Как считать и анализировать сотни гигабит трафика в секунду

Слава Николов,
UCDN.com

highload++ 2014

Конференция разработчиков
высоконагруженных систем

О компании UCDN.com

- UCDN.com входит в состав холдинга XBT (Webzilla)
- 4 года успешной работы
- 12 точек присутствия в мире
- сотни Gbps трафика
- более 7 млрд хитов в день

Немного обо мне

- технический директор и сооснователь компании UCDN
- сооснователь большого видео проекта (10ки Gbps в 2007 году)
- опыт работы в хостинговых компаниях
- первый сайт за который мне заплатили сделан в 1997 году

О чем будем говорить

- как мы 60 Gb логов/день собирались считать
- общая схема системы подсчета трафика
- кто что делает ?
- а почему не hadoop ?
- как бы мы написали сейчас такое же ?
- что из этого можно использовать в вашем проекте ?

Только наш опыт, он не абсолютная истина

Как мы 60 Gb логов/день собирались считать

- есть 12 DC, в каждом из которых находятся от 10ти до 100тни серверов
- каждый DC находится на rtt от 5 мс (в Европе) до 180 мс (Азия) друг от друга
- на каждом сервере все клиенты в перемешку
- каждую секунду проходят через сеть сотни Gbps (каждые 100 Gbps ~ 13 Gb данных/сек)
- каждые 24 часа проходят около 10ти млрд хитов (10 000 000 000 хитов)

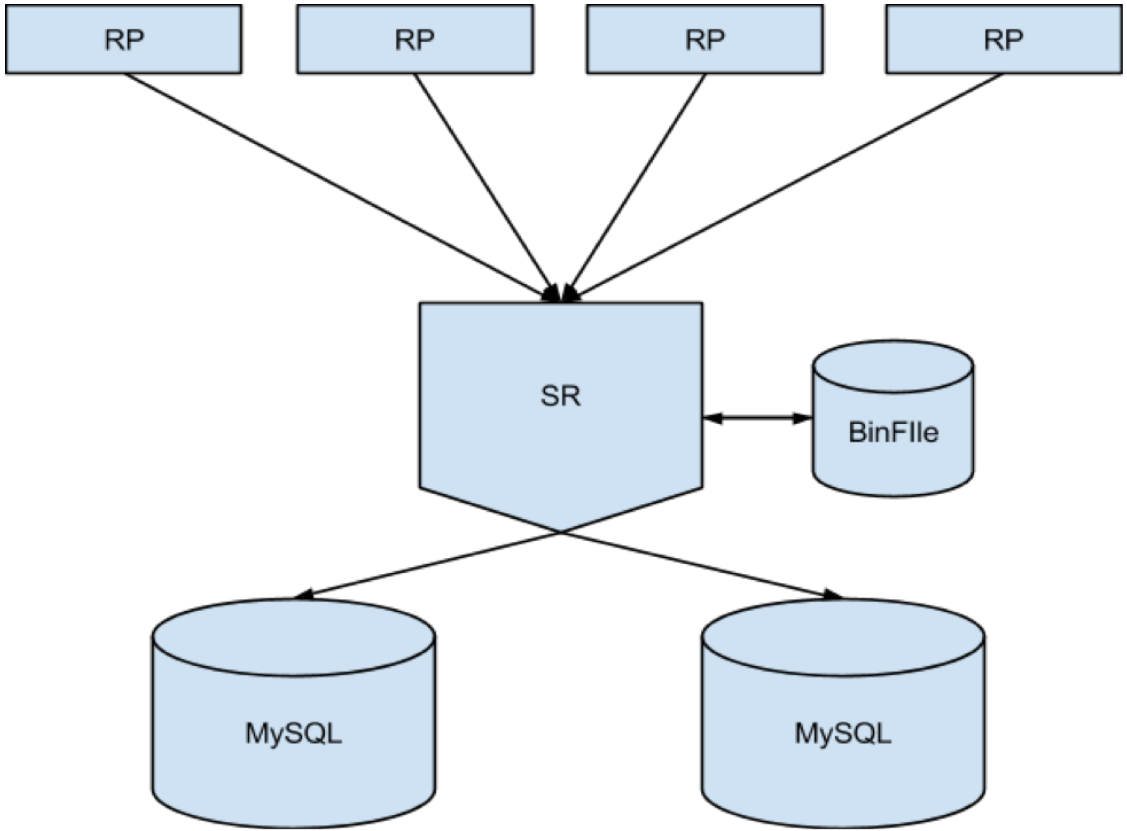
Что надо было считать ?

- каждые 5 минут
  - средную скорость отдачи по клиентам по DC (на все миллиарды хитов)
- в конце запроса
  - http return code
  - average speed (bytes)
  - total bytes served
  - и несколько других служебных метрик

Почему не получилось с логами ?

- слишком много хитов
- логи занимают место (на каждые 100 Gbps ~ 60Gb/день)
- всю кучу логов надо по латентным связям слать далеко, что не быстро
- если нет связи (или она плохая), то надо буферировать много информации, а диски не бесконечные
- логи весом 60 Гб в день, надо парсить, а CPU и дисков жалко

Общая схема



- RP - reverse proxy
- SR - stats receiver
- BinFile - binary log file
- MySQL

Общая схема - кто что делает ?

- Reverse Proxy (nginx модуль) - собирает первичную статистику в самом веб-сервере и шлет ее дальше
- Stats Receiver (standalone deamon) - агрегирует статистику собраную из веб-серверов и шлет ее в базу данных
- BinFile - бинарные логи, если Mysql не может обработать всю статистику
- База данных (MySQL) - делает последную агрегацию

Reverse proxy - internal stats

- Nginx с proxy cache модулем
- написан модуль (internal stats), который собирает 2 типа статистики
  - тип 1: статистика по средней скорости отдачи за интервал (каждые 5 мин)
  - тип 2: статистика в конце запроса

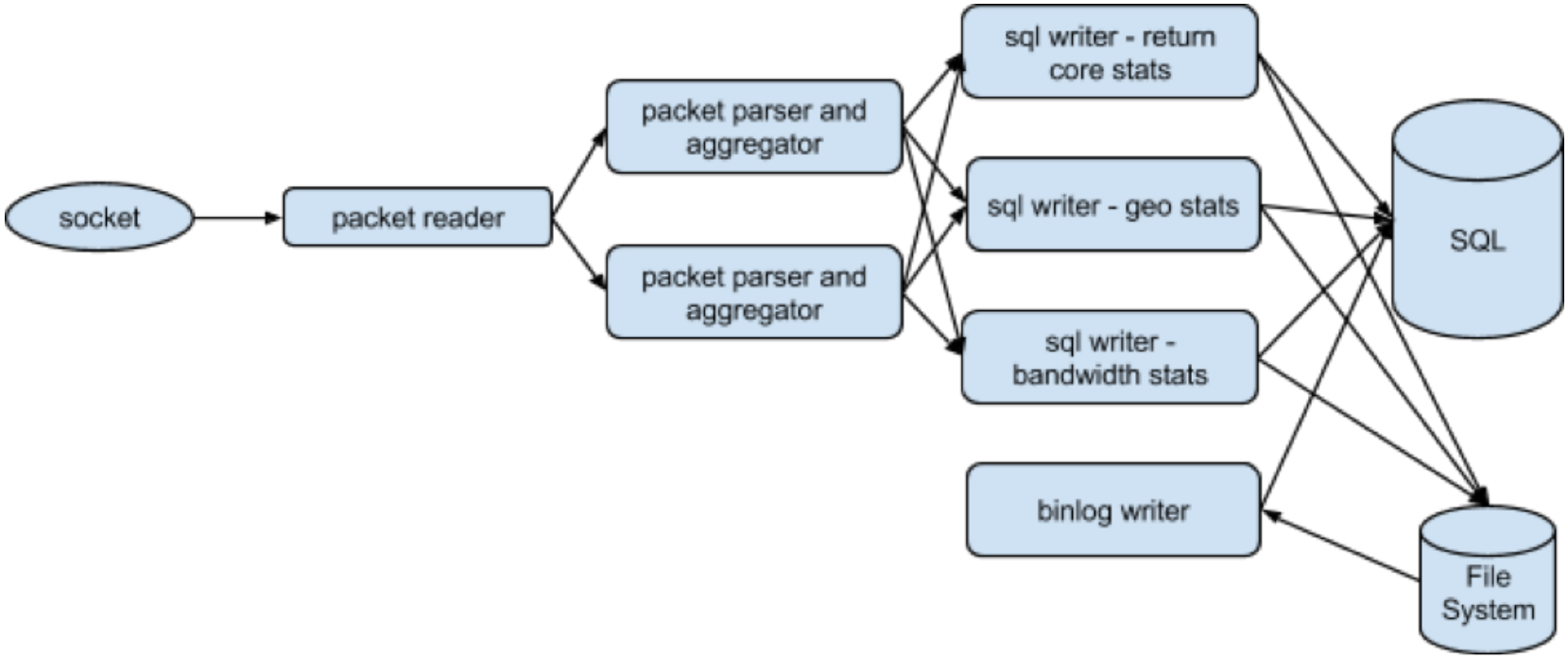
Internal stats module - сбор скорости

- модуль хранит в памяти список всех активных запросов
- каждую минуту он обходит этот список и собирает кол-во отданых байтов за последнюю минуту
- в конце этой минуты, он шлет данные по UDP stats receiver-у
  - UDP нормально, т.к. SR находятся в том же DC как RP
- протокол обмена данными бинарный
  - т.е. не нужно парсить stats receiver-у (он знает что где ожидать)
  - каждый пакет загружается полностью (payload), чтобы не слать полупустые
  - передается следующая информация:
    time interval, host_id, client_id, zone_id,
    bytes served, requests_served

Internal stats module - сбор данных per request

- модуль ждет окончания запроса
- в log phase есть hook, который собирает информацию:
  - host_id,
  - client_id,
  - zone_id,
  - Hit type - cache hit, cache miss, redirect
  - bytes sent
  - HTTP responce code
  - client IP
- в конце запроса шлет данные таким же образом (UDP) SR

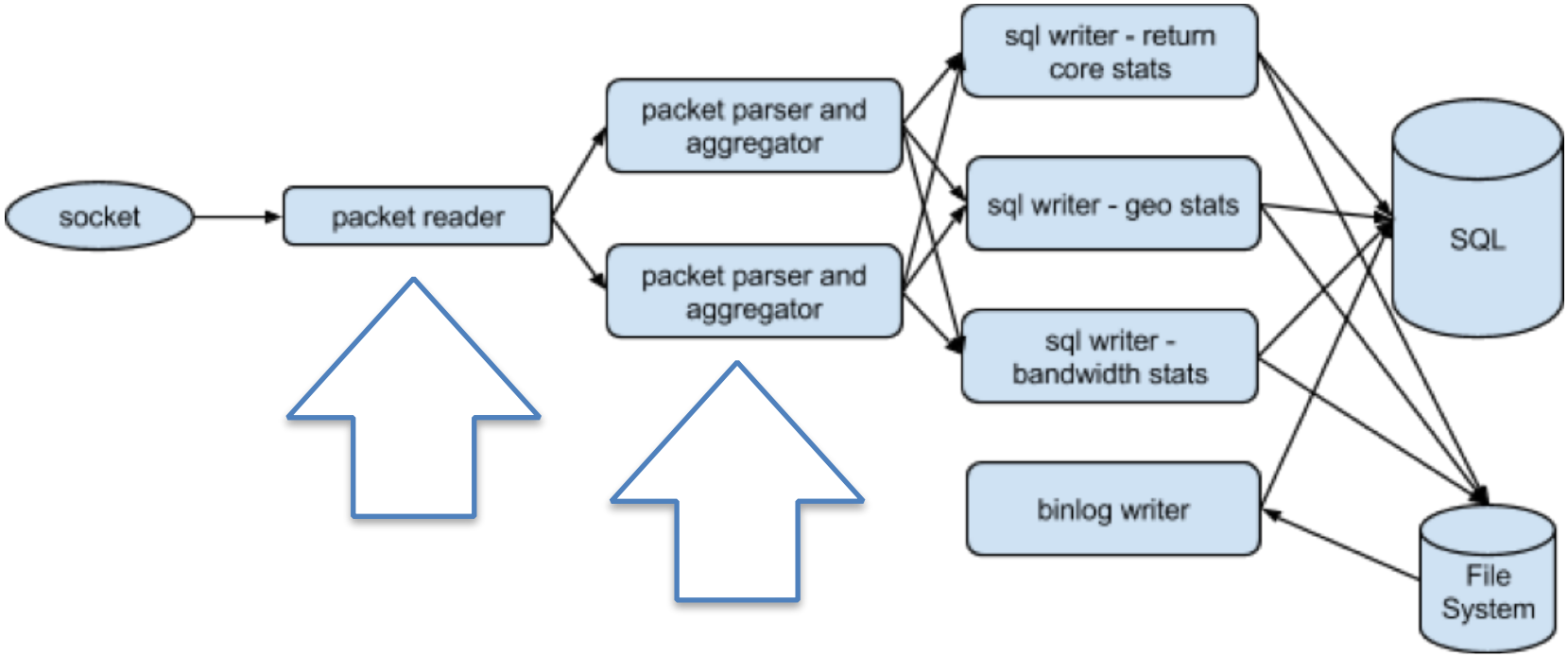
Stats receiver - общая архитектура



Stats receiver - общая архитектура

- агрегирует данные из reverse proxy
- реально 2 демона:
  - master SR demon:
    - читает конфиги
    - наблюдает за SR slave
    - перезапускает SR slave
  - threaded slave SR demon (worker)
    - socket reader thread
    - packet parser & aggregator thread
    - storage writer thread
    - binlog worker thread
    - telnet thread

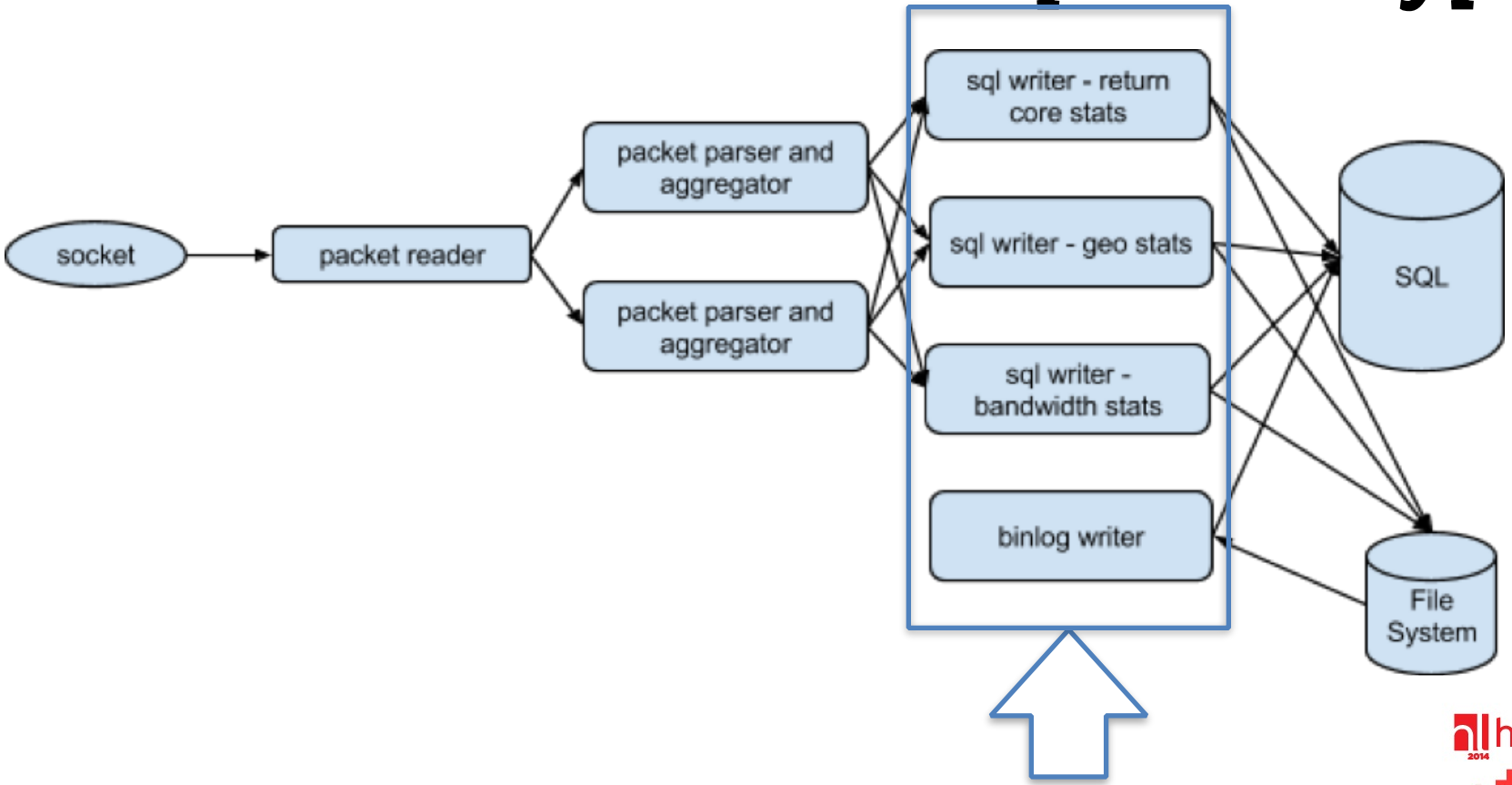
Stats receiver - общая архитектура



Stats receiver - что делает каждый тред

- socket reader thread (один)
  - читает пакеты быстро из сокета
  - записывает во внутренние буфера, для последующего пересчета
  - можно настраивать кол-во этих буферов в конфигурации
- packet parser & aggregator thread (несколько)
  - у каждого треда есть собственый буфер в котором свежие пакеты со статистикой
  - читает пакет со статистикой, проверяет целостность данных
  - есть структура данных (вид non-blocking tree), в котором записываются скорости
  - по ключу пакета находится его место в дереве и записывается скорость (тут реально второй Reduce)
  - если такого ключа нет (т.е. комбинации client_id, zone_id), то создается новый элемент в дереве и туда записывается скорость

Stats receiver - общая архитектура



Stats receiver - что делает каждый тред (2)

- storage writer thread
  - читает дерево с данными
  - генерирует SQL запросы и шлет их на ближайший DB сервер
  - так же, проверяет состояние связи и скорость записи DB
  - если скорость упала ниже граничного значения, то начинает писать в бинарные логи
  - бинлоги это все то, что не может вовремя записаться в базу данных
- binlog worker thread
  - проверяет есть ли бинлоги
  - если есть, то пытается связаться с DB и написать их туда
- telnet thread
  - дает возможность залезть и посмотреть счетчики, состояние тредов и всякие статистики
  - так же есть JSON статистическая страница, для внутреннего контролера

Database

- база данных для агрегированых статистик
- все записывается фиксированными интервалами продолжителностью 5 мин
- таким образом каждый день кол-во рядов постоянно (288 5min/24h)
- можно ротировать легче базу данных
- есть 2 основные агрегационные точки, которые связаны в master-master репликацию

Отказоустойчивость

- каждый RP может слать статистику разным SR
- есть failover в RP, если SR недоступен (за этим следит отдельный демон, т.к. UDP stateless)
- каждый SR может писать в >1 базы данных (паралельно)
- SR следит за состоянием записи в базу и создает бинлоги, когда не видит базу

Скалируемость

- RP практически не загружает nginx и т.к. код компилируется, то нет penalty интерпретирования
- каждый SR может агрегировать другому SR (если необходимо stack)
- каждый SR может писать в несколько баз данных шардя записи

Нагрузка

- так как все native, то практически не заметно
- в nginx: немного памяти на внутрение структуры + немного CPU на обход структур
  - реально не замечали, даже после первого пуска в production, не могли отличить сервера со считалкой от тех без нее
- SR мега быстрый
  - можно настраивать количество тредов
  - все native и не использует ничего внешнего
  - все структуры написаны самостоятельно и максимально упрощены
  - один сервер посчитал 124 Gbps с load average 0.1
- DB
  - есть горизональный шардинг, каждый SR может писать паралельно в несколько баз
  - последняя агрегация только на фиксированом кол-ве интервалов (288 интервалов на ключ)

Hadoop ?

- мы не против hadoop, даже скорее всего будем использовать его в анализе performance / мониторинг данных
- начали читать и настраивать, но испугались всех слоев
- не хотели покупать сервера, т.к. сейчас все работает на тех же серверах, которые отдают трафик
- данные структурированы и агрегация одна и та же, так что задача не меняется постоянно

Как бы мы сейчас написали такое же ?

- прошло почти 4 года
- модуль так же (никуда не деться, нужно лезть в nginx)
- SR можно написать на чем-то более легком, чем C, которое компилируется
  - erlang
  - go
  - если не жаль CPU и памяти и интерпретируемое можно поставить

Что из этого можно использовать в вашем проекте ?

- анализ данных
  - можно ли их фильтрировать (Map)
  - поддаются ли агрегации у источника ?
- не всегда нужно микроскопом забивать гвозди
  - иногда универсальные системы слишком громоздкие
  - часто задачу можно сделать проще, если смотреть в корень проблемы
- собственные алгоритмы агрегации в коде, могут быть быстрее NoSQL + latency

Спасибо! Вопросы ?

slava@ucdn.com