варианте Л.Н.Королевым внесены некоторые изменения по сравнению с версией опуликованной в 1976 году.

Редколлегия

Антипина А.В., Пашков В.Н.

Метод предотвращения DDoS атак на контроллер в программно-конфигурируемых сетях

Введение

Программно-конфигурируемые сети (ПКС) являются относительно новым перспективным направлением в развитии компьютерных сетей. Важной проблемой безопасности ПКС являются атаки на компьютерные системы, среди которых наиболее распространены DoS и DDoS атаки. DoS атака (атака типа "отказ в обслуживании") представляет собой целенаправленный комплекс действий для частичной или полной остановки работы веб-сайта или другого сетевого ресурса [1]. Атакующий может достичь своей цели разными способами, но в основном все сводится к перегрузке атакуемого ресурса большим количеством запросов. В результате сервер не может нормально выполнять свои функции. DDoS атака является распределенной DoS атакой и отличается от обычной только тем, что проводится с нескольких компьютерных устройств [2]. В ПКС появляется новый тип атак отказа в обслуживании — DDoS атака на контроллер. Так как все управляющие функции выполняются на контроллере, его перегрузка может повлечь вывод из строя всех сервисов сети.

В процессе выполнения задачи маршрутизации потоков трафика контроллер может добавлять, обновлять и удалять записи потоков в таблицах коммутаторов проактивно и реактивно. Проведение атаки возможно реактивным режимом работы: при поступлении нового потока, коммутатор буферизирует его пакеты и отправляет контроллеру запрос на добавление нового потока (packet_in). Получая его, контроллер рассчитывает новые маршруты и отправляет управляющие пакеты на коммутаторы (packet_out и flow_mod).

Злоумышленник может добиться перегрузки контроллера, когда он будет загружен обработкой лишь потоков с хостов злоумышленника, а время обработки потоков пользователей сети значительно возрастет, либо они не будут успевать обрабатываться, сбрасываясь по таймауту. Таким образом, происходит сбой работы сервисов сети, вследствие невозможности доставки трафика. Поэтому задача своевременного обнаружения атаки и противодействия ей является актуальной важной задачей безопасности ПКС, без решения которой невозможно говорить о полноценном внедрении данной концепции в

реальных сетях.

1. Задача обнаружения и предотвращения DDoD атаки на контроллер

Пусть имеется программно-конфигурируемая сеть с фиксированной топологией под управлением контроллера. Управление сетью осуществляется по протоколу OpenFlow не ниже версии 1.3 [3]. В узлах сети расположены коммутаторы, физические порты которых могут быть подключены к другим коммутаторам, сетевым сервисам и хостам сети. *Граничными коммутаторами* назовем те, которые имеют подключение к хостам. Порты граничных коммутаторов, связанные с другими коммутаторами и сетевыми сервисами – *доверенные*, остальные – *хостовые*. IP адреса хостов являются статическими.

В сети присутствует злоумышленник, который знает, что данная сеть является ПКС, имеет доступ к нескольким хостам (*зараженным*), но не имеет доступ к коммутаторам сети. Остальные хосты сети являются *пользовательскими*.

Предполагается, что злоумышленник использует следующую стратегию проведения DDoS атаки на контроллер – максимизировать количество запросов на установление новых потоков с зараженных хостов. Добиться этого он может с помощью частой отправки пакетов со случайно сформированными IP и MAC адресами источника и получателя с зараженных хостов.

Для решения задачи необходимо:

- Определить зараженные хосты и связанные с ними зараженные порты коммутаторов.
- Определить действия для предотвращения атаки с зараженных хостов.

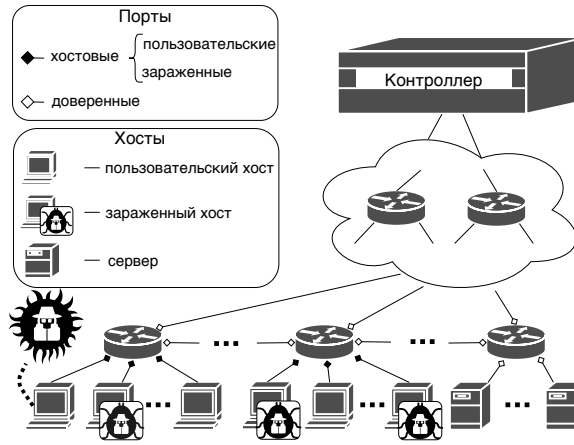

Рисунок 1. Топология сети для решаемой задачи

2. Обзор существующих методов предотвращения DDoS атак

В рамках работы был проведен обзор и сравнительный анализ существующих методов обнаружения и предотвращения DDoS атак по следующим критериям:

1. Информация о хостах — какая информация собирается об источниках.
2. Критерий классификации потоков — каким образом распознаются фиктивные и пользовательские потоки.
3. Оценка эффективности — оценка ошибки первого (ложное срабатывание) и второго рода (пропуск фиктивного трафика), оценка задержки обработки потоков.
4. Накладные расходы и требования реализации — как использование метода отражается на утилизации ресурсов контроллера и коммутаторов; какие архитектурные изменения необходимы; какие данные хранятся дополнительно.

2.1 Метод на основе разделения очереди обработки packet_in сообщений в контроллере

Авторы метода [4] предполагают логически разделить очередь обработки запросов в контроллере на k очередей, каждая из которых

соответствует одному коммутатору, имеющему канал связи с контроллером. Обслуживание этих очередей происходит в соответствии с какой-либо дисциплиной планирования, например, циклической. В итоге создается изолированное распределение вычислительной мощности контроллера для каждого коммутатора.

Информация о хостах. Для работы метода информация о хостах не требуется.

Критерий классификации потоков. Потоки явно не классифицируются. Вследствие создания логических очередей большинство запросов от фиктивных потоков будут сбрасываться по таймауту, не успев обработаться контроллером.

Оценка эффективности. Наименьшая доля сброшенных потоков на атакуемом коммутаторе видна на слабых атаках, превышающих обработки контроллера в 1.2 раза, — 20%; на более серьезных, превышающих в 8 раз, — 50-90%. На слабых атаках задержки обработки трафика при использовании метода и без него не различаются. На серьезных атаках задержка для атакуемого коммутатора равна 18 секундам, для остальных — менее 5 секунд, что гораздо лучше, чем без использования метода — 100 секунд.

Накладные расходы и требования реализации. Таблицы потоков атакуемых коммутаторов будут заполнены. Требуется возможность изменения архитектуры очереди обработки в контроллере.

К достоинствам метода можно отнести простоту реализации. Недостатком метода является сбрасывание пользовательских потоков на атакуемых коммутаторах с той же частотой, как если бы никакого метода защиты не применялось.

2.2 Метод на основе переноса table-miss потоков на соседние коммутаторы и фильтрации с помощью асимметрических функций

Метод состоит из четырех модулей [5]:

1. *Модуль обнаружения атак* контролирует состояние сети.

2. *Модуль управления потоками пропусков таблицы* перенаправляет часть новых потоков (table-miss потоков) на соседние коммутаторы, тем самым защищая таблицы атакуемого коммутатора от переполнения.

3. *Модуль пакетной фильтрации* идентифицирует фиктивный трафик и отфильтровывает его; содержит два компонента: *packet_in-буфер* для хранения packet_in сообщений и *двухфазный фильтр* для идентификации трафика атаки.

4. *Модуль управления правилами потоков* выделяет в области таблиц потоков коммутатора «область кэша». Модуль перехватывает правила потоков, инициированные другими приложениями («перехваченные» правила), и определяет для них правила отслеживания (с помощью асимметрических функций создаются правила для отслеживания ответных пакетов для потока). В область кэша попадают «перехваченные» правила потока; в последней таблице хранятся «защитные» правила потоков (сгенерированные модулем управления table-miss потоками)

Информация о хостах. *Packet_in-буфер* классифицирует пакеты на основе протокола передачи данных и использует B+ дерево для хранения потоков, ключом каждого узла которого является комбинация MAC-адреса и IP адреса отправителя и получателя, а значением конечного узла является сообщение packet_in и количество пакетов в данном потоке («частота потока»).

В качестве **критерия классификации** потоков используется логика двухфазного фильтра, который применяет две функции фильтрации: *фильтрация на основе частоты* (по количеству пакетов в потоке), *фильтрация на основе трафика* (по количеству ответных пакетов для потока).

Оценка эффективности Задержка обработки пакетов во время атаки не отличается от задержки при работе в обычном режиме. Сброса пользовательских потоков не наблюдается, а обработка фиктивных потоков близка к нулю.

Накладные расходы и требования реализации. Возникает нагрузка на контроллер, так как все сообщения packet_in пересылаются на обработку в контур управления (уменьшение утилизации ЦПУ благодаря использованию метода — менее 25%). Также происходит утилизация пространства таблиц коммутаторов за счет установки дополнительных правил отслеживания и правил переноса части table-miss пакетов.

Достоинством метода является эффективная классификация потоков и распределение table_miss потоков на соседние коммутаторы для освобождения буфера атакуемого коммутатора. Среди недостатков метода можно выделить то, что накладные расходы практически нивелируют эффект защиты для утилизации ЦПУ контроллера.

2.3 Метод на основе двухэтапной фильтрации на граничных коммутаторах и на контроллере

Особенностью метода является разделение фильтрации на два этапа [6]: первый выполняется на граничных коммутаторах, сбра-

сывая пакеты от источников, отсутствующих в *таблице привязок*; второй этап выполняется на контроллере, используя количество пакетов на поток и количество packet_in от конкретного хоста.

Информация о хостах. Для каждого хоста сети существует запись в таблице привязок, хранящая его MAC и IP адреса, номер входного порта коммутатора и MAC адрес коммутатора, к которому подключен хост.

Критерий классификации потоков/пакетов. Каждые T секунд контроллер отправляет коммутаторам запросы на получение значения счетчика количества пакетов для потока. Тогда *критерий* благонадежности хоста — доля «хороших» потоков (т. е. количество пакетов в которых выше некоторой константы — t). *Оценка* поведения хоста имеет отрицательную корреляцию с количеством packet_in и положительную с критерием. Область изменения значения оценки для адреса источника делится на три области: «фиктивный», «неопределенный» и «пользовательский». Потоки от источника, оцененного как «неопределенный», будут обрабатываться или сбрасываться в зависимости от использования ресурсов контроллера.

Оценка эффективности. Нулевая ошибка второго рода достигается при распознавании 72% пользовательских источников. Ошибка первого рода равна нулю при распознавании 67% трафика атаки.

Накладные расходы и требования реализации. При работе приложения затрачивается около 8% ресурсов контроллера, что сравнимо меньше, чем нагрузка при отсутствии защиты (25%). Задержка обработки потоков во время атаки остается равной задержке в нормальном состоянии сети. Количество дополнительных правил в таблицах коммутатора пропорционально количеству хостов, подключенных к данному коммутатору.

Среди достоинств можно выделить, что накладные расходы при работе приложения сведены к минимуму, так как большинство фиктивных потоков сбрасывается сразу на граничных коммутаторах. Недостатком является возможность принятия фиктивного трафика, если он попадает в область «неопределенных» источников.

3. Разработка метода обнаружения и предотвращения DDoS атаки на контроллер

Основой для разработки был выбран метод двухэтапной фильтрации на граничных коммутаторах и в контроллере, т.к. он удовле-

творяет требованиям 1–5. Цель реализации — добиться минимизации ошибок I и II рода и добавить модуль обнаружения атаки.

Работа метода разделена на три фазы.

Фаза инициализации. После запуска контроллера создается «таблица привязок» в контроллере, содержащая MAC и IP адреса всех хостов, которые присутствуют в сети, и соответствующие им номер физического порта и MAC адрес коммутатора. Порты на граничных коммутаторах разделяются на множества хостовых и доверенных портов в соответствие с постановкой задачи. В таблицах потоков граничных коммутаторов устанавливаются правила потоков для **I этапа фильтрации** (см. таб. 1): пакеты сбрасываются, если их MAC и IP адреса источника, входной порт и/или MAC адрес коммутатора не совпадают или отсутствуют в таблице привязок.

match	instructions	command
in_port = user_port, priotiy = 2, eth_type = 0x800, eth_src = host_MAC, ipv4_src = host_IP	GoToTabel(1)	OFPFC_ ADD
in_port = user_port, priotiy = 2, eth_type = 0x800, eth_src = host_MAC, ipv4_src = host_IP	GoToTabel(1)	OFPFC_ ADD
in_port = user_port, priotiy = 1, eth_type = 0x800	DropAction	OFPFC_ ADD
in_port = user_port, priotiy = 1, eth_type = 0x806	DropAction	OFPFC_ ADD
in_port = trusted_port, priotiy = 1,	GoToTable(1)	OFPFC_ ADD

Фаза мониторинга. Производится подсчет количества packet_in, и количества всех потоков в таблицах граничных коммутаторов и «хороших» потоков (количество пакетов в «хорошем» потоке превышает заданное значение для каждого хоста. Если количество packet_in от хоста превышает установленный порог, то метод переходит на фазу атаки.

Фаза атаки. DDoS атака может пройти через первый этап фильтрации, если злоумышленник не будет подменять MAC и IP адрес источника пакетов, а только генерировать различные адреса получателя. Тогда выполняется второй этап фильтрации, на котором через

интервал времени для каждого хоста из таблицы привязок рассчитываются *критерий* надежности хоста и *оценка* поведения хоста по следующим формулам:

- критерий $= \frac{\text{количество хороших потоков}}{\text{количество всех потоков}}$ (1) — чем меньше хороших потоков, тем выше вероятность, что хост зараженный, так как зараженный хост генерирует большое количество потоков, передавая в каждом минимальное количество пакетов;

- оценка $= \frac{\text{критерий}}{\text{количество packet\_in}}$

  (2) — оценка имеет отрицательную корреляцию с количеством packet_in, так как чем больше количество packet_in приходит от хоста за период времени, тем скорее этот хост будет зараженным;

- оценка = оценка $*(1-\alpha)$ + (предыдущая оценка) $*\,\alpha$, где $\alpha \in (0,1)$ (3) — вносит поправку на предыдущее значение оценки для хоста;

Далее значение оценки поведения хоста сравнивается с порогами `threshold_low`, `threshold_hight`, выбранными заранее:

```
if (score > treshold_hight)
   host.type = user;
else if (score > threshold_low)
      if (cpu_util.get() < threshold_cpu_util)
         host.type = ambiguous;
      else
         host.type = infected;
else
   host.type = infected;
```

где `score` — оценка поведения хоста, `threshold_cpu_util` — порог значения допустимой утилизации ЦПУ контроллера; `user`, `ambiguous` и `infected` — пользовательский, неопределенный и зараженный хост соответственно.

Для минимизации ошибок I и II рода была выявлена взаимосвязь пороговых значений для оценок в зависимости от количества итераций метода, за которое хост будет корректно классифицирован. Для уменьшения времени противодействия атаке необходимо минимизировать количество итераций фильтрации. Тогда справедливы выражения (4)–(5):

$$0 < \text{верхний порог } < 1,$$

$$\alpha * (\text{верхний порог}) < \text{нижний порог} < \text{верхний порог}$$

4. Приложение DDoS_Defender для контроллера RUNOS

Программная реализация метода выполнена в виде приложения DDoS_Defender для контроллера RUNOS [8].

Архитектура разработанного приложения DDoS_Defender представлена на рисунке 2.

Для реализации DDoS_Defender понадобились объекты приложений `Controller`, `SwitchManager` (предоставляет интерфейс работы с абстракциями коммутаторов) и `HostManager` (отслеживает появление в сети новых хостов и запоминает их IP и MAC адреса). Для работы приложения потребовалось написать собственную функцию-обработчик `MyHandler` для анализа packet_in.

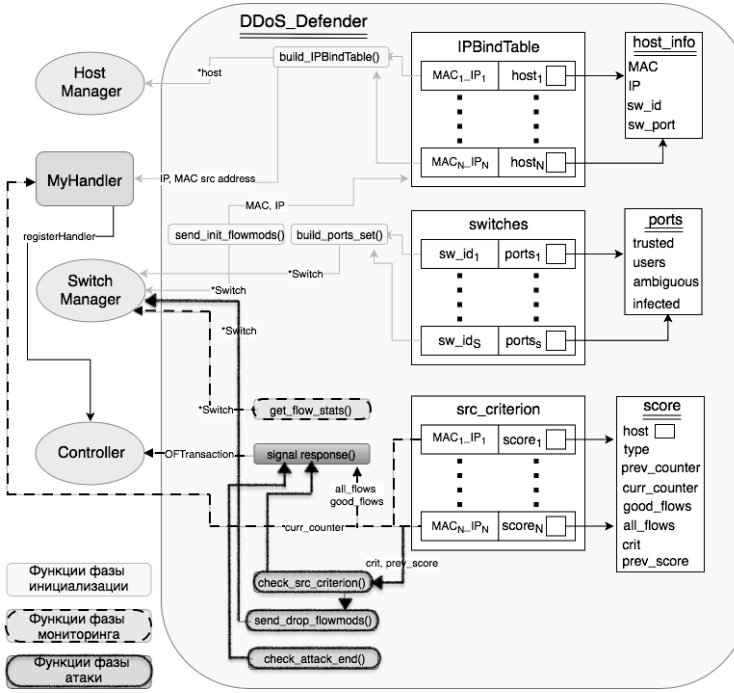

Рисунок 2. Архитектура реализации приложения DDoS_Defender

Приложение имеет следующие настраиваемые параметры, которые необходимо указать в конфигурационном файле *network-settings.json*: `crit_good_flows` — пороговое значение количества па-

кетов хороших потоков; `alpha` — константа корректировки оценки в диапазоне (0.0, 1.0); `threshold_low`, `threshold_hight` — нижнее и верхнее пороговые значения оценки (`curr_score`) для классификации хостов; `threshold_cpu_util` — пороговое значение утилизации ЦПУ контроллера; `THRESHOLD` — пороговое значение количества packet_in для начала атаки; `interval` — период времени, через который будет собираться статистика с коммутаторов и происходит пересчет оценки для хостов.

5. Экспериментальное исследование

Для проведения экспериментального исследования используются:

- OpenFlow-контроллер *RUNOS v0.6.5* [7].
- Эмулятор сетей *Mininet 2.3.0d5* [8].

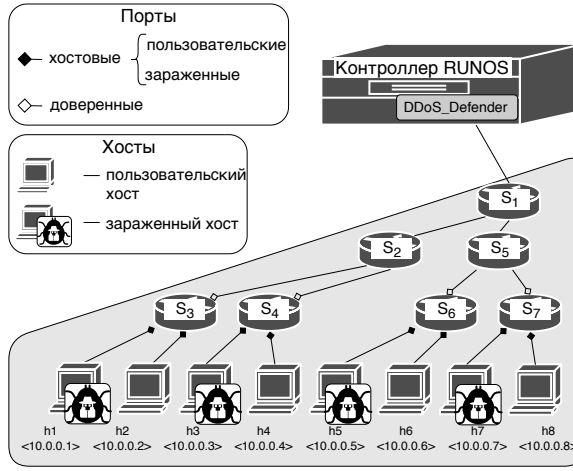

Рисунок 3. Экспериментальная топология в сети Mininet

В работе экспериментального стенда участвуют две виртуальные машины *Oracle VM VirtualBOX 5.2.16* [9] с операционной системой Ubuntu 16.04 LTS 32-bit. На машине 2 запущен контроллер RUNOS на порте 6653, на машине 1 — сеть Mininet с древовидной топологией, показанной на рисунке 3.

Проводится несколько проверок, направленных на:

1. Оценку задержки обработки пользовательских потоков.

2. Оценку доступной пропускной способности между пользовательскими хостами.

3. Оценку утилизации ЦПУ и памяти контроллера.

4. Оценка ошибки первого и второго рода.

Проверка 1. Значения RTT для первого пакета потоков пользователей приведены в таблицах 7.1 и 7.2. В отсутствии атаки работа приложения практически не оказывает влияния на задержку пакетов. Во время атаки показатели стали до 6 раз лучше.

Проверка 2. В отсутствии атаки приложение в среднем практически не оказывает дополнительной нагрузки на пропускную способность линий связи, а во время атаки уменьшает их утилизацию примерно на 36–52%.

Проверка 3. Утилизация ЦПУ во время атаки благодаря DDoS_Defender уменьшилась до 2 раз. Замечены накладные расходы в виде дополнительной утилизации ОП — до 1.34 раза больше. Значение зависит от количества хостов в сети (рассматривались топологии до 125 хостов).

Проверка 4. Было обнаружено, что значение оценки пользовательских хостов стремится к 1, но не достигает ее, а значение оценки зараженных хостов — к 0. Следовательно, константу `threshold_hight` в формулах (4)–(5) следует выбирать из диапазона (0.9, 1.0), если предполагается минимизировать количество итераций фильтрации до одной. Для топологий до 125 хостов при синхронном начале атаки и `threshold_hight` = 0.9 приложение распознает ∼90–100% зараженных хостов и ∼100% пользовательских хостов за время от 12 с; ∼50–100% зараженных хостов и ∼100% пользовательских хостов за время от 5 до 10 с.

Заключение

В данной работе был проведен обзор существующих методов предотвращения DDoS атак на контроллер в ПКС, в результате которого был выбран и реализован метод на основе двухэтапной фильтрации трафика, в который были внесены следующие усовершенствования: добавлен модуль мониторинга сети и выявлена связь на диапазоны значений задаваемых параметров. Метод реализован в виде приложения на языке C++ для контроллера RUNOS и позволяет уменьшить утилизацию ЦПУ контроллера до 50% во время атаки, уменьшить задержку обработки пользовательского трафика до 6 раз и увеличить пропускную способность каналов до 2 раз во

время атаки, а также добиться одновременной минимизации ошибок первого и второго рода. Метод имеет накладные расходы в виде дополнительной утилизации ОП контроллера для хранения таблиц с адресами хостов и классификацией портов граничных коммутаторов.

Литература

1. Encyclopedia by Kaspersky Lab `https://encyclopedia.kaspersky.ru/glossary/dos-denial-of-service-attack`
2. Encyclopedia by Kaspersky Lab `https://encyclopedia.kaspersky.ru/glossary/ddos-distributed-denial-of-service-attack/`
3. Open Networking Foundation. OpenFlow Switch Specification. Version 1.3.0 `https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf`
4. Lim S. et al. *Controller Scheduling for Continued SDN Operation under DDoS Attacks.* Electronics Letters. — 2015. — Т. 51. —№16. — С. 1259–1261.
5. Shang G. et al. *FloodDefender: Protecting Data and Control Plane Resources under SDN-Aimed DoS Attacks.* IEEE, 2017. — С. 1–9.
6. Zhang M. et al. *FloodShield: Securing the SDN Infrastructure Against Denial-of-Service Attacks.* IEEE, 2018. — С. 687–698.
7. RUNOS v0.6.5 `https://github.com/ARCCN/runos/tree/v0.6.5`
8. Mininet `https://github.com/mininet/mininet`
9. Oracle VM VirtualBOX 5.2.16 `https://www.virtualbox.org`

Балашов В.В., Антипина Е.А.

Итерационная схема планирования вычислений в модульных системах реального времени

Введение

Модульные вычислительные системы реального времени (МВС РВ) — это вычислительные системы, построенные из набора стандартизированных модулей, соединенных коммуникационной средой. В данной статье МВС РВ рассмотрены на примере интегрированной модульной архитектуры (ИМА) [1], в т.ч. на примере бортовых систем, в соответствии с которой в качестве коммуникационной среды выступает коммутируемая сеть передачи данных (СПД) с поддержкой виртуальных каналов (FC-AE-ASM-RT [2], AFDX [3]).

Вычислительные задачи, предназначенные для выполнения на МВС РВ, группируются в разделы согласно их принадлежности к программным подсистемам (приложениям). Процессорное время на ядре выделяется для каждого раздела в форме фиксированных (определенных до старта системы) интервалов времени – окон выполнения, в рамках каждого из которых могут выполняться задачи только одного раздела. Требования реального времени заключаются в том, что для задач также задаются их директивные сроки, строго внутри которых они должны быть выполнены. Задачи, выполняющиеся на разных модулях, создают трафик в межмодульной сети за счет обмена сообщениями между собой. Для каждого потока данных между модулями должен быть построен виртуальный канал, гарантирующий доставку сообщений за фиксированное время.

Планирование вычислений в МВС РВ состоит из трех этапов [5]: распределение разделов по модулям и процессорным ядрам; настройка СПД и конфигурирование виртуальных каналов для каждого потока данных; построение для каждого ядра расписания окон (временных интервалов) выполнения разделов. Прямая схема планирования заключена в последовательном решении данных задач и подразумевает невозможность выполнения планирования при отсутствии решения на одном из этапов.

В статье[1] представлена итерационная схема планирования вычислений в МВС РВ, расширяющая прямую схему за счет обратных связей между этапами. Обратные связи используются при от-

[1]Работа выполнена при поддержке РФФИ (проект №19-07-00614)

сутствии решений на втором или третьем этапах с целью уточнения входных ограничений для предыдущих этапах и их повторного выполнения.

1. Планирование вычислений в МВС РВ

Планирование вычислений в МВС РВ выполняется в три этапа. Целью планирования является построение статико-динамического расписание выполнений задач.

Решая задачу распределения вычислительной нагрузки по вычислительным модулям и процессорным ядрам, следует соблюдать ограничения на максимальную загрузку ядер и распределять разделы только на доступные для них ядра. Также возможно учитывать такие критерии оптимизации, как: нагрузка на сеть (минимизация), число используемых ядер (минимизация). Данная задача является NP-трудной [6]. Формальная постановка задачи и алгоритмы ее решения представлены в [4], [6].

Входные данные для задачи построения виртуальных каналов возникают сразу после выполнения распределения разделов по модулям. Для каждого потока данных, проходящих через сеть, необходимо определить маршрут и параметры виртуального канала, при этом соблюдая ограничения на пропускную способность физического канала, на максимальную длительность передачи каждого сообщения и максимальный джиттер. Алгоритм организации передачи сообщений в сетях AFDX представлен в [7].

Задача построения расписания выполнения вычислительных задач на процессорных ядрах решается после конфигурирования сети передачи данных, когда становятся известны задержки на передачу сообщений. В соответствии со стандартом ARINC 653, для каждого процессорного ядра должно быть построено статическое расписание окон. Построенное статико-динамическое расписание должно гарантировать выполнение всех работ (экземпляров вычислительных задач) в их директивные сроки. В работе [4] представлен жадный алгоритм, решающий данную задачу.

2. Проблемы при поэтапном планировании вычислений

Существующая прямая схема решения задачи планирования [5] заключена в том, что вышеописанные этапы выполняются по одному разу друг за другом, решения задач каждого из этапов являют-

ся входными данными для последующих этапов. Данная схема не всегда приводит к построению корректного статико-динамического расписания (при условии его существования). Это связано с тем, что алгоритмы каждого из этапов решают свои локальные задачи, не ориентируясь на всю структуру системы, а также с тем, что между этапами отсутствуют шаги анализа решения задач предыдущего этапа, необходимые для задания более точных ограничений и параметров последующих алгоритмов. Например, минимизируя нагрузку на сеть, алгоритм распределения вычислительной нагрузки может разместить разделы так, что весь сетевой трафик будет проходить между двумя модулями, тем самым сделав невозможным построение виртуальных каналов, удовлетворяющих ограничениям пропускной способности физических каналов.

Возможны два подхода, ведущих к избавлению от данных проблем: решение всей задачи планирования целиком, без разбиения ее на этапы, и расширение существующего поэтапного метода дополнительными шагами. В данной статье рассмотрен второй подход. Авторами предлагается использовать схему с обратной связью, с помощью которой можно уточнять ограничения и параметры алгоритмов при возникновении неудач решения на каждом из этапов планирования вычислений. Применение обратных связей и повторное выполнение одного или нескольких этапов делает схему итерационной.

3. Анализ причин неуспешного выполнения этапов планирования

Для того, чтобы ввести обратные связи между этапами планирования, необходимо рассмотреть каждый из этапов планирования вычислений в МВС РВ отдельно и выделить основные причины их неуспешного выполнения (невозможности построения решения задачи данного этапа, удовлетворяющего всем ограничениям). Для выявления причин можно вводить дополнительные шаги проверок, тогда данные причины будут устранены до того, как из-за них произойдет неуспешное выполнение схемы. Наличие других причин может быть обнаружено только после произошедшего неуспеха.

3.1 Локальная перегрузка СПД

Невозможность построения виртуальных каналов с заданными ограничениями (а именно с необходимостью гарантировать передачу сообщения за время, не превышающее заданного значения) свя-

зана с ограничениями на пропускную способность физических каналов и ограничением на максимальный джиттер выдачи кадров в сеть, заданную стандартом. Данная проблема может быть вызвана слишком большим объемом данных, передаваемым между парой модулей. Несмотря на то, что при распределении вычислительной нагрузки решается задача минимизации загрузки сети, большая часть трафика может быть локализована между одной или несколькими парами модулей. Соответственно необходимо уменьшать объем входящих/выходящих данных модуля в случае неудачи конфигурирования сети.

Для того, чтобы было возможно выполнить обратный переход с этапа построения виртуальных каналов на этап распределения вычислительной нагрузки, авторами был введен дополнительный параметр для вычислительных модулей — максимальная интенсивность потока трафика входящего и выходящего из модуля: W_i, i=$\overline{1, N^{mod}}$, N^{mod} — число модулей в системе. Соответствующее ограничение было добавлено к ограничениям задачи распределения вычислительной нагрузки:

$$\forall i = \overline{1, N^{mod}} : \sum_{msg \in \{MSG\}} X_{msg,i} * size_{msg}/T_{msg} \leq W_i,$$

$$X_{msg,i} = \begin{cases} 1, (snd_{msg} \in mod_i) \oplus (rcv_{msg} \in mod_i) \\ 0, \text{иначе} \end{cases},$$

где $\{MSG\}$ — множество всех сообщений системы, $size_{msg}$ — размер сообщения, T_{msg} — период сообщения, snd_{msg} и rcv_{msg} — задачи-отправитель и получатель сообщения.

Выявление локальной перегрузки сети возможно только после неудачной попытки построения каналов. В данной работе предложены следующие действия для осуществления обратной связи с этапа построения виртуальных каналов на этап распределения вычислительной нагрузки с целью балансировки трафика по каналам сети:

1. Получение данных от этапа построения виртуальных каналов, определение множества сообщений $F\_MSG = \{msg\}$, для которых не удалось построить виртуальные каналы.

2. Определение новых значений для ограничений на максимальный объем трафика, входящего и выходящего из модулей-отправителей и модулей-получателей сообщений из множества F_MSG:

   $\forall m \in \{mod | \exists\, msg \in F\_MSG : snd_{msg} \in mod \vee rcv_{msg} \in mod\}$

$$W_m = \sum_{msg \in \{MSG\}} X_{msg,m} * size_{msg}/T_{msg} - \sum_{msg \in F\_MSG} X_{msg,m} * size_{msg}/T_{msg}$$

3. Повторное выполнение алгоритма распределения вычислительной нагрузки с обновленными ограничениями.

3.2 Локальная перегрузка процессорных ядер

Невозможность выполнения всех работ (экземпляров вычислительных задач) в их директивные сроки может быть связана с перегрузкой процессорного ядра. С целью раннего выявления подобных локальных перегрузок предложено проводить подсчет суженных директивных интервалов работ и проверку критерия потребности в процессорном времени (Processor Demand Criterion, PDC) [8] непосредственно перед этапом построения расписания выполнения работ.

Суженные директивные интервалы работ — интервалы, внутри которых работы обязаны выполниться, чтобы все зависящие от них работы успели уложиться в свои директивные сроки. Ниже приведены формулы вычислений суженных директивных сроков для первых экземпляров задач (работ, у которых левый директивный интервал $l = 0$, правый $r = T_{task}$, где T_{task} — период задачи):

$$cl_{task} = \begin{cases} l, \text{ если задача task не имеет входных сообщений} \\ \max\limits_{msg:rcv_{msg}=task}(cl_{snd_{msg}} + C_{snd_{msg}} + t_{msg}), \text{ иначе} \end{cases}$$

$$cr_{task} = \begin{cases} r, \text{ если задача task не имеет исходящих сообщений} \\ \min\limits_{msg:snd_{msg}=task}(cr_{rcv_{msg}} - C_{rcv_{msg}} - t_{msg}), \text{ иначе} \end{cases},$$

где cl_{task} и cr_{task} суженные левые и правые директивные сроки, snd_{msg} и rcv_{msg} — задачи-отправители и получатели сообщений, C_{task} — длительность выполнения задачи на процессоре, t_{msg} — длительность передачи сообщения.

Для проверки существования расписания, гарантирующего выполнение всех работ в их директивные сроки, при рассмотрении процессорных ядер по-отдельности можно воспользоваться критерием потребности в процессорном времени. Суть критерия заключается в следующем: для того, чтобы все вычислительные работы могли быть выполнены на заданном процессорном ядре в их директивные сроки, необходимо, чтобы в любой временной промежуток сумма длительностей выполнений всех работ, директивные интервалы которых

принадлежат данному временном промежутку, была не больше, чем длительность этого промежутка. В представленной работе вместо директивных интервалов используются суженные директивные интервалы:

$$\forall t_1, t_2 \geq 0 : \sum_{cl_i \geq t_1 \,\&\, cr_i \leq t_2} C_i \leq t_2 - t_1,$$

где i — номер работы, выполняющейся на рассматриваемом ядре, cl_i и cr_i ее суженные левые и правые директивные сроки, C_i — длительность выполнения работы на процессоре.

При обнаружении перегруженных временных интервалов предлагается выполнять обратный переход на этап распределения вычислительной нагрузки или на этап построения виртуальных каналов. Обратный переход на первый этап заключен в введении дополнительных ограничений на привязку к процессорным ядрам работ, суженные директивные интервалы которых попадают в перегруженные временные интервалы. А именно решается задача поиска разделов, при исключении которых с рассматриваемого ядра, интервалы перестанут быть перегруженными. В данной работе обратный переход с этапа проверки PDC на этап распределения вычислительной нагрузки применяется только в случае, когда на ядре с перегрузкой существует раздел, изъятия которого достаточно, чтобы все перегруженные интервалы перестали быть таковыми.

Обратный переход на второй этап планирования выполняется с целью расширения длительностей суженных директивных интервалов работ за счет уменьшения ограничений на максимальные длительности передач сообщений. В отличие от обратного перехода на этап распределения вычислительной нагрузки, который влечет за собой большое количество изменений в исходной конфигурации системы, обратный переход на этап построения виртуальных каналов влияет только на длительности задержек при передаче сообщений.

3.3 Неуспешное планирование длинных цепочек задач

Еще одной проблемой при построении статико-динамического расписания являются длинные цепочки задач (последовательности задач, связанные между собой обменами сообщений). В работе предлагается использовать необходимое условие возможности выполнения каждой отдельной цепочки задач в модульной вычислительной системе.

Представим сообщения и вычислительные задачи в виде графа зависимостей G (Рисунок 1). Для разных значений перио-

да задач T строятся отдельные графы зависимостей. Пусть $G = (Task, Msg), Task$ — множество всех задач, Msg — множество всех сообщений, $Path \subset G$ — множество всех путей (под-графов) от задач-истоков (задач без входящих сообщений) к задачам-стокам (задачам без выходящих сообщений). Именно такие пути и будем считать цепочками.

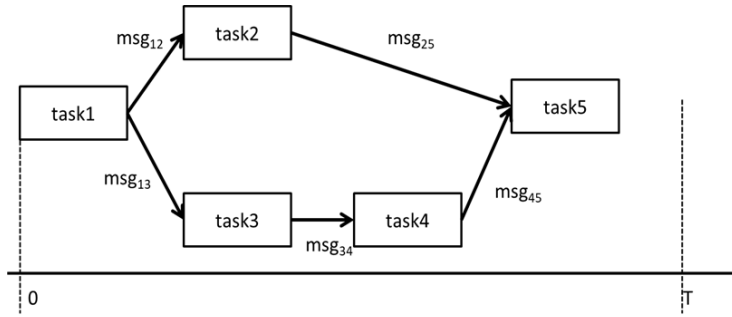

Рисунок 1. Пример графа зависимостей задач

Тогда необходимое условие выполнения всех работ каждой цепочки внутри своих директивных интервалов (у работ внутри одной цепочки данный интервал равен периоду любой задачи цепочки) является условием на длины критических путей в графах зависимостей:

$$\forall P \in Path : \sum_{msg \in P} t_{msg} \leq T_P - \sum_{task \in P} C_{task},$$

t_{msg} — длительность передачи сообщения, T_P — период задач цепочки, C_{task} — длительность выполнения задачи на процессоре.

На основе данного условия авторами предлагается задавать более точные ограничения на максимальные длительности передачи сообщений: свободный ресурс времени в цепочках задач (правая часть полученного ранее неравенства) распределятся между ограничениями на длительности передач сообщений пропорционально размерам сообщений. Это гарантирует построение расписания выполнений цепочек задач при условии отсутствия конфликтов между задачами из разных цепочек.

4. Итерационная схема планирования вычислений в МВС РВ

С учетом проведенного анализа поэтапная схема планирования вычислений в МВС РВ была расширена добавлением обратных свя-

зей и промежуточных вычислений. На Рисунке 2 изображена расширенная схема. Переходы с этапа проверки PDC выполняются последовательно: если переход на этап построения ВК был неуспешным (не удалось построить виртуальные каналы с обновленными ограничениями), выполняется переход на этап распределения ВН. Переход по обратной связи и последующее повторное выполнение одного или нескольких этапов будем называть итерацией.

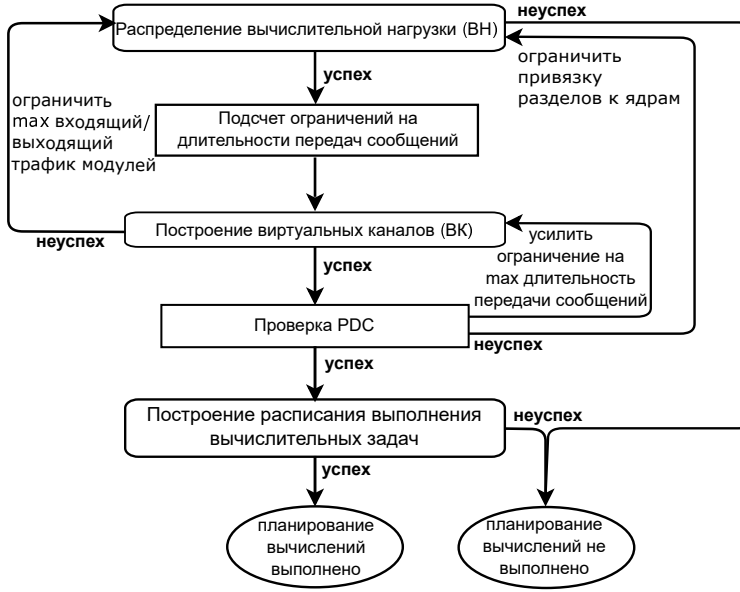

Рисунок 2. Итерационная схема планирования вычислений в МВС РВ

5. Экспериментальное исследование

Авторами был выбран следующий подход при исследовании предложенной схемы: сначала исследовались все представленные виды обратных связей по отдельности для выявления их областей применимости, а затем исследовалась вся схема в совокупности.

Генерация исходных данных для исследования проводилась следующим образом: к наборам данных, на которых прямая схема планирования вычислений строит корректные расписания, были применены операции по масштабированию их количественных характеристик, например: увеличение числа модулей, ядер, задач, а также

сообщений. Базовые исходные данные обладают следующими характеристиками: число модулей от 3 до 6, все модули одинаковые и одноядерные; число разделов от 4 до 10, в каждом разделе от 1 до 2 задач. Для каждого из видов обратных связей использовалось 7 наборов базовых исходных данных, к которым применялись разные наборы операций по изменениям системы, характеризующиеся коэффициентом масштабирования (целое число от 2 до 6).

Для исследования обратной связи с этапа построения виртуальных каналов на этап распределения вычислительной нагрузки (ВК→ВН) исходные данные были получены увеличением размеров сообщений между задачами. Для исследования обратной связи с этапа проверки PDC на этап построение виртуальных каналов (PDC→ВК) и на этап распределения вычислительной нагрузки (PDC→ВН) производилось деление задач и сообщений из состава исходных данных на несколько частей, за счет чего уменьшались суженные директивные интервалы задач.

На Рисунке 3 представлены результаты экспериментов, в них учтены только те наборы данных, на которых прямая схема планирования вычислений не строила решений. Для каждого вида обратных связей были найдены их недостатки, что также видно из графика. Так, для обратной связи ВК→ВН, было выявлено, что представленное изменение ограничений для максимального объема трафика, проходящего через модуль, на величину, равную произведению размера сообщения и его частоты, избыточно и может приводить к тупиковым ситуациям (к отсутствию решений задачи распределения вычислительной нагрузки). Обратная связь PDC→ВК успешно за одну итерацию решает проблему перегруженных интервалов на большинстве рассмотренных наборов данных. Но в некоторых случаях, при наличии нескольких сообщений, исходящих из одного модуля к разным задачам, обладающим одинаковыми суженными директивными сроками и выполняющимися на других модулях, алгоритм может заводить себя в тупик (отсутствие решений задачи построения виртуальных каналов). Исследование обратной связи PDC→ВН показало, что представленная в работе связь имеет узкую область эффективной применимости — она помогает только при наличии небольшого числа задач (до 5) внутри перегруженных интервалов. Для улучшения данной связи необходимо разработать другой алгоритм перераспределения вычислительной нагрузки, с помощью которого можно было одновременно переносить несколько задач с перегруженных ядер.

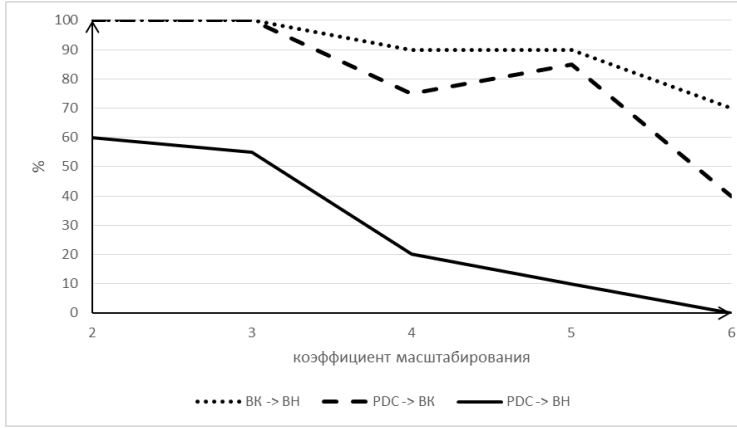

Рисунок 3. Зависимость процента успешных выполнений итерационной схемы от коэффициента масштабирования исходных данных

Исследование обратных связей в совокупности, т.е. исследование всей предложенной схемы было направлено на выявление проблем, возникающих при итерационном применении нескольких видов обратных связей. Например, когда примение обратной связи PDC→ВН приводило к необходимости применения обратной связи ВК→ВН. Исходные данные строились описанным ранее образом. В результате были выявлены наборы данных, для которых применение нескольких обратных связей, приводило к успешному решению задачи (данные с большим числом независимых задач), так и данные, для которых те же последовательности обратных связей заводили алгоритм в тупик (данные с сильной связанностью между задачами).

6. Заключение

В данной статье представлено расширение прямой схемы планирования вычислений в МВС РВ за счет добавления дополнительных проверок между этапами и применения обратных связей между ними. Важно отметить, что данный подход инвариантен относительно самих алгоритмов, выполняющихся внутри этапов исходной схемы, поскольку расширения, разработанные авторами, меняют исключительно исходные данные решаемых задач.

Экспериментальное исследование предложенной схемы показало область применимости каждой из обратных связей и их комбинаций. Также в результате исследований были выявлены недочеты разработанных обратных связей, причинами которых является сильная

взаимосвязь между параметрами задачи — изменение одиночного элемента входных данных этапа планирования ведет к изменению нескольких элементов выходных данных этапа.

В дальнейшем планируется более подробно изучить влияние параметров системы друг на друга и усовершенствовать предложенную схему, изменив и добавив обратные связи.

Литература

1. Парамонов П. П., Жаринов И. О. *Интегрированные бортовые вычислительные системы: обзор современного состояния и анализ перспектив развития в авиационном приборостроении.* Научно-технический вестник информационных технологий, механики и оптики. — 2013. — № 2. — С. 1—17.

2. Information technology. Fibre Channel – Part 312 *Avionics environment upper layer protocol MIL-STD-1553B Notice 2 (FC-AE-1553): Technical report TR 14165312:2009* ISO, 2009.

3. Aircraft Data Network Part 7 *Avionics Full Duplex Switched Ethernet (AFDX) Network.* ARINC Specification 664P7. — Annapolis : Aeronautical Radio, 2005. — 132 p.

4. Balashov V. V., Balakhanov V. A., Kostenko V. A. *Scheduling of computational tasks in switched network-based IMA systems.* Proc. International Conference on Engineering and Applied Sciences Optimization. – National Technical University of Athens (NTUA) Athens, Greece, 2014. – P. 1001–1014.

5. Балашов В. В. *Семейство систем автоматизации проектирования бортовых вычислительных систем реального времени.* Программные продукты, системы и алгоритмы. 2017. № 4. С. 1–19.

6. Balashov V., Antipina E. *Distribution of workload in IMA systems by solving a modified multiple knapsack problem.* Proc. 6th International Conference on Engineering Optimization. — Springer Nature Switzerland, 2018. — P. 1166–1177.

7. Вдовин П. М., Костенко В. А. *Организация передачи сообщений в сетях AFDX.* Программирование. — 2017. — № 1. — С. 3–18.

8. S. K. Baruah, L. E. Rosier, R. R. Howell *Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic Real-Time Tasks on One Processor*. Journal of Real-Time Systems, 4 (2) 301–324, 1990.

Волканов Д.Ю., Маркобородов А.А.

Исследование ячейки конвейера сетевого процессорного устройства на модели уровня регистровых передач[1]

Введение

В данной статье рассматривается модель вычислительной ячейки сетевого процессорного устройства (СПУ). СПУ представляет собой программируемую систему на кристалле, архитектура которой ориентирована на обработку заголовков пакетов данных, то есть на выполнение таких задач, как выделение заголовка из пакета, классификация заголовка по значению его полей, дальнейшая его модификация и выбор порта для отправки [3]. Структура большинства СПУ содержит интерфейсы ввода/вывода, набор программируемых вычислительных блоков, внутреннюю память и интерфейс для управляющего процессора [3]. Основные задачи по обработке заголовков пакетов решают программируемые блоки в составе СПУ.

Архитектура рассматриваемого в статье СПУ была разработана в ходе НИР, выполненной сотрудниками кафедры АСВК факультета ВМК и физического факультета МГУ. В этой архитектуре СПУ состоит из специализированных вычислительных ячеек. Данные ячейки способны выполнять загруженный в их память микрокод и специально разработаны для решения основных задач СПУ.

В процессе разработки СПУ возникает потребность исследования рассматриваемой архитектуры для выявления таких характеристик, как пропускная способность, энергопотребление, использование памяти и др.

Основной целью данной работы является исследование быстродействия вычислительной ячейки рассматриваемой архитектуры посредством разработки соответствующей модели уровня регистровых передач. Информация о скорости работы (количестве команд, выполняемых за единицу времени) позволит уточнить предполагаемую тактовую частоту в 1 ГГц и в дальнейшем позволит определить пропускную способность СПУ.

Статья имеет следующую структуру. В первом разделе содержится постановка задачи с перечислением подзадач, которые требовалось решить. Во втором разделе представлено краткое описание архитектуры СПУ на базе специализированных ячеек с более развер-

[1]Работа выполнена при частичной поддержке РФФИ, грант № 19-07-01076

нутым описанием самой ячейки и её системы команд. Третий раздел содержит описание модели уровня регистровых передач. В ней описываются характеристики, которые требуется получить, и общая структура модели. В последнем разделе представлены результаты исследования и методы их получения. В заключении перечислены выполненные задачи и приведен главный результат работы.

1. Постановка задачи

В качестве исходных данных имеется описание архитектуры СПУ и его вычислительной ячейки, система команд ячейки и оценка задержки логического элемента, предоставленная разработчиками архитектуры, равная 15 пс. Описание ячейки содержит информацию о функциях структурных блоков ячейки и схему их соединения между собой с указанием сигналов управления и шин данных. Требуется определить возможность работы ячейки на тактовой частоте в 1 ГГц. Для этого необходимо разработать модель уровня регистровых передач (RTL, register transfer level) вычислительной ячейки СПУ на языке описания аппаратуры.

Для решения этой задачи необходимо выполнить следующие подзадачи:

- разработать модель вычислительной ячейки;
- разработать тестовое окружение для моделирования ячейки на симуляторе;
- провести измерения скорости работы ячейки при помощи модели уровня регистровых передач.

2. Архитектура сетевого процессорного устройства

2.1 Общая структура

На рисунке 1 представлена структурная схема СПУ. В данной архитектуре СПУ состоит из следующих блоков:

- буферизующие логические устройства входной и выходной очередей;

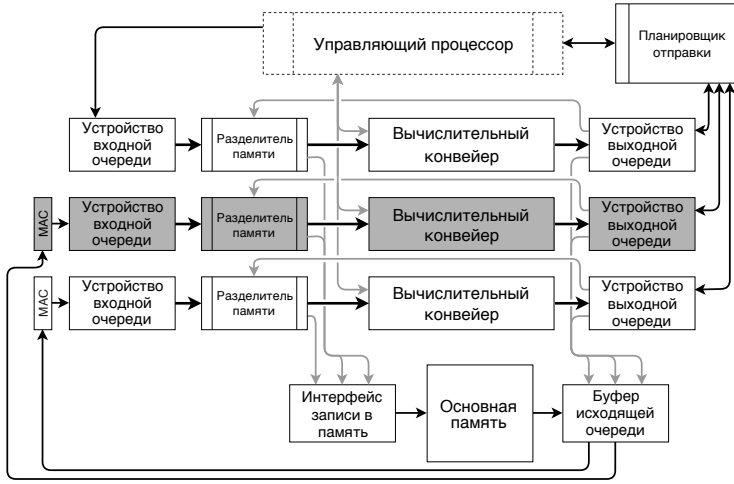

Рис. 1. Структура СПУ (стрелками указаны связи по данным)

- разделитель памяти;
- вычислительный конвейер;
- буфер исходящей очереди заголовков;
- планировщик отправки;
- основная память и интерфейс записи в неё;
- управляющий порт.

Для каждого входного интерфейса в СПУ выделен набор последовательно соединенных блоков (рис. 1, блоки серого цвета). Обработка каждого пакета начинается с буферизующего устройства входной очереди. Данный блок получает пакет через свой входной интерфейс, буферизует его и затем отправляет его следующему блоку — разделителю памяти. Этот блок отделяет от пакета его заголовок, а остальную часть (тело пакета) отправляет на запись в основную память СПУ. Далее заголовок и его метаданные (номер входного порта, маска выходных портов, размер пакета) отправляются в вычислительный конвейер. Данный блок выполняет основные задачи СПУ: разбор заголовка, классификация пакета по значениям полей в заголовке и его метаданных, модификация заголовка. После конвейера заголовок поступает в буферизующее устройство выходной очереди, где он соединяется с соответствующим телом пакета и ожидает отправки в порт.

Вычислительный конвейер состоит из набора последовательно соединенных вычислительных ячеек. Каждая ячейка выполняет одно и то же фиксированное число команд (каждая команда занимает один такт) после чего передает модифицированный заголовок и его метаданные следующей в конвейере ячейке, а последняя ячейка отправляет их на выход из конвейера в следующий блок СПУ.

2.2 Архитектура моделируемой вычислительной ячейки

На рисунке 2 представлены основные структурные элементы вычислительной ячейки и соединения между ними, рассмотрим функции каждого блока.

Блок памяти предназначен для хранения загруженного в ячейку микрокода и заголовка обрабатываемого пакета с его метаданными.

Счетчик команд содержит указатель на строчку в памяти, в которой находится исполняемая команда.

Исполнительный регистр служит для хранения текущей команды, загружаемой из памяти по указателю из счетчика команд.

Декодер интерпретирует команду в исполнительном регистре и посылает необходимые управляющие сигналы другим блокам.

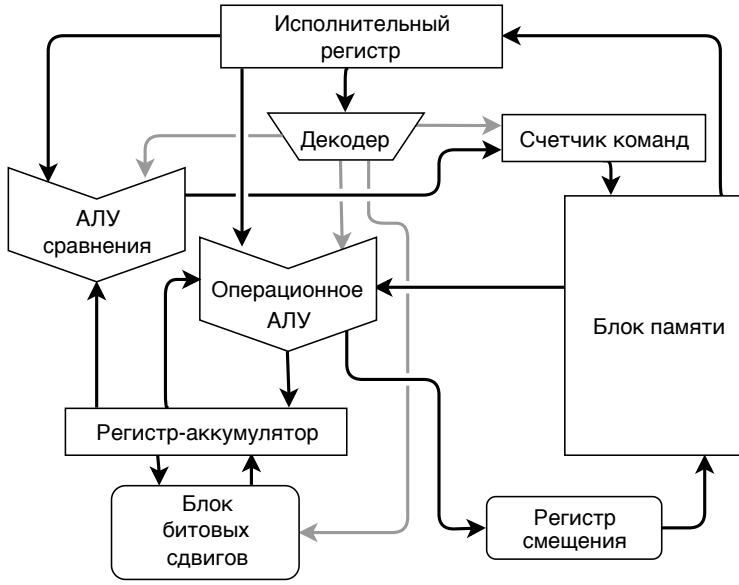

Рис. 2. Структурная схема ячейки конвейера СПУ (черные стрелки — связи по данным, серые — связи по управлению)

Регистр-аккумулятор используется для хранения результатов операций, а также может выступать в качестве операнда.

Операционное АЛУ — арифметико-логическое устройство, выполняющее арифметические и побитовые логические операции.

АЛУ сравнения — арифметико-логическое устройство, вычисляющее значение указателя счетчика команд для следующей команды при условных переходах, а также производящее инкремент указателя счетчика команд в остальных командах.

Блок битовых сдвигов производит логические и циклические сдвиги.

Регистр смещения хранит значение смещения в памяти, используемое для адресов операндов в командах.

В начальный момент времени обработки заголовка счетчик команд содержит указатель на первую строчку микрокода в памяти с нулевым адресом, слово данных по этому адресу загружается в исполнительный регистр. Загруженная команда исполняется, и далее каждый такт происходит загрузка в исполнительный регистр следующей команды, на которую указывает счетчик команд, и производится её выполнение.

Система команд

Вычислительная ячейка конвейера выполняет команды фиксированной длины в 128 бит. Такую же длину имеют регистр исполнения и регистр-аккумулятор.

Все команды можно разделить на несколько групп по типу производимых операций:

- чтение из памяти и запись в память (`load`, `store`, `loadbe`, `storebe`);
- арифметические операции (`add`, `sub`, `addi`, `subi`);
- логические побитовые операции (`and`, `or`, `xor`, `andbe`, `orbe`, `xorbe`, `andi`, `ori`, `xori`);
- битовые сдвиги (`rol`, `ror`, `rcl`, `rcr`);
- безусловный и условные переходы (`j`, `cmpj`, `cmpjn`, `cmpjg`, `cmpjg`, `cmpjlge`);
- специальные (`loadi` — загрузка значения в регистр-аккумулятор, `loadoffi` — загрузка значения в регистр смещения, `cpoff` — загрузка значения из регистра смещения в регистр аккумулятор, `nop` — пропуск такта).

Команды с суффиксом «be» работают с памятью, представляя байты в big endian порядке, а команды с суффиксом «i» используют значение, непосредственно прописанное в команде, в качестве операнда.

3. Модель уровня регистровых передач ячейки конвейера

В рамках данной работы было выполнено следующее:

- описана схема ячейки на языке Verilog;
- описан на языке Verilog тестовый блок для моделирования при помощи симулятора и инициализации микрокода в памяти модели;
- построена модель путем добавления задержек логических элементов схемы.

3.1 Описание измеряемых характеристик

Для того чтобы узнать, с какой скоростью СПУ будет обрабатывать пакеты, необходимо определить временные характеристики ячейки его вычислительного конвейера. Так как каждая команда выполняется ровно за один такт, то достаточно определить тактовую частоту ячейки. С помощью модели возможно оценить время выполнения команды и, исходя из него, получить максимальную тактовую частоту.

Таким образом, необходимо определить следующие характеристики:

- задержки распространения для структурных блоков ячейки, то есть максимальное время от начала изменения входа блока до момента, когда все его выходы достигнут установившихся значений [5], за вычетом задержек на распространение сигналов и усложнений схем в ходе их синтеза;
- максимальное время выполнения команды ячейкой.

3.2 Общая схема модели уровня регистровых передач

Для реализации модели использовалось такое же деление схемы ячейки на подсхемы, как и в её описании в пункте 2.2. Для каждой

подсхемы был написан специальный блок структурно-поведенческой модели на языке Verilog.

Так как существуют различия в описании на Verilog схем комбинационной и последовательностной логики, все блоки были поделены на две группы по типу схем, которые их реализуют. Таким образом, было получено следующее деление:

1. блоки с комбинационной логикой: АЛУ сравнения, операционное АЛУ, декодер, блок битовых сдвигов.

2. блоки с последовательностной логикой: счетчик команд, исполнительный регистр и регистр-аккумулятор, регистр смещения, блок памяти.

Исполнительный регистр и регистр-аккумулятор представляют собой 128-битные регистры с асинхронным сбросом в значение ноль и с разрешающим сигналом, то есть запись нового значения происходит при переднем фронте тактового сигнала на входе при условии, если разрешающий сигнал выставлен в 1 и отсутствует сигнала сброса.

Счетчик команд реализован в модели в виде 32-битного регистра с синхронным сбросом. Блок выдает на выход адрес команды.

Регистр смещения содержит регистр длиной 16 бит с сигналом разрешения записи. Блок выдает на выход адрес операнда с учетом смещения и передает их блоку памяти.

Блок памяти выдает на выход значение ячейки, соответствующей адресу на входе. Также при разрешающем запись сигнале и переднем фронте тактового сигнала производится сохранение нового значения.

АЛУ сравнения, операционное АЛУ и блок битовых сдвигов похожи по своей структуре. В них присутствует многовходовой мультиплексор, на который подается значение кода операции. В зависимости от этого значения выполняются необходимые преобразования входных сигналов.

Управляющим элементом ячейки является декодер. Ему на вход подается операционный код команды и сигналы состояния других блоков. Выходные значения декодера представляют собой управляющие сигналы и данные для других блоков.

Все описанные выше блоки затем соединяются в объемлющий блок при помощи сигнальных линий и шин. При этом у объемлющего блока есть два входа: тактовый сигнал и сигнал сброса, которые передаются внутренним блокам. Для синхронизации работы блоков тактовый сигнал на их входе может быть смещен на некоторую величину во времени относительно глобального тактового сигнала.

3.3 Задержки логических элементов

При моделировании с использованием симулятора схем, описанных на языке Verilog, по умолчанию все операторы выполняются моментально. Например, в непрерывном присваивании assign res = a + b при изменении операнда a или b моментально поменяется значение переменной (или выхода) res. Однако в физической реализации схемы потребовалось бы некоторое время на прохождение сигналов в сумматоре. По этой причине в код модели были добавлены задержки логических элементов.

В работе использовалась средняя оценка задержки логического элемента (например, двухвходовые AND или OR), равная 15 пс. С её помощью и, используя информацию о критических путях схем из [5], были получены значения для остальных элементов. Также из этого источника были использованы оценки задержек некоторых стандартных элементов.

4. Результаты измерений на модели

В данном разделе описываются методы проведения исследований на модели и приводятся полученные результаты. Для проведения исследований использовался симулятор ModelSim, разработанный Mentor Graphics [2].

4.1 Описание тестового окружения

Для запуска модели на симуляторе было необходимо разработать блок, имитирующий внешние воздействия на ячейку. В данном случае в этом блоке, во-первых, объявляется единица блока ячейки и, во-вторых, вырабатываются входные сигналы блока ячейки: вначале подается сигнал сброса и далее — тактовый сигнал через равные промежутки модельного времени.

Для проведения исследований был разработан микрокод, включающий в себя вариации всех типов команд из системы команд ячейки. Он позволяет задействовать все части схемы ячейки. Этот микрокод загружается в память ячейки в начальный момент модельного времени, после чего поступает сигнал сброса.

Для измерения времени работы каждого блока значение периода тактового сигнала было выставлено равным 10 нс (в 10 раз больше предполагаемого). Это позволило корректно измерить максимальные задержки сигналов каждого блока в отдельности. Также были

разнесены по времени синхронизационные сигналы для блоков с последовательностной логикой.

В качестве выходных данных симулятор выводит временные диаграммы для входов, выходов и внутренних переменных блоков. Используя диаграммы, можно найти максимальное время работы каждого блока, замеряя время от момента изменения входа блока до момента установления значений всех выходных сигналов блока.

Проверка полученного в данной работе значения тактовой частоты осуществлялась путем выставления полученного времени периода тактового сигнала и запуска тестового микрокода с проверкой значений регистров после каждой команды, а также с проверкой значений в памяти.

Для наглядного представления полученных результатов по скорости обработки заголовков была определена пропускная способность вычислительной ячейки на нескольких типичных сценариях обработки, взятых из [4] и реализованных в командах ячейки:

1. L2 switch + VLAN: обработка пакетов на коммутаторе в локальной сети с VLAN тегами. В процессе обработки из заголовка пакета выделяется MAC-адрес назначения и значение поля VID из тега VLAN. На основе этих значений принимается решение об отправке пакета на порт с определенным номером.

2. MPLS-P: обработка пакетов маршрутизатором внутри MPLS-сегмента сети. На вход поступают пакеты, содержащие один или более тегов MPLS в своем заголовке. При обработке пакета из первого тега MPLS выделяется значение поля номера метки. В зависимости от этого значения выполняется одно из трех действий: запись нового номера метки в тег, снятие тега MPLS или добавление нового тега MPLS, а также определяется номер порта для отправки.

3. Mirror: зеркалирование трафика на ПКС коммутаторе. На коммутатор поступают пакеты без тега VLAN или пакеты, содержащие от одного до двух тегов VLAN. В процессе обработки из заголовка пакета выделяются значения полей VID всех присутствующих в заголовке тегов VLAN. В зависимости от их значения может быть принято решение о зеркалировании — отправке копии пакета на выделенный порт. После этого обработка пакета продолжается по другому сценарию.

4.2 Результаты

Были получены следующие задержки распространения для блоков ячейки:

- АЛУ сравнения: 240 пс;
- операционное АЛУ: 465 пс (максимальная для команд сложения и вычитания);
- декодер: 285 пс;
- блок битовых сдвигов: 315 пс;
- регистр смещения: 240 пс (максимальная при вычислении адреса в памяти для операнда);
- блок памяти: 150 пс;
- исполнительный регистр, регистр-аккумулятор и счетчик команд: по 15 пс.

АЛУ сравнения, декодер, регистр смещения, блок памяти, исполнительный регистр и счетчик команд используются во всех командах и работают последовательно, так как зависят от данных друг друга. Их суммарная задержка равна 945 пс. Остальные блоки используются по одному в каждой команде. Среди них операционное АЛУ имеет самую большую задержку, поэтому максимальное время выполнения команды будет равно 1410 пс (соответствует тактовой частоте 709 МГц).

Дополнительно аналогичным способом были проведены замеры для предполагаемой задержки логического элемента, равной 10 пс. В этом случае максимальное время выполнения команды составило 940 пс.

Пропускная способность вычислительной ячейки на нескольких сценариях работы представлена в таблице 1.

Сценарий	Количество тактов	Пропускная способность, млн пакетов/с	Пропускная способность, Гбит/с
L2 switch + VLAN	55	12,89	7,68
MPLS-P	89	7,97	4,75
Mirror	14	50,64	30,18

Таблица 1. Пропускная способность ячейки на некоторых сценариях обработки пакетов

4.3 Выводы

Таким образом, было получено время в 1410 пс, необходимое для корректной работы ячейки со всеми имеющимися командами. Это позволяет использовать её на тактовой частоте в 709 МГц и ниже, что в 1,4 раза меньше предполагаемого значения.

Для достижения скорости работы на тактовой частоте в 1 ГГц можно предложить несколько путей решения. Во-первых, возможно сменить элементную базу схемы на более производительную. Замеры с задержкой элемента в 10 пс показали, что это позволит получить частоту в 1,06 ГГц. Во-вторых, можно внести изменения в систему команд ячейки для облегчения устройства некоторых блоков. Например, если все команды будут интерпретировать байты в памяти в одном порядке (Little Endian или Big Endian), часть подсхем станет не нужна, так как не будет необходимости менять порядок байтов в операндах, и, следовательно, уменьшится задержка распространения.

Заключение

В ходе работы были решены следующие задачи:

- разработана модель уровня регистровых передач вычислительной ячейки;
- при помощи модели были получены характеристики ячейки по времени работы и пропускная способность на некоторых сценариях обработки пакетов.

В результате была получена оценка тактовой частоты модели ячейки, равная 709 МГц.

Дальнейшие исследования данной ячейки конвейера СПУ могут быть направлены на увеличение тактовой частоты посредством внесения изменений в функциональность блоков и их внутреннего устройства. Также возможно изучить поведение целого вычислительного конвейера, состоящего из таких ячеек, и влияние других блоков СПУ на скорость обработки пакетов.

Литература

1. IEEE 1364-2005 *«IEEE Standard for Verilog Hardware Description Language»*

2. ModelSim ASIC and FPGA Design Simulator — Mentor Graphics `https://www.mentor.com/products/fpga/verification-simulation/modelsim/`

3. Orphanoudakis T., Perissakis S. *Embedded Multi-Core Processing for Networking//Multi-Core Embedded Systems* 2010. — CRC Press — p. 399-463.

4. Об одном подходе к построению сетевого процессорного устройства / С. О. Беззубцев, В. В. Васин, Д. Ю. Волканов и др. // *Моделирование и анализ информационных систем.* — 2019. — Т. 26, № 1. — С. 39–62.

5. Хэррис Дэвид М. *Цифровая схемотехника и архитектура компьютера : учеб. пособие* Дэвид М. Хэррис, Сара Л. Хэррис. — 2-е изд. — Нью-Йорк : Морган Кауфман, 2015. — 1628 с.

Галкина Е. В.

Средства статического анализа программ на функциональных языках с динамической типизацией

Введение

Последнее время происходит активное развитие языков функционального программирования. Они находят применение в различных прикладных областях, используются как в исследовательских, так и в коммерческих проектах. Функциональные языки позволяют работать с различными уровнями абстракции, имеют высокую степень выразительности, позволяют оперировать ***чистыми*** (*pure*) функциями. Все эти особенности приводят к повышению понятности исходного кода, уменьшению степени взаимозависимости между программными модулями, или ***сопряжения*** (*coupling*), и увеличению надёжности исполняемых программ.

Одним из свойств, согласно которому можно производить классификацию языков программирования, является способ типизации: различают языки со ***статической*** и ***динамической*** типизацией. При статической типизации анализ исходного кода происходит во время компиляции в предположении, что выражение (переменная, возвращаемое значение функции и т. п.) получает определённый тип в момент объявления, и этот тип не может изменяться в ходе выполнения программы. К функциональным языкам программирования со статической типизацией относятся, например, языки Haskell [1], OCaml [2], Standard ML [3].

В динамически типизированных языках переменная получает тип во время выполнения программы, а при компиляции не делается никаких предположений. Самыми распространёнными функциональными языками с динамической типизацией являются языки семейства Lisp, такие как Common Lisp [4], Racket [5] и Clojure [6], а также язык Erlang [7].

Основным преимуществом статической типизации является обнаружение значительной части ошибок во время компиляции, что повышает безопасность программ и минимизирует вероятность возникновения исключительных ситуаций во время выполнения. Кроме того, информация о типах позволяет генерировать более эффективный машинный код. К недостаткам можно отнести более высокий порог вхождения при изучении языка, громоздкие синтаксические конструкции и запутанные сообщения об ошибках. Всё это усложня-

ет и замедляет процесс написания программ на статически типизированных языках.

Динамически типизированные языки, в свою очередь, обычно проще в изучении, имеют более понятный синтаксис, лучше подходят для обобщённого программирования, основной идеей которого является возможность использовать одни и те же описания данных и алгоритмов без их изменения для различных типов данных. Среди областей применения функциональных языков с динамической типизацией можно выделить разработку систем управления данными, Web-приложений (как серверной, так и клиентской части), финансового программного обеспечения.

Главным недостатком языков с динамической типизацией является высокая вероятность появления ошибок времени выполнения из-за несоответствия типов. Поиск и устранение причин этих ошибок — нетривиальная задача для программиста. Чтобы минимизировать их количество, целесообразно использовать дополнительные средства для анализа текста программ или включить в язык опциональные конструкции, позволяющие повысить безопасность кода.

В данной статье рассматриваются подходы к решению задачи статического анализа кода на функциональных языках с динамической типизацией. Представлен обзор средств анализа для наиболее распространённых функциональных языков (Racket [5] и Clojure [6]). По результатам обзора формулируется заключение о целесообразности разработки средств статического анализа, одновременно включающих возможности рассмотренных подходов и основанных на формальной модели вычисления выражений в функциональных языках.

1. Подходы к статическому анализу

Для современных функциональных языков с динамической типизацией разработан набор методов и инструментов для статического анализа и повышения надёжности кода, которые можно разделить на две группы, представляющие два противоположных подхода: внутренние языковые конструкции и внешние программные инструменты.

К первому подходу относятся методы, требующие модификации исходного кода программы (например, расширения языка или специальный формат комментариев для объявления предполагаемых типов некоторых выражений). В данной работе рассмотрен наиболее широко применяющийся именно для динамически типизированных функциональных языков метод, получивший название ***постепенной типизации*** (*gradual typing*) [8]. Формальное описание метода

основывается на расширении λ-исчисления опциональными аннотациями типов, что применимо для любого динамически типизированного функционального языка. Отметим, что в рамках этого метода должны быть реализованы его основные возможности, которые будут рассмотрены в данном обзоре, а именно: поддержка иерархической системы типов, объявление пользовательских типов данных, аннотирование типов выражений, автоматический вывод типов и непосредственно проверка согласования типов.

Ко второму подходу отнесём методы и инструменты, не требующие модификации исходного кода, например, таковыми являются программные инструменты поиска «мёртвого» кода и неиспользуемых переменных, а также системы, предлагающие варианты упрощения кода.

В настоящей работе рассмотрены методы статического анализа программ на языках Clojure [6] и Racket [5], поскольку на данный момент последние являются одними из наиболее активно развиваемых потомков языка Lisp, используются в различных прикладных областях и для них существует множество библиотек и расширений. Эти языки существенно отличаются от языка Lisp, и поэтому не могут считаться диалектами, однако сохранили его основные свойства: они динамически типизированы, обладают свойством ***гомоиконности*** (текст программы может быть представлен как структура данных языка), работают с функциями как с объектами первого класса, синтаксис языков основан на ***S-выражениях*** [9].

Язык Clojure появился в 2007 году как современный диалект Lisp, поощряющий использование функциональной парадигмы и интегрированный в Java-платформу. В качестве особенностей языка можно указать поддержку многопоточности, наличие «ленивых» коллекций, возможность использования Java API. Язык Racket, ранее называвшийся PLT Scheme, был разработан в 1994 году и предоставляет обширную систему макросов, которая удачно используется как для расширения самого языка, так и для создания новых встроенных предметно-ориентированных языков.

2. Постепенная типизация

2.1 Особенности подхода

Постепенная типизация — способ типизации, при котором некоторым выражениям динамически типизированного языка могут быть присвоены типы как программистом в виде явной аннотации, так и автоматически, если статический анализатор, осуществляющий

проверку типа, поддерживает возможность автоматического вывода типа для неаннотированных выражений. Остальные выражения получают неопределённый тип, который обычно называется `Any` или `Dynamic`, для них тип определяется динамически и может быть уточнён. Проверка типа может осуществляться как самим компилятором, так и с помощью дополнительных библиотек.

При таком подходе система типов образует иерархию: если множество значений типа `X` является подмножеством значений типа `Y`, то `X` называется ***подтипом*** (*subtype*) типа `Y`, а `Y` — ***надтипом*** (*supertype*) типа `X`. Неопределённый тип находится на вершине иерархии, и все остальные типы считаются его подтипами. На нижнем уровне иерархии находится тип с пустым множеством значений, он является подтипом любого другого типа. Во время статической проверки типов считается, что использование подтипа корректно везде, где может быть использован его надтип, а неопределённый тип согласуется с любым другим.

Описанный подход позволяет комбинировать статическую и динамическую типизацию, делая язык программирования более гибким инструментом, подходящим для широкого круга задач. Постепенная типизация ведёт к повышению безопасности кода благодаря статическим проверкам, при этом не загромождая синтаксис языка излишними аннотациями типов (часто очевидными).

2.2 Typed Racket

Диалектом языка Racket, поддерживающим постепенную типизацию, является Typed Racket [10], который предоставляет набор встроенных типов, как базовых (числовые типы, строки, символы, ключевые слова и многие другие), так и параметризованных контейнеров (пара, список, вектор, множество и т. д.). Неопределённый тип носит название `Any`, пустой (ненаселённый) тип — `Nothing`.

Пользовательские типы данных могут быть сконструированы с использованием структур с типизированными полями, объединений и пересечений типов, в том числе рекурсивных, например:

```
; Объявление рекурсивного типа двоичного дерева
(define-type Leaf Number)
(define-type Tree (Rec Tree (U Leaf (Pair Tree Tree))))
```

В последнем примере ключевое слово `define-type` объявляет синоним типа, конструктор типа `U` образует объединение типов (`Leaf` и `Pair Tree Tree`), `Pair` — пару из двух элементов, `Rec` предназначен для объявления рекурсивных типов.

Для объявления типов функций используется конструктор типа `->`. Наиболее распространены следующие синтаксически эквивалентные варианты (здесь `arg-type-n` — типы аргументов, `return-type` — тип возвращаемого значения):

```
(-> arg-type-1 arg-type-2 ... return-type)
(arg-type-1 arg-type-2 ... -> return-type)
```

Аннотирование типов осуществляется с помощью спецформы, обозначаемой двоеточием (`:`) и указывающей тип переменной или выражения, как отдельно от объявления, так и совместно с определением (в том числе и анонимных функций). Для аннотирования типа выражения в произвольном месте программы вместо самого выражения используется конструкция `ann`. Если проверка типа прошла успешна, при дальнейшем вычислении будет возвращено значение исходного выражения. В противном случае компилятором будет зафиксирована ошибка на этапе проверки соответствия типов.

Поскольку в языке Typed Racket реализован механизм автоматического вывода типов, во многих случаях аннотацию типа можно опустить, и тогда тип выражения по возможности будет выведен автоматически. Тем не менее, существует несколько ситуаций, при которых настоятельно рекомендуется использовать аннотацию типов. К таким случаям относятся определения функций с использованием `define` и использование безымянных функций в качестве параметров или результатов работы функционалов. В противном случае результатом вывода типа выражения может стать наиболее общий тип (например, `Any`), и значительная часть статических проверок станет бесполезна.

Одной из отличительных особенностей диалекта Typed Racket является ***уточнение типа*** в зависимости от результата работы предиката для проверки типа (*occurrence typing* [11]). Если предикат используется для ветвления (например, в формах `cond` или `if`), то в соответствующей ветви проверяемое выражение получит более точный тип согласно семантике предиката. В нижеследующем примере функции `fn`

```
(: fn (-> Any Number))
(define (fn arg)
    (if (number? arg)
        (+ arg 1)       ; arg : Number
        (to-number arg))) ; arg : Any
```

в выражении `(+ arg 1)` тип символа `arg` уточняется как `Number`, поскольку это выражение будет вычисляться только в том случае, если предикат `number?` вернёт `#t`. В противном случае тип

символа остаётся `Any`, и сам аргумент должен быть преобразован к числу, поскольку функция `fn` возвращает значение типа `Number` (предполагается, что функция `to-number` уже описана и имеет тип `(-> Any Number)`).

2.3 Typed Clojure

Значительная часть решений, использованных в Typed Racket, послужила основанием для разработки средств типизации для языка Clojure, реализованных в виде библиотеки [12]. Так, иерархия типов аналогична используемой в Typed Racket, за некоторыми исключениями. Во-первых, в Typed Clojure есть гетерогенные коллекции, содержащие элементы разных типов. Во-вторых, можно указывать различные варианты сигнатуры функции, в которых отличается количество и тип аргументов:

```
; Тип функции, которая либо принимает в качестве аргумента строку и
; возвращает число, либо принимает числовой аргумент и возвращает строку.
(Fn [Str -> Num]
    [Num -> Str])
; Тип функции, которая принимает один или два аргумента любого
; типа и возвращает значение любого типа.
(Fn [Any -> Any]
    [Any Any -> Any])
```

Для создания пользовательских типов используются конструкторы пересечения и объединения типов. Роль структур исполняют ассоциативные списки, для которых при объявлении указывается тип значения, находящегося по каждому ключу.

Для более точного автоматического вывода типов все объявления переменных и функций должны быть аннотированы. Аннотация типов осуществляется как отдельно от определения выражения, с использованием функции `ann`, так и совместно с ним при помощи конструкции, обозначаемой `:-`. Она может использоваться в том числе в определении анонимных функций (λ-выражений) и конструкции LET. Кроме того, конструкции `ann-record` и `ann-form` используются для определении типов структур и указания типа произвольных выражений соответственно.

Если в описании типа функции отсутствует аннотация типа аргумента или возвращаемого значения, то по умолчанию используется тип `Any`. Таким образом, следующие аннотации эквивалентны:

```
; Типы аргумента и возвращаемого значения не указаны.
(fn [a] b)
; Тип аргумента -- Any, тип возвращаемого значения не указан.
(fn [a :- Any] b)
```

```
; Тип аргумента не указан, тип возвращаемого значения -- Any.
(fn [a] :- Any b)
; Тип аргумента и возвращаемого значения -- Any.
(fn [a :- Any] :- Any b)
```

Конструктор типа `All`, используемый для объявления полиморфных типов, имеет более широкое назначение, чем в диалекте Typed Racket, за счёт возможности указания явных ограничений на тип. Для указания того, что тип `A` является надтипом типа `B` используется обозначение `A :> B` или эквивалентное ему `B :< A`. Например, полиморфная функция, которая работает только с числовыми типами, имеет тип `(All [[x :< Num]] [x -> x])`. Ограничения можно опустить, тогда в качестве границ будут использованы типы `Nothing` и `Any`.

Подобно Typed Racket, в Typed Clojure реализован механизм уточнения типа в зависимости от потока управления и результатов вызова предикатов проверки типа (*occurrence typing*).

3. Внешние средства статического анализа

В данном разделе будут рассмотрены внешние по отношению к исходному коду инструменты, которые проводят статический анализ программы. Можно выделить следующие основные возможности таких анализаторов:

- Поиск конструкций, которые с большой вероятностью содержат ошибки. Для каждого языка существует свой специфический набор таких конструкций. Обычно к ним относят использование устаревшего (*deprecated*) функционала, обращение к неинициализированным данным, архитектурно-зависимый код. Анализаторы, обладающие этими возможностями, называются ***линтерами*** (*linter*). Изначально Lint — статический анализатор для языка C [13], но название стало нарицательным для всех подобных программ.

- Составление рекомендаций по упрощению кода.

- Поиск «мёртвого» (т. е. неиспользуемого) и повторяющегося кода.

- Проверка оформления кода по заданному шаблону.

- Проверка стандартов программирования, специфических для некоторой области индустрии, где требуется высокая надёжность программ (встроенное программное обеспечение, системы реального времени).

Обычно статические анализаторы кода сочетают в себе несколько из этих возможностей, но часть перечисленного функционала может быть реализована уже в компиляторе или интерпретаторе языка.

Для языка Clojure существует широкий набор инструментов для статического анализа, далее будут рассмотрены самые распространённые из них. Все эти анализаторы в свою очередь написаны на Clojure и являются проектами с открытым исходным кодом.

1. Eastwood [1] — статический анализатор для поиска подозрительных конструкций в коде проекта (линтер). Такие конструкции разбиты на несколько категорий, причём каждая категория может быть по отдельности включена или исключена из проверки. Категории включают:
   - константные выражения в условных конструкциях;
   - переопределение глобального имени в одном пространстве имён;
   - перекрытие глобального имени локальным;
   - неиспользуемые включения пространств имён;
   - неиспользуемые аргументы функций и локально связанные символы;
   - игнорирование результата, возвращаемого из чистой функции.
2. Yagni (You Aren't Gonna Need It) [2] — дополнительный модуль для системы Leiningen, используемой для автоматической сборки проекта и управления зависимостями, производящий поиск неиспользуемого кода в проекте.
3. Kibit [3] — библиотека для упрощения конструкций языка при помощи встроенных функций или более короткой эквивалентной записи. Библиотека позволяет автоматически изменять текст программы или выводить список возможных изменений.

[1] https://github.com/jonase/eastwood
[2] https://github.com/venantius/yagni
[3] https://github.com/jonase/kibit

4. Repetition Hunter [4] — библиотека для поиска повторяющихся фрагментов кода (с точностью до названий переменных) в заданном пространстве имён. В результате работы выводит список повторяющихся выражений, количество повторений и расположение каждого из них.

Для языка Racket набор инструментов значительно меньше. Как наиболее используемые можно выделить следующие:

1. Система так называемых ***контрактов*** [14] языка Racket. В простейшем случае контракт представляет собой предикат, накладывающий ограничение на какое-либо значение. Контракт для функционального значения — это пара контрактов для значения аргумента и результата вычисления функции.

2. Parlinter [5] — инструмент для проверки оформления кода. Проверяемые элементы оформления согласуются с большинством рекомендаций, которые можно найти в документации к соответствующим языкам. Отметим, что инструмент применяется и для других диалектов языка Lisp (Common Lisp, Racket, Scheme).

Заключение

В данной работе рассмотрены основные методы статического анализа кода на функциональных языках программирования, использующих динамический способ типизации, а именно Racket и Clojure. Выделены два основных подхода: добавление в язык новых конструкций (путём создания отдельного диалекта языка или библиотеки функций) и использование внешних программных инструментов, позволяющих производить статические проверки без изменения исходного кода программ. В рамках первого подхода рассмотрены соответственно диалект Typed Racket и библиотека Typed Clojure, которые реализуют механизм постепенной типизации и позволяют производить статическую проверку с возможностью вывода типов. Рассмотренные в работе внешние программные инструменты предназначены для решения только отдельных частных задач статического анализа (например, только упрощение кода или только поиск повторяющихся конструкций).

[4] https://github.com/mynomoto/repetition-hunter
[5] https://github.com/shaunlebron/parlinter

Стоит отметить, что функциональные возможности двух выделенных подходов не пересекаются. Таким образом, целесообразно создание более универсальных средств статического анализа для динамически типизированных функциональных языков, в частности, для языков семейства Lisp. Один из возможных путей состоит в использовании расширенной вычислительной модели, основанной на λ-исчислении. Такая модель может быть дополнена конструкциями, позволяющими не только работать с опциональными аннотациями типов, но и производить анализ потока управления, выявляя, например, недостижимые участки кода или повторяющиеся конструкции. Поскольку λ-исчисление является формальной моделью вычисления функциональных выражений, использование подобной расширенной модели позволит объединить возможности обоих подходов и реализовать их для разных функциональных языков.

Литература

1. Simon Peyton Jones, ed. Haskell 98 Language and Libraries: The Revised Report. — Cambridge University Press, 2003. — 263 p.
2. Xavier Leroy et al. The OCaml system release 4.07. Documentation and user's manual. — Institut National de Recherche en Informatique et en Automatique, 2018. — 750 p.
3. Robin Milner et al. The definition of standard ML: revised. — Cambridge, Massachusetts: The MIT Press, 1997. — 120 p.
4. G. L. Steele. Common Lisp the Language, 2nd edition. — Woburn, MA, USA: Digital Press, 1990. — 1029 p.
5. Racket Documentation [Электронный ресурс]. — Электрон. дан. — URL: https://docs.racket-lang.org (дата обращения 18.12.2018).
6. Chas Emerick, Brian Carper, Christophe Grand. Clojure Programming: Practical Lisp for the Java World. — Sebastopol, CA, USA: O'Reilly Media, 2012. — 630 p.
7. Francesco Cesarini, Simon Thompson. Erlang Programming. — Sebastopol, CA, USA: O'Reilly Media, 2009. — 498 p.
8. Jeremy G. Siek, Walid Taha. Gradual Typing for Functional Languages // Scheme and Functional Programming. — 2006. — P. 81–92.

9. J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine // Communications of the ACM. — 1960. № 3(4). — P. 184–195.

10. Sam Tobin-Hochstadt et al. The Typed Racket Reference [Электронный ресурс]. — Электрон. дан. — URL: https://docs.racket-lang.org/ts-reference/index.html (дата обращения 22.12.2018).

11. Sam Tobin-Hochstadt, Matthias Felleisen. Logical Types for Untyped Languages // ACM SIGPLAN Notices. — September 2010. № 45(9). — P. 117-128.

12. Ambrose Bonnaire-Sergeant. A Practical Optional Type System for Clojure. Honours Dissertation. — University of Western Australia, 2012. — 71 p.

13. Frans Kunst. Lint, a C Program CheckerLint, a C Program Checker. — Vrije Universiteit, Amsterdam, 1988. — 24 p.

14. Robert Bruce Findler, Matthias Felleisen. Contracts for higher-order functions // ACM SIGPLAN Notices. — September 2002. № 37(9). — P. 48-59.

Зайцева О.А., Антоненко В.А.

Разработка и реализация системы холодного старта функции для бессерверных вычислений

Введение

Введем несколько определений, следуя [1,2,3,4]:

Бессерверные вычисления – концепция создания и запуска приложений, которые не требуют управления сервером. Для того чтобы услуга или платформа считались бессерверными, в [5] выделены следующие возможности, которые должны обеспечиваться:

- Отсутствие управления сервером со стороны разработчика: нет необходимости выделять или обслуживать какие-либо серверы.
- Гибкое масштабирование: можно масштабировать приложение автоматически или настраивая его емкость, переключая единицы потребления (например, пропускную способность).
- Высокая доступность – приложения, разработанные с помощью концепции бессерверных вычислений, имеют встроенное решение высокой доступности и отказоустойчивости.
- Отсутствие простоя – плата не взимается, пока не запущена услуга или платформа.

Функция – небольшое приложение, предназначенное для конкретных задач и предоставляемое конечным пользователям по требованию.

Функция как услуга (Function-as-a-Service, FaaS) – основной компонент бессерверных вычислений,включающее управление функциями от имени пользователя, автоматическое масштабирование и выставление счетов. FaaS описывает основные характеристики платформы для запуска функций.

Холодный старт функции – первый вызов функции или вызов функции после продолжительного времени, когда функция не вызывается. Холодный старт функции подразумевает инициализацию всей инфраструктуры функции: выделение виртуальной машины, запуск контейнера, загрузка необходимых библиотек; или ее части, если один или несколько компонентов доступны для повторного запуска.

Теплый старт функции – запуск экземпляра функции из остановленного (но развернутого) состояния.

Масштабирование функции до N экземпляров – горизонтальное масштабирование функции, гарантирующее, что пользователю будет доступно ровно N экземпляров функции в режиме теплого старта, если будет использоваться большее количество, они будут вызваны в режиме холодного старта.

Контейнерная виртуализация – это технология виртуализации на уровне операционной системы, которая создает в ней полностью изолированную виртуальную среду без дополнительных затрат на полноценную виртуальную машину, далее контейнер.

Первый вызов функции происходит в режиме холодного старта. Последующие вызовы этой функции могут быть запущены в режиме теплого старта. Каждая такая загруженная среда, готовая к обработке событий, называется *экземпляром функции*. Теплый старт функции позволяет избежать нежелательной задержки при запуске.

Для уменьшения трат ресурсов впустую, которое возникает при постоянном поддержании функций в режиме теплого старта, введем ряд ограничений на используемые функции:

- Ограничение на количество функций, находящихся в режиме теплого старта;
- Ограничение времени, в течение которого функция находится в режиме теплого старта.

Первое ограничение можно ввести посредством аналогии с кэшированием контента. Самый популярный подход к кэшированию – least recently used (LRU) [7]. Данный подход прост в реализации, но не эффективен. Функции вызываются неравномерно, значит, можно посмотреть, какие функции наиболее популярны. Для этого стоит использовать подход last frequently used (LFU) [8]. Также существует подход к кэшированию, основанный на предсказании «популярности» хранимого контента. В данном случае мы будем понимать «популярность» функции как частоту вызова этой функции. Существующие алгоритмы кэширования, реализующие такой подход, например [9,10], показывают результаты значительно лучше других алгоритмов кэширования.

Второе ограничение можно ввести посредством проверки на время, прошедшее с последнего вызова функции. Данное ограничение, как видно в исследовании [6], уже введено по умолчанию на некоторых платформах.

На данный момент платформы, предоставляющие услуги бессерверных вычислений, минимизируют использование своих ресур-

сов, ограничивая время ожидания повторного запуска функции. Для минимизации времени старта функции большинство таких платформ предлагает избегать холодного старта функции. Предлагается уменьшить время запуска холодного старта функции, чтобы разница была не настолько существенна.

Дальнейшая структура статьи такова: в разделе 1 представлен обзор FaaS платформ. В разделе 2 описана разработанная система холодного старта функции. Для разработанной системы было проведено экспериментальное исследование, описание которого представлено в разделе 3.

1. Обзор FaaS платформ

Целью данного обзора является сравнение существующих FaaS-платформ по способам масштабирования функций и управлением холодным и теплым стартами функций.

Рассмотрим основные FaaS платформы с точки зрения следующих критериев:

- Открытый исходный код
- Автомасштабирование
- Пользовательское масштабирование
- Способ вызова функций
- Условия вызова функций в режиме холодного старта

AWS Lambda [10] автомасштабирует функции и после долгого периода бездействия функции вызываются с задержкой холодного старта функции.

По умолчанию Microsoft Azure [11] автоматически определяет нужные ресурсы и динамически масштабирует функции, которые после долгого периода простоя вызываются с задержкой холодного старта функции. Также эта платформа предлагает разработчику самостоятельно масштабировать функции в зависимости от своих нужд.

OpenWhisk [12] для масштабирования функций использует собственный контроллер OpenWhisk, который использует метрику QPS, которая обозначает количество входящих запросов функции в секунду. По какому принципу функции запускаются из холодного старта, помимо первого запуска, не известно.

OpenFaaS [13] масштабирует функции на основе использования ЦП. По умолчанию всегда один экземпляр функции находится в режиме теплого старта. Разработчики предоставляют возможность масштабирования до нуля с самостоятельным заданием времени бездействия, которое определяется посредством мониторинга контейнеров сервисом Prometheus. Также есть возможность полностью отказаться от автоматического масштабирования. [2] Управление функциями возможно с помощью http-запросов [14].

	Amazon Lambda	Microsoft Azure	OpenWhisk	OpenFaaS
Открытый исходный код	-	-	+	+
Автомасштабирование	+	+	+	+
Пользовательское масштабирование	-	+	-	+
Способ вызова функций	HTTP-запросы и др.	HTTP-запросы	Триггеры	HTTP-запросы
Период сохранения теплого старта	40 минут	40 минут	?	Всегда для одного экземпляра функции

Таблица 1: Сравнение FaaS платформ

По сравнению, представленному в таблице 1, для разработки и тестирования собственной системы холодного старта функции наиболее подходящей платформой оказалась OpenFaaS, так как у этой платформы доступен открытый исходный код и есть возможность управлять функциями внутри сторонней программы посредством http-запросов.

2. Реализация системы холодного старта функции

2.1 Платформа OpenFaaS

По умолчанию в OpenFaaS отключена возможность масштабирования до нуля экземпляров функции. Но данная платформа предлагает инструмент faas-idler, который можно настроить дополнительно для своего проекта. [2]

Модуль faas-idler масштабирует функции OpenFaaS до нуля экземпляров после периода бездействия, который задает пользователь. Запросы, получаемые функциями, faas-idler получает из метрик Prometheus через OpenFaaS API Gateway. [15]

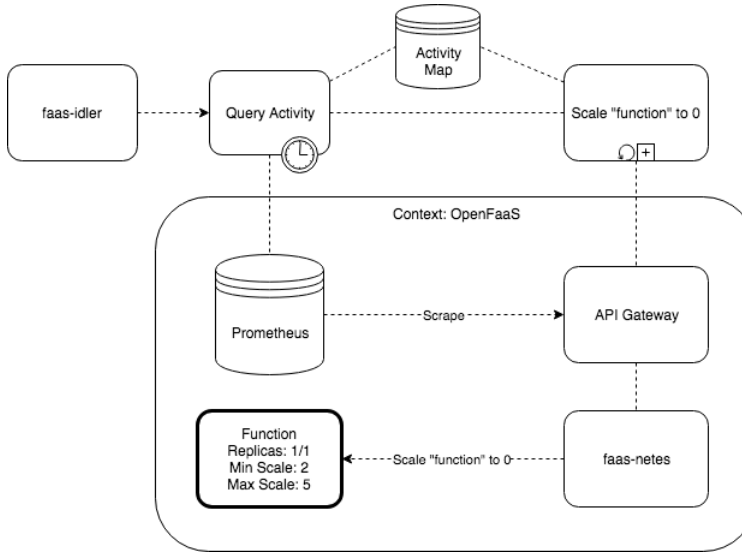

Рисунок 1. Архитектура решения OpenFaaS [15]

Для разрабатываемой системы холодного старта необходимо заменить модуль faas-idler. Далее в разделах 2.2, 2.3 и 2.4 будет предложена реализация модуля в соответствии с выбранными подходами и алгоритмами кэширования.

2.2 Реализация подхода LRU

Модуль, реализованный с помощью подхода LRU, состоит из 2 основных блоков:

- Отправление http-запросов на сервер для масштабирования и выполнения функций
- Управление кэшем

Алгоритм, реализующий подход LRU:

1. Модуль получает запрос с именем функции
2. Модуль проверяет, есть ли доступные экземпляры функции

3. Если не найден доступный экземпляр функции, модуль масштабирует последнюю использованную функцию до нуля и функцию, на которую получен запрос, до 1
4. Модуль обновляет кэш
5. Запрос выполняется

2.3 Реализация подхода LFU

Модуль, реализованный с помощью подхода LFU, состоит из нескольких основных блоков:

- Отправление http-запросов на сервер для масштабирования и выполнения функций
- Управление кэшем (Cache)
- Удаление функции с наименьшим количеством вызовов из кэша (delete)
- Количество вызовов функции из кэша (cache_func)

Алгоритм, реализующий подход LFU:

1. Cache получает запрос на выполнение функции
2. Cache проверяет, есть ли функция в локальном кэше
3. Если доступный экземпляр не найден, delete удаляет функцию с наименьшим количеством вызовов и обновляет cache_func
4. Cache отправляет http-запрос на масштабирование полученной функции
5. Cache отправляет http-запрос на выполнение полученной функции
6. Cache обновляет cache_func

2.4 Реализация алгоритма PopCaching

Алгоритм PopCaching определяет «популярность» функции по гиперкубу, которому она принадлежит. Гиперкуб обладает характеристиками, по которым определяется принадлежность функции к нему. [8] Через http-запросы мы можем получить некоторые основные данные OpenFaaS функций [14]. В данной статье рассмотрены

такие характеристики, как продолжительность функции и размер функции.

Модуль, реализованный алгоритма PopCaching, состоит из следующих основных блоков:

- Отправление http-запросов на сервер для масштабирования и выполнения функций
- Управление кэшем (Cache)
- Обновление кэша (Update)
- Кэширование на основе популярности функции (Estimate)
- Определение гиперкуба функции (Put_func)
- Деление гиперкуба на меньшие гиперкубы (Split)

Алгоритм:

1. Cache получает запрос на выполнение функции
2. Cache определяет гиперкуб, которому принадлежит функция
3. Если гиперкуб не найден, Cache отправляет запрос на определение гиперкуба Put_func
4. Cache обновляет популярность гиперкуба
5. Cache проверяет, есть ли функция в кэше
6. Если доступного экземпляра нет, Estimate решает, нужно ли кэшировать функцию, и обновляет кэш на основе принятого решения
7. Запрос выполняется
8. Если количество запросов превысило заданную границу, Update обновляет локальный кэш
9. Если количество функций в гиперкубе превысило заданную границу, Split делит гиперкуб на меньшие и задает необходимые характеристики

3. Экспериментальные исследования

Целью экспериментального исследования является выявление количества вызовов функций из теплого старта при изменяющихся ограничениях на количество кэшированных функций.

Было сгенерировано 100 различных функций. Каждый алгоритм обрабатывал 10000 вызовов функций. Для вызовов функций были использованы 3 различные функции распределения случайной величины:

1. Нормальное распределение
2. Заданное распределение, зависящее от одного параметра, с различными вероятностями по одному параметру
3. Заданное распределение, зависящее от двух параметров, с различными вероятностями по двум параметрам

Для каждого распределения рассматривались следующие ограничения на кэш:

1. 5% функций от общего количества
2. 10% функций от общего количества
3. 15% функций от общего количества
4. 20% функций от общего количества

Каждый эксперимент возвращал количество функций, вызванных в режиме теплого старта.

На диаграмме 1 сравниваются алгоритмы с различным ограничением на кэш при нормальном распределении номера функции для ее вызова.

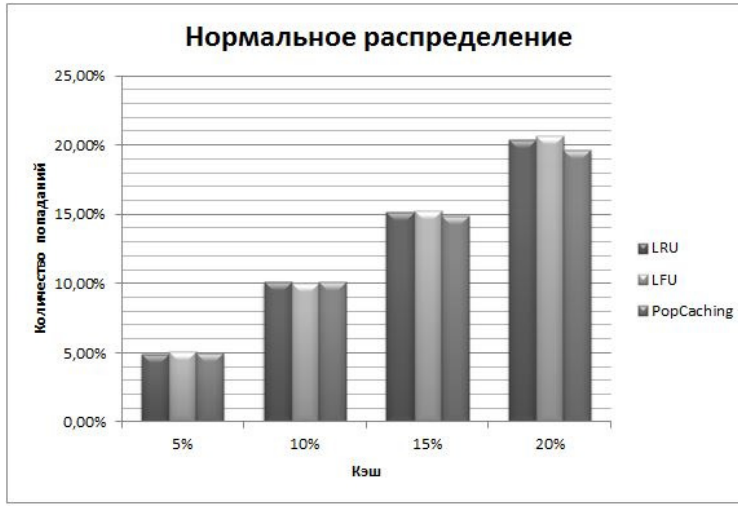

Диаграмма 1. Количество попаданий в кэш при нормальном распределении вызова функций.

На диаграмме 2 сравниваются алгоритмы с различным ограничением на кэш при случайном распределении с различными вероятностями, которые зависели от размера функции или времени выполнения функции. Были проведены тесты для 4 случаев значения вероятности:

- Чем меньше размер, тем больше вероятность
- Чем больше размер, тем больше вероятность
- Чем меньше время выполнения функции, тем больше вероятность
- Чем больше время выполнения, тем больше вероятность

На диаграмме 2 приведены средние значения попадания в кэшированные функции.

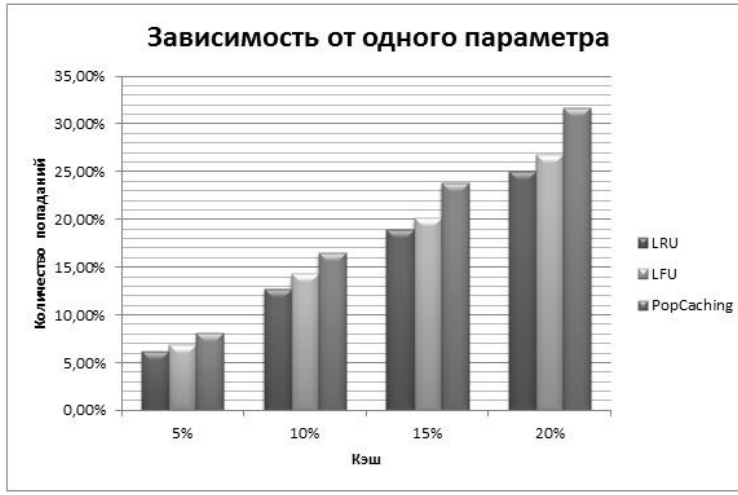

Диаграмма 2. Количество попаданий в кэш при заданном

распределении, зависящем от одного параметра, вызова функций.

На диаграмме 3 сравниваются алгоритмы с различным ограничением на кэш при случайном распределении с различными вероятностями, которые зависели от времени выполнения и размера функции. Были проведены тесты для 4 случаев значения вероятности:

- Чем меньше размер и чем меньше время выполнения функции, тем больше вероятность
- Чем меньше размер и чем больше время выполнения функции, тем больше вероятность
- Чем больше размер и чем меньше время выполнения функции, тем больше вероятность
- Чем больше размер и чем больше время выполнения функции, тем больше вероятность

На диаграмме 3 приведены средние значения попадания в кэшированные функции.

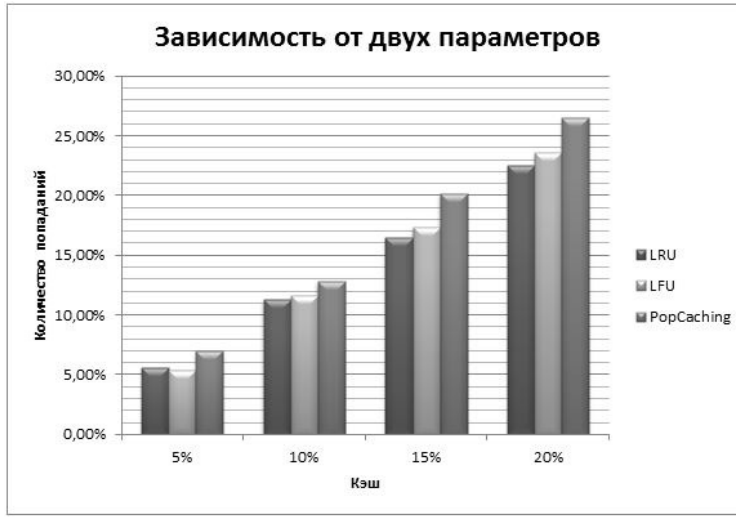

Диаграмма 3. Количество попаданий в кэш при заданном распределении, зависящем от двух параметров, вызова функций.

По результатам проведенного экспериментального исследования можно сделать следующие выводы:

- При нормальном распределении все алгоритмы имеют примерно такой же процент вызова кэшированных функций, как и процент функций, который может находиться в кэше.
- При случайном распределении с зависимостью частоты вызова функции от выбранных параметров явно видно, что PopCaching получает результат лучше алгоритмов LRU и LFU.
- Количество вызовов функций из режима теплого старта у алгоритма PopCaching увеличивается до 32,4% по сравнению с алгоритмом LRU и до 30,1% по сравнению с алгоритмом LFU.
- При кэшировании 20% функций с помощью алгоритма PopCaching до 32% вызовов происходит без задержки холодного старта

4. Заключение

В данной работе представлена система холодного старта функции для бессерверных вычислений с учетом ограничения на количество функций, находящихся в режиме теплого старта.

Был проведен обзор существующих решений проблемы холодного старта, который показал, что большинство существующих платформ масштабируют функции до нуля экземпляров, основываясь только на продолжительности бездействия функции. Решений, удовлетворяющих требованиям, сформулированным в работе, на данный момент не существует. На основании обзора была выбрана платформа OpenFaaS за предоставляемый функционал.

В соответствии с представленными требованиями была предложена реализация системы холодного старта функции на основании популярных подходов кэширования и алгоритма кэширования, основанного на подходе предсказания «популярности», который показывал хорошие результаты по сравнению с используемыми алгоритмами.

В соответствии с разработанной архитектурой реализованы программные модули, являющиеся системой холодного старта функции.

В результате экспериментального исследования была показана зависимость процента вызовов функций из режима теплого старта от вместимости кэша и выбранного алгоритма. При правильном подборе характеристик алгоритм PopCaching получает результат гораздо лучше рассмотренных алгоритмов. Характеристики следует подбирать под поведение пользователя.

Литература

1. CNCF Serverless Working Group *CNCF WG-Serverless Whitepaper v1.0* 2018.

2. Wang L., Li M., Zhang Y., Ristenpart T., Swift M. M. *Peeking Behind the Curtains of Serverless Platforms.* USENIX Annual Technical Conference., 2018.

3. Auto-scaling `https://docs.openfaas.com/architecture/autoscaling/` 28 August 2018

4. Chamberlain R., Schommer J. *Using Docker to support reproducible research.* 2014.

5. Amazon Web Services, Inc. *Serverless Architectures with AWS Lambda: Overview and Best Practices* 2017.

6. Lloyd W., Ramesh S., Chinthalapati S., Ly L., Pallickara S. *Serverless computing: An investigation of factors influencing*

microservice performance IEEE International Conference on Cloud Engineering (IC2E), pp. 159-169, 2018.

7. Jelenkovic P., Radovanovic A. *Asymptotic insensitivity of least-recently-used caching to statistical dependency.* IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428), vol. 1, pp. 438-447, 2003.

8. Alghazo J., Akaaboune A., Botros N. *SF-LRU Cache Replacement Algorithm* Records of the 2004 International Workshop on Memory Technology, Design and Testing, 2004., pp. 19-24, 2004.

9. Li S., Xu J., Schaar M., Li W. *Popularity-Driven Content Caching* IEEE International Conference on Computer Communications (INFOCOM), pp. 1-9, 2016.

10. Fan Z., Wu Q., Zhang M., Zheng R. *Popularity and gain based caching scheme for information-centric networks* International Journal of Advanced Computer Research, vol. 7, no. 30, pp. 71-80, 2017.

11. *Microsoft Azure* `https://azure.microsoft.com/`

12. *Apache OpenWhisk* `https://openwhisk.apache.org/`

13. *OpenFaaS* `https://www.openfaas.com/`

14. *swagger.yml* `https://github.com/openfaas/faas/blob/master/api-docs/swagger.yml` 25 April 2019.

15. *faas-idler* `https://github.com/openfaas-incubator/faas-idler` 4 December 2018.

Пантюхин Л.К., Антоненко В.А.

Разработка алгоритма распределения трафика для SD-WAN решения

Введение

В наше время компании и обычные пользователи всё больше зависят от сетевой инфраструктуры. Практически у всех есть доступ в глобальную сеть, и требования, которые предъявляются к данной услуге, становятся жёстче. Например, пользователь решил посмотреть фильм в большом разрешении на онлайн кинотеатре, или удаленный филиал банка постоянно обменивается информацией с главным отделением. В обоих случаях к услуге доступа в глобальную сеть предъявляются высокие требования такие, как бесперебойность, высокая пропускная способность и низкая задержка. Чтобы им удовлетворять, необходимо разрабатывать новые решения. Но, прежде чем перейти к одному из них, рассмотрим базовые определения.

Глобальная сеть (WAN, Wide Area Network) - сеть, которая охватывает большую географическую область [1].

WAN-подключение – услуга доступа в глобальную сеть, предоставляемая операторами связи.

WAN-соединение – соединение, обеспечивающее доступ в глобальную сеть.

Software-defined wide area network (SD-WAN) – это непосредственное применение технологии программно-конфигурируемых сетей к WAN-соединениям, которые соединяют сети через большие географические расстояния [2].

Под традиционной услугой WAN-подключения мы понимаем подключение с использованием одного WAN-соединения. Такое подключение имеет единую точку отказа, поэтому при обрыве единственного соединения доступ в глобальную сеть будет потерян. Также всему трафику предоставляется одинаковый сервис, независимо от того, какое приложение сгенерировало этот трафик. Традиционная услуга WAN-подключения не всегда обеспечивает соблюдение политик критических для пользователя приложений, или обеспечивает его неэффективно. Например, при отказе единственного WAN-соединения филиал компании может остаться на время без выхода в глобальную сеть, или важный разговор директора может быть сорван из-за того, что трафик VoIP идёт через то же соединение, что и другой трафик, и ему не предоставляется необходимый сервис из-за большой задержки.

В качестве альтернативы традиционному WAN-подключению была разработана технология SD-WAN. Её основная концепция состоит в том, что вместо одного WAN-соединения предоставляется несколько различных WAN-соединений и трафик пересылается через те или иные соединения так, чтобы соответствовать политикам QoS[1] приложения.

SD-WAN – развивающаяся технология и большинство существующих SD-WAN решений являются коммерческими, что предполагает закрытость реализации. При этом нет ни одного широко известного качественного решения с открытым исходным кодом. Под качественным решением имеется в виду проект с открытым исходным кодом, с хорошо написанной документацией и с возможностью интегиироваться с другими проектами, разработанными по стандартам ETSI[2]. Несмотря на отсутствие качественного решения, проблема распределения трафика между несколькими WAN-соединениями актуальна. И в связи с этим целью данной работы является создание алгоритма распределения трафика для решения SD-WAN, так как алгоритм распределения трафика является основой для SD-WAN решения.

В разделе 2 данной статьи представлен обзор существующих SD-WAN решений. В разделе 3 предложен алгоритм распределения трафика. В разделе 4 представлен экспериментальный стенд и проведено экспериментальное исследование предложенного алгоритма.

1. Обзор

Для выполнения поставленной задачи был проведён обзор с целью найти алгоритмы распределения трафика между несколькими WAN-соединениями, а также найти особенности SD-WAN решений, полезные при разработке алгоритма. Критерии обзора следующие:

- **Наличие открытого исходного кода.** Открытый исходный код, а также хорошая документация позволят нам понять, какой алгоритм используется в решении и каким образом происходит минимизация стоимости услуги WAN-подключения, а также соблюдение политик QoS.
- **Классификация трафика.** Работа с множеством потоков трафика не может обойтись без их классификации. Нас интересует, какая классификация трафика используется в каждом решении, а также какими способами обеспечивается.

[1] QoS – Quality of Service

[2] ETSI – European Telecommunications Standards Institute

- **Параметры оптимизации.** Оптимизация – это главное достоинство SD-WAN. Для того, чтобы эффективно обеспечивать соблюдение политик критических для пользователя приложений, необходимо оптимально использовать ресурсы. Помимо исходного кода и классификации нас интересует, по каким параметрам поставщики гарантируют оптимизацию.

SD-WAN решение	Открытый исходный код	Классификация трафика	Параметры оптимизации
Citrix NetScaler SD-WAN[3]	-	Realtime, Interactive, Bulk	C B L P J
Cisco (Viptela) SD-WAN[4]	-	Задается пользователем	C B L P
Riverbed SteelConnect[5]	-	Latency Sensitive, Streaming Media, Best Effort, Background Traffic	C B L
Versa Networks SD-WAN[6]	-	Информации нет	C B
Talari SD-WAN[7]	-	Информации нет	C B L P J
Silver Peak Unity[8]	-	Realtime, Interactive, AnyTraffic	C B L P J
Vrayo Systems VtrunkD[9]	+	Информации нет	B L J
Nante-WAN[10]	+	Информации нет	Информации нет

Таблица 1: Сравнение SD-WAN решений

В таблице 1 представлена классификация информации, полученной в ходе обзора. Условные обозначения для параметров оптимизации:

- C – Cost – Стоимость отправки трафика
- B – Bandwidth – Пропускная способность
- L – Latency – Задержка
- P – Packet loss – Потеря пакетов

- J – Jitter – Джиттер

Стоит отметить, что наличие открытого исходного кода обусловлено только реализацией SD-WAN решения, в то время, как остальные два критерия обзора обусловлены как реализацией, так и алгоритмом, который заложен в основу. Из результатов обзора можно сделать следующие выводы:

- Почти все решения не имеют открытого исходного кода в силу того, что являются коммерческими. А у решений с открытым исходным кодом скудная документация.
- Классификация трафика по умолчанию во всех решениях происходит похожим образом: в отдельный класс выделяются трафик, чувствительный к задержке и джиттеру, но не требующий большой пропускной способности, и трафик интерактивных приложений, чувствительный к потере пакетов и задержке.
- Параметры оптимизации почти во всех решениях одинаковы: поставщики SD-WAN заявляют, что их продукты эффективны по стоимости и гарантируют соблюдение параметров QoS для соответствующих классов трафика.

В ходе обзора необходимые алгоритмы распределения трафика между несколькими WAN-соединениями не были найдены. Поэтому было принято решение разработать свой алгоритм на базе алгоритмов обеспечения QoS в сетевых устройствах.

2. Описание алгоритма

В качестве краевого устройства SD-WAN будем рассматривать SDN[3] коммутатор. В качестве SD-WAN контроллера – SDN контроллер. Весь трафик одного приложения объединяется в поток. Движение потока трафика представлено на рисунке 1:

1. На входной порт коммутатора приходит пакет
2. Если пакет не принадлежит известным потокам:
   (a) Пакет отправляется на контроллер
   (b) Поток пакета классифицируется (**Стадия классификации**)

[3]SDN – Software-Defined Network

(c) Используя известную контроллеру информацию о текущем состоянии WAN-соединений, поток направляется на один/несколько портов (**Стадия планирования**)

(d) Формируются соответствующие OpenFlow[11] правила, которые отправляются на коммутатор

3. Пакет отправляется в соответствующую классу очередь на соответствующий его потоку порт

4. На каждом порту очередность отправки определяется по принципу CBWFQ + LLQ[4]

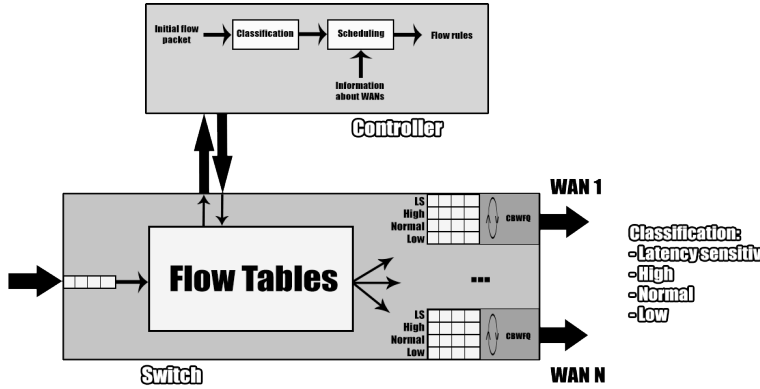

Рис. 1: Схема движения трафика

Далее подробно распишем наиболее сложные и значимые стадии предложенного алгоритма.

Стадия классификации

По результатам обзора сделан вывод, что в отдельный класс следует выделить трафик, чувствительный к задержке и джиттеру и трафик интерактивных приложений. В данной работе будем классифицировать трафик по DSCP[5] значению IP заголовка пакета. В общем случае DSCP значение применяется в механизме обеспечения

[4]CBWFQ – Class-Based Weighted Fair Queuing LLQ – Low-Latency Queue
[5]DSCP – DiffServ Code Point

QoS DiffServ [12]. В нём трафик разделяется на классы в зависимости от DSCP значения, и каждому классу предоставляется разный сервис. В реальных условиях классификация по DSCP значению будет корректно работать только при условии, что DSCP значениям можно доверять. Более надежный способ классификации реализуется при помощи DPI[6] или ML[7][13]. Для определенности используем классификацию одного из SD-WAN решений, рассмотренных в обзоре (Riverbed SteelConnect[5]), но переименуем классы следующим образом:

- Latency Sensitive – Latency Sensitive
- Streaming Media – High
- Best Effort – Normal
- Background Traffic – Low

Стадия планирования

Все правила в таблицах коммутатора имеют некоторое время, по истечению которого правило удаляется и на контроллер приходит событие FlowRemoved[14]. При событиях PacketIn или FlowRemoved соответствующему потоку назначается один из выходных портов по следующему принципу:

1. берётся набор параметров, которые важны при планировании для класса текущего потока
2. значения параметров корректируются для соответствия порядков
3. берётся квадратичная норма для подкорректированных значений
4. для каждого WAN-соединения вычисляется значение нормы
5. выбирается WAN-соединение с наибольшей/наименьшей нормой и на него назначается текущий поток

[6]DPI – Deep Packet Inspection
[7]ML – Machine Learning

3. Экспериментальное исследование

Экспериментальный стенд

Экспериментальный стенд представлен виртуальной SDN. Реализацией алгоритма является приложение, написанное на языке Python, для SDN-контроллера POX. В качестве виртуального коммутатора (vSwitch) будем использовать OvS[8] . Взаимодействие контроллера и коммутатора происходит по протоколу OpenFlow[11]. На рисунке 2 изображена топология виртуальной сети, построенной в Mininet, для тестирования.

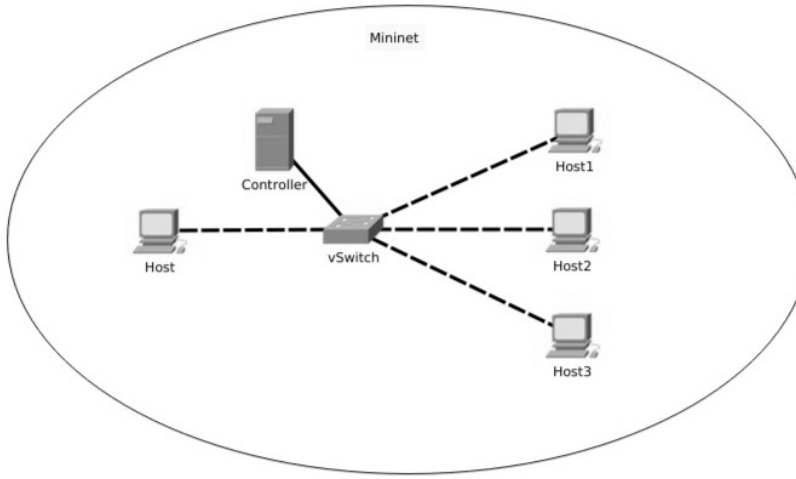

Рис. 2: Топология виртуальной сети

На хосте[9] слева установлен генератор трафика, который отправляет его на коммутатор. Коммутатор имеет несколько портов (в данном случае 3), имитирующих различные WAN-соединения. В процессе работы приложения в таблицах коммутатора появляются правила, которые распределяют трафик между соединениями. На хостах, расположенных справа, запущены ловушки (снифферы), которые фиксируют приходящие потоки.

[8] OvS – Open vSwitch

[9] Под «хостом» в данном случае подразумевается виртуальное конечное сетевое устройство

Результаты экспериментов

Продолжительность эксперимента – 100 секунд. Характеристики соединений приведены в таблице 2. Данные, полученные в ходе эксперимента, представлены ниже на рисунках 3, 4, 5, 6, 7, 8.

Номер соединения	Пропускная способность, Mb/s	Начальная задержка, ms	Стоимость, Rub/Gb
1	1	100	10
2	50	30	40
3	100	30	80

Таблица 2: Характеристики WAN-соединений

Анализ результатов

Стоимость использования WAN-соединений, указанная выше, является нашим предположением и необходима для наглядной демонстрации работоспособности алгоритма. Теперь, после получения результатов, рассчитаем стоимость пересылки трафика в двух случаях:

- в текущем (SD-WAN)
- если бы весь трафик шёл через одно соединение (Традиционный WAN)

Для расчёта используем формулу:

$$S = (R/2^{20}) * C$$

, где S – суммарная стоимость, R – количество байт, C – стоимость 1 Gb

Полученное количество байт делится на 2^{20} для перевода в гигабайты, и результат умножается на стоимость одного гигабайта для получения денежных затрат.

Расчёт представлен в таблице 3. Из анализа, очевидно, что использование нескольких WAN-соединений может дать преимущество в стоимости услуги WAN-подключения.

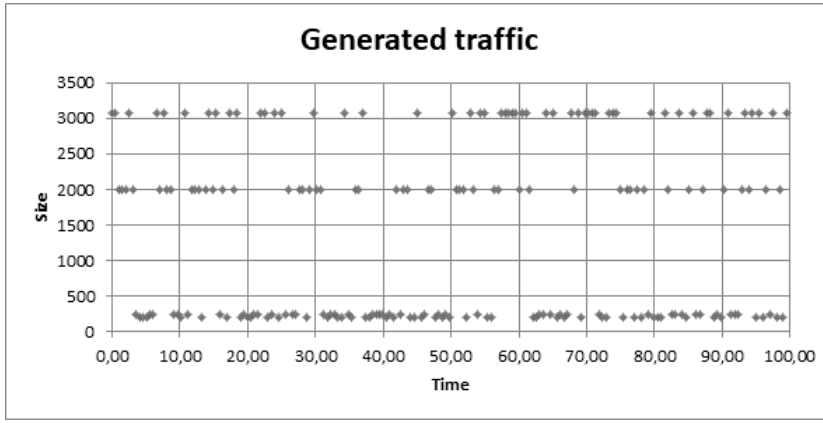

Рис. 3: Сгенерированный трафик

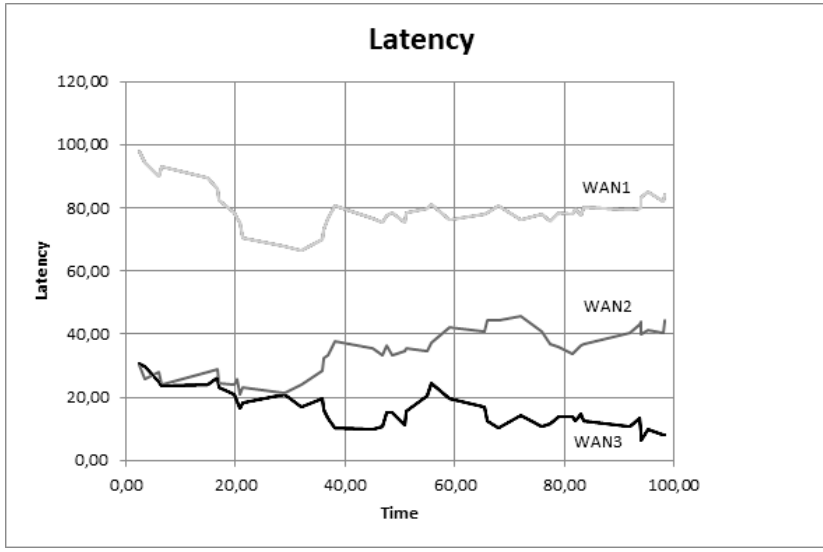

Рис. 4: Задержка WAN-соединений

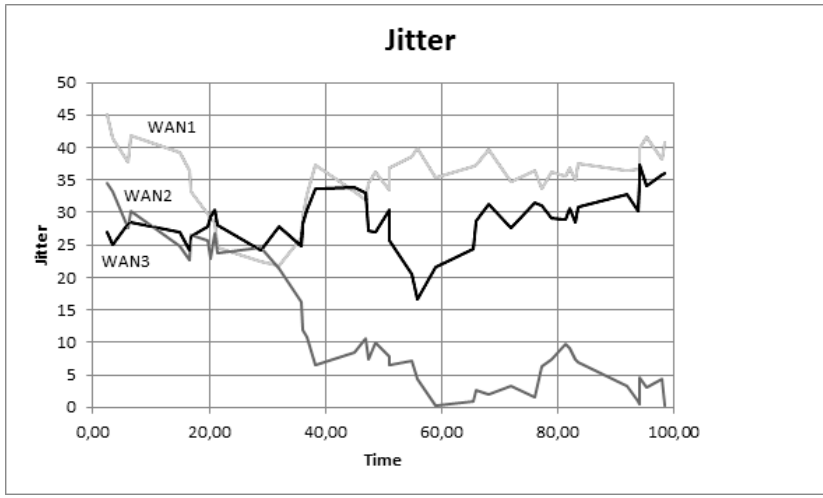

Рис. 5: Джиттер WAN-соединений

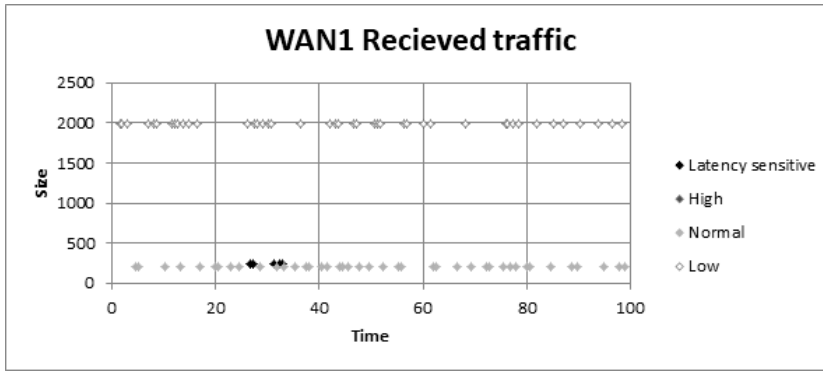

Рис. 6: Трафик, прошедший через WAN-соединение 1

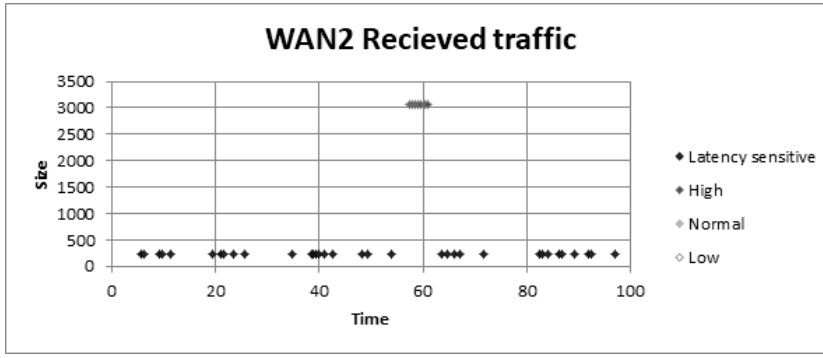

Рис. 7: Трафик, прошедший через WAN-соединение 2

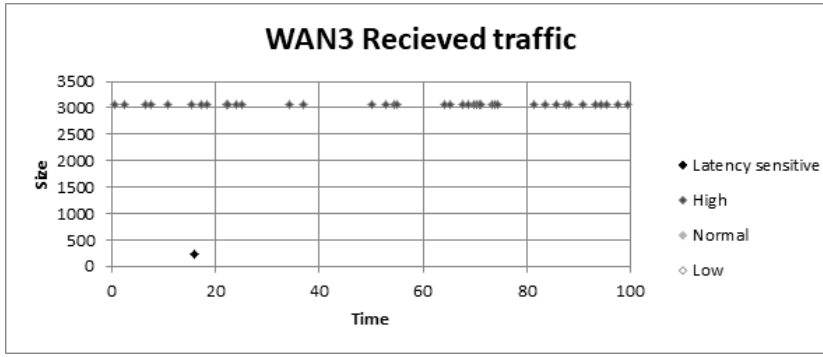

Рис. 8: Трафик, прошедший через WAN-соединение 3

	Традиционный WAN	SD-WAN		
		WAN1	WAN2	WAN3
Получено, byte	248 538	95 497	29 875	123 166
Стоимость услуги, Rub/Gb	80	10	40	80
Суммарная стоимость, Rub	$(248538/2^{20}) * 80 = \mathbf{18.962}$	$(95497/2^{20}) * 10 + (29875/2^{20}) * 40 + (123166/2^{20}) * 80 = \mathbf{11.448}$		

Таблица 3: Расчёт стоимости

4. Заключение

В работе была поставлена цель разработать и реализовать алгоритм распределения трафика для SD-WAN, который удовлетворяет политикам QoS и минимизирует стоимость услуги WAN-подключения. В рамках работы сделано следующее:

- проведён обзор существующих решений SD-WAN
- разработан простой алгоритм распределения трафика
- написано приложение для контроллера POX, реализующее алгоритм
- построена тестирующая среда при помощи Mininet и Open vSwitch
- проведено экспериментальное исследование, показавшее работоспособность предложенного алгоритма

В качестве возможных направлений для дальнейших исследований можно указать реализацию очередей на коммутаторе, распределение одного потока сразу на несколько соединений, а также приведение условий проведения эксперимента к реальным. Например, рассмотрение настоящих тарифов поставщиков услуг связи для параметров WAN-соединений, использование реального трафика вместо генератора и так далее.

Литература

1. RFC1392 *"Internet Users' Glossary"*
2. Kim D., Kim Y.H., Kim K.H., Kim J.B., You G., *"Logically isolated group network for virtual convergence environment over SD-WAN* The Journal of Supercomputing, 2018.
3. *"Citrix SD-WAN Product Documentation."* [Online]. Available: `https://docs.citrix.com/en-us/netscaler-sd-wan`
4. *"Cisco SD-WAN Product Documentation."* [Online]. Available: `https://sdwandocs.cisco.com/product_documentation`
5. *"SteelConnect Documentation."* [Online]. Available: `https://support.riverbed.com/content/support/software/steelconnect.html`
6. *"Versa Networks."* [Online]. Available: `https://www.versa-networks.com`
7. *"SD-WAN: Software-Defined WAN Solutions."* [Online]. Available: `https://www.talari.com/solutions/sd-wan`
8. *"Unity EdgeConnect SD-WAN Solution."* [Online]. Available: `https://www.silver-peak.com/sites/default/files/infoctr/silver-peak-datasheet-unity-edgeconnect-sd-wan-solution-0918.pdf`
9. *"Vrayo Systems VtrunkD."* [Online]. Available: `https://github.com/VrayoSystems/vtrunkd`
10. *"Nante-wan."* [Online]. Available: `"https://github.com/upa/nante-wan"`
11. Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, Jonathan Turner, *"OpenFlow: Enabling Innovation in Campus Networks,"* ACM SIGCOMM Computer Communication Review, vol. 38, no. 2, pp. 69-74, 2008.
12. RFC2474 *"Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers"*

13. Pu Wang, Shih-Chun Liny and Min Luoz, *"A Framework for QoS-aware Traffic Classification Using Semi-supervised Machine Learning in SDNs,"* IEEE International Conference on Services Computing, 2016.

14. *"OpenFlow Events: Responding to Switches."* [Online]. Available: `https://noxrepo.github.io/pox-doc/html/#openflow-events-responding-to-switches`

Синякова М.А., Степанов Е.П.

Оценка задержки потоков виртуального пласта

Введение

В настоящее время в связи с возросшими потребностями к качеству сервиса идет ативное усовершенствование существующих сетевых технологий и разработка новых. Частным примером является разработка сетей нового поколений 5G [1]. Одной из основных особенностей является распределение сетевых ресурсов от потребностей пользователей. Например, при просмотре видео в формате 4K скорость передачи данных является критической величиной и должна быть достаточно велика, чтобы видео воспроизводилось непрерывно. Возможен просмотр видео на разноформатных устройствах, что также требует различного качества сервиса. Например, разрешение на телефоне значительно отличается от разрешения на телевизоре, следовательно требуется передавать меньший объем данных в единицу времени. Кроме того, могут быть разные условия подключения к Интернет-провайдеру, где пользователь может быть подписан на недостаточную для просмотра видео в высоком разрешении скорость, что может быть обусловлено более дешевым тарифом. Как видно из вышеперечисленных примеров, для разных типов приложений необходимо свое качество сервиса - ограничение на доступную пропускную способность каналов передачи данных и требуемая максимальная задержка. Таким образом, некоторым пользователям или под определенный вид трафика необходимо выделять отдельные ресурсы сети - виртуальные пласты (совокупность сетевых ресурсов, предназначенных для передачи данных с определенным качеством сервиса).

В настоящей работе[1] рассматривается только проблема обеспечения требуемой максимальной задержки. Оценка задержки будет даваться с определенной достоверностью, что делает полученную оценку менее строгой, но более приближенной к реальной работе сети. Каждому виртуальному пласту выдается некоторая очередь с заданным приоритетом на коммутаторах для передачи данных. Приоритет очереди равен приоритету виртуального пласта. Вариация задержки происходит при помощи изменения приоритета виртуального пласта. Увеличивая приоритет виртуального пласта, уменьшается

[1]Работа выполнена при поддержке РФФИ (проект №19-07-01112)

задержка передачи данных внутри него. Но как проверить, что выбранного приоритета достаточно для обеспечения качества сервиса? Для этого для выбранного приоритета и множества потоков необходимо уметь вычислять оценку сквозной задержки виртуального пласта - максимальную задержку по всем потокам.

Есть множество методов для вычисления оценки задержки с использованием сетевого исчисления, но к задаче оценки задержки потоков в виртуальном пласте они напрямую не применимы, так как для виртуального пласта отсутствуют методы построения кривых нагрузки и обслуживания, без которых невозможно применение многих методов вычисления оценки задержки. Также для виртуального пласта нет средства для автоматического расчета оценки задержки исходя из этих кривых. Под термином “качество сервиса” понимается пропускная способность и сквозная задержка сети, которые характеризуют производительность сетевого соединения. Рассматриваются только сети прямого распространения. Под сетью прямого распространения понимаем такую топологию, в которой пакеты распространяются только в одном направлении - от "входов" к "выходам". В настоящей работе используется математический аппарат сетевого исчисления для нахождения оценок задержки. Существует два вида сетевого исчисления: детерминированное сетевое исчисление и стохастическое сетевое исчисление. Детерминированное сетевое исчисление дает оценку задержки в наихудшем случае, значение которой может быть недостижимо или выполнимо крайне редко. Поэтому в данной работе будет использовано стохастическое сетевое исчисление [2], которое позволяет вычислять оценку задержки, приближенную к работе реальной сети. Полученная оценка будет верна с некоторой вероятностью (достоверностью), её пользователь будет задавать вместе с остальными ограничениями для создания виртуального пласта. При формировании кривой нагрузки, для упрощения вычислений, в данной работе будем использовать только Пуассоновское распределение для генерации статистических данных, отражающих входной трафик.

1. Стохастическое сетевое исчисление

В данном разделе введем определения кривой нагрузки и кривой обслуживания, задавая их с использованием стохастического сетевого исчисления. Также получим оценку задержки для работы одного потока, которая была бы наиболее приближена к реальной работе сети.

Далее введем основные определения сетевого исчисления.

Кривой обслуживания (variable capacity node) обработчика s называется такая функция β_s, что для каждого временного интервала длины $\tau \geq 0$, величина $\beta_s(\tau)$ не превышает объёма данных, который s способен обработать в течение этого интервала:

$$\forall t : \forall \tau : S(t+\tau) - S(t) \geq \beta_s(\tau)$$

Кривая производительности β_s позволяет построить нижнюю (наихудшую с точки зрения качества сервиса) оценку функции обработчика S.

Кривой нагрузки (arrival curve) для потока с функцией прибытия A называется такая функция α, что для каждого временного интервала длины τ количество переданных в течение него данных этого потока не превышает величины $\alpha(\tau)$:

$$\forall t : \forall \tau : A(t+\tau) - A(t) \leq \alpha(\tau)$$

Кривая нагрузки ограничивает сверху скорость поступления данных потока на каждом интервале заданной длины.

Задержка (delay) $W(t)$ – время прохождения через обработчик той порции данных, которая поступила на него в момент времени t.

Представим ограничение для задержки в виде $P[W(t) > \omega] \leq \varepsilon'$, где ε' вероятность того, что в реальной сети задержка нарушит построенную нами оценку ω. Данную оценку можно получить из моделей EBB (exponentially bounded burstiness). Для фиксированного значения τ модель EBB будет иметь вид:

$$P[A(\tau, t) > \rho(t-\tau) + b] \leq \varepsilon(b), \quad \varepsilon(b) = \alpha e^{-\theta b} \tag{7.1}$$

где $A(\tau, t)$ – входной трафик, пришедший в момент времени от τ до t; ρ – скорость прибытия пакетов, которая зависит от свободного параметра $\theta \geq 0$; b – размер всплеска; $\varepsilon(b)$ – функция, обозначающая вероятность, с которой допускается нарушение построенной оценки для случайной величины $A(\tau, t)$ и зависит от параметра b, коэффициент $\alpha \geq 0$.

Обобщим неравенство, чтобы оно было верно для $\forall \tau \in [0, t]$:

$$P[\exists \tau \in [0, t] : A(\tau, t) > \rho'_A(t-\tau) + b_A] \leq \varepsilon'(b_A), \tag{7.2}$$

где $\rho'_A = \rho_A + \delta$, а δ - калибровочный параметр, $\varepsilon'(b_A)$ – вероятность, с которой может нарушаться построенная оценка. Величины ρ_A и b_A характеристики трафика.

Так как кривая нагрузки и кривая обслуживания имеют вид кусочно-линейных функций, то для получения конечной оценки задержки необходимо пересечение данных прямых или сохранение параллельности, то есть необходимо выполнение условия $\rho_A \leq \rho_S$,

где ρ_S - скорость работы обработчиков. Исходя из необходимого условия выполнения оценок, получаем диапазон для выбора δ, $\delta \in [0, \frac{\rho_A - \rho_S}{2}]$.

Для работы с величинами, которое задаются распределением, введем модель MGF (moment generation function) – производящая функция моментов. MGF - функция от случайной величины X с распределением $\mathbb{F}^X$, имеющая вид: $M_X(\theta) = \mathbb{E}[e^{\theta X}]$, где θ – случайный параметр.

Введем аффинную оценку для MGF трафика:

$$\mathbb{E}[e^{\theta A(\tau,t)}] \ \leq \ e^{\theta(\rho(t-\tau)+\sigma)} \tag{7.3}$$

Используя неравенство Чернова $P[X \geq x] \leq e^{-\theta x}\mathbb{E}[e^{\theta} X]$ можно из аффинной оценки MGF получить модели EBB и EBF для трафика и обработчика соответственно.

Используя выше рассмотренные подходы, авторы статьи [4] рассчитали вероятности для трафика и обработчика с которой будут нарушаться построенные оценки и вывели формулу для вычисления параметра b_A, который в дальнейшем будут использованы для вычислений задержки в сети.

$$b_A \ = \ \sigma_A - \frac{1}{\theta}\left(\ln\frac{\varepsilon'}{2} + \ln\left(1 - e^{-\theta\delta}\right)\right) \tag{7.4}$$

2. Обзор алгоритмов вычисления оценки задержки

Для вычисления оценки задержки для виртуального пласта необходимо уметь вычислять оценку задержки, когда в сети передается более чем один поток данных. В сетевом исчислении часто сначала вычисляют свертку [2] обработчиков, а затем вычисляют оценку задержку для потока, проходящего через один обработчик.

Введем основные понятия и теоремы, которые необходимы для применения ниже рассмотренных методов.

Определение: ***Система обработчиков*** - связное множество обработчиков, соединенное между собой каналами передачи данных.

Определение: ***Кросс-трафик*** - трафик потоков, маршруты которых пересекаются на рассматриваемом обработчике с маршрутом рассматриваемого потока.

Определение: ***Свертка двух функций*** - такая операция $\otimes$, что:

$$(f \otimes g)(t) = \inf_{0 \leq s \leq t}\{f(s) + g(t-s)\}$$

Теорема 1: (*Конкатенация системы обработчиков*) Рассмотрим поток, проходящий через тандем систем S_1 и S_2. Предположим, что S_i предлагает кривую обслуживания β_i, $i = 1, 2$ к рассматриваемому потоку. Тогда объединение двух систем дает кривую обслуживания $\beta_1 \otimes \beta_2$.

Теорема 2: (*Остаточная кривая обслуживания при произвольном мультиплексировании*) Рассмотрим узел, мультиплексирующий два потока 1 и 2 произвольным образом. Пусть β - минимальная кривая обслуживания, гарантируемая узлом; α_2 - кривая нагрузки для потока 2. $\beta_1 = [\beta - \alpha_2]_+$ является кривой обслуживания для потока 1. $\beta - \beta_1$ называют остаточной кривой обслуживания для рассматриваемого потока 2.

Но как учитывать кросс-трафик и как разделять ресурсы между потоками, проходящими через один обработчик? Было рассмотрено несколько методов оценки задержки, которые учитывают кросс-трафик, по следующим критериям: обработчики работают в режиме FIFO, какова точность полученной оценки и алгоритм должен иметь наименьшую вычислительную сложность.

2.1 Анализ общего потока (TFA - Total Flow Analysis)

Идея этого метода состоит в том, чтобы сначала оценить задержку для общего потока трафика (трафик всех потоков, проходящий через данный обработчик) для одного обработчика, а затем суммировать полученные оценки для интересующего нас потока. Если использовать произвольное мультиплексирование потоков, то оценка задержки для каждого узла должна быть вычислена как максимальный период занятости на узлах, потому что общий поток в этом методе рассматривается не как FIFO.

Рассмотрим данный метод на примере, изображенном на рисунке 1.

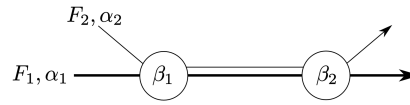

Рис. 1: *Простая модель сети из двух узлов и двух потоков*

Задержка для первого потока F_1 будет равна сумме задержек на каждом узле: $d^{TFA} = d_1 + d_2$, что подробнее описано в [4].

Данный метод не подходит для решения поставленной задачи,

так как не выполнен первый критерий обзора: обработчики обслуживают трафик не в модели FIFO.

2.2 Анализ раздельных потоков (Separated Flow Analysis - SFA)

Основным недостатком TFA является то, что он не разделяет услуги, предоставляемые потокам, то есть не использует теорему конкатенации, которая обеспечивает явное преимущество перед аддитивными методом вычисления оценки задержки. Метод SFA вместо обработки общего потока сначала определяет кривую обслуживания, которое будет предоставлено рассматриваемому потоку на каждом узле, а затем применяет теорему конкатенации к тандему кривых обслуживания. Разделение основано на теореме об остаточной кривой обслуживания при произвольном мультиплексировании. В отличие от метода TFA, в методе SFA предполагается, что каждый поток обслуживается по дисциплине FIFO. Оценка задержки 1 потока для сети, изображенной на рисунке 1, при помощи SFA может быть получена следующим образом:

$$d^{SFA} = h(\alpha_1, [\beta_1 - \alpha_2]^+ \otimes [\beta_2 - (\alpha_2 \oslash \beta_1)]^+)$$

При вычислении оценки задержки для каждого потока в рассматриваемом примере идет дважды учет второго потока, что уменьшает точность вычисления данного метода, описанного подробнее в [4]. Тем самым второй критерий обзора выполнен не оптимально.

2.3 SFA с учитыванием только одного вхождения мультиплексирования (Pay Multiplexing Only Once SFA - PMOO-SFA)

Один из недостатков SFA - это зависимость от порядка применения теоремы 2 (об остаточной кривой обслуживания при произвольном мультиплексировании) и теоремы 1 (о конкатенации системы обработчиков). Возможны две последовательности применения этих теорем. Например, в схеме на рисунке 1 сначала могла быть применена свертка двух узловых кривых обслуживания, а затем применено произвольное мультиплексирование к результирующей системе с одним узлом. Идея данного метода состоит в том, чтобы сначала объединить отдельные системы, чтобы иметь возможность платить за мультиплексирование с другими потоками только один раз. Тем самым при применении PMOO-SFA для случая на рисунке 1 полу-

чаем оценку:

$$d^{PMOO} = h(\alpha_1, [\beta_1 \otimes \beta_2 - \alpha_2]^+)$$

Для работы с ацикличным пересекающимся трафиком сложность данного алгоритма полиномиальна, что доказано в статье [5].

2.4 Линейное программирование

В этом разделе представлен альтернативный метод для вычисления оценки задержки в сетях прямого распространения (feed-forward сети) с произвольным мультиплексированием узлов. Этот метод состоит в формулировании задачи оптимизации, основанной на знании об ограничениях прихода других потоков трафика и гарантиях обслуживания, предоставляемых каждым узлом.

Кривые нагрузки и кривые обслуживания выражаются с помощью кумулятивных функций: кумулятивная функция $F(t)$ подсчитывает общий объем данных, который достиг некоторого условия до момента времени t. В частности для трафика это количество бит, поступивших в сеть к моменту времени t, а для обработчиков - это количество бит, отправленным данным обработчиком в сеть к моменту времени t. Ограничение на количество поступающих бит в систему, накладываемое кривой нагрузи, обозначают как $F(t)-F(s) \leq \alpha(t-s)$. Это означает, что число бит, поступающих между временем s и t, не больше, чем $\alpha(t-s)$. В качестве кривой нагрузки будем рассматривать линейные кривые вида $\alpha_{\sigma_A,\rho_A} = \rho_A t + \sigma_A$, где ρ_A - скорость прибытия пакетов, а σ_A - максимальное число пакетов, которые могут одновременно поступать в систему (всплеск трафика). Ограничение на количество бит, отправленных в систему обработчиком, накладываемое кривой обслуживания, обозначают как $F(t)-F(s) \geq \beta(t-s)$. Разница между выходным трафиком в момент времени t_π и $t_{j\pi}$ не меньше кривой обслуживания на этом интервале. Причем кривая обслуживания имеет вид $\beta_{\sigma_S,\rho_S} = \rho_S t - \sigma_S$, где ρ_S - скорость работы обработчика, а σ_S - задержка на обработку трафика.

Для вычисления оценки задержки задаются ряд ограничений на время, входной трафик, а также задаются соотношения между обработчиками. Целевой функцией для задачи линейного программирования является максимальная значение разности между временем входа трафика в сеть и временем его выхода ($max(t_\emptyset - u)$), что будет подробнее описано в следующем разделе. Метод линейного программирования имеет полиномиальную сложность [3]. Исходя из всех критериев был выбран именно метод линейного программирования вычисления оценки задержки для множественных потоков

3. Построение ограничений для метода линейного программирования

. Рассмотрим подробнее метод линейного программирования. Существует ряд ограничений, по которым строятся неравенства для линейного программирования: временные ограничения, траекторные ограничения, ограничения для входного трафика, ограничения для обработчиков и ограничения для построения оценки задержки.

Под траекторией будем понимать последовательность обработчиков, через которые проходит поток трафика.

Рассмотрим поток i, маршрут которого заканчивается на обработчике j. Пусть Π - множество всех траекторий в сети, заканчивающихся на обработчике j. Так же туда входит $\emptyset$. Введем переменные, которые будем использовать в данном методе:

- t_π - время начала периода отставания на траектории $\pi \in \Pi$. Под обозначением $t_\emptyset$ будем понимать момент, в который из системы выходит пакет с наибольшей задержкой. Для траектории $j\pi \in \Pi, t_{j\pi} = start_j(t_\pi)$ - начало периода отставания на обработчике j, который является началом траектории $j\pi$ и содержит время t_π.
- $F_i^{(j)}(t_\pi)$ для всех $j \in i, j \neq 0$ - накопительная функция для потока i на обработчике j.
- $F_i^{(0)}(t_\pi)$ - накопительная функция, соответствующая количеству байт, поступивших в поток i до момента времени t_π.

3.1 Временные ограничения

Для каждого потока i строится множество Π^i, которое содержит все t_π такие, что траектория π является частью маршрута потока i. Данное множество необходимо упорядочить в соответствии с некоторыми ограничениями:

- Для всех траекторий $j\pi \in \Pi \Rightarrow t_{j\pi} \leq t_\pi$.
- Для всех траекторий, выходящих из одного обработчика ($j\pi_1$, $j\pi_2 \in \Pi$), верно соотношение $t_{j\pi_1} < t_{j\pi_2} \Rightarrow t_{j\pi_1} \leq t_{\pi_1} \leq t_{j\pi_2} \leq t_{\pi_2}$. Так же необходимо учитывать ситуацию $t_{j\pi_1} = t_{j\pi_2}$, тогда возможны два варианта соотношений: $t_{j\pi_1} = t_{j\pi_2} \leq t_{\pi_1} \leq t_{\pi_2}$ и $t_{j\pi_1} = t_{j\pi_2} \leq t_{\pi_2} \leq t_{\pi_1}$.

Таким образом, для каждого потока строятся всевозможные комбинации времен соответствующих началам периодов отставания t_π, удовлетворяющие вышеописанным ограничением и отражающих траекторию потока. В зависимости от расстановки знаков неравенств получаются различные ограничения на переменные t_π, что приводит к нескольким вариантам задач линейного программирования.

3.2 Траекторные ограничения

- *Начало периода отставания:* для всех обработчиков, принадлежащих потоку i $(\forall j \in i)$, и для всех траекторий из этого потока $(\forall j\pi \in \Pi)$ верно $F_i^{(pred(j))}(t_{j\pi}) = F_i^{(j)}(t_{j\pi})$.

- *Ограничения потока:* для всех потоков i и $j\pi \in \Pi,\ j \in i$ добавляем ограничения $F_i^{(0)}(t_{j\pi}) \geq F_i^{(j)}(t_{j\pi}),\ F_i^{(0)}(t_\pi) \geq F_i^{(j)}(t_\pi)$.

- *Неубывающие функции:* для всех потоков и обработчиков, принадлежащих рассматриваемому потоку $(j \in i)$, и $\pi_1, \pi_2 \in \Pi^i$:

$$t_{\pi_1} = t_{\pi_2} \ \Rightarrow\ F_i^{(j)}(t_{\pi_1}) = F_i^{(j)}(t_{\pi_2})$$
$$t_{\pi_1} \leq t_{\pi_2} \ \Rightarrow\ F_i^{(j)}(t_{\pi_1}) \leq F_i^{(j)}(t_{\pi_2}).$$

3.3 Ограничения для входного трафика

Для всех потоков записываем ограничения для каждой пары времен. Для временны́х переменных, удовлетворяющих условиям: $t_{\pi_1} = t_{\pi_2}$ или $t_{\pi_1} \leq t_{\pi_2}$, записывается неравенство вида $F_i^{(0)}(t_{\pi_2}) - F_i^{(0)}(t_{\pi_1}) \leq \alpha_i(t_{\pi_2} - t_{\pi_1})$.

Будем считать, что кривая нагрузки α_i задается в виде $\rho_i t + \sigma_i$. Так как в качестве аргумента подается $t_\emptyset - t_\pi$, то кривая нагрузки принимает вид $\alpha_i(t_\emptyset - t_\pi) = \rho_i(t_\emptyset - t_\pi) + \sigma_i$.

3.4 Ограничения для обработчиков

Для каждого обработчика, суммируя по всем потокам, проходящим через данный обработчик, записываем неравенства вида $\sum_{i \ni j} F_i^{(j)}(t_\pi) - \sum_{i \ni j} F_i^{(j)}(t_\pi) \geq \beta_j(t_\pi - t_{j\pi})$. Если выполняется соотношение $t_{j\pi_1} = t_{j\pi_2}$,тогда для t_{π_1} и t_{π_2}, связанных соотношениями $\{=, \leq\}$, добавляем неравенство $\sum_{i \ni j} F_i^{(j)}(t_{\pi_2}) - \sum_{i \ni j} F_i^{(j)}(t_{\pi_1}) \geq \beta_j(t_{\pi_2} - t_{\pi_1})$.

Будем считать, что кривая обслуживания β_i задается в виде $\rho_i t + \sigma_i$. Так как в качестве аргумента подается $t_\pi - t_{j\pi}$, то кривая

обслуживания принимает вид $\beta_i(t_\pi - t_{j\pi}) = \rho_i(t_\pi - t_{j\pi}) + \sigma_i$.

3.5 Построение оценки задержки

Целевой функцией задачи линейного программирования является $max(t_\emptyset - u)$, где $t_\emptyset - u$ - задержка передачи данных, поступивших в сеть в момент времени u и остававшихся в сети до момента времени $t_\emptyset$. Для новой переменной u существует ряд дополнительных ограничений:

- *Время прихода:* $F_{i_0}^{(0)}(u) \geq F_{i_0}^{(last(i_0))}(t_\emptyset)$.
- *Позиция и монотонность:* для соотношения $t_\pi \leq u \Rightarrow F_{i_0}^{(0)}(t_\pi) \leq F_{i_0}^{(0)}(u)$, а для соотношений $u \leq t_\pi \Rightarrow F_{i_0}^{(0)}(u) \leq F_{i_0}^{(0)}(t_\pi)$.
- *Ограничения для входного трафика:* для соотношения $t_\pi \geq u \Rightarrow F_{i_0}^{(0)}(u) - F_{i_0}^{(0)}(t_\pi) \leq \alpha_{i_0}(u - t_\pi)$, для обратного соотношения $t_\pi \leq u \Rightarrow F_{i_0}^{(0)}(t_\pi) - F_{i_0}^{(0)}(u) \leq \alpha_{i_0}(t_\pi - u)$.

4. Алгоритм оценки задержки потока в виртуальном пласте

Была разработана модель сети для вычисления задержки в виртуальном пласте. Виртуальный пласт – виртуальная сеть, которая характеризуется: виртуальной топологией – неориентированный граф, не содержащий петель; множеством потоков, где поток представлен в виде временного ряда, где каждое значение ряда означает количество байт потока, поступивших на вход виртуального пласта за один интервал времени; требованиями по качеству сервиса для всех потоков виртуального пласта, приоритетом виртуального пласта. На вход модели подаются:

- физическая топология сети – удовлетворяющая термину «сети прямого распространения»;
- количество виртуальных пластов;
- пропускная способность каналов;
- множество виртуальных пластов.

Для каждого потока виртуального пласта строится кривая нагрузки, а для каждого обработчика в сети строится кривая обслуживания при помощи стохастического сетевого исчисления. В зависимости от конфигурации топологии сети и числа потоков, строятся ограничения для задачи линейного программирования для вычисления оценки задержки в сети. Решая эту задачу оптимизации, получаем значение задержки.

4.1 Формирование кривой нагрузки

Для построения кривой нагрузки в виде $\alpha = \rho_A t + b_A$ необходимо найти значения ρ_A и b_A по заданной статистике. Статистика представляет собой количество байт, пришедших за 1 секунду. Считаем, что замеры пришедшего трафика осуществлялись каждую секунду и всего таких замеров было сделано n, причем $n \geq 3$.

Для построения кривой нагрузки необходимо описать заданную статистику каким-то распределением. Для упрощения вычислений возьмем Пуассоновское распределение и при помощи метода проверки гипотезы о виде распределения, предложенного в [6], вычислим параметр λ Пуассоновского распределения. Стоит отметить, что возможно использование и других распределений с применением того же критерия согласия Пирсона. Для того чтобы можно было применить данный метод проверки гипотезы о виде распределения, необходимо преобразовать собранную статистику. Случайной величиной x_i считаем количество байт, пришедших за 1 секунду, а значением случайной величины n_i считаем количество раз, которое данное значение встретилось собранной в статистике, где i - индекс различных значений в файле с собранной статистикой.

Применим алгоритм, описанный в [6]. Данный метод основан на проверке гипотезы при помощи критерия согласия Пирсона. В качестве гипотезы возьмем Пуассоновское распределение с параметром $\lambda = x_{cp}$, где $x_{cp} = \dfrac{\sum_i x_i * n_i}{\sum_i n_i}$. Для начала вычислим вероятность p_i, что пришедшее количество байт соответствуют значению величины x_i. Данная вероятность вычисляется по формуле $p_i = \dfrac{\lambda^i}{i!} e^{-\lambda}$. После этого для каждой случайной величины x_i вычислим критерий согласия Пирсона $K_i = \dfrac{(n_i - np_i)}{np_i}$. Просуммировав K_i, получим величину $K = \sum K_i$ - эмпирическое значение критерия согласия для рассматриваемого распределения.

Критической областью называют совокупность значений критерия, при которых выдвинутую гипотезу отвергают. Областью приня-

тия гипотезы называют совокупность значений критерия, при которых гипотезу принимают. Критическими точками называют точки, отделяющие критическую область от области принятия выдвинутой гипотезы гипотезы. Определим границу критической области для данного распределения. Критическая область для данной гипотезы всегда правосторонняя: $[K_{kp}, +\infty)$, где K_{kp} является теоретическим значением критерия согласия Пирсона и находится по таблицам распределения χ^2, данный подход описан в [7]. Так как критерий Пирсона измеряет разницу между эмпирическим и теоретическим распределениями, то чем больше наблюдаемое значение $K_{current} = K$, тем сильнее довод против основной гипотезы. Если полученное значение $K \in [0, K_{kp})$, то построенная гипотеза верна и собранная статистика удовлетворяет Пуассоновскому распределению с параметром λ. Иначе наблюдаемое значение критерия Пирсона попадает в критическую область $K_{current} \in [K_{kp}, +\infty)$, поэтому есть основания отвергать построенную гипотезу.

Если по критерию согласия Пирсона построенная гипотеза о виде распределения собранной статистики верна, то получаем распределение для входящего трафика, которое имеет вид $p_i = \frac{(x_{cp})^i}{i!} e^{-x_{cp}}$. Используя полученное распределение, построим оценку MGF (7.3) для данного трафика, где $\rho_A = \frac{\lambda(e^{\theta/\nu} - 1)}{\theta}$, а $\sigma = 0$, ν^{-1} - размер пакета входящего трафика. Варьируемая величина θ необходима для получения оптимальной оценки задержки описанной в [2].

Переходим к представлению EBB (7.2) и вычисляем с помощью (7.4) величину всплеска трафика b_A, оптимизировав данную оценку по θ. Получим кривую нагрузки $\alpha = \rho_A t + b_A$ для данного трафика.

4.2 Формирование кривой обслуживания

Для построения кривой обслуживания в виде $\beta = \rho_S t + b_S$ необходимо найти значения ρ_S и b_S. Ограничение на скорость передачи данных задается в параметрах рассматриваемого виртуального пласта. Данное ограничение берется как значение величины ρ_S. Тем самым необходимо вычислить значение b_S.

Так как кривая обслуживания дает оценку снизу, то необходимо оценить наихудший случай загруженности каналов. Рассмотрим построение требуемой оценки на примере, изображенном на рисунке 2.

На рисунке 2(а) изображена последовательность обработки пакетов на коммутаторе. Для того чтобы получить оценку задержки

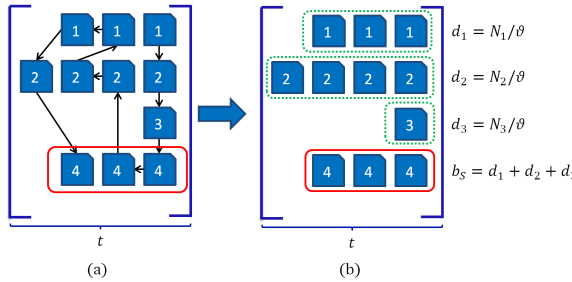

Рис. 2: *Пример оценки задержки обработчика с учетом приоритетов виртуальных пластов.*

на коммутаторе для виртуального пласта с меньшим приоритетом, необходимо знать время, которое будет затрачено на передачу пакетов для виртуальных пластов имеющих больший приоритет, что отражено на рисунке 2(b).

Рассмотрим подробнее преобразование, изображенное на рисунке 2(b). Для интервала времени t ($t = 1$ секунде) рассмотрим, какое количество пакетов может обработать обработчик за этот интервал времени. Максимальное количество пакетов зависит от пропускной способности, выделенной под каждый виртуальный пласт, то есть от скорости передачи данных внутри этого виртуального пласта. Тем самым задается максимальное количество обработанных пакетов за интервал времени для данного виртуального пласта. Для примера возьмем виртуальный пласт с наивысшим приоритетом (пакеты, обозначенные номером 1). Пусть скорость передачи данных внутри виртуального пласта 10 Mbps, тогда количество обработанных пакетов для данного виртуального пласта равен 10 Mb (на рисунке 2 $N_1 = 10$ Mb). Так как за 1 секунду из этого виртуального пласта больше чем 10 Mb данных обработчик обработать не может. Для того чтобы вычислить время d_i, необходимое на отправку пакетов с данным приоритетом, нужно поделить величину N_i на пропускную способность канала ϑ. Вычислив такую величину для всех виртуальных пластов и просуммировав эти значения у пластов с бóльшим приоритетом, получаем задержку перед отправкой пакетов на данном обработчике для виртуального пласта с приоритетом ниже. То есть если рассматривать очередь с приоритетом 4 на рисунке 2(b), то для вычисления b_S для 4-го виртуального пласта необходимо найти $\sum_{i=1}^{3} d_i$, что и будет величиной b_S.

Нерассмотренной задачей остался подбор параметра t. Один из возможных подходов основан на учете таких характеристик вирту-

ального пласта, как размер очереди и доступная пропускная способность. Пусть для виртуального пласта на коммутаторе выделана очередь i, размер которой равен V_i. Пропускная способность внутри виртуального пласта равна ρ_S, а пропускную способность канала обозначим как ϑ. Тогда время, необходимое для обработки пакетов очереди i, вычисляется как $t_i = \dfrac{V_i}{\vartheta}$. С другой стороны время t_i для отправки пакетов из очереди i - это часть временного интервала t, которая пропорциональна части пропускной способности, занимаемой данным виртуальным пластом, причем $\sum_i t_i = t$. Тогда t_i можно задать как $\dfrac{\rho_S}{\vartheta} t$. Тогда получаем формулу для вычисления временного интервала, исходя из физического размера очереди и пропускной способности внутри виртуального пласта, которая имеет вид: $\dfrac{V_i}{\vartheta} = \dfrac{\rho_S}{\vartheta} t \;\Rightarrow\; V_i = \rho_S t \;\Rightarrow\; t = \dfrac{V_i}{\rho_S}$.

4.3 Алгоритм вычисления оценки задержки

Рассмотрим поэтапно работу алгоритма по вычислению оценки задержки внутри виртуального пласта.

После считывания всех данных из входного файла вычисляются кривые нагрузки и обслуживания, затем начинаем составлять задачу линейного программирования на основе неравенств, описанных в главе 4.

Для каждого потока формируем множество временны́х переменных, которые имеют одинаковый конечный обработчик с данным потоком. Например, есть поток, который проходит через коммутаторы $[1, 2, 3]$, то время для обработчика $3 \Rightarrow t_3$, а для обработчика $2 \Rightarrow t_{23}$ и так далее всем коммутаторам в топологии (даже тем, которые не входят в маршрут рассматриваемого потока) ставятся в соответствие временны́е значения. Тогда искомое множество для рассматриваемого потока будет состоять из всех последовательности временны́х переменных, оканчивающихся в 3 обработчике + время $t_\varnothing$. Затем необходимо для рассматриваемого потока сформировать задачи линейного программирования. При построении множества времен для данного потока параллельно строим дерево соотношений этих времен.

Выполним обход по этому дереву, выбирая на каждом шаге обхода листовую вершину, которая затем удаляется из дерева, находящуюся на любом предыдущем уровне, на текущем или на один уровень ниже. Перебрав все вершины, получим последовательность обхода дерева, которая является возможной задачей линейного программирования. Множество всех получаемых последовательностей

описанным алгоритмом перебора даст все возможные задачи линейного программирования. Проверив выполнение временных ограничений, описанных в пункте 4.1 и расставив знаки $[\leq, =]$, получаем временны́е ограничения для задачи линейного программирования.

Возьмем одно временное ограничение и опишем подход формирования неравенств и решения задачи линейного программирования. Исходя из сформированного на прошлом шаге множества времен, для каждого потока в данном виртуальном пласте и для каждого обработчика в этих потоках формируем множество временны́х переменных для данного потока и из него выбираем относящиеся к данному обработчику. Такое разделение упростит нам построение ограничений линейного программирования.

Для каждого потока записываем траекторные ограничения и ограничения для входного трафика (описанные в пунктах 4.2 и 4.3 соответственно). Для каждого коммутатора, входящего в топологию, записываем ограничения для обработчиков (описанные в пункте 4.4). Теперь добавим значение u во временные ограничения, варьируя положение u между разными временны́ми переменными. Для каждого такого положения записываем дополнительные неравенства, описанные в пункте 4.5, и добавляем оптимизируемое значение $max(t_{\emptyset} - u)$, получаем готовый файл с задачей линейного программирования.

Каждый такой файл решается с помощью средства lp_solve [8]. Данное средство решает задачу линейного программирования и выдает значение оценки задержки для одного рассматриваемого потока. Перебрав все временны́е ограничения для одного потока, находим максимальную оценку задержку для него. Вычислив оценку задержки для каждого потока в рассматриваемом виртуальном пласте, выбрав из данных значений максимальное, получим оценку задержки для виртуального пласта.

4.4 Достоверность полученной оценки

При формировании кривой нагрузки задается значение ε - величина, с которой может нарушаться оценка для рассматриваемого потока. Необходимо оценить значение вероятности, с которой полученная нами оценка задержки для виртуального пласта будет корректна. Задержка для каждого потока вычисляется отдельно и единственным связывающим звеном являются ограничения на обработчики, но при построении неравенств считаем, что каждый поток проходит с уже выделенной для него пропускной способностью. Поэтому, учитывая одно из основных ограничений сетевого исчисления: отсутствие сброса и потерь пакетов, считаем потоки в виртуальном пласте независимыми. Тем самым вероятность корректности постро-

енной оценки вычисляется как произведение вероятностей, с которыми верны кривые нагрузки.

Пусть в виртуальном пласте проходят 2 потока, для которых построены кривые нагрузки с вероятностям нарушения данной оценки $\varepsilon_1 = 1\%$ и $\varepsilon_2 = 0,5\%$ соответственно. Тогда вероятности корректности кривых нагрузки равны $99,0\%$ и $99,5\%$ соответственно, а вероятность, с которой верна построенная оценка, равна $98,5\%$ $(0,99 \times 0,995 = 0.98505)$.

4.5 Перспективы применения алгоритма

В статье приведен алгоритм вычисления оценки задержки для виртуального пласта, но наиболее важной задачей является уметь варьировать величину задержки. На рисунке 3 представлен простой алгоритм изменения задержки в зависимости от приоритета виртуально пласта.

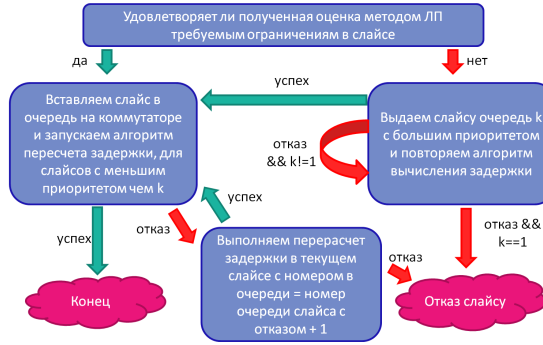

Рис. 3: *Пример алгоритма изменения задержки в сети.*

5. Заключение

В этой статье разработан алгоритм по вычислению оценки задержки виртуального пласта ПКС на основе объединения двух мощных математических аппаратов: стохастического сетевого исчисления и линейного программирования. Дальнейший интерес для исследования представляют: разработка алгоритма, который позволяет удовлетворить требования качества сервиса по задержке при помощи изменения приоритетов виртуальных пластов, модификация

предложенного алгоритма для работы с другими видами распределения трафика.

Литература

1. Zhang S. *An Overview of Network Slicing for 5G* //IEEE Wireless Communications. – 2019.

2. Fidler M., Rizk A. *A guide to the stochastic network calculus.* // IEEE Communications Surveys & Tutorials. – 2015. – Т. 17. – №. 1. – С. 92-105.

3. Bouillard A., Thierry É. *Tight performance bounds in the worst-case analysis of feed-forward networks* //Discrete Event Dynamic Systems. – 2016. – Т. 26. – №. 3. – С. 383-411.

4. Schmitt J. B., Zdarsky F. A., Fidler M. *Delay bounds under arbitrary multiplexing: When network calculus leaves you in the lurch.* // IEEE INFOCOM 2008-The 27th Conference on Computer Communications. – IEEE, 2008. – С. 1669-1677.

5. Bouillard A. et al. *Optimal routing for end-to-end guarantees using Network Calculus* //Performance Evaluation. – 2008. – Т. 65. – №. 11-12. – С. 883-906.

6. Распределение Пуассона [Сетевой ресурс] - URL: https://math.semestr.ru/group/poisson-examples.php (дата последнего обращения 15.12.2019)

7. Ковалевский А., Аркашов Н. *Теория вероятностей и случайные процессы.* – Litres, 2019. С.147 - С.159

8. ip_solver reference guide [Сетевой ресурс] - URL: http://lpsolve.sourceforge.net/5.5/ (дата последнего обращения 15.12.2019)

Степанов Е.П., Войнов Н.А.

Динамическое сегментирование транспортных соединений[1]

Введение

При работе существующих алгоритмов управления перегрузкой производительность TCP соединений деградирует с ростом задержки соединения и увеличением вероятности потери пакетов, не связанной с перегрузкой. Так, с увеличением задержки TCP агенту требуется больше времени, чтобы обнаружить потерю пакетов и провести их повторную передачу. Также TCP агенту требуется больше времени, чтобы занять доступные сетевые ресурсы. В случае, если алгоритм управления перегрузкой использует потерю пакетов в качестве индикатора перегрузки, ошибки передачи нарушают сходимость скорости, ограничиваемой алгоритмом управления перегрузкой, к доступной пропускной способности. [5]

Подход с сегментированием транспортного потока (Split TCP) [1] позволяет решить данную проблему, разбивая соединение на несколько последовательных при помощи прокси сервера. Таким образом, более короткие соединения быстрее реагируют на изменения в состоянии сетевого окружения и более оптимально работают с окном перегрузки.

Однако, такой подход не всегда оказывается эффективным. Улучшение качества исходного соединения зависит от задержки в сети, от количества прокси серверов на пути соединения и от текущей загруженности прокси серверов, через которые проходит это соединение [2]. Для каждого очередного соединения такие характеристики могут быть уникальными. Таким образом, становится актуальной задача разработки и реализации адаптивного метода проксирования TCP соединений, принимающего решение о необходимости и параметрах проксирования, таких как местоположение и количество проксисерверов для каждого соединения.

В данной статье представлен адаптивный подход сегментирования транспортных соединений, выбирающий параметры сегментирования, такие как местоположение и количество прокси серверов для каждого соединения, в зависимости от текущих условий в сети. Проводится экспериментальное исследование эффективности предложенного решения и показывается преимущество данного подхода

[1]Работа выполнена при поддержке РФФИ (проект №18-07-01255-А)

по сравнению с классическим статическим сегментированием в определенных сценариях.

1. Split TCP

Подход с сегментированием транспортного потока (Split TCP) [1] предлагает разбить TCP соединение с большим RTT на несколько последовательных, с меньшими RTT, с помощью использования дополнительных узлов — прокси серверов, которые перехватывают новое соединение и создают несколько последовательных. За счет уменьшения RTT на каждом участке, такое решение позволяет быстрее изменить окно перегрузки, отреагировать на потерю пакета и провести его повторную передачу. Также при установке прокси серверов на стыке отличных по качеству участках гетерогенной сети появляется возможность использовать различные алгоритмы управления перегрузкой для разных физических сред, например, для беспроводной и проводной сетей.

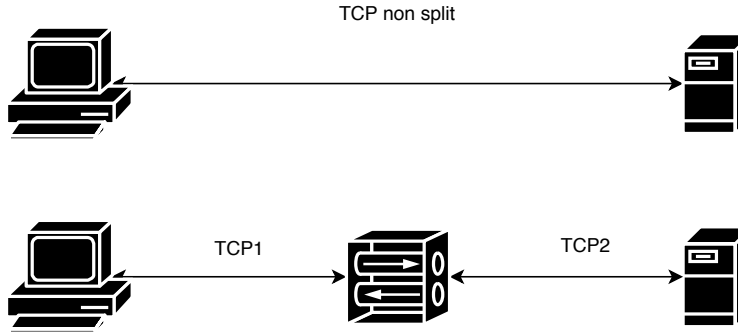

Рисунок 1. Схема работы Split TCP.

Объясним более подробно, как проксирование позволяет увеличить скорость TCP соединения. Рассмотрим соединение, изображенное на рисунке 1. Исходное end-to-end соединение без использования проксирования (обозн. non-split) делится на два последовательных: TCP1 и TCP2.

Положим:

R_{ns} — скорость end-to-end non-split соединения

R_{sp} — скорость split соединения

R_1 — скорость TCP1

R_2 — скорость TCP2

T — круговая задержка (RTT) для non-split соединения

q — вероятность потери пакета для non-split соединения

T_2 — круговая задержка (RTT) для TCP2

q_1 — вероятность потери пакета TCP1

q_2 — вероятность потери пакета TCP2

Согласно [5] пропускная способность TCP соединения без использования Split TCP подхода может быть оценена, как

$$R_{ns} = \frac{1}{T}\sqrt{\frac{3}{2q}}$$

При использовании Split TCP подхода пропускная способность соединения будет равна наименьшей из пропускных способностей на двух участках $R_{sp} = min(R_1, R_2)$. Предположим, что это участок TCP2.

$$R_2 = \frac{1}{T_2}\sqrt{\frac{3}{2q_2}}$$

Таким образом, преимущество, получаемое при проксировании через один узел, оценивается как

$$\frac{R_{sp}}{R_{ns}} = \frac{T}{T_2}\sqrt{\frac{q}{q_2}} > 1$$

Предполагая низкую вероятность потери пакета в проводных соединениях и гомогенность сети, на которой производится проксирование, формулу часто упрощают до

$$\frac{R_{sp}}{R_{ns}} = \frac{T}{T_2} > 1$$

Следует отметить, что полученная оценка имеет ряд ограничений.

Во-первых, оценка пропускной способности выведена для TCP Reno и может не подойти для соединений, использующих иные алгоритмы управления перегрузкой. Однако, согласно [5] качественная оценка, полученная в формулах выше, остается справедливой для таких алгоритмов, как BIC, Cubic и Compound TCP.

Также при оценке пропускной способности на самом медленном из участков предполагается, что происходит насыщенная передача данных, то есть у источника всегда есть данные для отправки, что может быть не так в случае, если отправитель — прокси-сервер, которому еще не дошли данные от источника. Но, так как данный участок является самым медленным на пути передачи, такая погрешность считается допустимой. [5]

Также стоит отметить, что полученные оценки могут быть неверны для короткоживущих соединений, которые завершаются до окончания фазы медленного старта. Также оценки не учитывают возросшее время на установку соединения, связанное с последовательной установкой соединений на каждом сегменте.

Однако, несмотря на данные ограничения, оценка наглядно показывает преимущество Split TCP подхода. Данный подход позволяет увеличить достигаемую пропускную способность по сравнению с классическим TCP протоколом для широкого класса соединений. Так, при использовании одного прокси-сервера пропускная способность может увеличиться до двух раз при условии гомогенности сети и несколько больше для гетерогенных. Нетрудно заметить, что согласно оценке увеличение количества равномерно распределенных прокси-серверов будет продолжать увеличивать достигаемую пропускную способность вплоть до достижения пропускной способности канала. Конечно, в практической реализации таких результатов сложно достичь из-за накладных расходов, возникающих при проксировании. Впрочем, согласно [2], даже при больших накладных расходах подход split TCP может давать преимущество.

Отметим, что данную оценку несложно расширить на случай нескольких прокси серверов. [5]

$$R_{sp} = \frac{1}{T_k}\sqrt{\frac{3}{2q_k}}$$

$$\frac{R_{sp}}{R_{ns}} = \frac{T}{T_k}\sqrt{\frac{q}{q_k}} > 1,$$

где

$$k = argmax\{i = 1 \dots N, T_i * \sqrt{q_i}\}$$

2. Динамическое сегментирование транспортных соединений

Приведенные выше формулы подтверждают практическую выгоду split TCP подхода и позволяют оценивать его возможное преимущество, выбирать расположение и количество прокси-серверов. Однако статический выбор такой конфигурации не всегда остается оптимальным: задержка соединения, служащая основной метрикой выбора таких конфигураций, меняется при различных условиях, таких как увеличение нагрузки на прокси-сервер, увеличение загрузки

очередей на пути соединения, изменение в топологии сети, маршрутах.

Существующие реализации адаптивных методов проксирования [3] обладают рядом ограничений, не позволяющих использовать их в классических сетях.

В качестве динамического метода сегментирования транспортных соединений для классических сетей предлагается следующее решение.

Предполагается, что архитектура предложенного решения строится для некоторого целевого сервера, взаимодействующего с клиентами посредством TCP протокола.

Решение подразумевает наличие нескольких прокси серверов, для возможного терминирования пользовательских соединений, направленных к целевому серверу. Для каждого очередного создания соединения к целевому серверу из данного множества прокси серверов выбирается подмножество, оптимальное с точки зрения оценки.

2.1 Общая схема

Схема предложенного решения состоит из нескольких компонентов:

1. Координатор (coordinator). Данный компонент является централизованным контроллером предложенного решения. Он отвечает за хранение метрик, полученных другими компонентами вычислением оптимальных путей и обслуживанием запросов от прокси-серверов. Данный компонент работает в системе в единственном экземпляре (за исключением сценариев отказоустойчивости) для всего множества прокси-серверов, обслуживающих целевой сервер.

2. Health-Checker. Данный компонент занимается вычислением необходимых метрик, таких как RTT и вероятность потерь пакетов между прокси-серверами. Также компонент производит передачу данных координатору. Данный компонент работает на каждом прокси сервере.

3. Точка присутствия (Point of Presence — PoP). Данный компонент занимается непосредственно проксированием соединений через оптимальные пути, вычисленные координатором. Данный компонент работает на каждом прокси сервере.

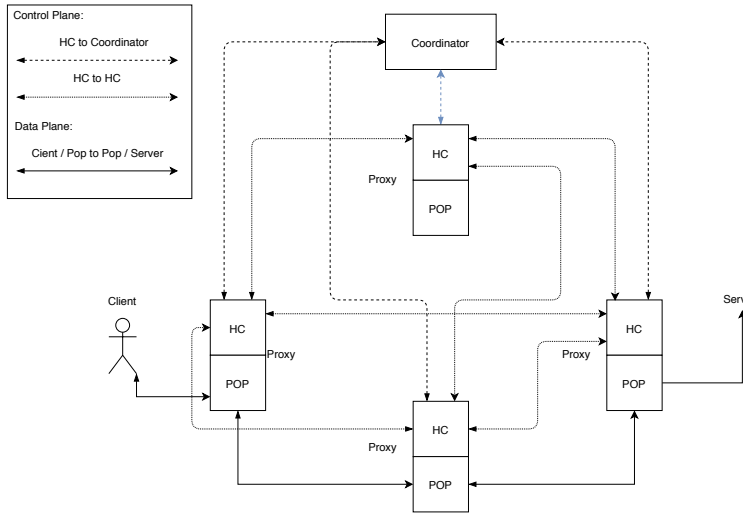

Рисунок 2. Схема предложенного адаптивного метода.

Взаимодействие компонентов схематично показано на рисунке 2.

Из данной схемы видна необходимость попадания запросов клиента на один из прокси серверов, обслуживающих целевой сервер. Предполагается, что это обеспечивает некоторый внешний компонент, не входящий в данное решение. Таким компонентом может служить DNS резолвер, направляющий клиента на ближайший в смысле задержки прокси сервер. Таким образом соединение будет всегда проходить через хотя бы один прокси сервер. Такой подход не всегда будет оптимальным, и обязательное проксирование может давать дополнительные накладные расходы без увеличения достижимой пропускной способности. Однако, вероятность данной проблемы можно существенно снизить, расположив один из прокси-серверов достаточно близко к целевому серверу, например, в том же ЦОД.

2.2 Координатор

Координатор содержит в себе два основных подкомпонента:

1. RPC сервер, обслуживающий запросы прокси серверов.
2. Подкомпонент, вычисляющий оптимальные пути от каждого прокси сервера до целевого сервера.

RPC сервер

RPC сервер реализует следующие удаленные процедуры:

1. `rpc GetNodes(Node) returns (NodeList)`. Данная процедура возвращает список прокси серверов, обслуживающий целевой сервер. Данную процедуру вызывает компонент health-checker, периодически обновляя множество серверов, между которыми считаются необходимые метрики.

2. `rpc RegisterNode(Node) returns (Empty)`. Данная процедура регистрирует обслуживающий прокси сервер в системе. Вызывается компонентом “PoP” при начале работы.

3. `rpc PutMetric(Metric) returns (Empty)`. Данная процедура доставляет вычисленные метрики на координатор. Вызывается компонентом “HealthChecker” периодично.

4. `rpc GetNextHop(Node) returns (Node)`. Данная процедура возвращает адрес следующей точки внутри вычисленного оптимального пути для прокси сервера, посылающего запрос. Вызывается компонентом “PoP” периодично.

Вычисление оптимальных путей

Также координатор отвечает за вычисление оптимальных путей от каждого прокси сервера до целевого сервера на основе вычисленных метрик. Из всевозможных путей выбирается оптимальный, согласно оценке выше. Пути вычисляются измененным алгоритмом Беллмана-Форда. Псевдокод решения приведен ниже. Предполагается, что ассоциативный массив Metrics содержит в себе произведение RTT и вероятности потерь пакетов для каждой пары прокси серверов.

```
for v in Nodes:
d[v] = inf
next_hop[v] = DEST
d[DEST] = 0
for i in range(len(Nodes)):
    for (u,v) in Edges:
    if max(d[u], Metrics(u,v)) + delta < d[v]:
        d[v] = max(d[u], Metrics(u,v)) + delta
        next_hop[v] = u
return d, next_hop
```

Таким образом вычисляются все оптимальные пути от всех прокси-серверов до целевого сервера. Вычисленная информация сохраняется в оперативную память, откуда ее получает при обработке

запроса `GetNextHop` RPC сервер. Данный процесс запускается периодично, перевычисляя оптимальные пути.

2.3 Health-Checker

Данный компонент занимается оценкой метрик между прокси серверами. Он начинает свою работу с получения списка прокси серверов, периодически обновляя эту информацию. Для каждого из полученных серверов и целевого сервера запускается отдельный процесс, вычисляющий необходимые метрики (RTT и вероятность потерь пакета) и периодично отправляющий полученную информацию, усредненную за некоторый промежуток времени, на RPC сервер.

2.4 Точка присутствия

Данный компонент занимается непосредственно проксированием пользовательских соединений к целевому серверу. При получении очередного запроса на подключение процесс локально получает информацию о следующем узле на вычисленном оптимальном пути и подключается к нему, передавая данные, отправляемые клиентом. Таким узлом может быть как очередной прокси сервер, так и сам целевой сервер. Информация о следующем хопе периодически обновляется с помощью удаленной процедуры `GetNextHop(Node)`.

3. Экспериментальное исследование

В качестве демонстрации работы предложенного метода были проведены следующие эксперименты.

3.1 Схема тестового стенда

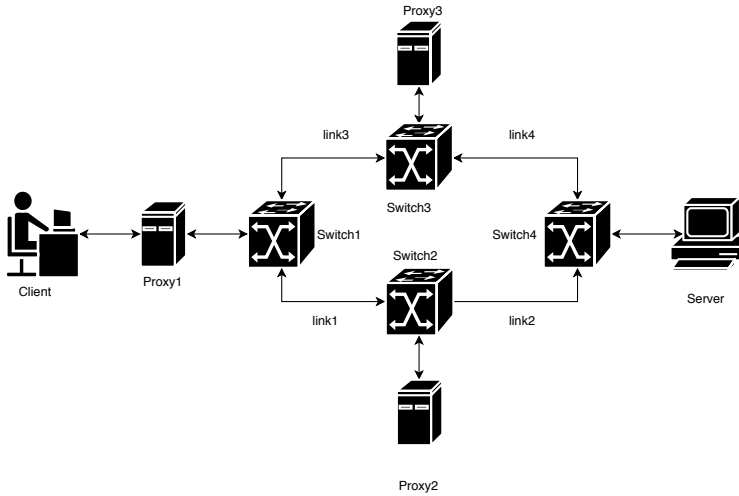

Рисунок 3. Схема тестового стенда.

Для проведения экспериментов была построена следующая топология сети. Для эмуляции представленной выше топологии использовались средства network namespaces и veth. Для задания характеристик линий связи использовались средства эмуляции Tc, Netem, TBF. Для эмуляции работы коммутаторов использовался программный коммутатор OVS. На данной топологии были заранее заданы таблицы маршрутизации: так, путь от Proxy1 до Server проходит через link1, link2. Остальные маршруты соответствуют единственным кратчайшим путям по метрике hop-count.

На прокси серверах, согласно предложенному методы были запущены компоненты Health-Checker и Point-of-Presence, компонент Coordinator был запущен на целевом сервере.

Клиентское приложение в цикле получало с HTTP-сервера файлы различного размера.

В экспериментах оценивалось время получения файлов различного размера, хранящихся на целевом HTTP-сервере. Для получения достоверной оценки, клиент выполнял GET запрос 100 различных файлов одного размера. В качестве времени получения файлов, указанного на графиках, выбиралось медианное время.

Указанные замеры проводились для следующих типов соединений:

1. Соединение без использования проксирования — в данном случае клиент отправлял GET запрос указывая IP адрес целевого

сервера.

2. Соединение с использованием динамичного проксирования — в данном случае клиент отправлял GET запрос указывая IP адрес ближайшего прокси-сервера, дальнейший путь и необходимость проксирования в других точках выбиралась предложенным адаптивным методом.

3. Соединение с использованием статического проксирования — в данном случае клиент также отправлял GET запрос, указывая IP адрес ближайшего прокси-сервера, однако во время начала экспериментов дальнейший путь и необходимость проксирования в других точках настраивалась статически, с помощью добавленных RPC процедур `rpc PutCustomNextHop(NextHop) returns (Empty)` и `rpc DropCustomNextHops(Empty) returns (Empty)`.

3.2 Результаты экспериментального исследования

Эксперимент 1

В данном эксперименте статическое проксирование проводилось через цепочку Proxy1-Proxy2-Server. Фоновая нагрузка на сеть не создавалась.

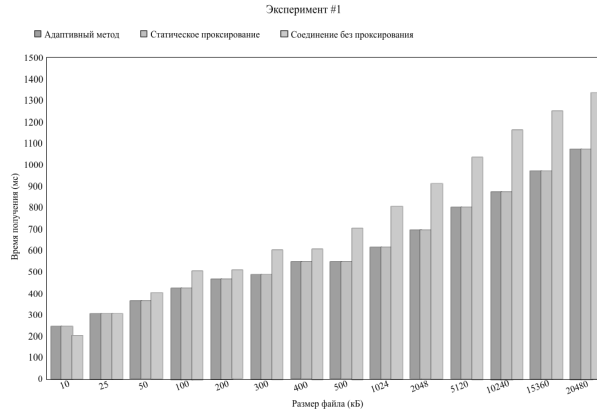

Рисунок 4. Результаты эксперимента 1.

Из графика с результатами замеров видно, что в случае стабильного состояния сети статическое и динамическое сегментирование дают одинаковые результаты, позволяя уменьшить время загрузки файлов по сравнению с соединением без использования проксирования.

Эксперимент 2

В данном эксперименте статическое проксирование также проводилось через цепочку Proxy1-Proxy2-Server. Создавалась фоновая нагрузка с помощью генератора трафика iperf между Proxy2 и целевым сервером. Таким образом эмулировалась увеличенная клиентская нагрузка на Proxy2.

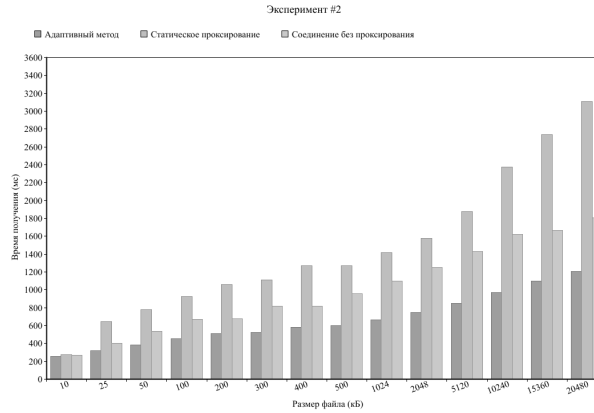

Рисунок 5. Результаты эксперимента 1.

При создании фоновой нагрузки статическое проксирование за счет повторного прохождения через узкий участок сети дает результат хуже, чем соединение без использования проксирования. Адаптивный метод, обнаруживает изменение задержки на участке между Proxy2-Server и выбирает цепочку Proxy1-Proxy3-Server в качестве оптимального пути. Таким образом адаптивный подход позволяет достичь лучших результатов при возникновении перегрузки в сети.

4. Заключение

В ходе проделанной работы был предложен и разработан метод адаптивной сегментации транспортных соединений для классических сетей, принимающий решение о необходимости и параметрах проксирования, таких как местоположение и количество прокси-серверов для каждого соединения, на основе аналитической модели.

Также было проведено экспериментальное исследование эффективности предложенного решения по сравнению с классическим сегментированием транспортных соединений, которое показано преимущество предложенного метода в сценариях с динамической нагрузкой в сети.

Литература

1. J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. *Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations.* June 2001, RFC 3135.

2. Siracusano G. et al. *On the fly tcp acceleration with miniproxy* Proceedings of the 2016 workshop on Hot topics in Middleboxes and Network Function Virtualization. – ACM, 2016. – С. 44-49.

3. Shimamura M., Ikenaga T., Tsuru M. *Splitting tcp connections adaptively inside networks* IEICE TRANSACTIONS on Information and Systems. – 2012. – Т. 95. – №. 2. – С. 542-545.

4. Siracusano G., Bifulco R., Salsano S. *TCP proxy bypass: All the gain with no pain!* Proceedings of the SIGCOMM Posters and Demos. – ACM, 2017. – С. 88-90.

5. Varma S. *Internet congestion control.* Morgan Kaufmann, 2015.

Титов Н.И., Антоненко В.А.

Разработка системы первичной настройки клиентских устройств для доступа к облачным сервисам

Введение

Основная идея системы первичной инициализации сетевых устройств (Zero Touch Provisioning, ZTP) – это упрощение процесса настройки до уровня доступа к ним рядовых пользователей.

В системах традиционного конфигурирования новых устройств, процесс обновления ОС, установки дополнительных патчей и конфигураций производились вручную системным администратором посредством CLI, после чего устройство подключалось в сеть и производились финальные конфигурации уже в рамках этой сети.

Парадигма ZTP подразумевает[1], что устройство, только что физически подключенное к сети, может без взаимодействия со стороны системного администратора получить доступ к серверу первичной настройки(далее ZTP-сервер), который сможет классифицировать устройство как новое и предоставить ему все ресурсы для обновления ПО, установки всех требуемых патчей, выбора необходимых ему конфигураций и интеграцию его в сетевую топологию.

Становится актуальной проблема массовой конфигурации устройств. Она позволит автоматизировать выполнение рутинных задач системных администраторов и сможет стать легкомасштабируемым и конфигурируемым решением. Основным направлением исследования является создание системы первичной настройки сетевых устройств с открытым исходным кодом на примере OpenWRT, используя проанализированные в обзоре коммерческие решения. Такая система позволит создать платформу для автоматической развертки вычислительных кластеров [2], кампусной сети Wi-Fi [2], автоматической конфигурации групп устройств для сетевого туннелирования.

В разделе 2 будут формально определены цель и задачи данной работы, в разделе 3 представлен обзор существующих подходов к решению данной проблемы, в 4 разделе представлено описание алгоритма предложенного решения и описана его реализация. Для реализации было проведено экспериментальное исследование и представлены его результаты в 5 разделе.

1. Цель исследования и постановка задач

Целью данного исследования является создание открытой системы *Zero Touch Provisioning* для *OpenWRT*, которая позволит без участия системного администратора проводить инициализирующее обновление и последующие конфигурирование устройств. Была поставлена задача анализа существующих решений, разработка и реализация предложенного алгоритма и экспериментальное исследование на предмет функционального тестирования полученного прототипа.

2. Обзор методов Zero Touch Provisioning

Выделим основные виды систем первичной настройки:

- Традиционный метод конфигурирования
- Идея *Zero Touch Provisioning*
- Реализации *ZTP* от *Cisco* и *Juniper*

Целью данного обзора является описание алгоритмов взаимодействия клиентского устройства с *ZTP*-сервером, выявление их общих признаков.

Критериями были выбраны: тип операционной системы, идентификация в сети, алгоритм первичной настройки и открытость исходного кода.

2.1 Традиционный метод конфигурирования

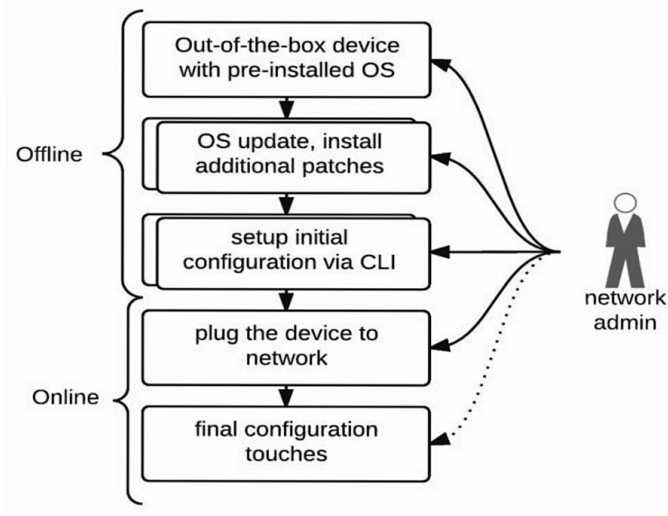

Рисунок 1. Классический метод настройки нового сетевого устройствах[2]

Сетевыми устройствами традиционно управляли через (*Command Line Interpreter*, далее – *CLI*). Например, коммутаторы подключаются с предварительно загруженными сетевыми операционными системами. Сетевые инженеры используют *CLI* для настройки устройства, данный процесс может быть разделен на следующие основные шаги[2](*см. рис 1*):

1. Новое устройство уже имеет предустановленную ОС для начальной загрузки устройства. При запуске "из коробки" устройство находится в автономном режиме, в то время как администратор проверяет версию операционной системы и устанавливает обновления, патчи исправления ошибок, или любые иные изменения функций по мере необходимости.

2. Выполняется начальная конфигурация, чтобы установить сетевое соединение. Включает в себя такие параметры, как: информация об администраторе и аутентификации пользователя, *IP*-адрес управления и шлюз по умолчанию, основные сетевые службы (*DHCP*, *NTP* и т.д.). Процессы включения выбранных сетевых протоколов также являются примерами процесса начальной загрузки.

3. После проверки исходной ОС устройство может быть установлено в сеть (подключено кабелем), после чего можно выполнить дальнейшую настройку (локально через консоль или с помощью протокола удаленного доступа). Эти финальные конфигурации являются срециализированными для расположения в сети.

2.2 Метод *Zero Touch Provisioning*

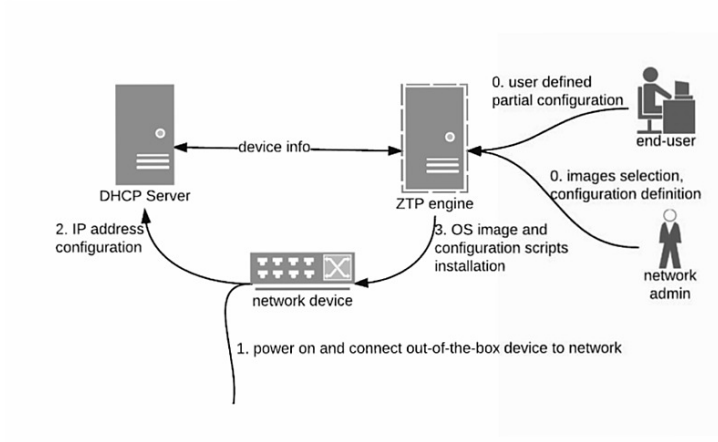

Рисунок 2. Процесс *ZTP*[2]

В случае концепции *ZTP*, администратор, получая оборудование, физически подключает кабель коммутатора в сеть. Как только устройство включено, оно автоматически использует стандартные сетевые протоколы для получения всего, в чем он нуждается для инициализации. Процесс обычно включает следующие шаги[2] (с небольшими вариациями в зависимости от поставщика)(*см. рис 2*):

1. Включение устройства и подключение к сети.

2. Устройство отправляет запрос *DHCP* и получает назначенный IP-адрес для сетевого подключения и управления.

3. Сервер *DHCP* предоставит устройству информацию о том, как связаться с сервером *ZTP* (с одним или несколькими, обычно это *TFTP*-сервера), чтобы:

   - получить правильный образ операционной системы

- получить и активировать нужный файл конфигурации на основе профиля конкретного приложения

Другая особенность *ZTP*, влияющая на универсальность и доступность этой концепции — интуитивно понятный интерфейс. Эта функция очень важна, так как она позволяет пользователям осуществлять контроль над сетевой инфраструктурой, когда этого требует конкретная служба.

2.3 *Cisco*

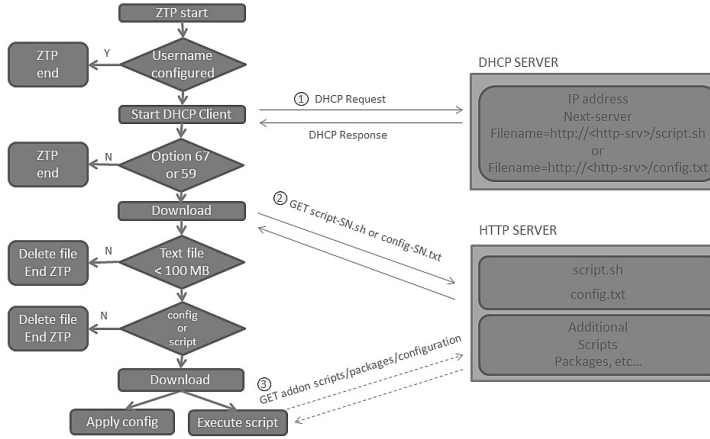

Рисунок 3. *Cisco Zero Touch Provisioning*[3]

Cisco Zero Touch Networking Services, как показано на рисунке(*см. рис 3*), предоставляет решение развертывания системы первичной настройки, где само устройство подключается к *ZTP*-серверу *Cisco*, чтобы принять полную конфигурацию[3]. Это возможно благодаря общему файлу начальной загрузки для конечных пользователей сервис-провайдеров, пользующихся данным сервисом. Фреймворки *Cisco Networking Services*, предоставляют клиентам сервиса возможность создавать конфигурации начальной загрузки без использования информации о устройстве или сети. Возможности *Cisco Zero Touch* позволяют принимать устройству конфигурационные файлы с удаленного *DHCP*-сервера во время начального развертывания без вмешательства конечного пользователя.

2.4 *Juniper*

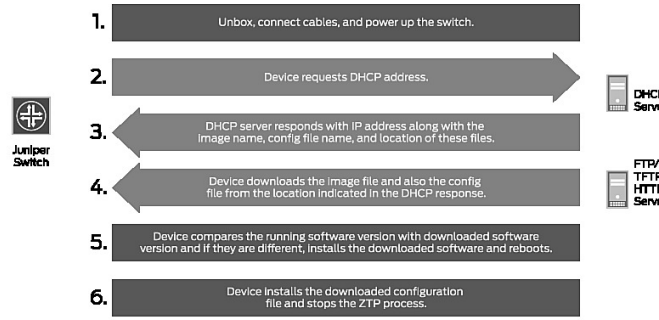

Рисунок 4. Процесс *Juniper Zero Touch Provisioning*[4]

В подходе *Juniper* новое устройство автоматически инициализируется конфигурациями портов и выделением IP-адреса исходя из требований к расположению в сети. [4] Когда клиент подключен к сети с заводской прошивкой, *ZTP*-процесс загружает необходимое ПО и конфигурационный файл. На первом этапе *ZTP* предоставляет стандартный файл настроек, основанный на типе оборудования (*см. рис 4*). Второй этап основан на особенностях аппаратной части, используя преимущество автоматизации *Juniper Networks*, формируются уникальные настройки из сохраненной на сервере информации о сетевой топологии сети. Если устройство было удалено из сети и заменено другим, то новое получает такой же набор специальных конфигураций, несмотря на различия в *MAC*-адресе или серийном номере.

2.5 Сравнение решений

На основе обзора были выделены общие подходы к реализации процесса автоматического конфигурирования: инициирующим действием является DHCP-запрос от устройства, начальная инициализация в системе происходит с помощью MAC-адреса. Сохранение информации о локальной сетевой топологии прзволяет производить замену устройства без ручного создания соединений.

Сравнительная таблица(*см. табл. 1*), характеризует каждый вид конфигурации по предложенным критериям.

	Cisco	Juniper	Предложенное решение
Тип лицензии	commercial	commercial	opensource
NOS	Cisco IOS	JunOS	OpenWRT
Тип ZTP-сервера	TFTP	FTP	HTTP
Доступ к ZTP-серверу	DHCP-запрос	DHCP-запрос	DHCP-запрос
Идентификация устройства	MAC/Serial number, LLDP	MAC/Serial number	MAC, id в базе данных

Таблица 1: Сравнение решений

3. Алгоритм ZTP для OpenWRT

По результатам обзора был предложен алгоритм, предоставляющий решение *Zero Touch Provisioning* для *OpenWRT*. Устройство должно получить адрес *ZTP*-сервера, после чего отправить на него запрос на регистрацию. Сервер, убедившись, что устройство поддерживает обновление с помощью *ZTP*, выполняет с устройством заранее заданные инструкции.

3.1 *DHCP*-сервер

В начале процесса клиентское устройство не имеет доступа к *ZTP*-серверу. Находясь в локальной сети, устройство отправляет запрос к *DHCP*-серверу. В процессе запроса устройство проверяет, является ли это именно тем *DHCP*-сервером, который участвует в *ZTP* процессе для этого устройства. Данная особенность предоставляет возможность использования нескольких *DHCP*-серверов как в рамках одной локальной сети, так и в рамках *ZTP*-процесса. Одновременно устройство получает *IP* от *DHCP*-сервера. Далее, валидировав корректный сервер, устройство через его опции получает адрес *Zero Touch Provisioning* сервера.

Необходимо отличать устройство, которое требует начальной настройки и устройство, уже прошедшее этот этап, так как в рамках локальной сети возможна ошибка работы системы. Важным остается вопрос идентификации клиента для *ZTP*-сервера, каким образом сервер, независимо от динамически меняющихся параметров должен однозначным образом идентифицировать устройство в своей базе, например, при физической замене устройства в топологии или при разрыве процесса конфигурирования.

3.2 *ZTP*-сервер

Получив адрес *ZTP*-сервера, устройство отправляет *http*-запрос с целью регистрации в базе данных. Сервер, в свою очередь, проверяет, впервые ли устройство подключается или же оно уже существует в базе.Для нового подключения сервер отправляет на устройство инициализирующий скрипт, который может как запрашивать обновление ПО, так и загружать другие скрипты для выполнения. Если же устройство было удалено из сети физически, то новое получит все пакеты обновления и конфигурации как для удаленного. После проведенной операции обновления устройство ожидает дальнейших интсрукций и обновлений со стороны сервера.

Со стороны использования *OpenWRT* также появляется ряд ограничений:

- Отсутствие поддержки *tftp*-клиента
- Упрощённый клиент *udhcpc* – ограниченная поддержка *dhcp*-опций
- Различия в поддержке встроенных пакетов в загрузчиках у вендоров.

3.3 Программная реализация

В качестве системы ОС для исследования была выбрана *OpenWRT x86-64*, так как она содержит в себе максимально полный набор пакетов и конфигураций, одинаковый для производителей аппаратной части[5]. С другой стороны, удобство в выборе именно этой версии связано с ее простотой в проведении экспериментов: данная *NOS* имеет созданные разработчиками образы, использующиеся для виртуальных машин *VirtualBox*.

OpenWRT устройство получает от *DHCP*-сервера адрес *ZTP*-сервера в опции *next-server*, а также *bootfile*, который используется как идентификатор, чтобы отсеять другие возможные *DHCP*-сервера. Далее устройство регистрируется по протоколу *HTTP* на *ZTP*-сервере, при этом посылает некоторые свои параметры (*mac, hostname, release*). Настройки полученные по *DHCP*, сохраняются в файле */etc/register.conf*. *ZTP*-сервер осуществляет раздачу файлов по *HTTP*-протоколу, например, обновление ОС.

ZTP-сервер, при поступлении запроса на регистрацию устройства, проверяет, не было оно уже зарегистрировано.Его параметром, однозначным образом ему соответствующим, служит *id*. При удалении из базы, новое подключенное получит последний свободный *id*.

Это позволяет сохранить назначение и топологию при физической замене элементов топологии. Если устройство уже зарегистрировано, сервер обновляет параметры устройства в БД, если нет, то регистрирует и выполняет инициализацию устройства путем выполнения скрипта *init.sh* на нем. Данный скрипт устанавливает *hostname* и обновляет прошивку, если она отличается от текущей.

4. Экспериментальное исследование

С помощью серии экспериментов оценивалась зависимость времени развертки устройств от количества. Для тестирования системы использовалась группа из 3, 6 и 9 коммутаторов, которые настраивались традиционным способом и с помощью предложенного в статье метода.

4.1 Окружение для экспериментов

Будем считать, что одно *ZTP*-процесс по времени сопоставим с настройкой устройства в классическом (*Manual*) режиме, когда каждый аппарат настраивается по очереди вручную. Система тестирования запускалась на гостевой операционной системе (*Ubuntu 18.04 64-bit*, виртуализация посредством *VirtualBox*: 4 ядра, ограничение нагрузки: 80%, 4 ГБ ОП) на компьютере (*Intel Core i5-6300HQ 2.30GHz x 4, Ubuntu 18.04 64-bit*). В хостовой операционной системе создавалось 2 виртуальных интерфейса. Каждая виртуальная машина посредством сетевого моста подключалась к локальной сети, созданной на хостовой машине. На хостовой машине располагался *ISC-DHCP* сервер и был запущен сервер *ZTP*.

4.2 Результаты

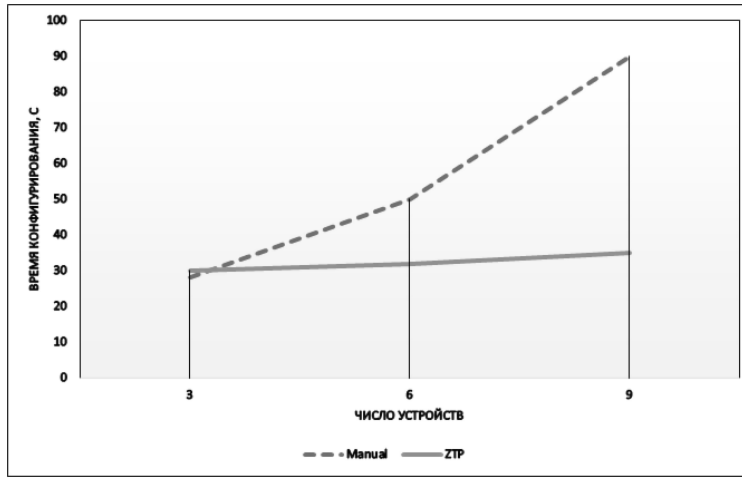

Рисунок 5. Среднее время конфигурирования 3, 6 и 9 устройств в секундах

По результатам эксперимента(*см. рис 5*) наблюдается линейная зависимость роста ускорения *ZTP* над *Manual* от количества устройств. Можно следать вывод, что методы параллелизации процессом управления каждым из устройств, работа с БД и время, требующееся на инициализацию устройства в системе конфигурирования, не оказывают существенного влияния на производительность предложенного решения с ростом числа устройств в рамках небольшой сети.

5. Заключение

По результатам исследования были изучены подходы к автоматической настройке сетевых устройств. На основе изученных на рынке решений был предложен алгоритм автоматической настройки устройств на базе *OpenWRT*. В рамках исследования была доказана возможность применения данного алгоритма для любых устройств с поддержкой *OpenWRT*, принимая особенности их аппаратной реализации. Логичным продолжением исследования будет дальнейшее развитие системы последующей конфигурации устройств, введение более эффективных алгоритмов доступа к устройствам, введение защиты передачи данных, оптимизация количества перезагрузок при обновлении.

Литература

[1] K. Watse, M. Abrahamsson, I. Farrer
Zero Touch Provisioning for Networking. - IETF, 2018.

[2] Y. Demchenko, S. Filiposka, R. Tuminauskas, A. Mishev, D. Regvart, K. Baumann, T. Breach
Enabling Automated Network Services Provisioning for Cloud Based Applications Using Zero Touch Provisioning. - IEEE/ACM, 2015.

[3] Cisco Networking Services
Zero Touch Cisco Networking Services Configuration Guide. - Cisco IOS Release, 2012

[4] Juniper Networks
Understanding Zero Touch Provisioning. - Juniper TechLibrary, 2016

[5] OpenWRT
`https://openwrt.org/docs/guide-user/installation/openwrt_x86` - OpenWRT Project, 2019

Королев Л.Н., Мельников В.А.

Об ЭВМ БЭСМ-6

Исходный вариант статьи был опубликован в 1976 году в журнале ВЫЧИСЛИТЕЛЬНЫЕ СРЕДСТВА И ВСПОМОГАТЕЛЬНОЕ ОБОРУДОВАНИЕ СИСТЕМ в связи с десятилетием ввода в эксплуатацию машины БЭСМ-6. В публикуемом ниже варианте Л.Н.Королевым внесены некоторые изменения в первоначальный текст, состоящие в замене настоящих времен глаголов на прошедшие и в добавлении нескольких фраз эмоционального содержания, навеянных современным временем. Эти редакторские правки сделаны для того, чтобы статья стала более удобочитаемой сегодня.

В 2002 году, в год столетия со дня рождения патриарха отечественной вычислительной техники, академика Сергея Алексеевича Лебедева, исполнилось 36 лет со дня ввода в эксплуатацию машины БЭСМ-6, созданной под его руководством и при его непосредственном участии.

Трудно переоценить то значение и то влияние на развитие вычислительной техники, на развитие других областей научно-технического прогресса в нашей стране, которое имело создание этой высокопроизводительной, оригинальной по архитектуре и структуре отечественной вычислительной машины.

Во второй половине ушедшего столетия основу вычислительных средств большинства крупных вычислительных центров нашей страны составляли машины БЭСМ-6. Сфера их использования превзошла самые смелые прогнозы ее разработчиков. Первоначально предполагалось, что небольшая серия БЭСМ-6 будет использована для решения крупных научных задач в нескольких крупных научных институтах Советского Союза, таких как Институт прикладной математики АН СССР и центры ядерных исследований. Реально эта машина нашла самое широкое применение.

На основе БЭСМ-6 были созданы центры коллективного пользования, на ее основе были организованы центры управления в реальном масштабе времени, координационно-вычислительные центры, системы телеобработки и т. д. Машина БЭСМ-6 широко использовались как инструментальная машина в системах проектирования, для разработки математического обеспечения новых ЭВМ, для моделирования сложнейших физических процессов и процессов управления.

Успех БЭСМ-6 определялся не только тем, что для решения перечисленных выше задач требовалось высокое быстродействие, которым эта машина обладала, но и тем, что принятые при ее создании

принципиальные решения предвосхитили то, что в архитектурах современных процессоров считается важными достижениями и достоинствами.

Чтобы не быть голословными, назовем следующие принципиальные особенности архитектуры БЭСМ-6:

- магистральный, или, как в свое время (1964 г.) назвал его академик С.А.Лебедев, «водопроводный» принцип организации управления, с помощью которого достигается глубокий внутренний параллелизм обработки потока команд и движения операндов (на современном языке это называется конвейерной обработкой потока команд; глубина конвейера составляла 14 стадий);

- впервые осуществленный в БЭСМ-6 принцип использования ассоциативной памяти на сверхбыстрых регистрах с логикой управления, позволяющей аппаратно экономить число обращений к ферритовой памяти и тем самым осуществлять локальную оптимизацию в динамике счета (это явилось прообразом кэш-памяти для данных – неотъемлемой части процессоров новых поколений);

- «расслоение» оперативной памяти, что позволяет осуществить одновременное обращение из нескольких устройств машины к разным блокам памяти;

- принцип страничной организации виртуальной памяти и разработанные на его основе механизмы защиты памяти по числам и командам, что дало возможность осуществлять динамическое распределение оперативной памяти в процессе вычислений; аппаратный механизм преобразования виртуального («математического») адреса в физический адрес;

- использование индексных регистров для базирования и модификации адресов позволило строить свободно перемещаемые программы и вложенные процедуры;

- развитая система прерываний и индикации состояния внешних и внутренних устройств машины, контроль передачи данных между оперативной памятью и центральным устройством машины, между оперативной и внешней памятью позволили достаточно хорошо вести диагностику работы аппаратуры;

- возможность одновременной работы парка устройств ввода-вывода и внешних запоминающих устройств на фоне работы центрального процессора.

Сейчас эти особенности БЭСМ-6, возможно, кажутся не столь удивительными. Развитые ЭВМ сегодняшнего дня в той или иной степени обязаны обладать перечисленными характеристиками. Но идеи такой организации были предложены и воплощены в БЭСМ-6 намного раньше времени их широкой реализации в будущих машинах.

Четко выраженное в БЭСМ-6 разделение виртуальной и физической памяти, наличие нескольких параллельно работающих каналов обмена между оперативной и внешней памятью позволило наращивать оперативную память этой машины и объемы внешних запоминающих устройств (быстрых магнитных барабанов, магнитных дисков и магнитных лент).

Перечисленные характеристики позволили машине БЭСМ-6 рекордно долгое время, вплоть до начала 80-х годов прошлого столетия, оставаться одной из наиболее мощных и высокоразвитых вычислительных систем нашей страны. При всем этом БЭСМ-6 обладала рекордным коэффициентом отношения быстродействия к стоимости вычислений, так как при ее проектировании и конструировании учитывались вопросы технологии производства, серийного выпуска и стоимости эксплуатации машины.

В этом еще раз сказался замечательный стиль творчества академика С.А.Лебедева - стиль сочетания смелого творческого полета мысли, научной принципиальности н одновременного видения условий реализации, возможностей технологии, путей наискорейшего внедрения научных достижений в производство.

В смысле такого подхода к разработке машина БЭСМ-6 является развитием семейства ЭВМ БЭСМ-1, БЭСМ-2, М-20 и его преемников БЭСМ-ЗМ и БЭСМ-4, созданных под руководством С. Л. Лебедева. Архитектуру и структуру этих машин отличает концептуальная целостность, прозрачность построения и изящные инженерные решения. Наиболее полно это проявилось в БЭСМ-6. Несмотря на то, что эта машина является сложной вычислительной системой, механизмы функционирования ее устройств, их функциональные связи легко понимаются, четко интерпретируются, и, следовательно, наладка и эксплуатация машины БЭСМ-6 переставала быть проблемой.

Общеизвестна высокая надежность по критериям того времени этой машины. Время наработки на отказ достигает нескольких сотен часов.

Гениальность академика С. А. Лебедева как инженера состояла в том, что он ставил цель с учетом перспективы развития структуры будущей машины, умел правильно выбрать технические средства для ее реализации применительно к возможностям отечественной

промышленности. Была предложена элементная база и конструкция, которая могла быть реально освоена в заданное время. Схемотехника машины БЭСМ-6 по тем временам была совершенно новой, принцип разделения сложной логики, построенной на диодных блоках, и достаточно однотипной усилительной части обеспечил машине новое качество в смысле простоты ее создания.

Высокая надежность машины БЭСМ-6 и ее быстродействие в первую очередь определяются высокочастотной системой элементов и оригинальной конструкцией. Впервые в СССР была достигнута тактовая частота машины 10 МГц, в то время как машины, разрабатываемые на тех же диодах и транзисторах, имели тактовую частоту 4 - 6 МГц. Надежность БЭСМ-6 в значительной степени обеспечивалась большим запасом мощности основных элементов (диоды и транзисторы нагружены па 25-40

Все схемы машины БЭСМ-6 записаны формулами булевой алгебры. Это составляет основу се эксплуатационной и наладочной документации. В руках инженеров-наладчиков мы не видим полотен схем электрических соединений - в их руках тетрадки с формулами, компактно и точно описывающими функционирование устройств ЭВМ.

Принципиальное значение перехода на такой уровень описания машины состоит не только в том, что это прямое и успешное внедрение математического формализма в повседневную инженерную практику, но и в том, что это открыло широкие возможности автоматизации проектирования и подготовки монтажной и производственной документации с помощью ЭВМ. Такая возможность была использована при проектировании и внедрена в производственный процесс. Этот опыт схемотехники может быть взят на вооружение при создании машин новых поколений. Рост логической сложности отдельных устройств, модулей и агрегатов современных машин настоятельно потребовал формализации их описания.

Второй важный, с нашей точки зрения, момент, на котором следует остановить внимание, состоит в проблеме использования отечественного и зарубежного опыта при создании новых ЭВМ. Машина БЭСМ-6 не является копией какой-либо отечественной или зарубежной установки ни по системе команд, ни по внутренней структурной организации. Но при ее создании и проектировании был изучен и проанализирован опыт создания ЭВМ высокой производительности, накопленный к тому времени. Из этого опыта было взято на вооружение то, что соответствовало целям, поставленным при разработке этой машины. Принятые решения были всесторонне, комплексно обоснованы.

К разработчикам БЭСМ-6 в свое время обращались с вопросами

типа: а почему вы это сделали не так, как оно сделано, например, в машинах фирм IBM, CDC? Мы далеки от того, чтобы критиковать задававших подобные вопросы с позиций патриотизма или с других эмоциональных позиций. Техника есть техника, и ее нужно делать наиболее разумным способом.

При разработке машины БЭСМ-6 по инициативе С.А.Лебедева было проведено детальное моделирование, определившее ее структурные характеристики, на классе тех задач, для решения которых машина была предназначена. Результаты этого моделирования дают обоснованные ответы на большинство такого рода вопросов. Характер предназначенных для решения задач в ряде случаев позволял принимать более простые структурные решения, в ряде случаев требовал создания сложных схем, не имевших аналога в опыте разработки вычислительной техники.

Как нам кажется, принцип обоснованности принятых решений не потерял своего значения и по сей день. Этот принцип требует творческого, научного подхода, которым в полной мере обладал С. А. Лебедев.

Поясним сказанное на одном примере, связанном с организацией управления работой внешних запоминающих и вводных-выводных устройств БЭСМ-6. Способ сопряжения внешних устройств, принятый в машине БЭСМ-6, подвергался наибольшим нападкам со стороны критиков этой машины. Действительно, подключение всякого нового устройства требует определенных инженерных доработок в устройстве управления внешними устройствами, и пользователь, сумевший заполучить в свои руки какое-либо «нестандартное» устройство, испытывает большие затруднения с его подсоединением.

Кажется, что такому решению нет оправданий, тем более, что к моменту разработки БЭСМ-6 фирмой IBM был реализован стандартный интерфейс, снимавший в значительной степени проблему подключения новых устройств и замену одних другими.

Но если внимательно подумать или просто сравнить по стоимости и объему оборудование, необходимое для реализации сопряжения с внешними устройствам в том и другом случае, то решения, принятые в БЭСМ-6, могут оказаться не столь плохими. В самом деле, каждое внешнее устройство для обеспечения стандартного сопряжения имеет в своем составе контроллер или подключено к этому весьма дорогому и сложному устройству, обеспечивающему стандартный выход на мультиплексный или селекторный каналы. Если просуммировать затраты на дополнительное оборудование и занимаемые им площади, необходимые при подключении устройств по стандартному интерфейсу, то окажется, что система БЭСМ-6 многократно

экономнее. Иными словами, централизованное устройство управления БЭСМ-6 в несколько раз дешевле, чем стоимость контроллеров стандартного набора внешних запоминающих и вводных/выводных устройств машин аналогичного класса.

Впрочем, в ряде моделей машин IBM был введен так называемый интегрированный адаптер файла, позволяющий подключить к вычислительной машине дисковое запоминающее устройство без использования стандартного канала и контроллера. В этом факте можно усмотреть некоторую аналогию с рациональным способом подключения устройств, принятым в БЭСМ-6.

Не следует понимать приведенный пример как мнение авторов, касающееся перспектив распределения электронного оборудования по элементам системы. Развитие технологии производства микропроцессоров, удешевление электроники требует принятия иных решений и в принципе делает более рентабельным усложнение логических функций периферии. Это потребует пересмотра сложившегося к настоящему времени и названного стандартным интерфейса между периферией и центральным устройством, потребует творческого подхода и принятия новых решений, обоснованных пре- дыдущим опытом и правильно выбранными целями.

Вернувшись к машине БЭСМ-6, уместно сообщить здесь, что в состав ее внешних устройств были включены стандартные магнитные диски, стандартные магнитные ленты, алфавитно-цифровые дисплеи, новейшие печатающие устройства, т. е. усилиями завода-изготовителя преодолены трудности, отмеченные в вышеприведенном примере, и машина была снабжена парком современных внешних устройств.

Говоря о наследии С.А.Лебедева, которое должно быть взято на вооружение в перспективных разработках, нельзя не сказать о той атмосфере коллективизма и творческого воодушевления, которое умел создавать вокруг себя Сергей Алексеевич. Машины, а тем более такие сложные, как ЭВМ, делаются большими коллективами людей. С.А.Лебедев умел показать каждому значимость его усилий, умел поощрять творческую инициативу, оставаясь при этом принципиально требовательным. Он считал и неоднократно подчеркивал, что лучшая школа для специалиста - это участие в конкретных разработках, и не боялся привлекать к самым серьезным проектам молодежь, не боялся, что очень и очень важно, полагаться на своих учеников, доверяя их чувству ответственности и, тем самым, воспитывая это чувство.

В подтверждение тезиса, высказанного в начале статьи, об огромном влиянии, которое оказало создание машины БЭСМ-6 на разви-

тие различных областей науки и техники, особо следует сказать о ее роли в развитии работ в области математического обеспечения.

Прежде всего, назначение этой машины, ее архитектурные и структурные особенности, отвечающие современным идеям, потребовали создания соответствующей операционной системы и системы программирования, т. е. потребовали создания математического обеспечения, удовлетворяющего требованиям пользователя. Это требование, возникшее из существа дела, было подкреплено некоторыми административными мерами: БЭСМ-6 стала первой отечественной вычислительной машиной, которая была принята государственной комиссией как система аппаратных средств совместно с ее математическим обеспечением. Она и поставлялась затем потребителям вместе с программным обеспечением.

Без особого преувеличения можно сказать, что работы по исследованию и разработке операционных систем, стратегий распределения ресурсов и планирования вычислений широким фронтом в нашей стране начались с появлением БЭСМ-6. Это и понятно. Первые машины БЭСМ-6 предназначались для установки в центрах, обладавших наиболее сильными коллективами специалистов в области программирования и использования вычислительных машин: ВЦ АН СССР, ВЦ МГУ, ИПМ АН СССР, ОИЯИ, ИК АН УССР, СО АН СССР и др.

Появилась возможность реализовать многие созревшие к тому времени идеи на машине, обладавшей аппаратными возможностями для организации мультипрограммирования, режима разделения времени, аппаратными средствами интерпретации стекового способа обращения к памяти.

Коллективными усилиями советских программистов уже к 1968 г. была создана система математического обеспечения, включавшая в свой состав операционную систему пакетной обработки, трансляторы с машинно-ориентированных языков и с универсальных языков АЛГОЛ-60 и ФОРТРАН.

На протяжении всего времени существования машины БЭСМ-6 ее математическое обеспечение непрерывно совершенствовалось, и по качеству, объему и возможностям не уступало математическому обеспечению лучших отечественных и зарубежных серийных ЭВМ того времени. Основания к такому суждению дает перечень элементов, входящих в состав математического обеспечения БЭСМ-6. Для универсальных языков программирования АЛГОЛ-60, ФОРТРАН, ЛИСП использовалось несколько вариантов трансляторов, генерирующих программы разной степени эффективности, в том числе оптимизирующие трансляторы с языков АЛГОЛ-60 и ФОРТРАН

и компилятор с языка ЛИСП. В состав программных средств машины БЭСМ-6 входил спектр проблемно-ориентированных языков СИМУЛА-67, ГРАФОР, ГРАФАЛ, язык типа EPSILON и ряд других языков, более узко специализированных. Пошаговые трансляторы с некоторых других языков использовались для построения диалоговых систем.

В состав математического обеспечения БЭСМ-6 входили системы, обеспечивающие режим мпогопультового доступа к машине с удаленных терминалов. К ним относится система ПУЛЬТ, обеспечивающая отладку программ в терминах входного языка; система МУЛЬТИДОСТУП, обеспечивающая связь задачи пользователя с терминалами; система ДИМОН (диалоговый монитор), используемая для многих целей, в том числе для редактирования файлов и запуска задач в решение с удаленных пультов.

В состав математического обеспечения БЭСМ-6 входят системы управления заданиями и системы управления данными, дающие в руки программиста средство высокого логического уровня. Эти средства развиты на основе двух широко используемых, дополняющих друг друга операционных систем: ДИСПАК и ДУБНА (ДД), имеющих несколько разную ориентацию.

ОС ДИСПАК предназначена главным образом для управления режимом пакетной и дистанционной пакетной обработки, ОС ДУБПА — фортранно-ориентированная операционная система, используемая и развиваемая в центрах обработки данных физического эксперимента. Родоначальником этих двух ОС явилась первая операционная система Д-68. Обе операционные системы ДИСПАК и ДД сохраняли преемственность в отношении своего родоначальника.

Для БЭСМ-6 разработано несколько ОС, обеспечивших реализацию параллельных процессов в задачах и использование машины в системах реального времени и в многомашинных вычислительных комплексах (НД-70, ОС ИПМ, ДИАПАК).

В библиотеках машины БЭСМ-6 содержатся пакеты прикладных задач, с помощью которых были решены задачи наиболее передовых направлений научно-технического прогресса. Накоплен огромный фонд программ пользователей и опыт его эксплуатации. Можно вполне определенно сказать, что на базе БЭСМ-6 создан «золотой фонд» программного обеспечения, значение которого трудно переоценить.

В области создания математического обеспечения велика роль С.А.Лебедева. По его инициативе в Институте точной механики и вычислительной техники была создана сильная по своему составу лаборатория математического обеспечения, и математики-

программисты являлись полноправными участниками разработки структуры и архитектуры этой машины. Академик С.А.Лебедев одним из первых понял значение совместной работы математиков и инженеров по созданию вычислительных систем и воплотил в жизнь это необходимое и продуктивное сотрудничество. Значение этого особенно четко вырисовывается сегодня, когда все более и более становится очевидным, что создание эффективных вычислительных средств перерастает из проблемы инженерно-технологической в проблему в большей степени математическую, проблему, которую можно решить только комплексными усилиями инженеров и математиков.

Приведенный выше внушительный и далеко не полный перечень возможностей и средств математического обеспечения БЭСМ-6 сделан авторами не в целях рекламы - она излишня, а для того, чтобы подчеркнуть следующую, на наш взгляд, важную мысль. Математическое обеспечение БЭСМ-6 создано усилиями специалистов Советского Союза и специалистов социалистических стран. Как известно, тот, кто держит в своих руках математическое обеспечение, держит в своих руках ключевые позиции в деле эффективного использования вычислительной техники в наиболее важных, решающих областях научно-технического прогресса.

Не потому ли именно на плечи БЭСМ-6 легли наиболее ответственные задачи, от сопровождения космических кораблей в международной программе ЭПАС, до проектирования разработки нефтяных месторождений с прямой экономией, исчисляемой сотнями миллионов рублей. И надо прямо сказать, что эта машина Главного конструктора академика С.А.Лебедева с честью справлялась с ответственными обязанностями в течение двадцати лет.

У читателя может возникнуть вопрос, имеет ли смысл говорить столь много хороших слов о машине, построенной на элементной базе второго поколения, в то время, когда произошел невероятный скачок перехода к вычислительным системам шестого поколения, построенным на сверхбольших интегральных схемах с тактовой частотой, превосходящей несколько Ггерц, с числом активных логических элементов на одном кристалле, приближающимся к сотне миллионов. Нам кажется, что говорить о БЭСМ-6 и ее главном конструкторе академике С.А.Лебедеве имеет смысл именно сейчас не только для того, чтобы отдать должное создателю лучших отечественных машин прошлого поколения, но, главным образом, для того, чтобы дать пищу для размышления о будущем вычислительной техники в нашей стране.

Хотелось бы обратить внимание на ряд проблем ее развития..

Известна расхожая фраза пессимистов о том, что в развитии вычислительной техники мы отстали навсегда. Трудно сейчас опровергать этот тезис с учетом состояния развития отечественной элементной микроэлектронной базы. Но это совсем не так с точки зрения разработки новых архитектурных принципов построения вычислительных систем.

Вычислительная техника была, есть и будет наиболее бурно развивающимся направлением научно-технического прогресса. Идет непрерывное обновление идей, появление новых проблем, требующих своего решения. Речь идет не о замкнутых научных или технических проблемах, связанных с решением отдельных пусть и очень важных задач. Речь идет о проблемах комплексных, затрагивающих целый круг вопросов, связанных не только с индустрией производства, но и с глобальными целями использования объединенного в неразрывное целое мира компьютеров, архитектурной поддержкой математического обеспечения, то есть об обширном круге вопросов взаимоотношения человека с машиной, вплоть до вопросов, если угодно, социологических.

В этой ситуации особое значение приобретает широта взглядов, направленность разработок, умение находить правильное и оптимальное в сложившихся реальных условиях решение, учитывающее перспективы развития «компьютерного социума». Необходимо уметь дать четкий ответ на вопрос, для каких целей, для решения каких перспективных задач разрабатывается та или иная архитектура системы, тот или иной модуль вычислительных средств. Важно умение достигать наиболее экономным путем поставленную цель с детальным учетом обозначившихся возможностей. Вот то, что должно быть взято на вооружение из опыта разработки БЭСМ-6, из арсенала высочайших человеческих черт, глубокой научной методологии, оптимального подхода к реализации идей, всего того, чем был в полной мере наделен Академик Сергей Алексеевич Лебедев.

Аннотации

Антипина А.В., Пашков В.Н. Метод предотвращения DDoS атак на контроллер в программно-конфигурируемых сетях // Программные системы и инструменты. Тематический сборник № 19, М.: Изд-во факультета ВМиК МГУ, 2019.

В статье рассматривается программно-конфигурируемые сеть и методы предотвращения DDoS атак на контроллер.

Ил.: 3 рис., 1 табл., Библиогр.: 9.

Балашов В.В., Антипина Е.А. Итерационная схема планирования вычислений в модульных системах реального времени // Программные системы и инструменты. Тематический сборник № 19, М.: Изд-во факультета ВМиК МГУ, 2019.

В статье представлена итерационная схема планирования вычислений в модульных вычислительных системах реального времени, основанная на существующей поэтапной схеме планирования (распределение вычислительной нагрузки, построение виртуальных каналов, построение расписания выполнений вычислительных задач) и расширяющая эту схему за счет наличия обратных связей между этапами. Обратная связь применяется в случае неуспешного выполнения этапа планирования и задает новую итерацию выполнения одного или нескольких этапов. Проведенное экспериментальное исследование позволило выявить области эффективного применения каждого из предложенных видов обратных связей и их сочетаний.

Ил.: 3 рис., Библиогр.: 8.

Волканов Д.Ю., Маркобородов А.А. Исследование ячейки конвейера сетевого процессорного устройства на модели уровня регистровых передач // Программные системы и инструменты. Тематический сборник № 19, М.: Изд-во факультета ВМиК МГУ, 2019.

В статье рассматривается архитектура сетевого процессорного устройства (СПУ) на базе однотипных специализированных вычислительных ячеек. Для вычислительной ячейки данного СПУ была

проведена оценка её скорости работы и тактовой частоты с помощью модели уровня регистровых передач, разработанной на языке описания аппаратуры Verilog.

Ил.: 2 рис., 1 табл., Библиогр.: 5.

Галкина Е.В. Средства статического анализа программ на функциональных языках с динамической типизацией // Программные системы и инструменты. Тематический сборник № 19, М.: Изд-во факультета ВМиК МГУ, 2019.

В работе рассматриваются методы статического анализа программ на функциональных языках с динамической типизацией, а именно Racket и Clojure. Выделяются два основных подхода: добавление в язык новых конструкций и использование внешних программных инструментов. В рамках первого подхода рассмотрены диалект Typed Racket и библиотека Typed Clojure, которые реализуют механизм постепенной типизации. Далее рассматриваются внешние программные инструменты, предназначенные для решения отдельных частных задач статического анализа кода на указанных языках.

Библиогр.: 14.

Зайцева О.А., Антоненко В.А. Разработка и реализация системы холодного старта функции для бессерверных вычислений // Программные системы и инструменты. Тематический сборник № 19, М.: Изд-во факультета ВМиК МГУ, 2019.

В статье рассматривается проблема построения системы «холодного старта функции» для бессерверных вычислений. Представлено сравнение существующих FaaS-платформ по способам масштабирования функций и управления холодным и теплым стартами функций. Представлена система холодного старта для платформы OpenFaaS. Проведено экспериментальное исследование полученной реализации.

Ил.: 4 рис., 1 табл., Библиогр.: 15.

Пантюхин Л.К., Антоненко В.А. Разработка алгоритма распределения трафика для SD-WAN решения // Программные системы и инструменты. Тематический сборник № 19, М.: Изд-во факультета ВМиК МГУ, 2019.

В данной статье рассматривается алгоритм распределения трафика для SD-WAN решения.

Ил.: 8 рис., 3 табл. Библиогр.: 14.

Синякова М.А., Степанов Е.П. Оценка задержки потоков виртуального пласта. // Программные системы и инструменты. Тематический сборник № 19, М.: Изд-во факультета ВМиК МГУ, 2019.

Актуальность работы связана с возросшими потребностями к качеству сервиса в современных сетях, в частности, с необходимостью выделять некоторым пользователям или под определенный вид трафика логически обособленные ресурсы сети - виртуальные пласты (совокупность сетевых ресурсов, предназначенных для передачи данных с определенным качеством сервиса). работе рассматривается проблема обеспечения требуемой сквозной максимальной задержки по всем потокам виртуального пласта. Задержка регулируется за счёт изменения приоритета пласта (и, соответственно, приоритета очередей пакетов виртуального пласта на сетевых коммутаторах). Для оценки задержки выбора предложено использовать аппарат стохастического сетевого исчисления. Проведено сравнение базовых методов построения алгоритмов расчёта задержки. Обоснован выбор метода линейного программирования, на его основе построен алгоритм расчёта задержки.

Ил.: 1 рис., Библиогр.: 7.

Степанов Е.П., Войнов Н.А. Динамическое сегментирование транспортных соединений // Программные системы и инструменты. Тематический сборник № 19, М.: Изд-во факультета ВМиК МГУ, 2019.

В данной статье представлен адаптивный подход сегментирования транспортных соединений. Проводится экспериментальное исследование эффективности предложенного решения и показывается преимущество данного подхода по сравнению с классическим статическим сегментированием в определенных сценариях..

Ил.: 5 рис., Библиогр.: 5.

Титов Н.И., Антоненко В.А. Разработка системы первичной настройки клиентских устройств для доступа к облачным сервисам // Программные системы и инструменты. Тематический сборник № 19, М.: Изд-во факультета ВМиК МГУ, 2019.

В рамках сценариев использования крупномасштабных наборов клиентских сетевых устройств, система их первичной настройки (*Zero Touch Provisioning*, далее – *ZTP*) является эффективным средством автоматизации процесса удаленного развертывания, уменьшает вероятность сбоев и увеличивает скорость ввода оборудования в эксплуатацию. С помощью *ZTP* становится возможным удаленно и в автоматическом режиме реализовать процесс установки программного обеспечения, выполнять скрипты на основе единого шаб-

лона, выделять группы и подгруппы устройств, требующих различные версии операционных систем и инструкций. На рынке сетевых операционных систем (*Network Operation Systems*, далее – *NOS*) существуют проприетарные коммерческие реализации *ZTP* от *Cisco*, *Juniper* и других сетевых вендоров. Все они созданы для *NOS* своего производителя, лишая разработчика возможности создания мультивендорных систем. В рамках исследования создана система *ZTP* для устройств на базе open-source *NOS OpenWRT*.

Ил.: 5 рис., 1 табл., Библиогр.: 5.

Королев Л.Н., Мельников В.А. Об ЭВМ БЭСМ-6. // Программные системы и инструменты. Тематический сборник № 19, М.: Изд-во факультета ВМиК МГУ, 2019.

Исходный вариант статьи был опубликован в 1976 году в журнале ВЫЧИСЛИТЕЛЬНЫЕ СРЕДСТВА И ВСПОМОГАТЕЛЬНОЕ ОБОРУДОВАНИЕ СИСТЕМ в связи с десятилетием ввода в эксплуатацию машины БЭСМ-6. В публикуемом ниже варианте Л.Н.Королевым внесены некоторые изменения в первоначальный текст.

Software systems and tools : Tematic collection / Ed. by R. L. Smelyansky. – Moscow : Publishing Department of the Faculty of Computational Mathematics and Cybernetics (license ID № 05899 from 24.09.2001); MAKS Press, 2019. – № 19. – 136 p.

ISBN 978-5-89407-604-1 (CMC MSU)

ISBN 978-5-317-06324-5 (MAKS Press)

Theese proceedings consists of student's works, who have received the recommendation of the Department's scientific seminars, which he created and directed Korolev L.N. This edition continues the tradition of publishing in memory of this outstanding man. The proceedings contains articles devoted to creating information computing infrastructure for scientific research, problems of modern computer networks, methods and tools for organizing and managing cloud computing, tools that support the operation of real-time systems (RTS), static analysis tools.

These papers will be of interest to students, graduate students and professionals in the development of application software systems using new information technologies.

Keywords: information telecommunication technology, software-defined networking, OpenFlow switch, centralized controller, cloud computing, data center, virtual resources, network functions virtualization, cloud platform, real-time systems, integrated modular architecture, job scheduling, DDOS attacks, network processor, static analysis tools, functional languages, computer architecture, history of computers, BESM-6.

Научное издание

ПРОГРАММНЫЕ СИСТЕМЫ И ИНСТРУМЕНТЫ

Тематический сборник

№ 19

Под общей редакцией чл.-корр. РАН,
профессора Р. Л. Смелянского

Издательство «МАКС Пресс»
Главный редактор: *Е. М. Бугачева*

Напечатано с готового оригинал-макета
Подписано в печать 20.12.2019 г.
Формат 60x90 1/16. Усл.печ.л. 8,5.
Тираж 40 экз. Заказ 004.

Издательство ООО “МАКС Пресс”
Лицензия ИД N 00510 от 01.12.99 г.
119992, ГСП-2, Москва, Ленинские горы, МГУ им. М.В. Ломоносова,
2-й учебный корпус, 527 к.
Тел. 8(495)939-3890/91. Тел./Факс 8(495)939-3891.

Отпечатано в полном соответствии с качеством
предоставленных материалов в ООО «Фотоэксперт»
115201, г. Москва, ул. Котляковская, д.3, стр. 13.