Петрозаводский государственный университет
Институт математики и информационных технологий
Кафедра информатики и математического обеспечения

01.03.02 Прикладная математика и информатика

Выпускная квалификационная работа на степень бакалавра

# Разработка и реализация много-агентной системы для прогнозирования производительности сетей передачи данных

Выполнил:
студент 4 курса группы 22403

Д.О. Карлов _____
*подпись*

Руководитель:
О. Ю. Богоявленская, к.т.н., доцент

_____
*подпись*

Итоговая оценка

_____
*оценка*

Петрозаводск — 2020

# Содержание

# Введение

~~Цель практики:~~ Разработать и реализовать много-агентную систему для прогнозирования производительности сетей передачи данных

Задачи практики:

1. Изучить механизм работы TCP;
2. Разработать архитектуру многоагентной системы

С каждым годом стремительно увеличивается число пользователей и кол-во сетевых устройств интернета. Соразмерно этому растет и количество трафика. По данным отчета Cisco VNI 2017 в период с 2017—2022гг. объем интернет трафика вырастет троекратно (с 29Гб. в месяц на пользователя до 85 Гб.).

Балансирование общей нагрузки сети важная задача, однако конечного пользователя больше интересует производительность на ~~некоторых конечных~~ маршрутах. Такая информация может указать ему на выбор среди альтернативных поставщиков контента, например просмотр выпуск новостей на новостном сайте или на youtube.com, или на время использование ресурса, обеспечивающее наилучший опыт использования. К сожалению, для обычных пользователей практически не существует подобного инструментария, что и ~~объясняет~~ актуальность разработки систем мониторинга и прогнозирования(СМП).

# 1 Описание системы

СМП, описанная в [1], представляет из себя распределенную систему, в которой каждая сущность(агент) регистрирует данные о конкретных маршрутах на уровне точка-точка, сжимает их, предоставляет пользователю статистику, результаты анализа или прогноза. Распределенность системы позволяет уменьшить объем информации собираемой на одном устройстве, улучшить качество прогноза, опираясь на данные от других агентов, получить данные о недоступных или еще не зарегистрированных на данном устройстве маршрутах. В центре внимания системы протоколы транспортного уровня, так как именно они оказывают существенное внимание на производительность соединения.

## 1.1 Гипотетические сценарии использования системы

Можно выделить три гипотетических сценариев использования системы использования системы.

В первом случае пользователь целенаправленно устанавливает на свое устройство агента, регистрирует интересующие его маршруты. Этот сценарий имеет ряд больших недостатков.

Во первых обычному пользователю для регистрации маршрута проще всего использовать имя конечного маршрута (например vk.com). Однако в большинстве случаев трафик к пользователю поступает с большого пула ip адресов, которые dns не может сопоставить с названием исходного маршрута. Следовательно требуется более детальное описание конечных маршрутов на уровне ip адресов или диапазон сетевых адресов в виде сетевой маски (CIDR).

Вторым недостатком (наиболее существенным) этого сценария является невозможность распределенности. Для того, что бы измерения и прогнозы с других агентов имели какой-то смысл для текущего агента необходимо, что бы они находились в одной локальной сети. У среднестатистического пользователя банально будет отсутствовать множество активных сетевых устройств работающих независимо и кроме того настройку портов и агентов на разных устройствах никто не отменял. Тем не менее одиночную систему все еще можно будет использовать для мониторинга и сбора статистики.

Во втором случае, на мой взгляд наиболее интересном, системный администратор некого предприятия, офиса устанавливает агентов на необходимые устройства и задает им маршруты для наблюдений. Это позволяет избежать "недосказанности" маршрута, атак же дает простор для распределенности. Использование СМП в данном сценарии позволит

мониторить производительность критически важных маршрутов и своевременно сообщать об отклонениях от нормы показателей на отдельных узлах, что в свою очередь может сигнализировать например о неправильно конфигурации сети или неполадках на физическом уровне.

Третий возможный сценарий использования СМП заключается в следующем. Поставщик услуг (например интернет провайдер) устанавливает агентов на абонентские устройства и собственно мониторит производительность на своём канале. Это позволит ~~опять таки~~ своевременно выявить неполадки, а также собирать информацию о качестве предоставляемых услуг. Критическим недостатком такого подхода является сложность настройки удаленных агентов и кроме этого не все клиенты будут рады установке агентов, потенциально позволяющих собирать информацию не только о сетевом маршруте. Так же наверняка могут возникнуть проблемы с информационной безопасностью обоих сторон.

В дальнейшей работе будут рассмотрены первый и второй сценарии использования.

## 1.2 Основные модули и требования

Несмотря на очевидное отличие сценариев 1 и 2, они имеют много общих требований в особенности касающихся сбора информации, а именно:

1. возможность работы как с tcp так и с udp трафиком;
2. процесс захвата трафика и управлением системы должен происходить в реальном времени;
3. потенциально неограниченное время работы;
4. возможность обмена данными между агентами, находящихся в одной системе.
5. анализ собранных данных.

Для выполнения описанных выше функций можно выделить следующие логически цельные блоки (модули):

1. Монитор – основной задачей которого является регистрирование данных о сетевом потоке. На вход ему поступают данные с сетевой карты. Для уменьшения вычислительных расходов мониторинг ведется только по маршрутам, заранее добавленным в реестр соединений. Для добавления маршрута достаточно указать ip адреса хоста или же диапазон адресов.

2. Модуль прогноза, осуществляющий статистический анализ и прогноз интересующих характеристик потока, например пропускной способности, на основе данных полученных с монитора и/или других агентов.

3. Интерфейс пользователя, позволяющий добавлять маршруты в реестр и предоставляющий пользователю информацию о текущих соединениях.

4. Модуль взаимодействия с агентами, ведущий учет активных агентов в сети и отвечающий за обмен данными между ними.

В зависимости от сценария использования, модули могут образовывать ~~цельный~~ программный продукт (обязательно для первого сценария) [illegible] и разбиты на группы. Например, во втором сценарии скорее всего конечные узлы, на которых будет осуществляться сбор информации, будут иметь только монитор и связь с другими агентами, а модуль прогнозирования и интерфейс пользователя будут располагаться у системного администратора. Отсюда в дополнение к описанным ограничениям возникает ~~желания~~ иметь возможность опционального включения и отключения нужных модулей.

## 1.3 Ключевые параметры регистрируемые системой

Ключевым параметром, определяющим производительность сети является пропускная способность, определяющая объем переданной информации в секунду, которая напрямую зависит от работы протоколов транспортирного уровня (в особенности TCP).

В независимости от протокола сетевого уровня монитор должен собирать следующую информацию о соединении:

1. время начало сессии;

2. общее кол-во переданной информации;

3. длительность соединения;

4. и как следствие среднюю пропускную способность.

Если речь идет о tcp сесии то к этому набору добавляются следующие немаловажные характеристики:

1. RTT – задержка между временем отправки сегмента и временем получения подтверждения (Round trip time).

2. P – вероятность потери пакета

3. MSS – максимальный размер TCP сегмента. Обычно равен 1460 байт.

Описанный выше параметры поваляют вычислить теоретически максимально возможную пропускную способность по формуле М.Матиса

$$T = \frac{MSS}{RTT}\sqrt{\frac{3}{2P}} \tag{1}$$

Величина MSS содержится в заголовке сегмента TCP. Вероятность потерь и RTT непосредственно замеряются агентом.

# 2 Обзор

## 2.1 Обзор TCP

TCP – Протокол транспортного уровня модели OSI, осуществляющий сегментацию и передачу данных между двумя процессами, ключевыми особенностями которого являются гарантия доставки и контроль перезагрузки [2].

Надежность доставки осуществляется путем нумерования байтов в потоке (приемник и источник ведут собственные нумерации) и посылке подтверждений, содержащих в себе номер последовательности, начиная с которого отправитель подтверждения желает получить. В случае потери сегмента TCP организует повторную отправку[2].

Контроль перегрузки реализуется за счет алгоритмов управления таким параметром как окно перегрузки. Размер окна – количество байт, которые источник может отправить, не дожидаясь подтверждения от второй стороны, так например алгоритм медленного старта (Slow Start) экспоненциально увеличивает размер окна при каждом получении подтверждения и уменьшает до порогового или начального размера в случаи потери [2].

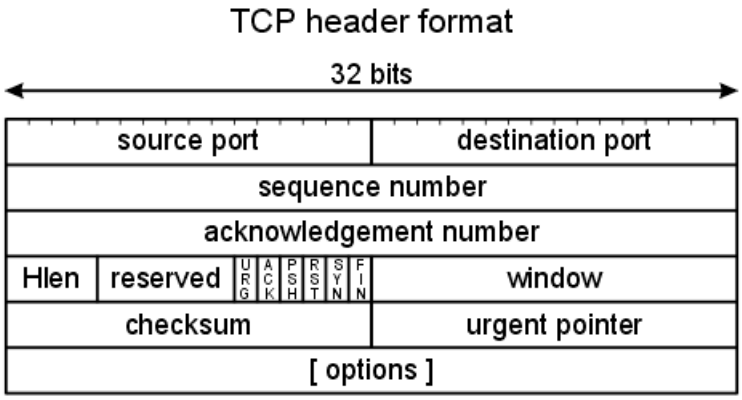


Рис. 1: Заголовок TCP сегмента

## 2.2 Модель акторов

Учитывая требования, системе придется иметь дело с множеством tcp/udp соединений с множеством хостов. Для каждого соединения необходимо хранить его внутреннее состояние. Проектируя выполнение системы в одном потоке влечет за собой невозможность её масштабирования (скорее всего и проектирования), однако используя множество потоков можно столкнутся с не меньшими проблемами в виду блокировок, доступов к разделенным ресурсам и тп. Однако уже довольно давно была придумана математическая концепция,

позволяющая избежать всех этих проблем на корню.

Модель акторов – математическая модель конкурентных вычислений впервые описанная в 1973 году Карлом Хьюиттом[5]. В ее основе лежит понятия актора(Actor) – сущности общающейся с другими сущностями(акторами) путем неизменяемых сообщений. Варианты действия актора в ответ на полученное сообщение:

- отправить конченное число сообщений другим акторам
- создать конечное число новых акторов
- опредилить свое поведение на дальнейшие сообщения (сохранение внутреннего состояния)

Эта модель позволяет избежать проблем описанных выше и кроме того дает возможность абстрагироваться от низкоуровневых понятий потоков и процессов, что в свою очередь дает возможность избавиться от локальности. Действительно, на уровне модели ни что не мешает актором находится в одном потоке, в разных и даже на разных машинах, открывая тем самым путь к распределенным системам.

Кроме того акторы хороши для описания систем с множеством диалогов (в нашем случае соединений)в которых необходимо хранить внутренне состояние. Именно по этому данная модель и была взята за основу архитектуры.

## 2.3 Libpcap

Согласно требованиям, система должна иметь доступ к сетевому трафику. Использование программ анализаторов таких как wireshark, tcpdump затруднено, так как требуется мобильное добавление и удаление маршрутов, и получение данных "налету". Альтернатива – использование библиотек, для получения трафика напрямую с сетевой карты. Наиболее популярной является Libpcap (для unix систем) и ее аналог Winpcap(для Windows). Они предназначены для использования совместно с языками C/C++, однако существуют интерфейсы (зачастую сомнительного качества) для многих других языков.

## 2.4 Akka

Несмотря на то, что реализации модели акторов существуют для множества языков, наиболее популярными и используемые на практике является фреймворк Akka для jvm,

и языки Erlang/Elixir, в которых модель акторов напрямую встроена в архитектуру языка и виртуальной машины Beam, на которой интерпретируется байткод, полученный в результате компиляции кода на этих языках.

На данном этапе разработки и проектирования весомым преимуществом в пользу выбора Akka стала наличие стабильного и поддерживаемого интерфейса для pcap библиотек (Pcap4j), позволяющей работать как и с Libpcap, так и с Winpcap, а так же в целом наличие большего опыта с экосистемой jvm.

Akka – популярный фреймфорк, реализующий модель акторов для jvm. Написан на языках Scala и Java и предоставляет интерфейс для них же. Приятным бонусом при использование Akka является наличие легковесного http сервера, использующего акторы для обработки запросов, и модуля кластеризации, который позволяет подключать логику распределенных акторов. В качестве языка программирования был выбран Scala за счет более удобного интерфейса с Akka.

## 2.5 Архитектура акторов в Akka

Актор по своей сути является сущностью, инкапсулирующей логически завершенный функционал, атомарно выполняющийся в однопоточном режиме. К его составным частям можно отнести состояния, поведение, почтовый ящик.

Под состоянием понимается как набор некоторых внутренних переменных/структур данных так и явно определенный конечный автомат.

Каждый раз, когда сообщение обрабатывается, оно сопоставляется с текущим поведением актора, которое означает функцию, определяющую действия, которые необходимо предпринять при обработке сообщение в данный момент времени.

Цель актера – обработка сообщений, полученых от других акторов или извне системы. Элемент, соединяющий отправителя и получателя, является почтовым ящиком актера: у каждого актора есть ровно один почтовый ящик, в который все отправители помещают свои сообщения в очередь.

Постановка в очередь происходит во временном порядке операций отправки, что означает, что сообщения, отправленные разными субъектами, могут не иметь определенного порядка во время выполнения из-за очевидной случайности распределения участников по потокам. Однако имеется гарантия, что порядок сообщений, отправленных одному и тому же получателю от одного и того же отправителя, сохранится в почтовом ящике.Так как доступ к внутреннему состоянию актора запрещен, то для коммуникации между ак-

торами используется специальный тип ссылки на актора, которая инкапсулирует в себе "адрес"актора и содержит механизм для отправки сообщений ему, что позволяет на уровне программной логике не делать различий между локальными акторами, находящимися в этой же системе, и удаленными. Эта ссылка может безопасно передаваться в сообщениях.

Когда очередь сообщений пуста акторы не делают никакой работы. Добавление сообщения в очередь тригерит диспетчера, назначенному для этого акторы или их группы, и тот помещает актора в fork join pool обычных jvm потоков.

Важной частью актора являются определение того, что будет происходить при ошибке в дочернем акторе, так называемая Supervision Strategy. У родительского актора обычно есть следующие варианты:

- сохранить дочернего актора (игнорировать ошибку)
- перезапустить с обнулением текущего состояния
- удалить его
- эскалировать ошибку дальше по иерархии

Как было описано ранее, акторы могут порождать других акторов что ведет к древовидной струтуре системы акторов.

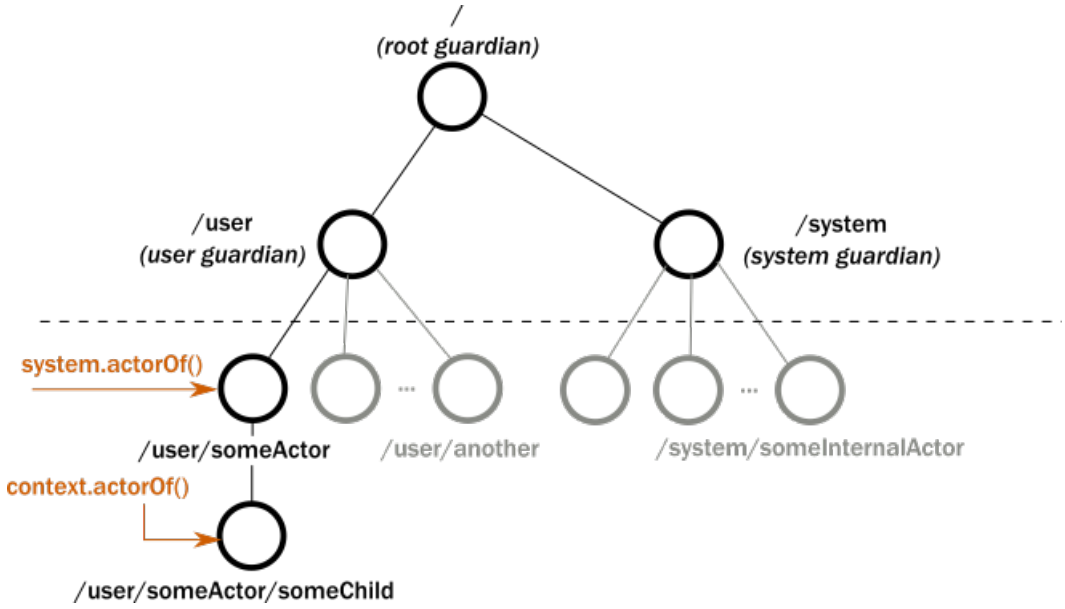


Рис. 2: Ситемма акторов в Akka

На рис. 2 представлена система акторов в Akka. Все созданные пользователем акторы, являются потомками user актора, все системные являются потомками system актора.

# 3 Архитектура

Как было упомянуто ранее, роль модуля статистики/прогнозирования может в простом случае играть отдельный скрипт. Все что ему необходимо это интерфейс для получения данных. Поэтому в этой главе будет дано описание всех остальных модулей.

Проанализировав требования, я спроектировал следующую архитектуру. В нее входят следующие акторы, иерархически представленные на рис.3:

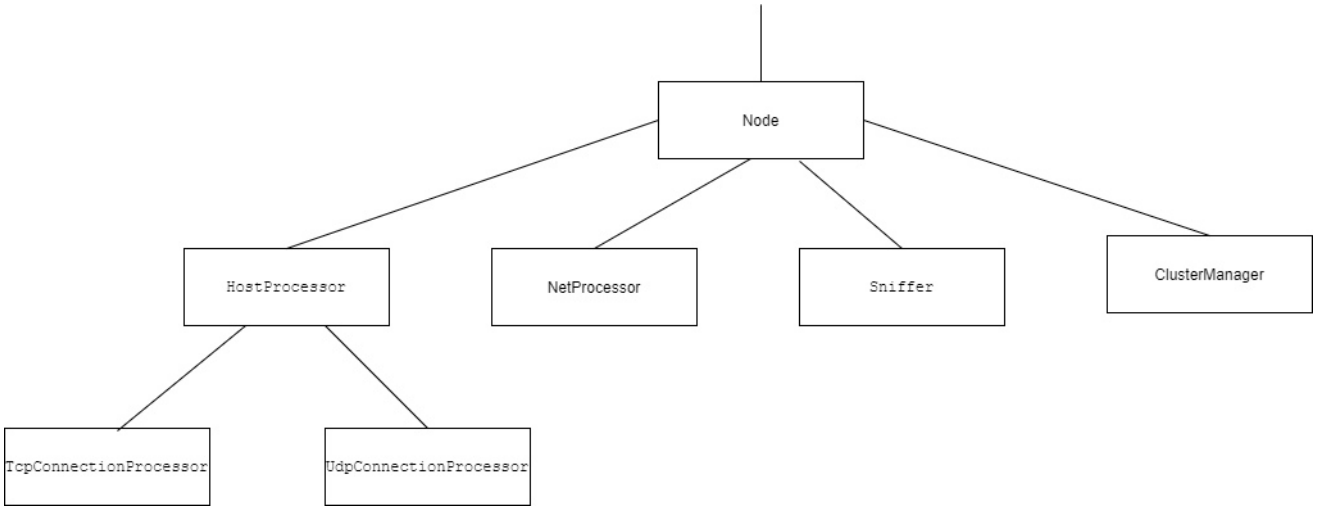


Рис. 3: Архитектура монитора

- Node – актор ответственный за создание других акторов, прием сообщений с интерфейса и других удаленных акторов;
- Sniffer, роль которого заключается в захвате сетевых пакетов и отправке их на обработку нужному актору;
- ClusterManager отвечающий за сбор данных с других агентов и хранящий список активных агентов;
- HostProcessor, создающий акторов для обработки tcp или udp диалога, в простом варианте так же хранящий данные собранные с них;
- NetProcessor, актор создающий HostProcessor на каждый хост в диапазоне маршрутов;
- TcpConnectionProcessor и UdpConnectionProcessor акторы ответственные за обработку tcp и udp сессий.

В следующих пунктах будет описана роль каждого из них, их набор состояний и проведений, но перед этим стоит сказать о стартовой точке и конфигурационном файле.

## 3.1 Конфигурация

В стартовой функции main считывается файл конфигурации. На основе него специализируется система акторов, создается главный Node актор и запускается Http сервер, который играет роль интерфейса пользователя для взаимодействия с системой. Базовый файл конфигурации в HOCON формате представлена на рис.4.

```
akka {
  actor {
    provider = cluster
    deployment {
      /node/clusterManager/router{
        router = broadcast-group
        routees.paths = ["/user/node"]
        cluster {
          enabled = on
          allow-local-routees = true
        }
      }
    }
  }
  remote {
    netty.tcp {
      hostname = ${clustering.ip}
      port = ${clustering.port}
    }
  }
  cluster {
    seed-nodes = [
      "akka.tcp://"${clustering.cluster.name}"@"${clustering.seed-ip}":"${clustering.seed-port}
    ]
  }
}
http {
  ip = "127.0.0.1"
  ip = ${?SERVER_IP}
  port = 8800
  port = ${?SERVER_PORT}
}
clustering {
  ip = "127.0.0.1"
  ip = ${?CLUSTER_IP}
  port = 2552
  port = ${?CLUSTER_PORT}
  seed-ip = "127.0.0.1"
  seed-ip = ${?CLUSTER_SEED_IP}
  seed-port = 2552
  seed-port = ${?CLUSTER_SEED_PORT}
  cluster.name = "TrafficAnalyzer"
}
sniffer {
  device-id = 0
  local-ip = "192.168.1.103"
}
```

Рис. 4: application.conf

В секции akka указаны указаны опции кластеризации, а именно включение данной опции, описание настроек как к этой систему будут обращаться удаленно, описания ключевых хостов, через которые будет происходить процесс автоматического присоединения

к кластеру (в данном случае этот хост будет вялятся ключевым) и настройки роутера, через который будут отправятся сообщения другим агентам. В секции http указан адрес и порт сервера, который будет играть роль интерфейса.

Секция sniffer должна содержать перечисления идентификаторов или имен сетевых устройств, с которых будет производится захват трафика, а так же ip адрес, который на данном сетевом устройстве будет распознаваться как локальный. Помимо этого эта секция может содержать обязательные для захвата маршруты.

Опционально возможно добавление данных для подключения к базе данных, однако в простейшем случае запись в бд рассматриваться не будет, так как во-первых необходимость бд в простейшем варианте (захват трафика, получение данных через http интерфейс) попросту отсутствует. Во-вторых требуется строго определить какой набор информации необходимо сохранять, как часто это делать, и с каких источников и т.д., что в свою очередь увеличивает сложность и не привносит в систему каких-либо качественно новых изменений, собственно кроме возможности длительного хранения данных. Кроме того возможны сценарии (например гипотетический сценарий 2) в котором необходимость локальной базы данных и вовсе отсутствует, так как сбор и хранение информации с агентов будет осуществятся централизованно администратором сети.

## 3.2 Node актор

Node актор является первым и главным в иерархии. Он ответственен за создание всех остальных акторов в системе, а так же все сообщения отправленные с пользовательского интерфейса и удаленных агентов в первую очередь будут поступать к нему в очередь сообщений.

К непосредственно его внутреннему состоянию относится список маршрутов, добавленных в наблюдение а также ссылки на акторов, ответственных за обработку пакетов с соответствующего маршрута. Таким образом образуется структура данных словарь, в котором ключи это ip адреса/сеть а значения те самые ссылки на акторов. При сборе информации с нескольких сетевых устройств структура описанная ранее добавляется в виде значения в другой словарь, ключами в котором являются id или имя сетевого устройства (в прототипе этого не было реализовано).

Основные сообщениями являются добавления маршрутов для наблюдений, при этом создается соответствующий актор (HostProcessor или NetProcessor) и отправляется сообщение актору Sniffer о перезапуске процесса захвата трафика, при этом передовая ему

множество маршрутов. Другие типы сообщений это получение данных об определенном хосте или сети. Если данные имеются то они передаются отправителю, иначе запрашиваются данные у актора ClusterManager.

```
sealed trait NodeResponse

case class AlreadyExist(message: String) extends NodeResponse
case class NotFound(message: String)     extends NodeResponse
case object Ok                           extends NodeResponse

case object GetClusterMembers
case class GetData(host: String)
case class GetDataFromNet(net: String)
case class AddHosts(hosts: List[String])
case class AddRanges(ranges: List[String])
```

Рис. 5: сообщения Node актора

## 3.3 Sniffer

Внутренним состоянием этого актора является класс PacketSniffer наследуемый от класса Thread. При создании он инициализируется словарем маршрутов. По списку маршрутов Формируется строка pcap фильтра.

Пример фильтра – net 99.181.64.0/20 or 185.42.204.0/22 or host 192.168.1.6. Затем эта строка компилируется, для выполнения фильтра на сетевой карте. Используя библиотеку Pcap4j, открывается сетевой интерфейс с заданным id (по умолчанию нулевым) и стартует бесконечный цикл приема пакетов.

При получении пакета с сетевого интерфейса определяется тип пакета (tcp или udp), используя словарь находится ссылка на актора, отвечающего за диалог с данным хостом или диапозоном хостов и отправляются сообщения следующего вида рис.5, содержащие в себе заголовок пакета, размер передаваемых данных и время получения.

```
case class TcpPacket(src: String, dst: String, tcpHeader: TcpHeader, payload: Long, time: Long)
case class UdpPacket(src: String, dst: String, udpHeader: UdpHeader, payload: Long, time: Long)
```

Рис. 6: Сообщения, содержащие информацию о пакетах

Единственным сообщением, обрабатываемым актором Sniffer, является запуск процесса захвата пакетов путем создания нового экземпляра класса PacketSniffer с новым набором

маршрутов, при этом старый останавливается вызовом специального метода окончания цикла захвата, предусмотренном для вызова с другого потока.

```
object Sniffer {
  case class StartSniffer(hostToActor: Map[String, ActorRef], rangeToActor: Map[String, ActorRef])
  def props(): Props = Props(new Sniffer)
}

class Sniffer extends Actor {
  var sniffer: Option[PacketSniffer] = None

  override def receive: Receive = {
    case StartSniffer(hs, rs) =>
      sniffer match {
        case Some(ps) =>
          ps.handle.breakLoop(); ps.join()
          val packetSniffer = new PacketSniffer(hs, rs)
          packetSniffer.start()
          sniffer = Some(packetSniffer)
        case None =>
          val packetSniffer = new PacketSniffer(hs, rs)
          sniffer = Some(packetSniffer)
          packetSniffer.start()
      }
  }
}
```

Рис. 7: Sniffer актор

## 3.4 ClusterManager

Роль этого актора заключается в отправке и сборе информации с других агентов. Для этого в свою очередь необходимо иметь перечень доступных хостов.

Для решения этой задачи в akka реализован протокол, при котором все узлы через определенные моменты времени обмениваются информацией о состоянии кластера со случайными другими узлами, при этом приоритет отдается хостам, от которых давно не было сообщений. В место того, что бы использовать метки доступен недоступен, для определения достижимости узла используются значение $\phi$ которое вычисляется по следующей формуле[4]:

$$\phi(t) = -\log_{10}(1 - F(t))$$

, где t – время с момента получения последнего сообщения, F – функция распределения нормальной случайно величины, со средним и дисперсией оцененными по историческим интервалам полученных сообщений. Таким образом $\phi$ определяет вероятность недоступности узла.

Для отправки сообщения удаленным акторам необходимо использовать интерфейс специального актора роутера, предоставляемым библиотекой akka или знать путь актора. В качестве примера используется широковещательный роутер, отправляющий сообщения всем зарегистрированным в кластере узлам.

Запрашивая информацию с удаленных агентов, нельзя ожидать точного кол-ва ответов, так как в процессе отправки узел мог стать недоступным, поэтому при запросе информации, актор ClusterManager переходит в состояние ожидания ответов и по достижении определенного интервала времени возвращается в обычное состояние, передовая отправителю сообщения собранные данные.

## 3.5 HostProcessor и NetProcessor

Роль HostProcessor заключается в создании и соответствующих акторов обработки Tcp или Udp соединений, а так же в отсутвие бд хранение полученных результатов. Его поведение довольно просто: если получен tcp пакет с флагом Syn создать обработчик соединения с соответствующим портом, во всех остальных случаях пересылка сообщений уже существующим обработчиком tcp соединений, а если такого актора не существует то просто проигнорировать сообщение. С ucp пакетами все еще проще, если обработчика нет то создать его, и пересылать им пакеты.

Информация о соединениях хранится в виде представленном на следующей картинке.

```
case class TcpData(T: Double, Rtt: Double, packetLoss: Double, duration: Double, size: Long, startTime: Long)
case class UdpData(T: Double, duration: Double, size: Long, startTime: Long)
case class Data(udpData: List[UdpData], tcpData: List[TcpData])
```

Рис. 8: Внутренние представление данных о соединениях

NetProcessor в свою очередь ответственен за создание акторов HostProcessor для каждого ip из диапазона с которым состоялось соединение и пересылку сообщений им.

На первый взгляд эти акторы выполняют довольно простую роль и можно было и обойтись без них и это действительно так. Однако они значительно упрощают архитектуру, что является несомненным плюсом их использования.

## 3.6 UdpConnectionProcessor

UdpConnectionProcessor аккумулирует данные об одной udp сессии началом которой является первый принятый системой пакет, а завершением считается отсутствие получен-

ных пакетов в течении некоторого промежутка времени. Для этого используется таймер, посылающий сообщения через определенный интервал времени. Если с времени получения последнего пакета прошло больше чем некоторое количество времени, то актор завершает свою работу и посылает данные родительскому актору HostHostProcessor. Другим возможным подходом является ожидание момента, когда порт на локальной системе освободится, однако его пока не удалось реализовать. На рис. 9 представлен полный код данного актора.

```
class UdpConnectionProcessor(host: String, port: Int) extends Actor with Timers with ActorLogging {
  var initTime = 0L
  var totalSize = 0L
  var lastTime = 0L
  val interval = 5 seconds

  timers.startPeriodicTimer(TickKey, Tick, interval)

  override def preStart(): Unit = {
    log.info( template = "UDP connection actor started for host {} with port {}", host, port)
  }
  override def postStop(): Unit = {
    log.info( template = "UDP connection actor closed for host {} with port {}", host, port)
  }

  override def receive: Receive = {
    case UdpPacket(_, _, _, payload, time) =>
      totalSize += payload
      if (initTime == 0L) {initTime = time}
      lastTime = time
    case Tick if (System.currentTimeMillis - lastTime) / 1000 > interval.toSeconds =>
      val duration = (lastTime - initTime).toDouble / 1000
      context.parent ! Data(totalSize.toDouble / duration, duration, totalSize, port, initTime)
      self ! PoisonPill
  }
}
```

Рис. 9: актор UdpConnectionProcessor

Хост в данном случае это ip адрес второй стороны соединения, а порт это номер порта на локальном устройстве. Объем информации выражается в байтах. В качестве время начала соединения используется время с начала эпохи Unix в миллисекундах, что позволяет без проблем конвертировать его в любой другой формат времени.

## 3.7 TcpConnectionProcessor

Этот актор призван хранить внутреннее состояние tcp сессии и помимо средней пропускной способности замерять так же значение Rtt и количество потерянных пакетов, а

точнее количество отправленных и принятых ретрансмиссий, так как определить точное число потерянных пакетов, использую только одну сторону, невозможно.

Так как номера последовательностей в tcp потоке возрастает, то каждый повторно отправленный пакет несомненно будет нарушать этот порядок, однако для выявления ретрансмиссий недостаточно условий нарушения порядка. Так для определения ретрансмиссий связанных с потерей пакетов в середине окна необходимо зарегистрировать наличие подряд отправленных подтверждений с одинаковыми номерами и в целом ориентиром для выявление рестрансмиссий должно служить большее время ожидание, чем при просто пакетах пришедших в неправильном порядке. Основные виды ретрансмиссий:

- TCP Retransmission – классический тип повторной передачи пакетов. Отправитель, не получив подтверждения от адресата, по истечению времени таймера (retransmission timer) предполагает, что пакет потерян и отправляет его снова
- TCP Fast Retransmission – отправитель повторно отправляет данные после предположения, что отправленные пакеты потеряны. Обычно на это указывает получение (обычно трех) подряд подтверждений с одинаковыми номерами.

Для простоты в прототипе был реализован только подсчет пакетов пришедших или отправленных в неправильном порядке.

Значения круговой задержки напрямую вычисляются как разница во временных метках между отправленным сегментом с номером n и размером l и получении подтверждения с номером n + l.

В отличии от udp сесии, в tcp диалоге можно определить начало и конец. Началом служит так называемая процедура 3-х этапного рукопожатия, которую повторяет актор. Он создается создается при отправке сегмента от локальным узла с флагом SYN, дожидается подтверждения SYN+ACK, если получен сегмет с флагом RST то завершает свою работу. В классическом варианте завершения tcp сеанса клиент посылает сегмент с флагом FIN, получает сегмент с флагом FIN+ACK и подтверждает его отправкой ACK. Для завершения работы актора достаточно получить от севера сегмент с флагами FIN+ACK, так как сервер к этому моменту уже закроет соединение и никаких данных больше не будет отправлено. Так же при получении сегмента с флагом RST работа актора завершается.

## 3.8 Интерфейс пользователя

В качестве интерфейса пользователя используется http сервер akka удобно интегрированный в систему акторов. На данный момент поддерживается следующее Rest Api:

- POST /addhosts body: {hosts:[]} добавление хостов для монитора
- POST /addranges body: {ranges:[]} добавление диапазонов ip в формате cidr
- GET /gethostdata?host=ip получение данных о хосте . Если локальные данные отсутствуют опрашивается кластер.
- GET /netdata?net=net получение данных о диапазоне.

```
val dataRoutes =
  concat(
    path( pm = "hostdata") {
      parameters( pdm = 'host) { host =>
        val future = (node ? GetData(host)).mapTo[Data]
        onSuccess(future) { data =>
          complete(data)
        }
      }
    },
    path( pm = "netdata") {
      parameters( pdm = 'net) { net =>
        val future = (node ? GetDataFromNet(net)).mapTo[Data]
        onSuccess(future) { data =>
          complete(data)
        }
      }
    }
  )
```

Рис. 10: Пример обработки запроса

GET localhost:8800/netdata?net=185.42.204.0/22

Pretty Raw Preview Visualize JSON


```
{
    "tcpData": [
        {
            "Rtt": 59.0,
            "T": 1045974.2925565711,
            "duration": 20.461,
            "packetLoss": 0.0012441546183877827,
            "size": 21401680,
            "startTime": 1589536313204
        },
        {
            "Rtt": 59.0,
            "T": 939613.9985869338,
            "duration": 281.657,
            "packetLoss": 0.0021805625055922873,
            "size": 264648860,
            "startTime": 1589536404532
        }
    ],
    "udpData": []
}
```


Рис. 11: Результат запроса

# 4 Результаты

Перед использованием системы был проведен ряд тестов на сравнение с программой анализатором трафика WireShark. Тесты показали, что средняя пропускная способность, замеры rtt и количество пакетов полученных в неправильном порядке системой верны.

В качестве практического эксперимента было решено понаблюдать за изменением пропускной способности при просмотре онлайн трансляций или записей на платформе twitch.tv использующей в качестве протокола транспортного уровня tcp. Экспериментально было обнаружено, что видео живых трансляция транслируется с ip диапазона 185.42.204.0/22. Был написан баш скрипт включающий и выключающие браузер через примерно 5 минут. В результате были получены такие данные, которые в среднем показывают, что пропускная способность с утра и ближе к вечеру лучше.

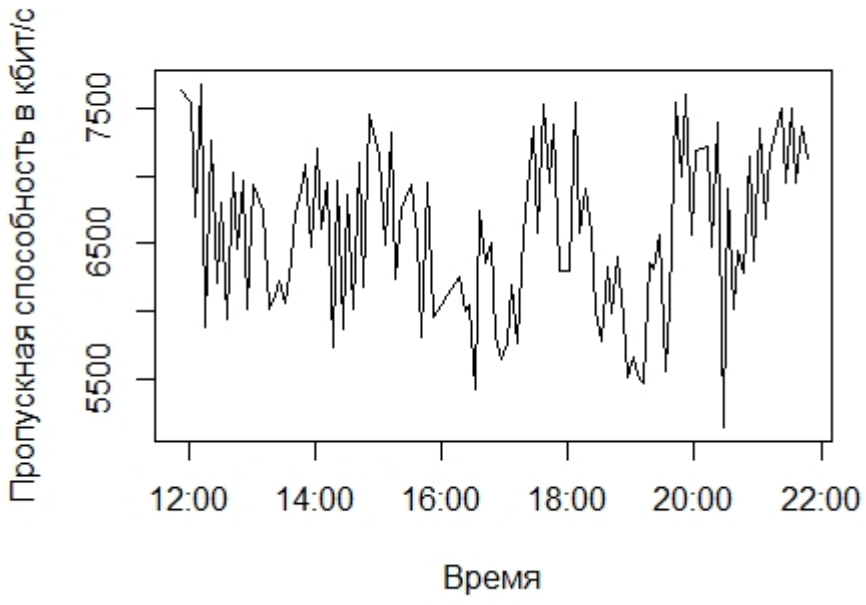


Рис. 12: График средней пропускной способности

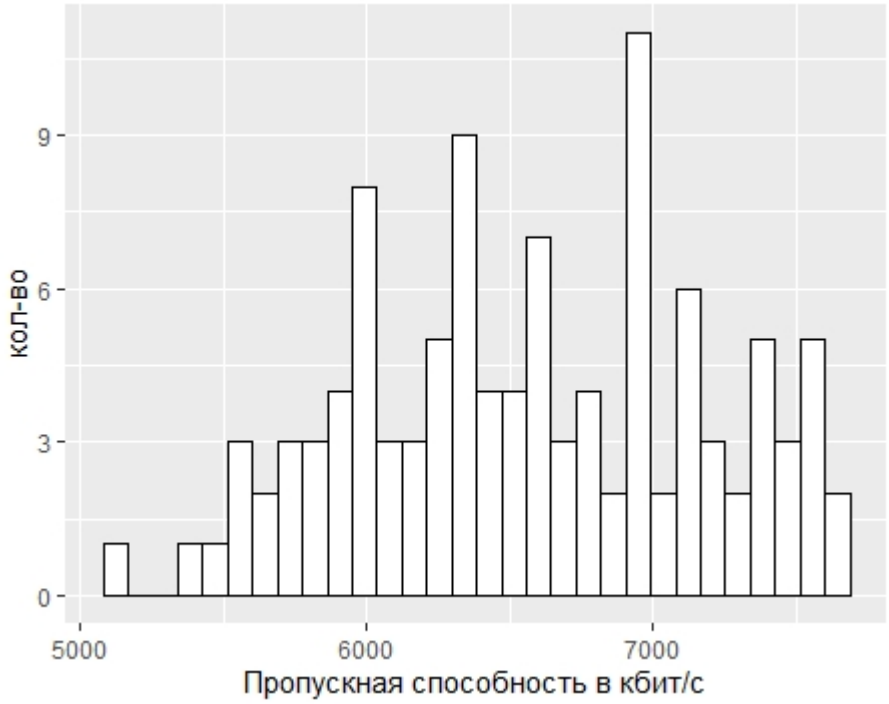


Рис. 13: Гистограмма средней пропускной способности

Данные полученные о средней пропускной способности согласуются с нормальным распределением. Тест Шапиро-Уилка дает p value равное 0,066.

## 4.1 Метрики прототипа

- Количество файлов: 12
- Строк кода: 607
- Вес собранного jar файла со всеми зависимостями, кроме системных libpcap/winpcap: 44.8 МБ

# 5 Заключение

В результате ~~научно исследовательской работы~~ было получены навыки в разработке систем с использованием акторов, спроектирован и реализован прототип, позволяющий получать данные о зарегистрированных маршрутах и при необходимости делится ими. Дальнейшие пути по улучшению системы

- Отделить повторно отправленные пакеты от пришедших в неправильном порядке.
- Добавить графический интерфейс
- Предложить и реализовать методы для анализа полученных данных, для случая когда данные будут собираться регулярно.
- Добавить поддержку БД.

# Список литературы


1. Богоявленская, О.Ю. Распределенная многоагентная система мониторинга и прогнозирования производительности транспортного уровня сетей передачи данных [Текст] / О.Ю. Богоявленская // Программная инженерия. - Москва, 2018. - Т.9, №.1. - С.11-21. - ISSN 2220-3397. - Режим доступа: http://novtex.ru/prin/rus/10.17587/prin.9.11-21.html (https://elibrary.ru/item.asp?id=32385387) (ВАК, РИНЦ)

2. Олифер В.Г., Олифер Н.А. Компьютерные сети. Принципы, технологии, протоколы: Учебник для вузов. 3-е изд. — СПБ.: Питер, 2008. — 958 с.

3. [Электронный ресурс]: URL: `https://en.wikipedia.org/wiki/Actor_model`

4. Naohiro Hayashibara, Xavier Défago, Rami Yared: The $\phi$ Accrual Failure Detector: [Электронный ресурс] URL: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.80.7427&rep=rep1&type=pdf`

5. [Электронный ресурс]: URL: `https://akka.io/docs`