ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Департамент программной инженерии

СОГЛАСОВАНО
Преподаватель факультета компьютерных наук, доцент базовой кафедры «Системное программирование» ИСП ран, канд. техн. наук

_____ А. И. Гетьман
«___» _____2022 г.

УТВЕРЖДАЮ
Академический руководитель образовательной программы «Программная инженерия», канд. техн .наук

_____ В. В. Шилов
«___» _____ 2022 г.

СОГЛАСОВАНО
Руководитель департамента программной инженерии, доцент факультета компьютерных наук, канд. экон. наук

_____ С. А. Лебедев
«___» _____2022 г.

Подп. и дата	
Инв. № дубл.	
Взам. инв. №	
Подп. и дата	
Инв. № подл	

ПРОГРАММНЫЙ МОДУЛЬ ДЛЯ ВЫДЕЛЕНИЯ ЛОГИЧЕСКИ СВЯЗАННЫХ ПОТОКОВ В ВЫСОКОСКОРОСТНОМ СЕТЕВОМ ТРАФИКЕ

Текст программы

ЛИСТ УТВЕРЖДЕНИЯ

RU.17701729.05.03-01 01-1-ЛУ

Исполнитель:
студент группы БПИ197
_____ / Глущенко З. С./
«____»_____ 2022 г.

Москва 2022

УТВЕРЖДЕНО
RU.17701729.05.03-01 81 01-1-ЛУ

ПРОГРАММНЫЙ МОДУЛЬ ДЛЯ ВЫДЕЛЕНИЯ ЛОГИЧЕСКИ СВЯЗАННЫХ ПОТОКОВ В ВЫСОКОСКОРОСТНОМ СЕТЕВОМ ТРАФИКЕ

Текст программы

RU.17701729.05.03-01 81 01-1

Листов 20

Подп. и дата	
Инв. № дубл.	
Взам. инв. №	
Подп. и дата	
Инв. № подл	

Москва 2022

СОДЕРЖАНИЕ

1. Назначение программы

1.1. Директория ftp_extractor

1.1.1. Код файла ftp_ ftp_extractor.h

```
#ifndef FTP_EXTRACTOR_H
#define FTP_EXTRACTOR_H

#include "common/ProcessModule.h"

namespace Pr {

/**
 * @brief The FTP extractor is a class for extracting information from packets
 * and sending it to the appropriate modules.
 */
class ftp_extractor : public ProcessModule, public MessageProcessor<ftp_extractor> {

public:
    ftp_extractor();
    virtual ~ftp_extractor();
    /**
     * @brief declareResourcesUsage - function to declare queues, statistics and other module
resources
     * @return true if all declarations are successful
     */
    virtual bool declareResourcesUsage();
    /**
     * @brief acquireResources - function to get parameters from config and connect queues
     * @return true in case of success
     */
    virtual bool acquireResources();

protected:
    /**
     * @brief processMessage - function to process a single message from the input queue
     * @param msg - pointer to the message in the queue of general type QueueMessage
     * @param rec - pointer to the PacketRecord in the packet buffer
     * @return zero in the end of the message processing (here we chose not to return any error
codes)
     */
    int processMessage(QueueMessage* msg, PacketRecord* rec);
```

```
    /**
     * @brief sendSignal - function to send a signal to the output queue; will send until the success
of the operation
     * @param output_queue - pointer to the output queue of type T
     * @param signal - pointer to signal to send of type T
     * @return
     */
    template<typename T>
    void sendSignal(MessageQueue<T>* output_queue, T signal);

    /**
     * @brief parseFTP - function to extract an address from control connection packet
     * @param packMess - pointer to the message in the queue of type PacketMessage
     * @param rec - pointer to the PacketRecord in the packet buffer
     * @return pairSignal - reference to the pair signal variable
     * @return shift - offset depending on connection mode
     */
    void parseFTP(PacketMessage* packMess, PacketRecord* rec, PairStreamsSignalV4&
pairSignal, int shift);


    // queues
    /**
     * @brief m_packets - input packets queue
     */
    MessageQueue<StreamPacket>* m_packets;
    /**
     * @brief m_input_queue - The queue of output pair signals
     */
    MessageQueue<PairStreamsSignalV4>* m_output_merger;
    /**
     * @brief m_input_queue - The queue of output pair signals
     */
    MessageQueue<PairStreamsSignalV4>* m_output_converter;
    /**
     * @brief m_input_queue - The queue of output SelectSignalV4 signals
     */
    MessageQueue<SelectSignalV4>* m_output_selector;

    // module statistics;
    /**
     * @brief m_lostPackets - counter of lost packets
     */
    uint64_t *m_lostPackets;

    /**
```

```
    * @brief m_mutex - mutex to work with inout packets
    */
   boost::mutex m_mutex;

};
}

#endif
```

1.1.2. Код файла ftp_extractor.cpp

```
#include "ftp_extractor.h"

// config parameters
// parameter with '?' is optional
const char * spec = R"SPEC(
{
   "packetQueue" : "string",
   "packetBuffer" : "string",
   "pairSignals_merger" : "string",
   "SelectSignals_selector" : "string",
   "pairSignals_converter" : "string"
}
)SPEC";

Pr::ftp_extractor::ftp_extractor() : ProcessModule("ftp_extractor", spec),
   MessageProcessor<ftp_extractor>(const_cast<ftp_extractor*>(this)) {
   m_packets = nullptr;
   m_output_merger = nullptr;
   m_output_converter = nullptr;
   m_output_selector = nullptr;
}

Pr::ftp_extractor::~ftp_extractor() {
   // flush the output queue
   m_output_merger->flush();
   m_output_converter->flush();
   m_output_selector->flush();
   m_packets = nullptr;
   m_output_merger = nullptr;
   m_output_converter = nullptr;
   m_output_selector = nullptr;
}

// here we create queues and statistical parameters to watch
bool Pr::ftp_extractor::declareResourcesUsage() {
```

```
    // here we declare the input queue of type PacketMessage
    // names of the queue and the packet buffer are derived from the config file
    // it also automatically starts counting some statistics for this queue
    declareQueueUsage(m_config.value<std::string>("packetQueue"),
                      {"PacketMessage", true, false, false, -1,
                       "Packets received", "Packets lost",
                       "Avg. processing time", "Max throughput",
                       (MessageQueueInterface**)&m_packets,
                       m_config.value<std::string>("packetBuffer"), false});

    // declare queues
    declareQueueUsage(m_config.value<std::string>("pairSignals_merger"),
"PairStreamsSignalV4", false, true, (MessageQueueInterface**)&m_output_merger);
    declareQueueUsage(m_config.value<std::string>("SelectSignals_selector"), "SelectSignalV4",
false, true, (MessageQueueInterface**)&m_output_selector);
    declareQueueUsage(m_config.value<std::string>("pairSignals_converter"),
"PairStreamsSignalV4", false, true, (MessageQueueInterface**)&m_output_converter);

    return true;
}

bool Pr::ftp_extractor::acquireResources()
{
    m_lostPackets = acquirePointerToStatParam("Packets lost");

    // set input queue listener
    if(!setQueueListener(m_packets, this)) {
        return false;
    }

    return true;
}

// this is the function that is called each time we receive a new packet in the input queue
// it is the main processing function
int Pr::ftp_extractor::processMessage(QueueMessage* msg, PacketRecord* rec)
{
    PacketMessage* mess = (PacketMessage*)msg;

    // we use the mutex to separate the processing of each packet
    boost::lock_guard<boost::mutex> guard(m_mutex);

    IpHeader* header = (IpHeader*)rec->payload();
    if (header->proto != TCP)
    {
        return 0;
    }
```

```
    TcpHeader* tcp = (TcpHeader*)(header->payload());
    int ftp_payload = header->headerLength() + tcp->headerLength();
    if (ftp_payload + 4 < rec->dataSize())
    {
        // interpret first 4 ftp_payload's chars as hex digit
        const int32_t& hx= *reinterpret_cast<int32_t*>(rec->payload() + ftp_payload);
        const int32_t hex_port = 0x54524f50;
        const int32_t hex_pasv = 0x20373232;
        const int port_shift = ftp_payload + 5;
        const int pasv_shift = ftp_payload + 27;

        PairStreamsSignalV4 pairSignal;

        if(hx == hex_port)
        {
            Logger::log(Logger::Info, "Identify PORT");
            parseFTP(mess, rec, pairSignal, port_shift);
            pairSignal.childAddr.srcPort = htons(20);

            SelectSignalV4 sv4;
            sv4.addr = pairSignal.childAddr;
            sv4.flags = pairSignal.flags;

            sendSignal(m_output_merger, pairSignal);
            sendSignal(m_output_selector, sv4);
        }
        else if (hx == hex_pasv)
        {
            Logger::log(Logger::Info, "Identify PASV 227");
            parseFTP(mess, rec, pairSignal, pasv_shift);
            sendSignal(m_output_converter, pairSignal);
        }
        else
        {
            Logger::log(Logger::Error, "recieved wrong packet!");
            (*m_lostPackets)++;
        }
    }
    m_output_merger->flush();
    m_output_converter->flush();
    m_output_selector->flush();
    return 0;
}

template<typename T>
void Pr::ftp_extractor::sendSignal(MessageQueue<T>* output_queue, T signal)
{
```

```
    // try sending until we reach the success status
    while(output_queue->send(signal) != Infra::MessageQueueInterface::Sent) {}
    output_queue->flush();
}

void Pr::ftp_extractor::parseFTP(PacketMessage* packMess, PacketRecord* rec,
PairStreamsSignalV4& pairSignal, int shift)
{
    IpHeader* header = (IpHeader*)rec->payload();

    int32_t tmpNum = 0;
    int32_t ip = 0;
    int16_t port = 0;
    int curIteration = 0; // 4 - Ip bytes, 6 - port bytes
//    bool error = true;
    for (__uint32_t i = shift; i < rec->dataSize(); ++i)
    {
        char curSymbol = *(rec->payload() + i);
        // если это цифра
        if(curSymbol >= '0' && curSymbol <= '9')
        {
            tmpNum = tmpNum * 10 + (curSymbol - '0');
        }
        else
        {
            // сдвигаем на 8 бит исходное и выгружаем tmpNum
            ++curIteration;
            if(curIteration <= 4)
            {
                ip = (ip << 8) + tmpNum;
                tmpNum=0;
            }
            else if(curIteration <= 6)
            {
                port = (port<< 8) + tmpNum;
                tmpNum=0;
            }
            else
            {
                // если дошли до сюда, значит удачно считали ip & port
//                error = false;
                break;
            }
        }
    }

    pairSignal.creationTime = rec->timestamp();
```

```
    pairSignal.flags = 1u;
    // parent - info from recieved packet
    //pairSignal.parentId = mess->streamId;
    rec->toAddrInfo(pairSignal.parentAddr);

    // child - parsed info
    pairSignal.childAddr.proto = TCP;
    pairSignal.childAddr.destIp = ip;
    pairSignal.childAddr.destPort = port;
    pairSignal.childAddr.srcIp = header->destIp;
    //pairSignal.childAddr.srcPort = 0;

    pairSignal.childAddr.destIp = htonl(pairSignal.childAddr.destIp);
    pairSignal.childAddr.destPort = htons(pairSignal.childAddr.destPort);

    // Это packet_filter -> нужно подать сигналы на отбор
    // This is control connection!
    SelectSignalV4 selector;
    selector.addr = pairSignal.parentAddr;
    selector.creationTime = pairSignal.creationTime;
    selector.flags = pairSignal.flags;
    sendSignal(m_output_selector, selector);
}
```

1.1.3. Код файла main.cpp

```
#include "ftp_extractor.h"

int main(int argc, char* argv[])
{
    return Pr::ftp_extractor().run(argc, argv);
}
```

1.2. Директория stream_merger

1.2.1. Код файла StreamMerger.h

```
#ifndef STREAMMERGER_H
#define STREAMMERGER_H

#include <pcap.h>
#include <unordered_map>

#include <common/ProcessModule.h>
#include <infra/includes/OldestFirstPool.h>
#include <common/StreamTypes.h>
```

```
namespace Pr
{

   /**
    * @brief The StreamMerger is a class for merging packets of several connected flows into one
pcap file
    */
   class StreamMerger : public ProcessModule, public MessageProcessor<Pr::StreamMerger>
   {

      struct PairStreamNode
      {
         StreamDescrV4 parent;
         uint32_t childCount;
         uint32_t recievedChildCount = 0;
         bool isVisited = false;
         Dequeue children;
      };

      struct StreamNode : StreamDescrV4, DequeueNode
      {
         PairStreamNode* parent;
      };


   public:
      /**
       * @brief PacketManager - simple constructor
       */
      StreamMerger();

      ~StreamMerger();

      virtual bool declareResourcesUsage();

      virtual bool acquireResources();

   private:
      void removeStream(StreamDescrV4* stream);

      void mergeStreams(PairStreamNode* parent);

      int processMessage(QueueMessage* msg, PacketRecord* rec);

      int processSignal(QueueMessage* msg, PacketRecord* rec);
```

```
void startMerge(PairStreamNode* parent);

// resources
/**
 * @brief m_addr2parent - map from stream addrs to parent streams
 */
std::unordered_map<AddrInfoV4, PairStreamNode*> m_addr2parent;
/**
 * @brief m_childStreams - pool for child streams
 */
OldestFirstPool<StreamNode> m_childStreams;
/**
 * @brief m_parentStreams - pool for parent streams
 */
OldestFirstPool<PairStreamNode> m_parentStreams;
/**
 * @brief m_streamIn - input queue for stream packets
 */
MessageQueue<SelectSignalV4>* m_streamIn;
/**
 * @brief m_streamIn - input queue for stream packets
 */
MessageQueue<PairStreamsSignalV4>* m_pairsIn;

/**
 * @brief m_path - path for pcap files
 */
std::string m_pathFrom;
/**
 * @brief m_path - path for pcap files
 */
std::string m_pathTo;

// module statistics
/**
 * @brief m_maxStreamsCount - max streams count
 */
uint32_t m_maxStreamsCount;
/**
 * @brief m_lostPackets - pair merged count
 */
uint64_t* m_pairsMerged;
/**
 * @brief m_notPairedStreams - streams with no pair
 */
uint64_t* m_notPairedStreams;
/**
```

```
        * @brief m_maxChilds - maximum children count with one parent
        */
       uint64_t* m_maxChilds;
       /**
        * @brief m_lostParents - lost parent streams count
        */
       uint64_t* m_lostParents;
       /**
        * @brief m_lostChildren - lost child streams count
        */
       uint64_t* m_lostChildren;
       /**
        * @brief m_mergeResults - merge results count
        */
       uint64_t* m_mergeResults;
       /**
        * @brief m_mutex - mutex for change paired streams
        */
       boost::mutex m_mutex;
   };
}


#endif //STREAMMERGER_H
```

1.2.2. Код файла StreamMerger.cpp

```
#include "StreamMerger.h"

#include <cstdint>
#include <cstdio>
#include <boost/format.hpp>


const char * spec = R"SPEC(
{
   "maxStreamsCount" : "int",
   "inputPath" : "string",
   "outputPath" : "string",
   "streamsIn" : "string",
   "pairSignalsIn" : "string"
}
)SPEC";
```

```
Pr::StreamMerger::StreamMerger() : ProcessModule("StreamMerger", spec),
   MessageProcessor<StreamMerger>(const_cast<StreamMerger*>(this))
{
}

Pr::StreamMerger::~StreamMerger()
{
   m_childStreams.deinit();
   m_parentStreams.deinit();
}

bool Pr::StreamMerger::declareResourcesUsage()
{

   m_maxStreamsCount = (uint32_t)m_config.value<int>("maxStreamsCount");
//   m_pathFrom = dataPath(m_config.value<std::string>("inputPath"));
//   m_pathTo = outputFile(m_config.value<std::string>("outputPath"));

   m_pathFrom = m_config.value<std::string>("inputPath");
   m_pathTo = m_config.value<std::string>("outputPath");

   declareStatParam("Pairs merged", &m_pairsMerged);
   declareStatParam("Not paired streams", &m_notPairedStreams);
   declareStatParam("Maximum children", &m_maxChilds);
   declareStatParam("Lost parents", &m_lostParents);
   declareStatParam("Lost children", &m_lostChildren);
   declareStatParam("Merge results", &m_mergeResults);

   declareQueueUsage(m_config.value<std::string>("streamsIn"), {"SelectSignalV4", true, false,
false, -1, "Streams received", "", "", "", (MessageQueueInterface**)&m_streamIn});
   declareQueueUsage(m_config.value<std::string>("pairSignalsIn"), {"PairStreamsSignalV4",
true, false, false, -1, "Signals received", "", "", "", (MessageQueueInterface**)&m_pairsIn});

   return true;
}

bool Pr::StreamMerger::acquireResources()
{
   if(!m_parentStreams.init(m_maxStreamsCount))
   {
      std::string err= boost::str(boost::format("Can't create pool for parent streams with size
%1%") % m_maxStreamsCount);
      Infra::Logger::log(Logger::Error, err);
```

```
        return false;
    }
    if(!m_childStreams.init(m_maxStreamsCount))
    {
        std::string err= boost::str(boost::format("Can't create pool for child streams with size
%1%") % m_maxStreamsCount);
        Infra::Logger::log(Logger::Error, err);
        return false;
    }

    if(!setQueueListener(m_streamIn, this))
    {
        return false;
    }
    if(!setQueueListener(m_pairsIn, this, function(&Pr::StreamMerger::processSignal)))
    {
        return false;
    }
    return true;
}

void Pr::StreamMerger::removeStream(StreamDescrV4* stream)
{
    std::unordered_map<AddrInfoV4, PairStreamNode*>::iterator iter;
    iter =  m_addr2parent.find(stream->addr);
    if(iter != m_addr2parent.end())
    {
        m_addr2parent.erase(iter);
    }

    std::string fileName = boost::str(boost::format("%1%%2%") % m_pathFrom % stream-
>toString());
    if(remove(fileName.c_str()) != 0)
    {
        std::string err= boost::str(boost::format("Can't remove stream file %1%") % fileName);
        Infra::Logger::log(Logger::Error, err);
    }
}

void Pr::StreamMerger::mergeStreams(PairStreamNode* parent)
{
    DequeueNode* cur = parent->children.front();
    if(cur != 0)
```

```
    {
        std::string parentFile = parent->parent.toString();
        std::string command = boost::str(boost::format("mergecap -w %1%%2% %3%%4%") %
m_pathTo % parentFile % m_pathFrom % parentFile);
        int count = 0;
        while(cur != 0)
        {
            StreamNode* node = (StreamNode*)cur;
            command += boost::str(boost::format(" %1%%2%") % m_pathFrom % node-
>toString());
            count++;
            cur = cur->next;
        }
        std::string info = boost::str(boost::format("Merging (%1%) streams with parent (%2%)") %
count % parentFile);
        Infra::Logger::log(Logger::Warning, info);
        if(system(command.c_str()) == 0)
        {
            (*m_pairsMerged) += count;
            (*m_mergeResults)++;
        }
        else
        {
            (*m_lostParents)++;
            (*m_lostChildren) += count;
        }
        info = boost::str(boost::format("Merged"));
        Infra::Logger::log(Logger::Warning, info);
    }
}

int Pr::StreamMerger::processSignal(QueueMessage* msg, PacketRecord* rec)
{
    PairStreamsSignalV4* mess = static_cast<PairStreamsSignalV4*>( msg);
    boost::lock_guard<boost::mutex> guard(m_mutex);

    std::unordered_map<AddrInfoV4, PairStreamNode*>::iterator iter;
    iter = m_addr2parent.find(mess->parentAddr);
    if(iter == m_addr2parent.end())
    {
        PairStreamNode* parent = m_parentStreams.get();
        if(!m_parentStreams.isEmpty(parent))
        {
```

```
            (*m_lostParents)++;
            iter = m_addr2parent.find(parent->parent.addr);
            if(iter != m_addr2parent.end())
            {
               m_addr2parent.erase(iter);
            }
            DequeueNode* cur = parent->children.front();
            while(cur != 0)
            {
               StreamNode* node = (StreamNode*)cur;
               removeStream(node);
               m_childStreams.free(node);
               (*m_lostChildren)++;
               cur = cur->next;
            }
         }
         parent->childCount = 0;
         parent->children.clear();
         parent->parent.addr = mess->parentAddr;
         parent->parent.creationTime = 0;
         iter = m_addr2parent.insert(std::pair<AddrInfoV4, PairStreamNode*>(mess->parentAddr,
parent)).first;
      }
      // Now parent exist
      PairStreamNode* parent = iter->second;
      parent->childCount++;
      if(parent->childCount > (*m_maxChilds))
      {
         (*m_maxChilds) = parent->childCount;
      }
      std::string info = boost::str(boost::format("Pair (%1%) + (%2%)") % mess-
>childAddr.toString() % mess->parentAddr.toString());
      Infra::Logger::log(Logger::Warning, info);
      m_addr2parent.insert(std::pair<AddrInfoV4, PairStreamNode*>(mess->childAddr, parent));

      return 0;
   }

   int Pr::StreamMerger::processMessage(QueueMessage* msg, PacketRecord* rec)
   {
      boost::lock_guard<boost::mutex> guard(m_mutex);
      SelectSignalV4* mess = static_cast<SelectSignalV4*>(msg);
```

```
    std::unordered_map<AddrInfoV4, PairStreamNode*>::iterator iter;
    iter = m_addr2parent.find(mess->addr);
    if(iter == m_addr2parent.end())
    {
        (*m_notPairedStreams)++;
        //std::string info = boost::str(boost::format("Received unpaired stream (%1%)") % mess-
>addr.toString());
        //Infra::Logger::log(Logger::Warning, info);
        StreamDescrV4 stream;
        stream.addr = mess->addr;
        stream.creationTime = mess->creationTime;
        removeStream(&stream);
        return 0;
    }
    PairStreamNode* parent = iter->second;
    //Received parent
    if(parent->parent.addr == mess->addr)
    {
        parent->parent.creationTime = mess->creationTime;
        if (parent->recievedChildCount == parent->childCount)
        {
            startMerge(parent);
        }
        else
        {
            parent->isVisited = true;
        }
    }
    // Recieve child
    else
    {
        StreamNode* child = m_childStreams.get();
        if(!m_childStreams.isEmpty(child))
        {
            removeStream(child);
            child->parent->children.stichQueue(child);
            (*m_lostChildren)++;
        }
        child->addr = mess->addr;
        child->creationTime = mess->creationTime;
        parent->recievedChildCount++;
        parent->children.pushBack(child);
        if((parent->recievedChildCount == parent->childCount) && parent->isVisited)
```

```
        {
            startMerge(parent);
        }
    }
    return 0;
}

void Pr::StreamMerger::startMerge(PairStreamNode* parent)
{
    mergeStreams(parent);
    // delete children
    DequeueNode* cur = parent->children.front();
    while(cur != 0)
    {
        StreamNode* node = (StreamNode*)cur;
        removeStream(node);
        m_childStreams.free(node);
        cur = cur->next;
    }
    // delete parent
    removeStream(&(parent->parent));
    m_parentStreams.free(parent);
}
```

1.2.3. Код файла main.cpp

```
#include "StreamMerger.h"
int main(int argc, char* argv[])
{
    Pr::StreamMerger().run(argc, argv);
}
```

ЛИСТ РЕГИСТРАЦИИ ИЗМЕНЕНИЙ

Лист регистрации изменений									
Номера листов (страниц)					Всего листов (страниц в докум.)	№ документа	Входящий № сопроводительного докум. и дата	Подп.	Дата
Изм.	Измененных	Замененных	Новых	Аннулированх					