课程设计报告

1. 课题小组分工：需要明确说明各成员在整个课题中分工负责完成的内容

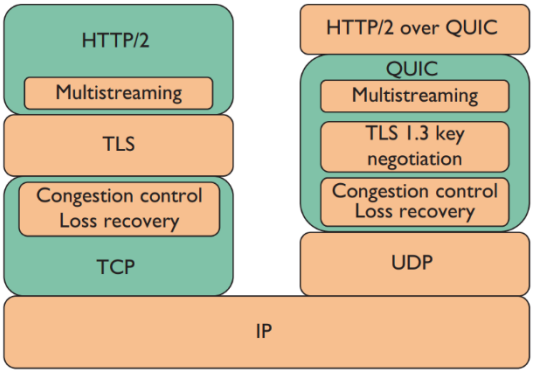
2. 课程设计题目

基于大数据分析的 QUIC 初始参数配置

3. 摘要

Quic 协议是谷歌公司基于 UDP 开发的新一代传输协议，相对于 TCP+TLS 的组合，Quic 大大减少了建立连接所需的握手次数，同时提供了可按需配置的多种拥塞控制算法，以及连接迁移，连接级别的多路复用等等特性，目前 IETF 在 HTTP/3 中已经提议使用 Quic 取代 TCP 作为 HTTP/3 的默认传输协议。本文在 Google 提供的 stand-alone 版本的 Proto-Quic 为基础，结合网络仿真器 ns3，以及大数据处理系统，根据 ns3 仿真生成的 quic 连接日志信息，利用 spark 以及相关机器学习算法对于这些日志的分析，实现了在建立连接之初对 Quic 拥塞控制算法的相关参数的配置，并对配置前后的性能做了评价实验。

4. 研究问题背景



众所周知，HTTP 协议一直以 TCP 协议作为其默认的传输层协议，纵然 HTTP 只是一个应用层协议，是可以实现在其他传输层协议之上，比如 UDP，但是各大浏览器厂商只支持基于 TCP 的 HTTP 协议，因为 HTTP 协议需要传输层提供可靠的数据交付，伴随着互联网的发展，网页内容不在仅仅是简单的文字和图片，视频以及现代化的 web 应用甚至基于网络的操作系统比如 chrome os，在现代互联网中占据越来越大的比重，而 HTTP 协议之下的传输层协议 TCP 也伴随着 web 的发展暴露出越来越多的的问题，首先 TCP+TLS，握手次数太多，至少需要四次握手才能客户端发送 HTTP 请求，HTTP1.1 一次连接只能发送一个请求，对于有众多网页元素的网页来说需要建立多个 TCP 连接，才能在合理的时间之内发送完请求；HTTP\2 支持在一个 TCP 连接之中发送多个 HTTP 请求，但是当 TCP 本身阻塞之时，在其上传输的所有 HTTP 请求都会被阻塞， 这被称为对头阻塞问题，另外互联网社会，客户端有可能处于多种网络环境之下，比如手机客户端可能由于 wifi 信号不稳定，在 4G，wifi 之间来回切换，而每一次切换，都需要传输层重新

建立一次连接，又因为 TCP 实现在端系统内核之中，并非应用层所能控制，所以应用层只能通过各种方式来避免 TCP 的种种问题，针对 TCP 的种种缺点，谷歌公司在 spdy 协议(HTTP/2 协议的前身)的基础之上开发了 Quic 协议，如图所示，Quic 协议在 UDP 层之上实现了 TCP 的可靠性和拥塞控制，TLS 的全部，以及 HTTP/2 的 stream 级别的多路复用，Quic 协议具有以下特点：

1. 减少了 TCP 三次握手及 TLS 握手时间
   整个的握手过程需要 0-1RTT
2. 改进的拥塞控制
   拥塞控制算法是可插拔的，可以根据应用需求选择合适的拥塞控制算法
3. 避免队头阻塞的多路复用
   QUIC 在在 UDP 内实现了和 Http2 相对应的 stream 级别的多路复用(一个 connection 可以有多个 stream)，同时由于使用的是 UDP，所以没有 TCP 固有的传输层队头阻塞问题，而且 QUIC 支持更多的 SACK 块，对于乱序更加友好
4. 连接迁移
   QUIC 使用 connection ID 来标识一条连接，而非传统的四元组(srcIP, srcPort, dstIP, dstPort)，所以当客户端在不同网络状态下切换之时(比如 4G 和 wifi 之间的相互切换)不需要重新建立连接(0-RTT)

由于 QUIC 本身基于 UDP，相关的拥塞控制，错误恢复等等机制都在应用层实现，所以开发者就可以根据自身的需要定制相关的算法，参数来优化应用的性能，目前来说 QUIC 主要应用于一些 web 应用或服务，而浏览器在和提供这些服务的服务端交互的时候，会将客户端相关的一些信息，附在 User Agent 里面传送给服务端，比如说客户端所处的网络环境，所使用的操作系统，浏览器类型和版本等等信息，结合服务端自身产生的日志信息比如该连接的丢包率，请求资源的类型大小，发包数，平均拥塞窗口大小，重传数目等等网络信息就有可能对每一个(种\条)客户端(网络环境\连接)提供定制化服务，我们此次课程设计主要针对拥塞控制算法的一些初始参数进行配置，关于这些初始参数对请求-响应时延(latency)的影响，W3C 主席 Ilya Grigorik 所著"High Performance Browser Networking" 中有详细介绍，以初始拥塞窗口为例，由于大部分请求资源的大小都比较小，如果初始拥塞窗口过小，不能在第一次响应时发送完所有字节，就需要另外一个 roundtrip 来发送完其余字节，而 latency 的主要组成部分便是 round trip time，因此如果能合理地设置初始拥塞窗口大小，对于小的请求便能够大大地减少 latency。

我们通过 NS3 仿真器来仿真 QUIC 流量数据，NS3 是一种网络仿真器，通过 NS3，可以根据需要配置从客户端至服务端经过的所有设备的参数，比如链路带宽，丢包率，等等，在 Diego de Araújo Martinez Camarinha（葡萄牙语）的 quic 模块(解决了谷歌提供的 proto-quic 的依赖问题，集成到了 ns3 之中，几乎就是真实的 Quic)之上，我们模拟了 Quic 客户端和服务端在真实世界中的行为配置等等细节信息，以其得到接近真实世界的结果。并且在 ns3 Quic 模块的基础之上设计了一个机制可以根据 spark 大数据处理系统对日志进行分析的结果来对新建连接的的拥塞控制算法初始参数进行动态配置，并在优化前后做了实验对比。

5. 主要技术难点和拟解决的问题，尤其要解释说明哪些地方、为什么需要采用 MapReduce

我们希望通过对大量的 QUIC 连接日志信息的分析来对每一个(种)客户端\IP(网络环境\连接)来提供个性化的初始连接参数配置，比如说，最合适的拥塞控制算法，初始拥塞窗口，初始 RTT 值，初始 pacing rate，FEC rate 等等，来尽量减少 round trip 数目

技术难点：

1，服务端拥有的 log 数目通常是巨量的（24h 数 GB），如何快速有效地分析出相关的信息，对于单机算法而言几乎是不可能的，所以需要使用 mapreduce 或者 spark，同时，通常而言，长时间来看连接状态对于时间而言是有周期性的，而短时间来看，最近的相关连接信息的可信度是更高的，因此需要综合考量这些 insight。

2，另外，如何快速响应新到来的连接也是一个相当棘手的问题，因为通常而言，请求的大部分 web 页面元素大小都不是很大(但是一旦发生重传丢包就很有可能增加 round trip 数目，大大增加了传输时间)，所以一个连接的生命周期通常很短，所以需要在尽可能短的时间查询到该连接最适用的初始参数配置。

3、另外一个难点是系统集成的问题，通过何种策略将 spark ns3 quic 结合到一起也是一个难点。

6. 主要解决方法和设计思路，尤其要解释说明如何采用 MapReduce 并行化算法解决问题

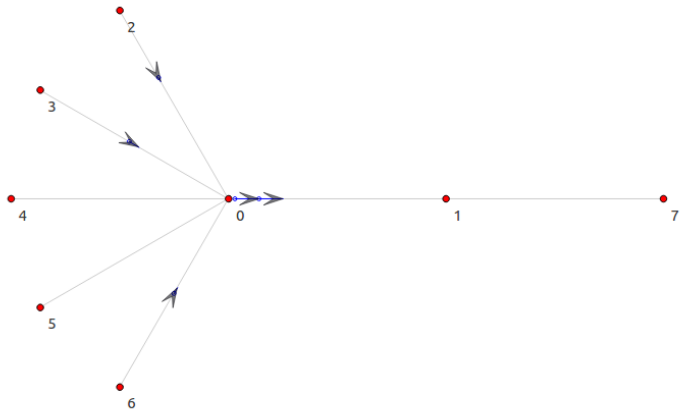
服务端收到客户端请求之时，可以看到客户端客户端 IP，Port，以及客户端请求头信息，一个客户端可能有多个请求到达，客户端 IP 可以看作客户端的唯一标识，于是日志信息中客户端 IP 一致的即为同一个客户端在不同时间的连接状态信息，而客户端 IP 保持不变之时，意味着其网络环境，设备信息等信息维持不变，于是我们对于服务端日志信息进行分类之时，利用 map-reduce 先根据客户端 IP 进行预分类，而后根据预分类的连接状态信息的均值，使用 map-reduce 进行 k-means 聚类，这样做的原因很显然，不同的客户端虽然 IP 不同，但是他们的网络状态可能相近，比如都在同一个地域，使用同一家运营商提供的 4G 服务，在相近的时间请求相同的资源，如何根据请求的 IP，请求头部信息进行判断它属于哪一个类别，从而对他的拥塞控制算法的一些参数进行配置，我们的策略是控制 k-means 输出信息，为其簇标识，和属于该簇的所对应连接状态信息的均值向量，与最值向量，另外还有 client IP 与簇 ID 的对应信息，这样产生的信息的数据量仍然很大，我们实现了时间复杂度为 O(1) 的 LRU cache 来缓存最近请求的 client IP 与簇 ID 的对应信息，以及，最近查询的簇 ID 与其对应的连接状态信息，cache 命中的话，就利用得到的其所属簇的连接状态信息来配置该连接拥塞控制算法的初始参数(Quic 实现中，server 收到请求之后，会交付给 dispatcher，dispatcher 会创建一个新的 session，和 connection，每个 connection 拥有一个 quic_send_packet_manager 的实例，quic_send_packet_manager 会创建一个 send algorithm 的实例(也就是拥塞控制算法的类))，另外实际情况中，由于客户端请求随着时间的增长不断到来，所以我们原本打算利用 spark 或者 flink 的流处理能力，然而由于时间关系，仅仅实

现了 hadoop mapreduce 和 spark mapreduce 版本，所以对于新到来的未见的请求进行分类的策略就是周期性地 re run 我们的 map reduce 程序。

在 ns3 quic 模块中完成了相关参数配置的接口实现之后，我们在 dispatcher 初始化 connection 之后，从新创建的 connection 入手取出实例化的 send algorithm 进行相应的配置工作。map reduce 的输出我们目前放在了本地磁盘之上，之后 LRU cache 在服务端启动，dispacher 初始化的时候也会进行磁盘读取进行初始化，当然在现实情境之中服务端启动之时使用 spark sql 或者直接通过 hdfs 提供的接口进行 LRU cache 的初始化更加合理，我们此处暂时没有实现。

7. 详细设计说明，包括详细算法设计、程序框架、功能模块、主要类的设计说明，包括主要类、函数的输入输出参数、尤其是 map 和 reduce 函数的输入输出键值对详细数据格式和含义，主要功能和算法代码中加清晰的注释说明

由于 ns3 和 quic 都太过庞大，所以此处仅仅介绍涉及到的重要的部分



上图所示，为使用 ns3 生成的网络拓扑图，其中节点 7 为服务端，节点 2，3，4，5，6 为客户端，为了尽可能模拟真实环境，客户端数目，以及客户端请求资源大小，客户端与 0 号节点之间链路的带宽，丢包率，时延等等都是随机生成，同时又考虑到实际情况，客户端请求资源大小，客户端与 0 号节点之间的链路带宽，丢包率，传播时延为相互独立的高斯分布，每次运行都以时间作为随机种子，避免出现配置碰撞，具体细节见 quic-mapreduce.cc (ns3 是一个使用 c++语言实现的网络仿真器，因为要详细介绍拓扑生成细节涉及到的 ns3 相关细节太多，同时所写仿真代码类似伪代码，所以此处不赘述)
服务端启动成功，调用 int QuicSimpleServer::Listen(const IPEndPoint& address)，在其内部初始化 QuicSimpleDispatcher,同时初始化 ConnectionStatsQuery，ConnectionStatsQuery 的细节如下:

```
class ConnectionStatsQuery{
    public:
        ConnectionStatsQuery();
        ConnectionStatsQuery(size_t size);
        ConnectionStatsQuery(size_t size, size_t clusterNum);
        void Initialize(const string clusterStatsLogPath,
                const string ipToClusterLogPath);
        ~ConnectionStatsQuery();
        bool Query(string ip, vector<double> &stats);
    private:
        size_t size_ = 2000;
        size_t clusterNum_=100;
        LRUCache<string, vector<double>> *cluster_to_stats_;
        LRUCache <string,string> *ip_to_cluster_;
};
```

size_为 ip_to_cluster_的缓存容量，clusterNum_为 cluster_to_stats_的缓存容量，Initialize 方法为初始化 ip_to_cluster_和 cluster_to_stats_，参数意义即为其字面意义，Query 根据客户端 ip 地址查询连接状态信息，并将结果写入 stats，成功返回 true，失败返回 false。

Initialize 的实现细节如下：

```
void ConnectionStatsQuery::Initialize(const string clusterStatsLogPath,
                const string clusterToIPLogPath){
    std::ifstream clusterStatsLog( clusterStatsLogPath, std::ios::in);

    if(!clusterStatsLog.is_open()){
      cout << "Error opening file: " << clusterStatsLogPath <<std::endl;
      exit (1);
    }
    string s;
    while(getline(clusterStatsLog,s)){
        vector<string> sv;
        split(s,sv,';');
        vector<string> sv1;
        vector<string> sv2;
        vector<string> sv3;
        split(sv[0],sv1,',');
        split(sv[1],sv2,',');
        split(sv[2],sv3,',');

        string clusterID = sv1.front();
        sv1.erase(sv1.begin());
        vector<double> stats;
        for(unsigned int i=0; i<sv3.size(); i++){
            stats.push_back(atof(sv1[i].c_str()));
            stats.push_back(atof(sv2[i].c_str()));
            stats.push_back(atof(sv3[i].c_str()));

        }
        cluster_to_stats_->Put(clusterID,stats);
    }

    clusterStatsLog.close();
```

```
    std::ifstream clusterToIPLog( clusterToIPLogPath, std::ios::ate);
    if(!clusterToIPLog.is_open()){
      cout << "Error opening file: " << clusterToIPLogPath <<std::endl;
      exit (1);
    }
    clusterToIPLog.seekg(-2, clusterToIPLog.cur);
    int nCurentPos = -2;
    for(unsigned int i = 0; i < size_; i++){
        while( clusterToIPLog.peek() != clusterToIPLog.widen('\n') ){
            nCurentPos = clusterToIPLog.tellg();
            if (nCurentPos == clusterToIPLog.beg)
                break;
            clusterToIPLog.seekg(-1, clusterToIPLog.cur );
        }
        if (nCurentPos == clusterToIPLog.beg)
            break;
        clusterToIPLog.seekg(-1, clusterToIPLog.cur);
    }
    if (nCurentPos == clusterToIPLog.beg){
      clusterToIPLog.clear();
      clusterToIPLog.seekg(0,clusterToIPLog.beg);
    }
    else{
        clusterToIPLog.seekg(2, clusterToIPLog.cur);
    }
    vector<string> ip_clusterID;

    while(getline(clusterToIPLog, s)){
        split(s,ip_clusterID,',');
        ip_to_cluster_->Put(ip_clusterID[0], ip_clusterID[1]);
        // std::cout << ip_clusterID[0] << " " << ip_clusterID[1] <<  std::endl;
    }

    clusterToIPLog.close();
}
```

实现过程是文件格式相关的，格式会在下文介绍，前半部分顺序读行分割填充 cluster_to_stats_，后半部分逆序读行填充 ip_to_cluster_

Query 的实现细节如下：

```
bool ConnectionStatsQuery::Query(string ip, vector<double> &stats){
    string clusterID;
    if (!ip_to_cluster_->Get(ip, clusterID)){
        return false;
    }
    // std::cout << clusterID << std::endl;
    if(!cluster_to_stats_->Get(clusterID, stats)){
        return false;
    }
    // for (long unsigned int i=0;i<stats.size();i++){
    //       std::cout << stats[i]<< " ";
    //     }
    //       cout << endl;
    return true;
}
```

两次 Cache 查询，如前文 7 所述，此处不再赘述。

LRUCache 为实现的最近最少使用策略的 cache，工具类，利用 unordered_map 实现 O(1) 时间复杂度，由于对于本文文意的论述帮助不大，且实现较为常见，此处不赘述

```
QuicSimpleDispatcher::QuicSimpleDispatcher(
    const QuicConfig& config,
    const QuicCryptoServerConfig* crypto_config,
    QuicVersionManager* version_manager,
    std::unique_ptr<QuicConnectionHelperInterface> helper,
    std::unique_ptr<QuicCryptoServerStream::Helper> session_helper,
    std::unique_ptr<QuicAlarmFactory> alarm_factory,
    QuicHttpResponseCache* response_cache)
    : QuicDispatcher(config,
                     crypto_config,
                     version_manager,
                     std::move(helper),
                     std::move(session_helper),
                     std::move(alarm_factory)),
      response_cache_(response_cache) {
        connection_stats_query_ = new ConnectionStatsQuery();

        connection_stats_query_->Initialize("./cluster-min-max-mean","./ip-cluster");
      }
```

ConnectionStatsQuery 随 QuicSimpleDispacher 初始化，如前文所述。

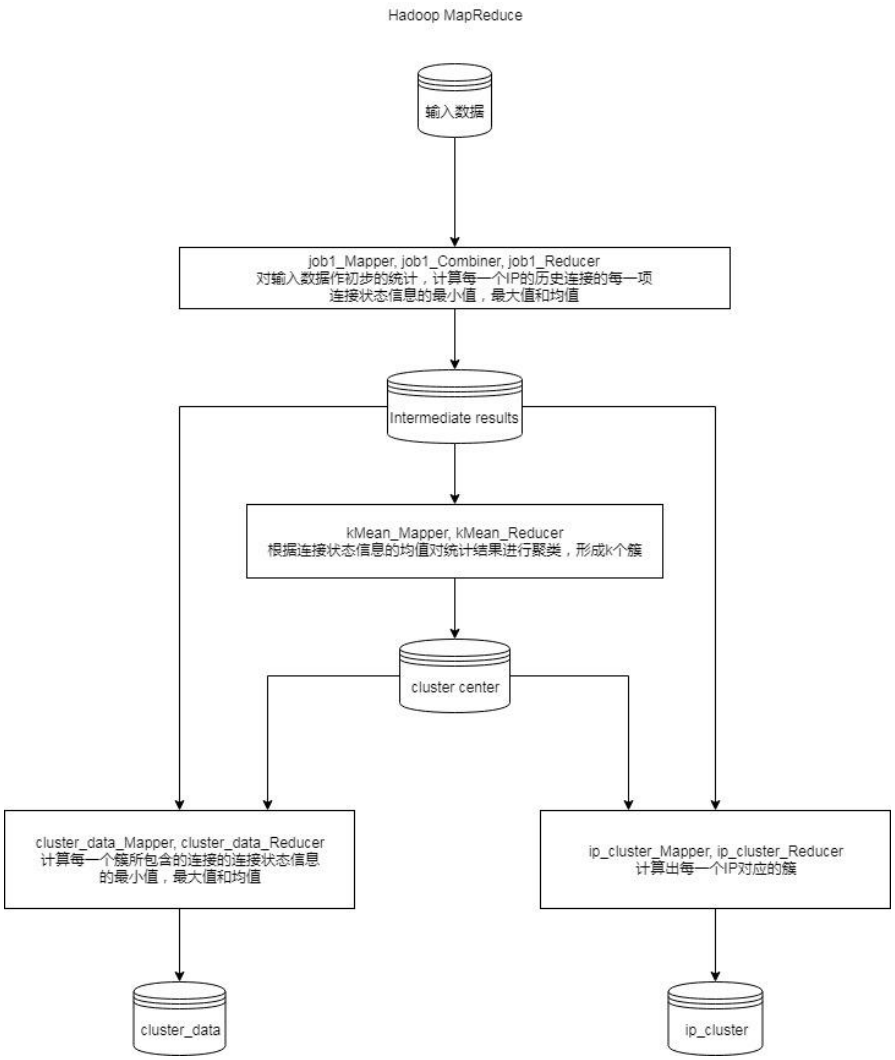
```
// The QuicServerSessionBase takes ownership of |connection| below.
QuicConnection* connection = new QuicConnection(
    connection_id, client_address, helper(), alarm_factory(),
    CreatePerConnectionWriter(),
    /* owns_writer= */ true, Perspective::IS_SERVER, GetSupportedVersions());
bool enableQuery = true;
if(enableQuery){
  string clientIp = client_address.host().ToString();
  vector<double> stats;
  // std::cout << clientIp << std::endl;
  if(connection_stats_query_->Query(clientIp,stats)){
    TcpCubicSenderBytes * sendAlgorithm = dynamic_cast<TcpCubicSenderBytes *> (
      connection->sent_packet_manager_not_const().GetSendAlgorithmNotConst());
    RttStats * rttstats = connection->sent_packet_manager_not_const().GetRttStatsNotConst();

    // sendAlgorithm->SetInitialCongestionWindowInBytesPublic(140*kDefaultTCPMSS);
    sendAlgorithm->SetInitialCongestionWindowFromBandwidthAndRttPublic(
     QuicBandwidth::FromBitsPerSecond(1000*stats[ESTIMATED_BANDWIDTH_MAX]),
       QuicTime::Delta::FromMicroseconds(stats[MIN_RTT_US_MAX]));
    sendAlgorithm->SetCongestionWindowFromBandwidthAndRttPublic(
     QuicBandwidth::FromBitsPerSecond(1000*stats[ESTIMATED_BANDWIDTH_MAX]),
       QuicTime::Delta::FromMicroseconds(stats[MIN_RTT_US_MEAN]));
    // sendAlgorithm->SetSlowstartthresholdFromBandwidthAndRttPublic(
    //  QuicBandwidth::FromBitsPerSecond(1000*stats[ESTIMATED_BANDWIDTH_MAX]),
    //    QuicTime::Delta::FromMicroseconds(stats[MIN_RTT_US_MAX]));
    sendAlgorithm->SetMinSlowstartExitWindowFromBandwidthAndRttPublic(
      QuicBandwidth::FromBitsPerSecond(1000*stats[ESTIMATED_BANDWIDTH_MEAN]),
        QuicTime::Delta::FromMicroseconds(stats[MIN_RTT_US_MEAN]));

    rttstats->set_initial_rtt_us(stats[MIN_RTT_US_MEAN]);
    rttstats->set_initial_srtt_us(stats[SRTT_US_MEAN]);
  }
}
QuicServerSessionBase* session = new QuicSimpleServerSession(
    config(), connection, this, session_helper(), crypto_config(),
    compressed_certs_cache(), response_cache_);
session->Initialize();
```

初始参数配置，如前文 7 所述，取得 sendAlgorithm 而后调用相应的方法进行初始参数配置，方法作用参数即为其字面意思，具体细节见相关代码。

下面介绍 map reduce 相关部分的实现细节，其整体框架如下：



Hadoop MapReduce 中主要包含了以下的类文件:

job1_Mapper, job1_Combiner, job1_Reducer, kMeans_Mapper, kMeans_Reducer, cluster_data_Mapper, cluster_data_Reducer, ip_cluster_Mapper, ip_cluster_Reducer, job1_Main.

下面分别对这些类文件进行详细介绍(详细的实现请参考对应的源代码，已包含注释)

job1_Mapper :

map 方法的输入的 key 为文本文件的行号，value 为文本行，其格式为(IP data_vector)，表示源地址为 IP 的一个连接，data_vector 保存了此连接的状态信息。

map 方法的输出的 key 为 IP，value 为(data_vector;data_vector;data_vector;1)，第一个 data_vector 用于统计每一项状态的最小值，第二个 data_vector 用于统计每一项状态的最大值，第三个 data_vector 和 1 用于统计每一项状态的均值。

job1_Combiner:

此类的 reducer 方法主要做一些中间合并，减少通信开销。其输入 key 值和输出 key 值为均为 IP，其输入 value 为(data_vector;data_vector;data_vector;1)，输出 value 为对输入的 value 初步统计的结果，格式为(data_vector;data_vector;data_vector;count)

job1_Reducer

此类的 reducer 方法的功能和 job1_Combiner 方法基本一致。其输入 key 值和输出 key 值为均为 IP，其输入 value 为(data_vector;data_vector;data_vector;count)，输出 value 为对输入的 value 初步统计的结果，格式为(data_vector;data_vector;data_vector)，为此 IP 的所有历史连接数据的统计结果，此时三个 data_vector 依次表示为每一项状态的最小值，最大值和均值。

kMean_Mapper

Setup 方法的主要功能是初始化聚类中心。第一次聚类操作时，会直接从数据中读取前 k 个数据项作为初始聚类中心。后面的聚类操作直接读取上一次聚类的结果作为聚类中心。

map 方法的输入的 key 为文本文件的行号，value 为文本行，其格式为(IP data_vector; data_vector; data_vector;)，表示源地址为 IP 的连接状态信息的最小值，最大值和均值。使用均值向量进行聚类，计算此 IP 属于哪一类。

map 方法的输出的 key 为 cluster_id, value 为 data_vector，为此 IP 的均值向量。

kMean_Reducer

此类的 reducer 的输入 key 为 cluster_id, value 为均值向量 data_vector.输出 key 为 cluster_id, value 为新的聚类中心，新的聚类中心由同一个 cluster_id 中的所有均值向量求和平均得到。

cluster_data_Mapper

Setup 方法的主要功能是读取 kMeans 得到的最终的簇中心。

map 方法的输入的 key 为文本文件的行号，value 为文本行，其格式为(IP data_vector; data_vector; data_vector;)，表示源地址为 IP 的连接状态信息的最小值，最大值和均值。根据均值向量计算出此 IP 属于哪一类。

map 方法的输出的 key 为 cluster_id, value 为(data_vector;data_vector;data_vector)。

cluster_data_Reducer

Reducer 方法的输入 key 为 cluster_id, value 为(data_vector;data_vector;data_vector)。使用输入的数据计算一个簇中所有 IP 的所有状态信息的最小值向量和最大值向量。

输出的 key 值为 cluster_id, value 为(data_vector;data_vector;data_vector)

ip_cluster_Mapper:

Setup 方法的主要功能是读取 kMeans 得到的最终的簇中心。

map 方法的输入的 key 为文本文件的行号，value 为文本行，其格式为(IP data_vector; data_vector; data_vector;)，表示源地址为 IP 的连接状态信息的最小值，最大值和均值。根据均值向量计算出此 IP 属于哪一类。

map 方法的输出的 key 为 IP, value 为 cluster_id

Reducer 方法的输入 key 为 IP, value 为 cluster_id。直接输出即可，表示此 IP 所属的 cluster

输出的 key 值为 IP, value 为 cluster_id

job1_Main

由于统计和计算的运行过程需要若干次 MapReducer 并且需要多个 job。故此类文件主要对上面的不同 MapReducer 进行封装。

Client_statistic(String input_path, String_path) : 对 job1_Mapper, job1_Combiner, job1_Reducer 组成的 job 进行封装。两个参数分别为数据输入路径，数据输出路径。

Calcucate_error(String oldcenter, String newcenter) : 计算旧的聚类中心和新的聚类中心之间的误差。两个参数分别为旧聚类中心的文件路径，新聚类中心的文件路径。

kmeans(String datapath, String newcenterpath, String k, String oldcenterpath) : 对 kMean_Mapper, kMean_Reducer 组成的 job 进行封装。四个参数分别为数据输入路径，存放新的聚类中心的文件聚类，k，旧的聚类中心的文件路径。

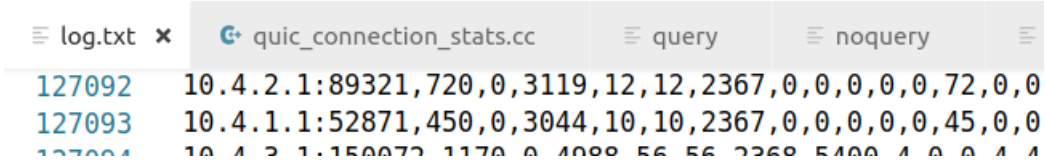
cluster_data(String input_path, String output_path, String centerpath) : 对 cluster_data_Mapper, cluster_data_Reducer 组成的 job 进行封装。三个参数分别为数据输入路径，数据输出路径，聚类中心路径。

ip_cluster(String input_path, String output_path, String centerpath): 对 ip_cluste_Mapper, ip_cluste_Reducer 组成的 job 进行封装。三个参数分别为数据输入路径，数据输出路径，聚类中心路径。

本次实验中还使用了 scala 实现一份可以在 Hadoop Spark 平台上运行，并且具有相同功能的程序，关于 scala 的实现详细见源代码中的 Lab6-spark.zip，已在代码文件中加上注释。

8. 输入文件数据和详细输入数据格式，输出结果文件数据片段和详细输出数据格式（必须清晰描述）

仿真运行，在正确安装了本次课程设计所需的运行环境，以及正确配置了相关环境变量之后，运行脚本自动执行，无需其他输入，脚本运行过程中产生的日志文件格式即相关含义如下(绿色的是编辑器的行号):



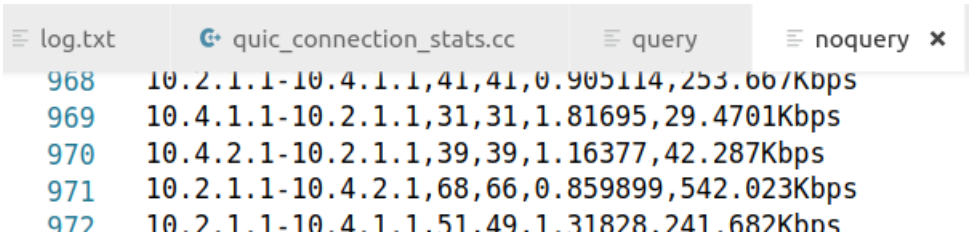
由于显示空间有限，输出格式为 IP:数据,数据,…(即只有第一个是冒号，其余均为逗号)

含义:

CLIENTIP:BYTES_SENT,PACKETS_SENT,STREAM_BYTES_SENT,PACKETS_DISCARDED,
BYTES_RECEIVED,PACKETS_RECEIVED,PACKETS_PROCESSED,STREAM_BYTES_RECEIVED,
BYTES_RETRANSMITTED,PACKETS_RETRANSMITTED,BYTES_SPURIOUSLY_RETRANSMITTED,
PACKETS_SPURIOUSLY_RETRANSMITTED,PACKETS_LOST,SLOWSTART_PACKETS_SENT,
SLOWSTART_PACKETS_LOST,SLOWSTART_BYTES_LOST,PACKETS_DROPPED,
CRYPTO_RETRANSMIT_COUNT,LOSS_TIMEOUT_COUNT,TLP_COUNT,RTO_COUNT,
MIN_RTT_US,SRTT_US,MAX_PACKET_SIZE,MAX_RECEIVED_PACKET_SIZE,
ESTIMATED_BANDWIDTH,PACKETS_REORDERED,MAX_SEQUENCE_REORDERING,
MAX_TIME_REORDERING_US,TCP_LOSS_EVENTS,CONNECTION_CREATION_TIME,

BLOCKED_FRAMES_RECEIVED,BLOCKED_FRAMES_SE

除此之外，脚本还会产生此次仿真所产生流(四元组 IP port IP port)的统计信息，输出样例：



含义：

SRCIP-DSTIP,PACKETSSENT,PACKETSRECV,DELAY,AVGBANDWIDTH

ConnectionStatsQuery 初始化时需要两个文件，一个是 clusterID 与 connectionStats 之前的对应数据，一个是 IP 与 clusterID 之间的对应关系，具体格式及样例见下文

Mapreduce 相关输入输出文件及数据介绍如下：

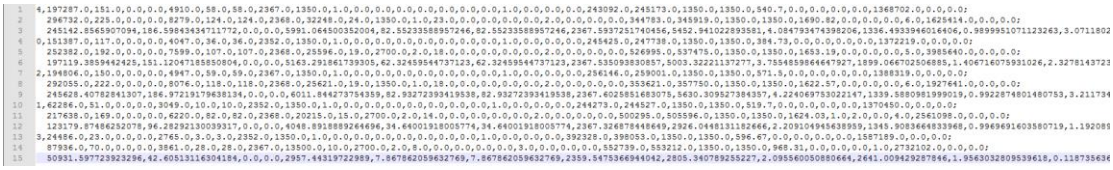
输入文件数据为每一个连接的连接状态信息构成的状态向量。

其输入数据格式即为 9 中 log.txt 格式。

输出结果的文件有两个：

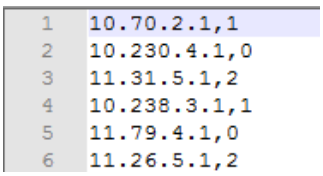
第一种文件的格式为 :(可用于对参数进行实时查询)

Cluster_id,状态向量的最小值;状态向量的最大值;状态向量的均值



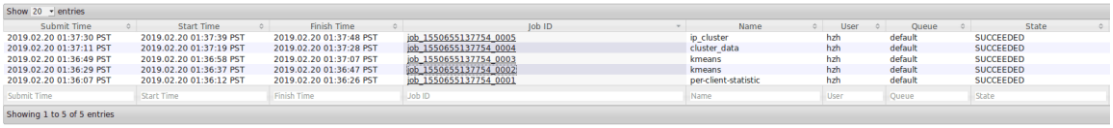
第二种文件的格式为 :(用于表示 IP 属于哪一个簇)

IP,cluster_id

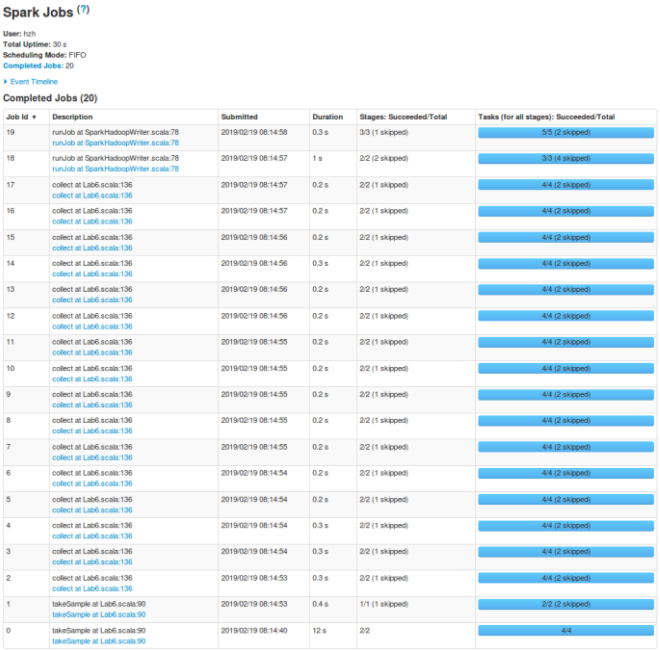


9. 程序运行实验结果说明和分析

Hadoop mapreduce 运行结果如下：



Spark mapreduce 运行结果如下：



Spark Jobs (?)

User: hzh
Total Uptime: 30 s
Scheduling Mode: FIFO
Completed Jobs: 20

▸ Event Timeline

Completed Jobs (20)

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
19	runJob at SparkHadoopWriter.scala:78 runJob at SparkHadoopWriter.scala:78	2019/02/19 08:14:58	0.3 s	3/3 (1 skipped)	5/5 (2 skipped)
18	runJob at SparkHadoopWriter.scala:78 runJob at SparkHadoopWriter.scala:78	2019/02/19 08:14:57	1 s	2/2 (2 skipped)	3/3 (4 skipped)
17	collect at Lab6.scala:136 collect at Lab6.scala:136	2019/02/19 08:14:57	0.2 s	2/2 (1 skipped)	4/4 (2 skipped)
16	collect at Lab6.scala:136 collect at Lab6.scala:136	2019/02/19 08:14:57	0.2 s	2/2 (1 skipped)	4/4 (2 skipped)
15	collect at Lab6.scala:136 collect at Lab6.scala:136	2019/02/19 08:14:56	0.2 s	2/2 (1 skipped)	4/4 (2 skipped)
14	collect at Lab6.scala:136 collect at Lab6.scala:136	2019/02/19 08:14:56	0.3 s	2/2 (1 skipped)	4/4 (2 skipped)
13	collect at Lab6.scala:136 collect at Lab6.scala:136	2019/02/19 08:14:56	0.2 s	2/2 (1 skipped)	4/4 (2 skipped)
12	collect at Lab6.scala:136 collect at Lab6.scala:136	2019/02/19 08:14:56	0.2 s	2/2 (1 skipped)	4/4 (2 skipped)
11	collect at Lab6.scala:136 collect at Lab6.scala:136	2019/02/19 08:14:55	0.2 s	2/2 (1 skipped)	4/4 (2 skipped)
10	collect at Lab6.scala:136 collect at Lab6.scala:136	2019/02/19 08:14:55	0.2 s	2/2 (1 skipped)	4/4 (2 skipped)
9	collect at Lab6.scala:136 collect at Lab6.scala:136	2019/02/19 08:14:55	0.2 s	2/2 (1 skipped)	4/4 (2 skipped)
8	collect at Lab6.scala:136 collect at Lab6.scala:136	2019/02/19 08:14:55	0.2 s	2/2 (1 skipped)	4/4 (2 skipped)
7	collect at Lab6.scala:136 collect at Lab6.scala:136	2019/02/19 08:14:55	0.2 s	2/2 (1 skipped)	4/4 (2 skipped)
6	collect at Lab6.scala:136 collect at Lab6.scala:136	2019/02/19 08:14:54	0.2 s	2/2 (1 skipped)	4/4 (2 skipped)
5	collect at Lab6.scala:136 collect at Lab6.scala:136	2019/02/19 08:14:54	0.2 s	2/2 (1 skipped)	4/4 (2 skipped)
4	collect at Lab6.scala:136 collect at Lab6.scala:136	2019/02/19 08:14:54	0.3 s	2/2 (1 skipped)	4/4 (2 skipped)
3	collect at Lab6.scala:136 collect at Lab6.scala:136	2019/02/19 08:14:54	0.3 s	2/2 (1 skipped)	4/4 (2 skipped)
2	collect at Lab6.scala:136 collect at Lab6.scala:136	2019/02/19 08:14:53	0.3 s	2/2 (1 skipped)	4/4 (2 skipped)
1	takeSample at Lab6.scala:90 takeSample at Lab6.scala:90	2019/02/19 08:14:53	0.4 s	1/1 (1 skipped)	2/2 (2 skipped)
0	takeSample at Lab6.scala:90 takeSample at Lab6.scala:90	2019/02/19 08:14:40	12 s	2/2	4/4

仿真运行(实际使用脚本运行，此处作为展示)

```
ross@ubuntu:~/ns-allinone-3.27/ns-3.27$ ./waf --run quic-mapreduce
Waf: Entering directory `/home/ross/ns-allinone-3.27/ns-3.27/build'
Waf: Leaving directory `/home/ross/ns-allinone-3.27/ns-3.27/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (7.157s)
10.4.1.1-10.2.1.1,31,31,1.49497,30.3588Kbps
10.4.2.1-10.2.1.1,44,44,1.55187,34.9811Kbps
10.4.3.1-10.2.1.1,71,71,1.80281,38.5927Kbps
10.2.1.1-10.4.3.1,122,116,1.59159,622.8Kbps
10.2.1.1-10.4.2.1.78.75.1.52602.393.349Kbps
```

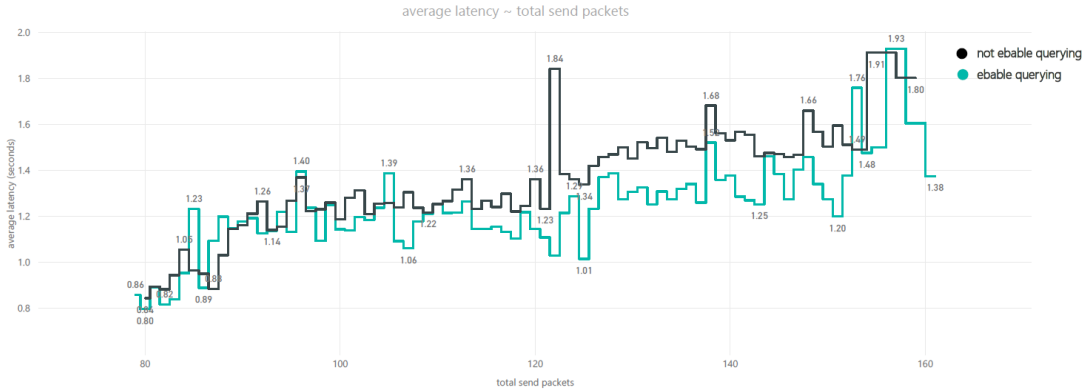
仿真产生 trace(pcap)文件：

No.	Time	Source	Destination	Protocol	Le
1	0.000000	10.4.5.1	10.2.1.1	GQUIC	
2	0.007338	10.4.3.1	10.2.1.1	GQUIC	
3	0.010775	10.4.2.1	10.2.1.1	GQUIC	

```
▸ Frame 1: 1380 bytes on wire (11040 bits), 1380 bytes captured (11040 bit
▸ Point-to-Point Protocol
▸ Internet Protocol Version 4, Src: 10.4.5.1, Dst: 10.2.1.1
▸ User Datagram Protocol, Src Port: 49153, Dst Port: 6121
▾ GQUIC (Google Quick UDP Internet Connections)
  ▸ Public Flags: 0x0d
    CID: 12950477315985208484
    Version: Q041
    Packet Number: 1
    Payload: d916f12a4735ae814f72e9b3c105140143484c4f0d000000...
```

实验对比结果：

我们在 1000 次请求里面针对某一客户端在启用我们的策略前后的 Latency(发起请求->响应结束)做了对比，横坐标是服务端总共发送数据包的数目，纵坐标是相对于某个数据包数目的 latency 的平均值，实验对比如下(绿色的为启用我们策略之后的结果)：



实验结果发现，启用我们策略之后，latency 下降有 20%左右，效果显著。所以通过对 quic server 端日志数据的分析，如果能够准确判断新的连接请求的拥塞控制层面的一些初始参数，并加以配置，对于减少 lantency，提交用户体验具有显著帮助。

10. 总结：特点总结，功能、性能、扩展性等方面存在的不足和可能的改进之处

关于 mapreduce：

从上面的结果可以看出，Hadoop MapReduce 需要启动多个 job 来完成统计和计算，而 Spark 则将自动划分 jobs。并且 Spark 无需通过 hdfs 交换数据，其运行时间更短。若后面要实现实时更新簇中心并且对用户的每一个连接请求进行参数的实时查询，使用 Spark 更可能满足实时性。

关于本系统：

作为以实验为目的的系统，该系统已经基本满足了实验环境的要求，对于最终实验结果的不及预期，我们需要后续的验证实验，来检验 10 中可能原因的正确性。

11. 参考文献

1. https://hpbn.co/?utm_source=igvita&utm_medium=referral&utm_campaign=igvita-homepage
2. https://www.chromium.org/quic
3. https://gitlab.com/diegoamc/ns-3-quic-module