**广州大学学生实验报告**

开课学院及实验室：计算机科学与工程实验室电子楼518室 2020年\*\*月\*\*日

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 学院 | 计算机科学与网络工程学院 | | 年级、专业、班 | 软件211 | 姓名 | 张景致 | 学号 | 32106300004 |
| 实验课程名称 | | 计算机网络实验 | | | | | 成绩 |  |
| 实验项目名称 | | 网络程序设计 | | | | | 指导老师 | 綦科 |

## 实验目的

通过程序模拟网桥的工作原理以及检验和的计算，或者编写数据包的监听与分析程序，使学生加深对网络知识的理解。

## 实验环境

操作系统Arch Linux和Windows 11、以太网；

## 实验内容

1. **写一个程序来模拟网桥功能。**

模拟实现网桥的转发功能，以从文件中读取帧模拟网桥从网络中收到一帧，即从两个文件中读入一系列帧，从第一个文件中读入一帧然后从第二个文件中再读入一帧，如此下去。对每一帧，显示网桥是否会转发，及显示转发表内容。

**要求：**Windows或Linux环境下运行，程序应在单机上运行。

**分析**：用程序模拟网桥功能，可以假定用两个文件分别代表两个网段上的网络帧数据。而两个文件中的数据应具有帧的特征，即有目的地址，源地址和帧内数据。程序交替读入帧的数据，就相当于网桥从网段中得到帧数据。

对于网桥来说，能否转发帧在于把接收到的帧与网桥中的转发表相比较。判断目的地址后才决定是否转发。由此可见转发的关键在于构造转发表。这里转发表可通过动态生成。

1. **编写一个计算机程序用来计算一个文件的16位效验和。最快速的方法是用一个32位的整数来存放这个和。记住要处理进位（例如，超过16位的那些位），把它们加到效验和中。**

**要求**：1）以命令行形式运行：check\_sum infile

其中check\_sum为程序名，infile为输入数据文件名。

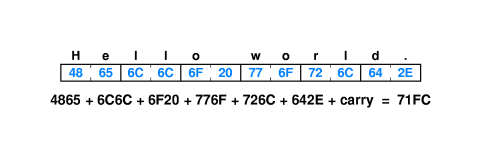
2）输出：数据文件的效验和

附：效验和（checksum)

**参见RFC1071 - Computing the Internet checksum**

* 原理：把要发送的数据看成16比特的二进制整数序列，并计算他们的和。若数据字节长度为奇数，则在数据尾部补一个字节的0以凑成偶数。
* 例子：16位效验和计算，下图表明一个小的字符串的16位效验和的计算。

为了计算效验和，发送计算机把每对字符当成16位整数处理并计算效验和。如果效验和大于16位，那么把进位一起加到最后的效验和中。



## 设计思路

### 实验5.1

#### 基本思想

网桥接受来自不同网段的帧，首先使用源地址和网段更新储存地址-网段映射关系的转发表，然后在表中查询目标地址。若查询得到网段，则进行转发。若没有，则检查等待队列中是否存在该地址，若没有则广播该帧，而后将其添加到等待队列。若存在网段接受该地址，则更新转发表，将等待队列中所有该地址的帧进行转发，否则执行丢弃。

#### 实现要点

1. 随机生成大量地址、网段和帧
2. 模拟网桥运行环境（硬件、系统软件的行为）的逻辑
3. 模拟网桥主要软件运行的逻辑
4. 在比较大（1000万帧及以上）的规模，比较短（30秒以内）的时间进行仿真。
5. 记录网桥运行的各种数据
6. 分析记录数据，给出便于观察的统计结果

#### 技术规划

编程语言：Rust（用于模拟）

* 静态编译型，性能强大，适合大规模的模拟
* 整合紧密，不需要第三方项目管理工具
* 原生支持多线程、MPSC开发，适合现代化项目的开发
* 具有无可比拟的第三方开源生态

编程语言：Python（用于数据分析）

* 具有matplotlib等第三方生态便于制图

开发方法：

* 多线程：解耦网桥软件和硬件的模拟
* MPSC：进程间高效通讯
* 拆分数据生成和模拟阶段，分工更明确

数据定义：

* 使用4字节表示网络地址
* 使用2字节表示网段
* 使用4字节表示帧数据
  + 帧数据在模拟中作用较小，故调整为较小长度，便于数据产生

性能观测：

* 记录每个操作的时间点，统计为散点图
* 记录任何进行等待的帧的等待时间
* 记录等待队列的长度变化规律

### 实验5.2

#### 基本思想

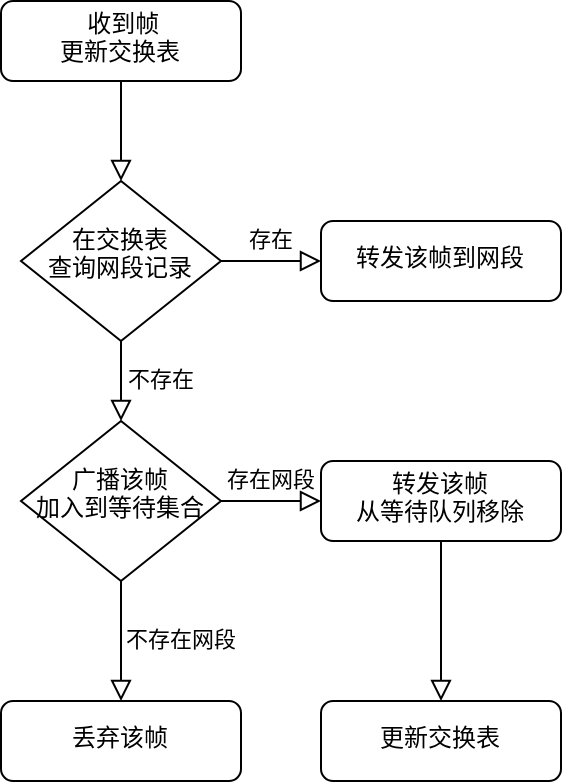
实验要求所提出的校验和算法主要需要将文件分为16位，实行累加，但是遇到溢出需要额外加1补偿。鉴于每一次操作有2个16位数字参与，计算结果仍然是16位数字，并且可以参与进一步运算（属于同一域），不同计算步骤之间没有必然联系，计算顺序可以打乱，也可以通过利用已有的数据减少空间分配。

#### 实现要点

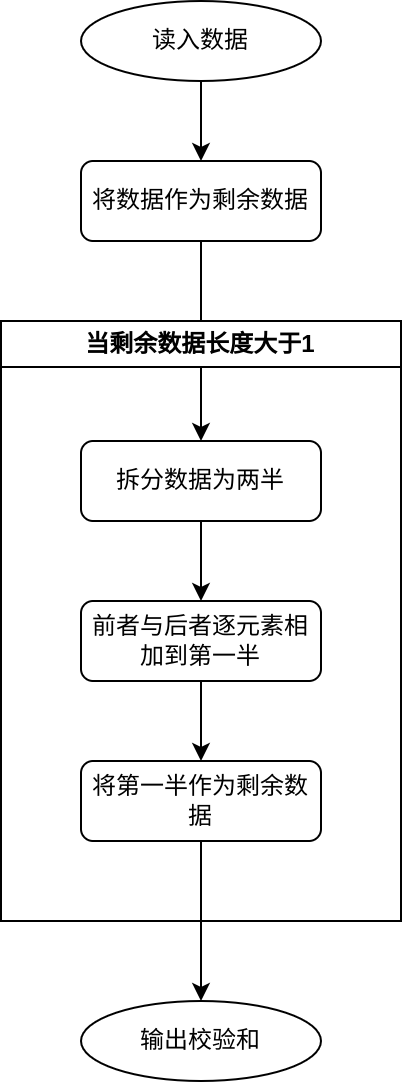
* 两个一组进行相加的话，大规模的计算可以并行化
* 数组拆分为前后两部分，（逐元素）后加前的话，可以无须额外分配空间

## 运行流程

### 实验5.1（仅列出网桥软件的）



### 实验5.2



## 问题记录

### 实验5.1

1. 网桥线程的终止逻辑难以定夺：从网桥行为记录的硬件在记录到全部的请求已经得到完成之后，向网桥发送终止信号。
2. 延迟统计计算的性能低下：换用哈希表记录帧和时间对应关系，时间复杂度少许下降，实际时间占用大幅改善。

### 实验5.2

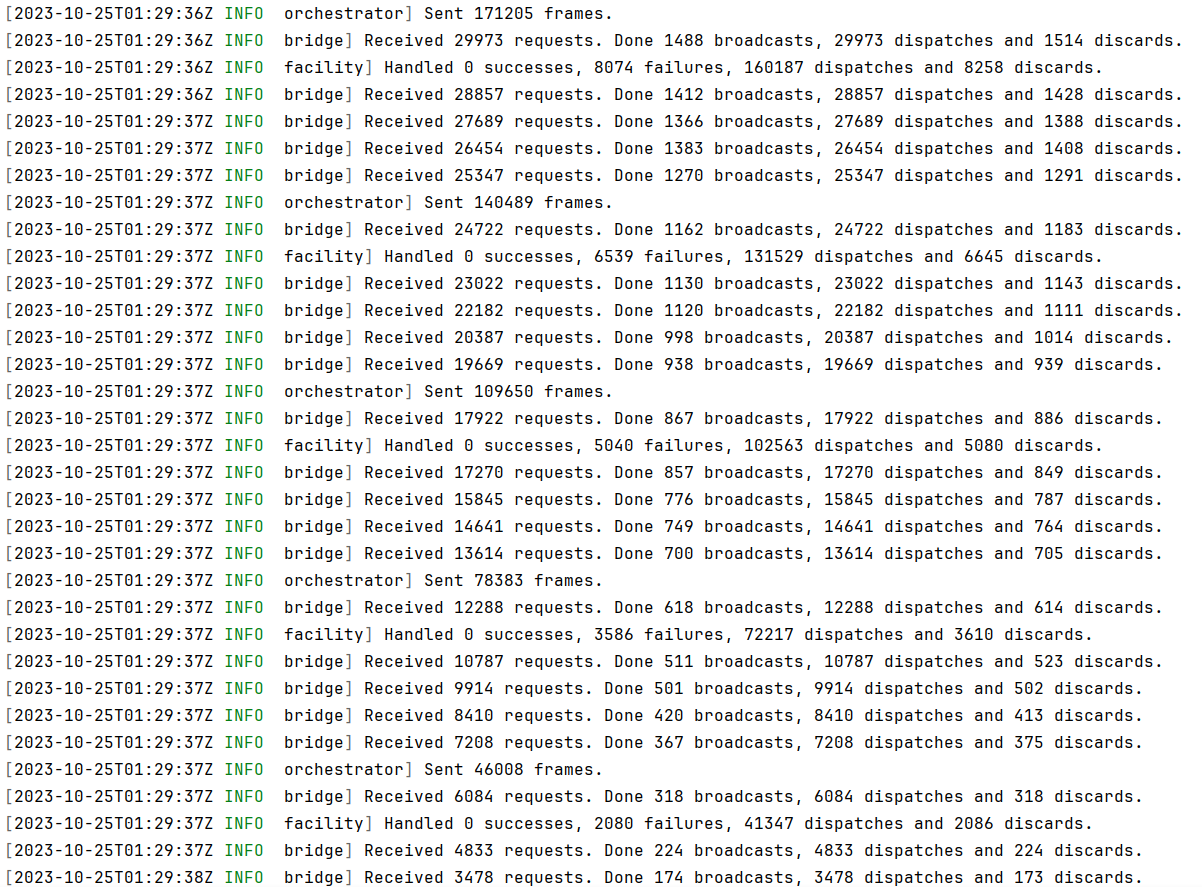
1. 将文件解释为16位无符号整数串，需要关注大小端问题：实验要求默认大端。
2. 实际上不进行优化比进行优化效率要高很多：推测是编译器产生了优化作用。

## 功能展示

### 实验5.1

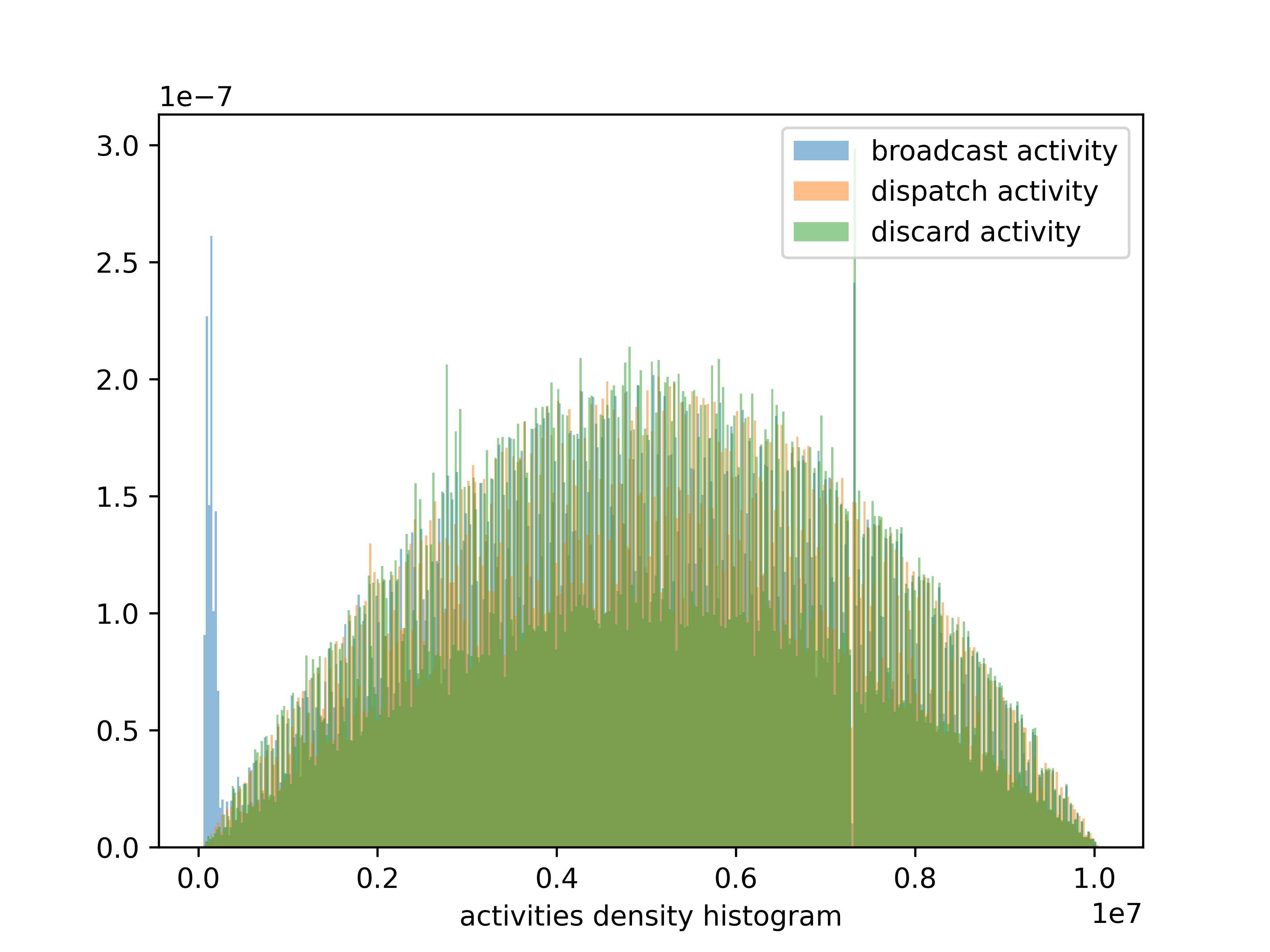
#### 模拟程序Log节选

解释：orchestrator会回报250ms内发送的帧数；bridge会回报50ms内受到的请求，发送的广播和实行的操作；facility会回报250ms内受到来自bridge的操作。



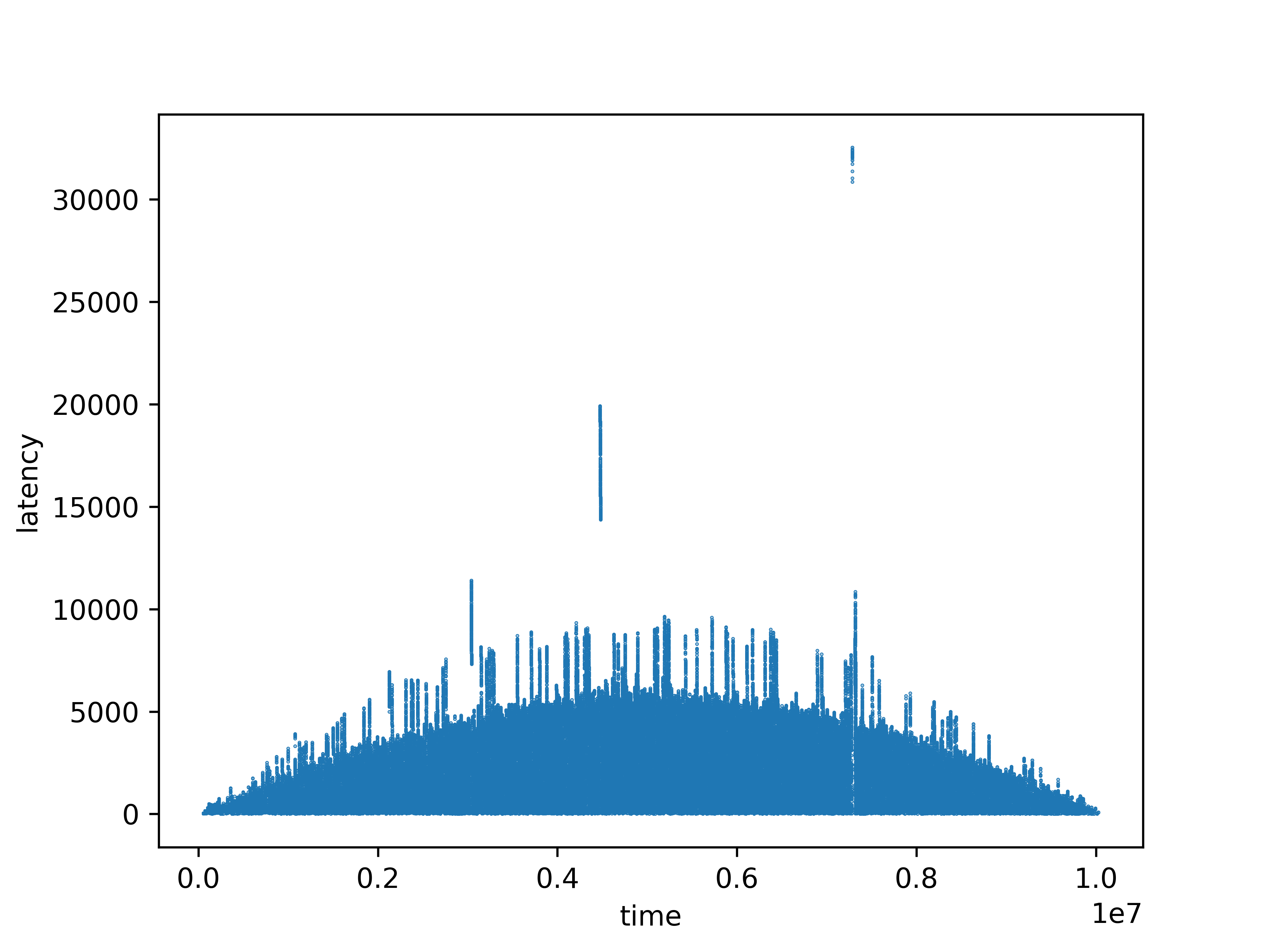
#### 网桥活动图

解释：该图展示了网桥执行的操作（“活动”），由于orchestrator发送信息的规律（见下文），整体的分布呈现上半圆的形状，其中各种操作的频率是不断波动的。在开端的时候出现了比较多的广播操作，基本上是由（当时）较为贫瘠的转发表造成的。



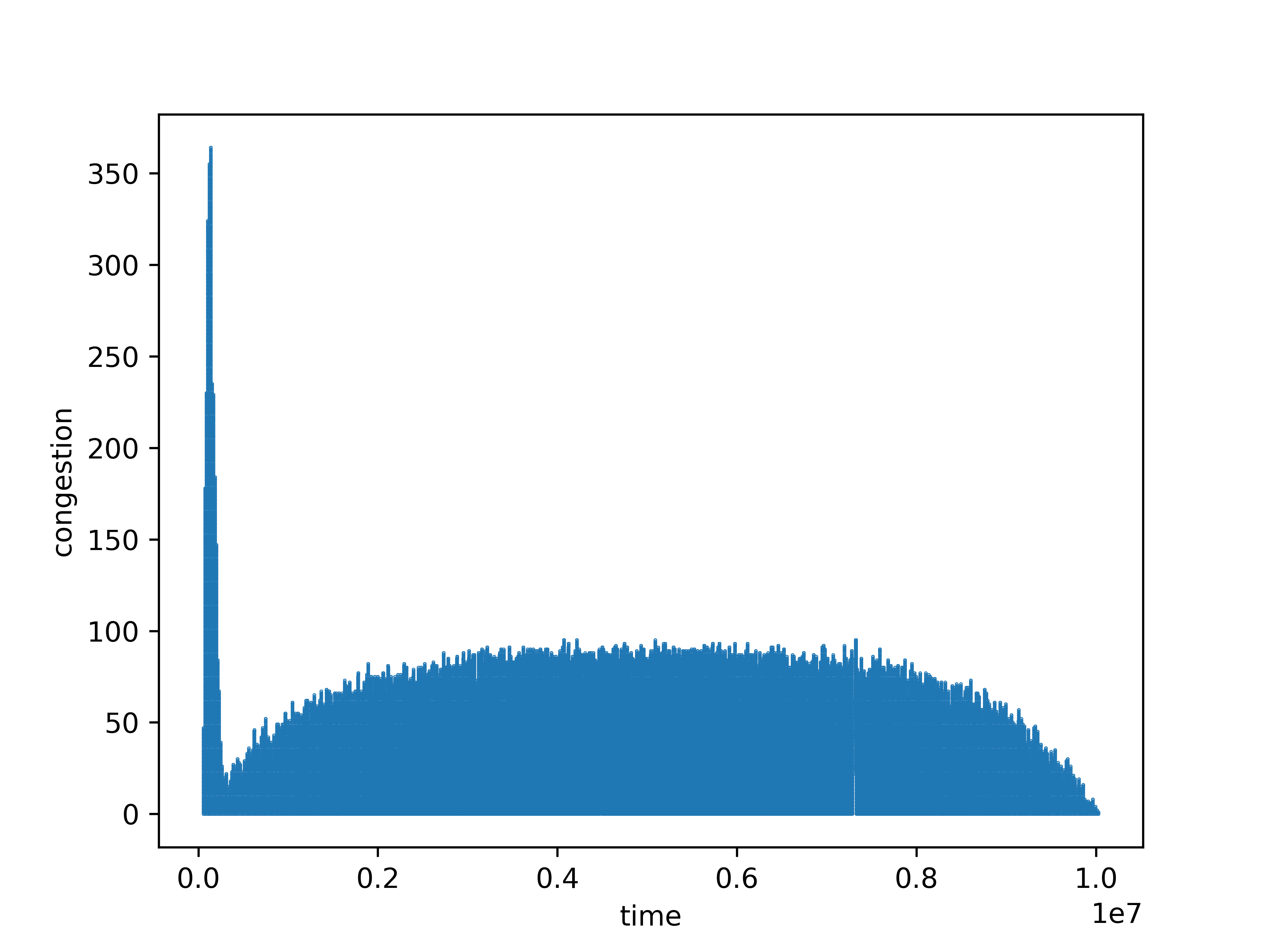
#### 网桥延迟图

解释：该图说明了被广播的帧的从被广播到被转发（或者丢弃）的延迟。总体上呈现出稳定的趋势，但是偶尔会出现延迟十分高的帧（未得到及时处理）。



#### 网桥拥塞情况图

解释：这里的“拥塞”指的是指定时间（等待队列发生变化时）此刻等待队列的长度。



### 实验5.2

下图：实验要求中的字符串，储存为文本文档，作为输入输入进该应用程序得出的结果。



下图：同理，但处理的是装有 Windows XP 的 VirtualBox 虚拟机磁盘镜像。



## 未来工作

### 实验5.1

1. 将网桥软件本身改为多线程操作，进一步提高处理效率
2. 使用改进的转发表数据结构
3. 在更大的规模上进行测试

### 实验5.2

1. 从LLVM IR或者汇编输出的角度分析性能的变化

## 源代码

### 实验5.1

#### src/lib.rs

use std::fmt::{Display, Formatter};  
use serde::{Serialize, Deserialize};  
  
#[derive(Debug, Copy, Clone, Eq, PartialEq, Ord, PartialOrd, Hash, Serialize, Deserialize)]  
#[repr(transparent)]  
pub struct Address {  
 pub data: [u8; 4]  
}  
  
impl Display for Address {  
 fn fmt(&self, f: &mut Formatter<*'\_*>) -> std::fmt::Result {  
 let a1 = self.data[0];  
 let a2 = self.data[1];  
 let a3 = self.data[2];  
 let a4 = self.data[3];  
 write!(f, "{a1:02x}:{a2:02x}:{a3:02x}:{a4:02x}")  
 }  
}  
  
impl TryFrom<&str> for Address {  
 type Error = ();  
  
 fn *try\_from*(value: &str) -> Result<Self, Self::Error> {  
 if value.len() != 11 {  
 return *Err*(());  
 }  
 let mut data = [0; 4];  
 data[0] = u8::*from\_str\_radix*(&value[0..2], 16).map\_err(|\_| ())?;  
 data[1] = u8::*from\_str\_radix*(&value[3..5], 16).map\_err(|\_| ())?;  
 data[2] = u8::*from\_str\_radix*(&value[6..8], 16).map\_err(|\_| ())?;  
 data[3] = u8::*from\_str\_radix*(&value[9..11], 16).map\_err(|\_| ())?;  
 *Ok*(Address { data })  
 }  
}  
  
#[derive(Debug, Copy, Clone, Eq, PartialEq, Ord, PartialOrd, Hash, Serialize, Deserialize)]  
#[repr(transparent)]  
pub struct Segment {  
 pub data: [u8; 2]  
}  
  
impl Display for Segment {  
 fn fmt(&self, f: &mut Formatter<*'\_*>) -> std::fmt::Result {  
 let a1 = self.data[0];  
 let a2 = self.data[1];  
 write!(f, "{a1:02x}:{a2:02x}")  
 }  
}  
  
impl TryFrom<&str> for Segment {  
 type Error = ();  
  
 fn *try\_from*(value: &str) -> Result<Self, Self::Error> {  
 if value.len() != 5 {  
 return *Err*(());  
 }  
 let mut data = [0; 2];  
 data[0] = u8::*from\_str\_radix*(&value[0..2], 16).map\_err(|\_| ())?;  
 data[1] = u8::*from\_str\_radix*(&value[3..5], 16).map\_err(|\_| ())?;  
 *Ok*(Segment { data })  
 }  
}  
  
pub type FrameData = [u8; 4];  
  
#[derive(Debug, Clone, Eq, PartialEq, Hash, Serialize, Deserialize)]  
pub struct Frame {  
 pub src: Address,  
 pub src\_seg: Segment,  
 pub dst: Address,  
 pub data: FrameData  
}  
  
impl Display for Frame {  
 fn fmt(&self, f: &mut Formatter<*'\_*>) -> std::fmt::Result {  
 write!(f, "{} {} {} {:02x?}", self.src, self.src\_seg, self.dst, self.data)  
 }  
}  
  
impl TryFrom<&str> for Frame {  
 type Error = ();  
  
 fn *try\_from*(value: &str) -> Result<Self, Self::Error> {  
 let mut seg = value.trim().split(' ');  
 let src = if let *Some*(val) = seg.next() { val } else { return *Err*(()) };  
 let src\_seg = if let *Some*(val) = seg.next() { val } else { return *Err*(()) };  
 let dst = if let *Some*(val) = seg.next() { val } else { return *Err*(()) };  
 let data\_s = if let *Some*(val) = seg.next() { val } else { return *Err*(()) };  
 let mut data = FrameData::default();  
 for i in 0..16 {  
 data[i] = u8::*from\_str\_radix*(&data\_s[i \* 2..i \* 2 + 2], 16).map\_err(|\_| ())?;  
 }  
 *Ok*(Frame {  
 src: src.try\_into()?,  
 src\_seg: src\_seg.try\_into()?,  
 dst: dst.try\_into()?,  
 data  
 })  
 }  
}

#### src/bin/generate.rs

use std::collections::HashSet;  
use std::fs::File;  
use rand::prelude::\*;  
use net\_exp\_bridge::{Address, Frame, FrameData, Segment};  
use std::io::{BufWriter, Write};  
use log::info;  
  
*/// Count of valid addresses*const *VALID\_ADDR\_CNT*: usize = 5000;  
*/// Count of invalid addresses*const *INVALID\_ADDR\_CNT*: usize = 100;  
*/// Count of segments*const *SEG\_CNT*: usize = 100;  
*/// Count of valid frames*const *VALID\_FRAME\_CNT*: usize = 1000\_0000;  
*/// Count of invalid frames*const *INVALID\_FRAME\_CNT*: usize = 10\_0000;  
  
*/// Generate random byte array of specified size with `fastrand` API.*fn gen\_byte\_arr<const *N*: usize>() -> [u8; *N*] {  
 let mut data = [0u8; *N*];  
 data.iter\_mut().for\_each(|x| \*x = fastrand::u8(..));  
 data  
}  
  
*/// Generate a physical address.*fn gen\_addr() -> Address {  
 Address { data: gen\_byte\_arr() }  
}  
  
*/// Generate a pool of physical addresses, unique.*fn gen\_addr\_pool(count: usize) -> HashSet<Address> {  
 let mut unique\_set: HashSet<Address> = HashSet::*with\_capacity*(count);  
 while unique\_set.len() < count {  
 unique\_set.insert(gen\_addr());  
 }  
 unique\_set  
}  
  
*/// Generate a pool of invalid addresses, unique and not clashing with valid ones.*fn gen\_invalid\_addr\_pool(addr\_pool: &HashSet<Address>, count: usize) -> HashSet<Address> {  
 let mut unique\_set: HashSet<Address> = HashSet::*with\_capacity*(count);  
 while unique\_set.len() < count {  
 let addr = gen\_addr();  
 if !addr\_pool.contains(&addr) {  
 unique\_set.insert(addr);  
 }  
 }  
 unique\_set  
}  
  
*/// Generate a segment.*fn gen\_seg() -> Segment {  
 Segment { data: gen\_byte\_arr() }  
}  
  
*/// Generate a pool of segments.*fn gen\_seg\_pool(count: usize) -> HashSet<Segment> {  
 let mut unique\_set: HashSet<Segment> = HashSet::*with\_capacity*(count);  
 while unique\_set.len() < count {  
 unique\_set.insert(gen\_seg());  
 }  
 unique\_set  
}  
  
*/// Generate frame data.*fn gen\_data() -> FrameData {  
 gen\_byte\_arr()  
}  
  
*/// Generate frame with specified pools for source and destination addresses.*fn gen\_frame(src\_pool: &[Address], src\_seg\_pool: &[Segment], dst\_pool: &[Address]) -> Frame {  
 let src = src\_pool[fastrand::usize(0..src\_pool.len())];  
 let src\_seg = src\_seg\_pool[fastrand::usize(0..src\_seg\_pool.len())];  
 let mut dst = src;  
 while dst == src {  
 dst = dst\_pool[fastrand::usize(0..dst\_pool.len())];  
 }  
 let data = gen\_data();  
 Frame { src, src\_seg, dst, data }  
}  
  
*/// Generate a sequence of frames with `gen\_frame` function.*fn gen\_frame\_seq(src\_pool: &[Address], src\_seg\_pool: &[Segment], dst\_pool: &[Address], count: usize) -> Vec<Frame> {  
 let mut seq = Vec::*with\_capacity*(count);  
 for \_ in 0..count {  
 seq.push(gen\_frame(src\_pool, src\_seg\_pool, dst\_pool));  
 }  
 seq  
}  
  
*/// Generate a mapping from address to segment from their pools.*fn gen\_addr\_seg(addr\_pool: Vec<Address>, seg\_pool: &[Segment]) -> Vec<(Address, Segment)> {  
 let mut seq = Vec::*with\_capacity*(addr\_pool.len() \* seg\_pool.len());  
 let least = addr\_pool.len() / seg\_pool.len();  
 *// assign segment for addresses* for (i, seg) in seg\_pool.iter().enumerate() {  
 let begin = i \* least;  
 for j in 0..least {  
 seq.push((addr\_pool[begin + j], \*seg));  
 }  
 }  
 *// treat remaining ones* if seq.len() < addr\_pool.len() {  
 let begin = seq.len();  
 for i in begin..addr\_pool.len() {  
 seq.push((addr\_pool[i], seg\_pool[fastrand::usize(0..seg\_pool.len())]));  
 }  
 }  
 seq  
}  
  
*/// Serialize data for use with simulation binary & human analysis.*fn serialize(addr\_seg\_seq: &[(Address, Segment)], inv\_addr\_pool: &[Address], frame\_seq: &[Frame]) {  
 *// encode binary format for use with simulation* let addr\_seg\_rmp = File::*create*("addr\_seg.rmp").unwrap();  
 let inv\_addr\_rmp = File::*create*("inv\_addr.rmp").unwrap();  
 let frame\_rmp = File::*create*("frame.rmp").unwrap();  
 rmp\_serde::encode::write(&mut BufWriter::*new*(addr\_seg\_rmp), addr\_seg\_seq).unwrap();  
 rmp\_serde::encode::write(&mut BufWriter::*new*(inv\_addr\_rmp), inv\_addr\_pool).unwrap();  
 rmp\_serde::encode::write(&mut BufWriter::*new*(frame\_rmp), frame\_seq).unwrap();  
  
 *// encode text for human-based analysis* let addr\_seg\_file = File::*create*("addr\_seg.txt").unwrap();  
 let inv\_addr\_file = File::*create*("inv\_addr.txt").unwrap();  
 let mut addr\_seg\_bw = BufWriter::*new*(addr\_seg\_file);  
 let mut inv\_addr\_bw = BufWriter::*new*(inv\_addr\_file);  
 for (addr, seg) in addr\_seg\_seq {  
 writeln!(addr\_seg\_bw, "{} {}", addr, seg).unwrap();  
 }  
 for addr in inv\_addr\_pool {  
 writeln!(inv\_addr\_bw, "{}", addr).unwrap();  
 }  
}  
  
  
fn main() {  
 env\_logger::init();  
  
 *// create pools* info!("Address pool...");  
 let addr\_pool = gen\_addr\_pool(*VALID\_ADDR\_CNT*);  
 info!("Invalid address pool...");  
 let inv\_addr\_pool = gen\_addr\_pool(*INVALID\_ADDR\_CNT*);  
 info!("Segment pool...");  
 let seg\_pool = gen\_seg\_pool(*SEG\_CNT*);  
  
 let addr\_pool = addr\_pool.into\_iter().collect::<Vec<\_>>();  
 let inv\_addr\_pool = inv\_addr\_pool.into\_iter().collect::<Vec<\_>>();  
 let seg\_pool = seg\_pool.into\_iter().collect::<Vec<\_>>();  
  
 *// fabricate frames* info!("Frame sequence...");  
 let frame\_seq = {  
 let mut frame\_seq = gen\_frame\_seq(  
 &addr\_pool, &seg\_pool, &addr\_pool, *VALID\_FRAME\_CNT*);  
 let inv\_frame\_seq = gen\_frame\_seq(  
 &addr\_pool, &seg\_pool, &inv\_addr\_pool, *INVALID\_FRAME\_CNT*);  
 frame\_seq.extend\_from\_slice(&inv\_frame\_seq);  
 frame\_seq.shuffle(&mut thread\_rng());  
 frame\_seq  
 };  
  
 *// generate segment mapping* let addr\_seg\_seq = gen\_addr\_seg(addr\_pool, &seg\_pool);  
 info!("Serialization...");  
 serialize(&addr\_seg\_seq, &inv\_addr\_pool, &frame\_seq);  
}

#### src/bin/simulate.rs

use std::collections::{BTreeMap, HashMap};  
use std::fs::File;  
use std::io::{BufReader, BufWriter};  
use std::sync::mpsc::{Receiver, Sender};  
use std::{fs, thread};  
use std::f64::consts::*PI*;  
use std::time::{Duration, Instant};  
use log::info;  
use serde\_pickle::SerOptions;  
use net\_exp\_bridge::{Address, Frame, Segment};  
  
const *ELAPSE\_SEC*: usize = 10;  
  
*/// Event that bridge receives.*enum Event {  
 */// Incoming request of routing a frame.  
 Request*(Frame),  
 */// Found segment that accept an address.  
 Success*(Address, Segment),  
 */// No segment accepts an address.  
 Failure*(Address),  
 */// Simulation finishing and the bridge should be exiting.  
 Shutdown*,  
}  
  
*/// Command that bridge emits.*enum Command {  
 */// Broadcast an address to segments  
 Broadcast*(Address),  
 */// Dispatch a frame to a segment  
 Dispatch*(Frame, Segment),  
 */// Discard a frame  
 Discard*(Frame),  
}  
  
*/// Waiting list of frames.*struct Holder {  
 map: BTreeMap<Address, Vec<Frame>>  
}  
  
impl Holder {  
 fn *new*() -> Self {  
 Holder { map: BTreeMap::*new*() }  
 }  
  
 */// Check if there exist frames of a specific address.* fn exist\_addr(&self, addr: &Address) -> bool {  
 self.map.contains\_key(addr)  
 }  
  
 */// Hold a frame.* fn hold(&mut self, frame: Frame) {  
 let frames = self.map.entry(frame.dst)  
 .or\_insert\_with(Vec::*new*);  
 frames.push(frame);  
 }  
  
 */// Release frames of the same address.* fn release(&mut self, addr: Address) -> Vec<Frame> {  
 self.map.remove(&addr).unwrap\_or\_default()  
 }  
  
 fn len(&self) -> usize {  
 self.map.len()  
 }  
}  
  
*/// Statistics of bridge*pub enum BridgeStatRecord {  
 *Broadcast*(Frame),  
 *Dispatch*(Frame),  
 *Discard*(Frame),  
}  
  
impl BridgeStatRecord {  
 pub fn frame(&self) -> &Frame {  
 match self {  
 BridgeStatRecord::*Broadcast*(frame) => frame,  
 BridgeStatRecord::*Dispatch*(frame) => frame,  
 BridgeStatRecord::*Discard*(frame) => frame,  
 }  
 }  
}  
  
*/// Record of bridge statistics.*pub struct BridgeStat {  
 pub records: Vec<BridgeStatRecord>,  
 pub times: Vec<Instant>,  
 pub init: Instant,  
}  
  
impl BridgeStat {  
 fn *new*() -> Self {  
 BridgeStat { records: Vec::*new*(), times: Vec::*new*(), init: Instant::*now*() }  
 }  
  
 fn broadcast(&mut self, frame: Frame) {  
 self.records.push(BridgeStatRecord::*Broadcast*(frame));  
 self.times.push(Instant::*now*());  
 }  
  
 fn dispatch(&mut self, frame: Frame) {  
 self.records.push(BridgeStatRecord::Dispatch(frame));  
 self.times.push(Instant::now());  
 }  
  
 fn discard(&mut self, frame: Frame) {  
 self.records.push(BridgeStatRecord::Discard(frame));  
 self.times.push(Instant::now());  
 }  
  
 fn len(&self) -> usize {  
 self.records.len()  
 }  
  
 */// Export scatter of different types of activities.* fn export\_activity\_scatter(&self) {  
 let sc\_src = self.records.iter()  
 .zip(self.times.iter())  
 .map(|(x, y)| (x, y.duration\_since(self.init).as\_micros()));  
  
 let mut sc\_broadcast = Vec::with\_capacity(self.records.len());  
 let mut sc\_dispatch = Vec::with\_capacity(self.records.len());  
 let mut sc\_discard = Vec::with\_capacity(self.records.len());  
  
 for (x, y) in sc\_src {  
 match x {  
 BridgeStatRecord::Broadcast(\_) => sc\_broadcast.push(y as i64),  
 BridgeStatRecord::Dispatch(\_) => sc\_dispatch.push(y as i64),  
 BridgeStatRecord::Discard(\_) => sc\_discard.push(y as i64),  
 }  
 }  
  
 let mut w\_broadcast = BufWriter::new(File::create("sc\_broadcast\_activity.pkl").unwrap());  
 let mut w\_dispatch = BufWriter::new(File::create("sc\_dispatch\_activity.pkl").unwrap());  
 let mut w\_discard = BufWriter::new(File::create("sc\_discard\_activity.pkl").unwrap());  
  
 serde\_pickle::to\_writer(&mut w\_broadcast, &sc\_broadcast, SerOptions::default()).unwrap();  
 serde\_pickle::to\_writer(&mut w\_dispatch, &sc\_dispatch, SerOptions::default()).unwrap();  
 serde\_pickle::to\_writer(&mut w\_discard, &sc\_discard, SerOptions::default()).unwrap();  
 }  
  
 */// Export scatter of latencies of frames broadcast.* fn export\_latency\_scatter(&self) {  
 let mut hold\_map = HashMap::<Frame, u128>::new();  
 let mut latencies = Vec::with\_capacity(self.records.len());  
 for (rec, t) in self.records.iter().zip(self.times.iter()) {  
 let t = t.duration\_since(self.init).as\_micros();  
 match rec {  
 BridgeStatRecord::Broadcast(frame) => {  
 hold\_map.insert(frame.clone(), t);  
 }  
 BridgeStatRecord::Dispatch(frame) | BridgeStatRecord::Discard(frame) => {  
 let begin = if let Some(val) = hold\_map.remove(&frame) { val } else {  
 continue  
 };  
 let lat = t - begin;  
 latencies.push(vec![begin as i64, lat as i64]);  
 }  
 }  
 }  
 serde\_pickle::to\_writer(&mut BufWriter::new(File::create("sc\_latency.pkl").unwrap()),  
 &latencies, SerOptions::default()).unwrap();  
 }  
}  
  
*/// Statistics of pending frames of bridge.*pub struct BridgePendingStat {  
 pub records: Vec<usize>,  
 pub times: Vec<Instant>,  
 pub init: Instant,  
}  
  
impl BridgePendingStat {  
 fn new() -> Self {  
 BridgePendingStat { records: Vec::new(), times: Vec::new(), init: Instant::now() }  
 }  
  
 fn rec(&mut self, count: usize) {  
 self.records.push(count);  
 self.times.push(Instant::now());  
 }  
  
 fn len(&self) -> usize {  
 self.records.len()  
 }  
  
 */// Export scatter of congestion, the changing pressure of waiting list.* fn export\_congestion\_scatter(&self) {  
 let sc\_congestion = self.records.iter()  
 .zip(self.times.iter())  
 .map(|(x, y)| (x, y.duration\_since(self.init).as\_micros()))  
 .map(|(x, y)| vec![y as i64, \*x as i64])  
 .collect::<Vec<\_>>();  
 serde\_pickle::to\_writer(&mut BufWriter::new(File::create("sc\_congestion.pkl").unwrap()),  
 &sc\_congestion, SerOptions::default()).unwrap();  
 }  
}  
  
*/// Launch network bridge*fn bridge(tc: Sender<Command>, re: Receiver<Event>) {  
 info!(target: "bridge", "Bridge started.");  
 let mut mapping = BTreeMap::new();  
 let mut pending = Holder::new();  
 let mut stat = BridgeStat::new();  
 let mut pending\_stat = BridgePendingStat::new();  
 let mut req\_cnt = 0;  
 let mut b\_cnt = 0;  
 let mut dp\_cnt = 0;  
 let mut dc\_cnt = 0;  
 let mut last\_t = Instant::now();  
 while let Ok(event) = re.recv() { *// receive an event* match event {  
 Event::Request(frame) => {  
 if mapping.get(&frame.src).is\_none() {  
 *// correlate the source address with incoming segment* mapping.insert(frame.src, frame.src\_seg);  
 }  
 if let Some(segment) = mapping.get(&frame.dst) {  
 *// dispatch if source found in mapping* stat.dispatch(frame.clone());  
 tc.send(Command::Dispatch(frame, \*segment)).unwrap();  
 req\_cnt += 1;  
 dp\_cnt += 1;  
 } else if !pending.exist\_addr(&frame.dst) {  
 *// broadcast if no frames of same source are waiting* stat.broadcast(frame.clone());  
 tc.send(Command::Broadcast(frame.dst)).unwrap(); *// <- actual command* pending\_stat.rec(pending.len());  
 pending.hold(frame);  
 b\_cnt += 1;  
 } else {  
 stat.broadcast(frame.clone());  
 pending\_stat.rec(pending.len());  
 pending.hold(frame);  
 }  
 }  
 Event::Success(address, segment) => {  
 *// update the mapping* mapping.insert(address, segment);  
 for frame in pending.release(address) {  
 *// dispatch all frames with the same segment* stat.dispatch(frame.clone());  
 tc.send(Command::Dispatch(frame, segment)).unwrap();  
 dp\_cnt += 1;  
 }  
 pending\_stat.rec(pending.len());  
 }  
 Event::Failure(address) => {  
 for frame in pending.release(address) {  
 *// discard them all* stat.discard(frame.clone());  
 tc.send(Command::Discard(frame)).unwrap();  
 dc\_cnt += 1;  
 }  
 pending\_stat.rec(pending.len());  
 }  
 Event::Shutdown => {  
 info!(target: "bridge", "Received shutdown signal.");  
 *// export statistics* stat.export\_activity\_scatter();  
 stat.export\_latency\_scatter();  
 pending\_stat.export\_congestion\_scatter();  
 break;  
 }  
 }  
 if last\_t.elapsed() > Duration::from\_millis(50) {  
 info!(target: "bridge", "Received {} requests. Done {} broadcasts, {} dispatches and {} discards.",  
 req\_cnt, b\_cnt, dp\_cnt, dc\_cnt);  
 req\_cnt = 0;  
 b\_cnt = 0;  
 dp\_cnt = 0;  
 dc\_cnt = 0;  
 last\_t = Instant::now();  
 }  
 }  
 info!(target: "bridge", "Bridge exiting.");  
}  
  
*/// Cumulative distribution function of the distribution of "half circle".  
///  
/// Its PDF (Probability Density Function)'s graph will look like one top half of a circle fitted  
/// in the square of x from 0 to 1 and y from 0 to 1.*fn half\_circle\_dist\_cdf(x: f64) -> f64 {  
 let x = x \* PI - PI / 2.0;  
 (x.sin() + 1.0) / 2.0  
}  
  
*/// Distribute the frames per milliseconds in specified duration with a distribution function.*fn distribute(frame\_seq: Vec<Frame>, dur\_sec: usize, dist: fn(f64) -> f64) -> Vec<Vec<Frame>> {  
 let mut buckets = vec![Vec::new(); dur\_sec \* 1000];  
 let mut last\_pos = 0;  
 let dur = dur\_sec \* 1000;  
 for (i, vec) in buckets.iter\_mut().enumerate() {  
 let pos = (dist(i as f64 / dur as f64) \* frame\_seq.len() as f64) as usize;  
 vec.extend\_from\_slice(&frame\_seq[last\_pos..pos]);  
 last\_pos = pos;  
 }  
 *// collect remaining bits if any* if last\_pos < frame\_seq.len() {  
 buckets.last\_mut().unwrap().extend\_from\_slice(&frame\_seq[last\_pos..]);  
 }  
 buckets  
}  
  
*/// Orchestration service that send frames to the bridge with distributed frame sequence.*fn orchestrator(frame\_seq: Vec<Frame>, te: Sender<Event>) {  
 info!(target: "orchestrator", "Orchestrator started.");  
 let frame\_seq = distribute(frame\_seq, ELAPSE\_SEC, half\_circle\_dist\_cdf);  
 let begin = Instant::now();  
 let mut last = 0;  
 let mut last\_t = Instant::now();  
 let mut count = 0;  
 loop {  
 let now = Instant::now();  
 let dur = now.duration\_since(begin);  
 let cur = dur.as\_secs() \* 1000 + dur.subsec\_millis() as u64;  
 if cur >= frame\_seq.len() as u64 {  
 for buckets in frame\_seq[last..].iter() {  
 for frame in buckets {  
 te.send(Event::Request(frame.clone())).unwrap();  
 }  
 }  
 break;  
 }  
 if cur > last as u64 {  
 for buckets in frame\_seq[last..cur as usize].iter() {  
 for frame in buckets {  
 te.send(Event::Request(frame.clone())).unwrap();  
 count += 1;  
 }  
 }  
 last = cur as usize;  
 }  
 if now.duration\_since(last\_t) > Duration::from\_millis(250) {  
 info!(target: "orchestrator", "Sent {} frames.", count);  
 count = 0;  
 last\_t = now;  
 }  
 thread::sleep(Duration::from\_millis(1));  
 }  
 info!(target: "orchestrator", "Orchestrator exiting.");  
}  
  
*/// Meter to count facility statistics within some time.*struct FacilityMeter {  
 s\_cnt: usize,  
 f\_cnt: usize,  
 dp\_cnt: usize,  
 dc\_cnt: usize,  
}  
  
impl FacilityMeter {  
 fn new() -> Self {  
 FacilityMeter { s\_cnt: 0, f\_cnt: 0, dp\_cnt: 0, dc\_cnt: 0 }  
 }  
  
 fn inc\_success(&mut self) {  
 self.s\_cnt += 1;  
 }  
  
 fn inc\_failure(&mut self) {  
 self.f\_cnt += 1;  
 }  
  
 fn inc\_dispatch(&mut self) {  
 self.dp\_cnt += 1;  
 }  
  
 fn inc\_discard(&mut self) {  
 self.dc\_cnt += 1;  
 }  
  
 fn report(&mut self) {  
 info!(target: "facility", "Handled {} successes, {} failures, {} dispatches and {} discards.",  
 self.s\_cnt, self.f\_cnt, self.dp\_cnt, self.dc\_cnt);  
 self.s\_cnt = 0;  
 self.f\_cnt = 0;  
 self.dp\_cnt = 0;  
 self.dc\_cnt = 0;  
 }  
}  
  
*/// Facilitation service that handle commands from the bridge.*fn facility(count: usize, mapping: BTreeMap<Address, Segment>, te: Sender<Event>, rc: Receiver<Command>) {  
 info!(target: "facility", "Facility started.");  
 let mut cur\_n = 0;  
 let mut meter = FacilityMeter::new();  
 let mut last\_t = Instant::now();  
 while let Ok(command) = rc.recv() {  
 match command {  
 Command::Broadcast(addr) => {  
 if let Some(segment) = mapping.get(&addr) {  
 te.send(Event::Success(addr, \*segment)).unwrap();  
 meter.inc\_success();  
 } else {  
 te.send(Event::Failure(addr)).unwrap();  
 meter.inc\_failure();  
 }  
 }  
 Command::Dispatch(\_, \_) => {  
 meter.inc\_dispatch();  
 cur\_n += 1;  
 }  
 Command::Discard(\_) => {  
 meter.inc\_discard();  
 cur\_n += 1;  
 }  
 }  
 if last\_t.elapsed() > Duration::from\_millis(250) {  
 meter.report();  
 last\_t = Instant::now();  
 }  
 if cur\_n == count {  
 te.send(Event::Shutdown).unwrap();  
 break;  
 }  
 }  
 info!(target: "facility", "Facility exiting.");  
}  
  
*/// Load segment mapping from disk.*fn load\_mapping() -> BTreeMap<Address, Segment> {  
 let addr\_seg = BufReader::new(File::open("addr\_seg.rmp").unwrap());  
 let addr\_seg: Vec<(Address, Segment)> = rmp\_serde::from\_read(addr\_seg).unwrap();  
 BTreeMap::from\_iter(addr\_seg)  
}  
  
*/// Load generated frames from disk.*fn load\_frames() -> Vec<Frame> {  
 let frame = BufReader::new(File::open("frame.rmp").unwrap());  
 rmp\_serde::from\_read(frame).unwrap()  
}  
  
fn main() {  
 env\_logger::init();  
 let (tc, rc) = std::sync::mpsc::channel();  
 let (te, re) = std::sync::mpsc::channel();  
 let frames = load\_frames();  
  
 let facility = {  
 let mapping = load\_mapping();  
 let te = te.clone();  
 let len = frames.len();  
 thread::spawn(move || facility(len, mapping, te, rc))  
 };  
  
 let bridge = {  
 let tc = tc.clone();  
 thread::spawn(move || bridge(tc, re))  
 };  
  
 let orchestrator = {  
 let te = te.clone();  
 thread::spawn(move || orchestrator(frames, te))  
 };  
  
 orchestrator.join().unwrap();  
 facility.join().unwrap();  
 bridge.join().unwrap();  
}

#### datagen.py

import matplotlib.pyplot as plt  
import numpy as np  
import pickle  
import os  
import datetime  
from tqdm import tqdm  
  
attempt\_num = 20  
pb = tqdm(total=attempt\_num, desc='')  
  
for i in range(20):  
 os.chdir('tmp')  
  
 for file in os.listdir():  
 os.remove(file)  
  
 pb.set\_description('Generating')  
 os.system('..\\generate.exe')  
 pb.set\_description('Simulating')  
 os.system('..\\simulate.exe')  
  
 pb.set\_description('Reading data')  
 sc\_bc\_act = np.array(pickle.load(open('sc\_broadcast\_activity.pkl', 'rb')))  
 sc\_dp\_act = np.array(pickle.load(open('sc\_dispatch\_activity.pkl', 'rb')))  
 sc\_di\_act = np.array(pickle.load(open('sc\_discard\_activity.pkl', 'rb')))  
 sc\_lat = np.array(pickle.load(open('sc\_latency.pkl', 'rb')))  
 sc\_cong = np.array(pickle.load(open('sc\_congestion.pkl', 'rb')))  
  
 pb.set\_description('Plotting')  
 fig, ax = plt.subplots()  
 ax.hist(sc\_bc\_act, bins=400, density=True, alpha=0.5, label='broadcast activity')  
 ax.hist(sc\_dp\_act, bins=400, density=True, alpha=0.5, label='dispatch activity')  
 ax.hist(sc\_di\_act, bins=400, density=True, alpha=0.5, label='discard activity')  
 ax.legend(loc='upper right')  
 ax.set\_xlabel('activities density histogram')  
 fig.savefig('activity.png', dpi=600)  
  
 fig, ax = plt.subplots()  
 ax.scatter(sc\_lat[:, 0], sc\_lat[:, 1], s=0.1)  
 ax.set\_xlabel('time')  
 ax.set\_ylabel('latency')  
 fig.savefig('latency.png', dpi=600)  
  
 fig, ax = plt.subplots(dpi=150)  
 ax.scatter(sc\_cong[:, 0], sc\_cong[:, 1], s=0.1)  
 ax.set\_xlabel('time')  
 ax.set\_ylabel('congestion')  
 fig.savefig('congestion.png', dpi=600)  
  
 plt.close('all')  
  
 pb.set\_description('Saving')  
 time\_str = datetime.datetime.now().strftime('%Y-%m-%d-%H-%M-%S')  
 out\_subdir\_name = '{}#{}'.format(time\_str, i)  
 os.chdir('..')  
 os.mkdir('out\\' + out\_subdir\_name)  
  
 os.rename('tmp\\activity.png', 'out\\' + out\_subdir\_name + '\\activity.png')  
 os.rename('tmp\\latency.png', 'out\\' + out\_subdir\_name + '\\latency.png')  
 os.rename('tmp\\congestion.png', 'out\\' + out\_subdir\_name + '\\congestion.png')  
  
 pb.update(1)

### 实验5.2

use std::env::args;  
use std::fs::File;  
use std::io::{BufReader, Read};  
use std::time::Instant;  
  
fn main() {  
 let path = args().nth(1).unwrap();  
 let file = File::*open*(path).unwrap();  
 let mut file = BufReader::*new*(file);  
 let begin = Instant::*now*();  
 let mut buf = [0; 2];  
 let mut sum = 0\_u16;  
 while let *Ok*(n) = file.read(&mut buf) {  
 if n == 0 { break; }  
 if n == 1 { buf[1] = 0; }  
 let val = u16::*from\_be\_bytes*(buf);  
 let (mut ns, of) = sum.overflowing\_add(val);  
 if of { ns += 1; }  
 sum = ns;  
 }  
 let time = begin.elapsed();  
 println!("Elapse: {time:?}");  
 println!("Checksum: {sum:x}");  
}