ECS 132 - Project

Teresa Li, sqtli@ucdavis.edu
Wenjing Fu, luffu@ucdavis.edu

2018-05-28

Contents

Design **2**
Question 1 . . . 2
Question 2 . . . 2
Question 3 . . . 4
Question 4 . . . 4
Question 5 . . . 6

Detection **6**
Step 1 . . . 6
Step 2 . . . 7
Step 3 . . . 8
Step 4 . . . 9
Step 5 . . . 10
Step 6 . . . 12
Step 7 . . . 12
Step 8 . . . 13

Implementation **14**
Implementation 1 . . . 14
Implementation 2 . . . 16
Implementation 3 . . . 18
Implementation 4 . . . 21

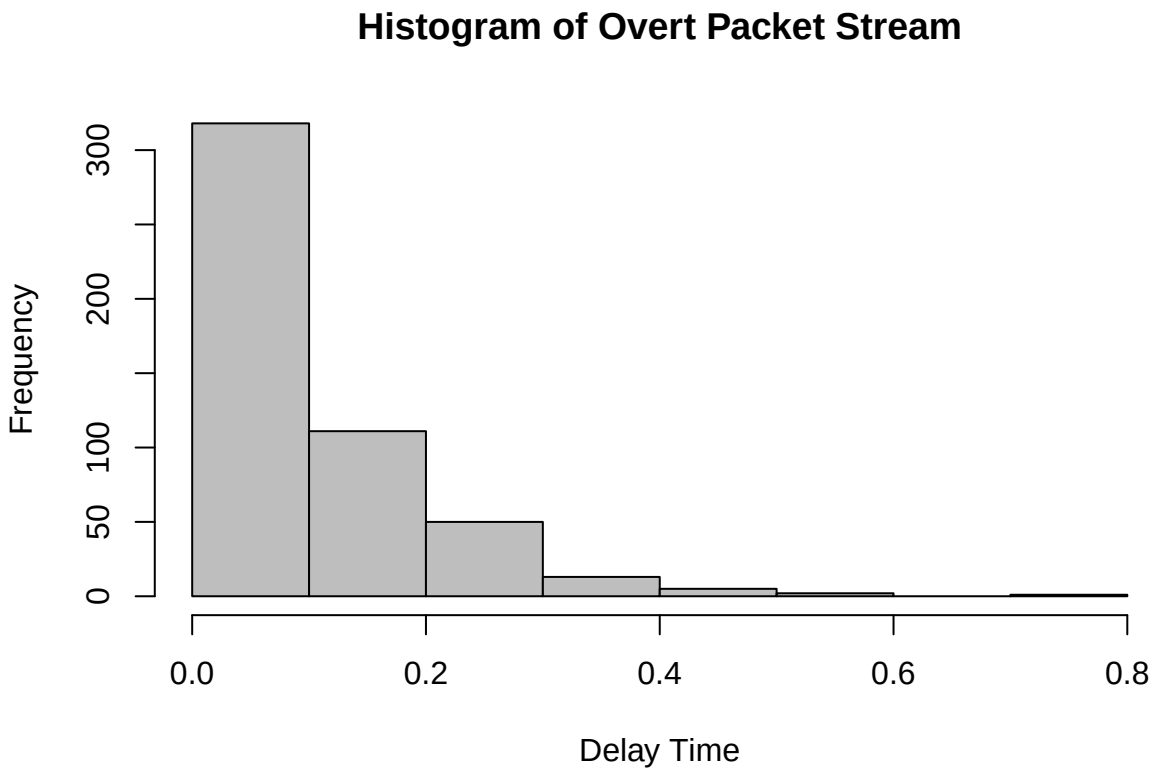
Design

Question 1

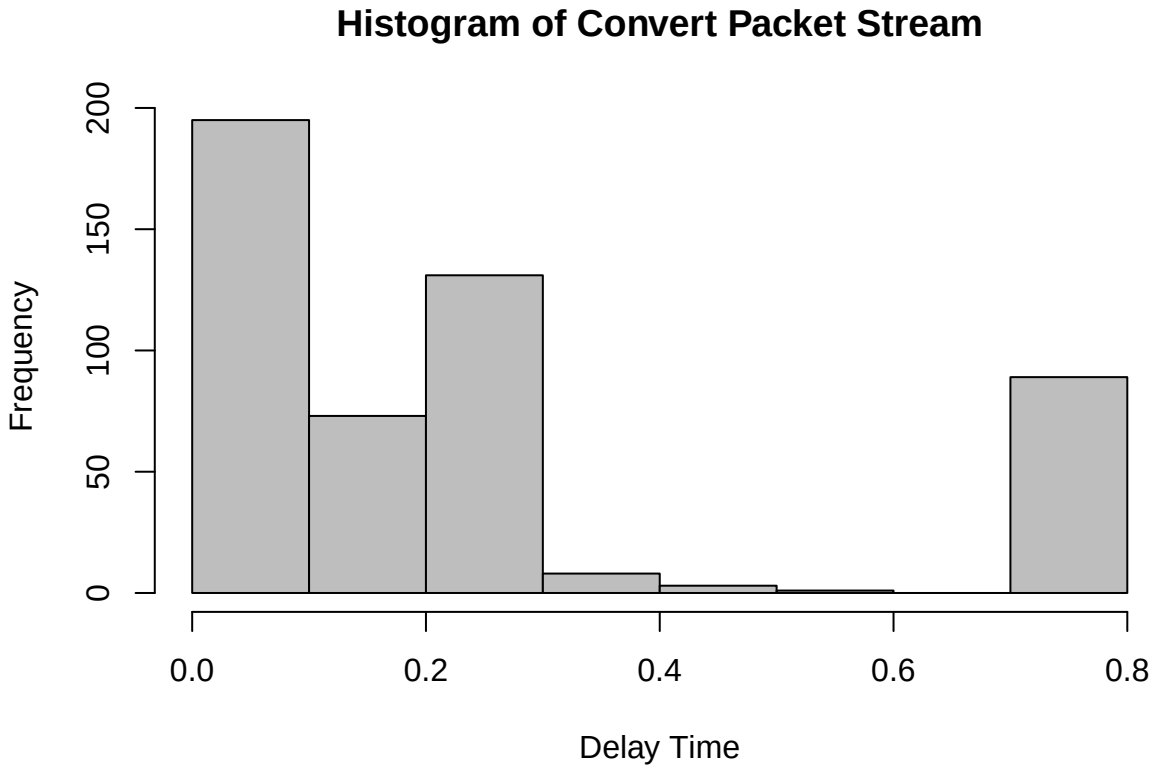
```
Traffic_data_orig <- read.csv("Traffic_data_orig.csv", header=TRUE)
message <- "this is a secret message"
raw <- charToRaw(message)
time = Traffic_data_orig$Time
num = as.integer(rawToBits(raw))

delays = numeric(length(time) - 1)
for (i in (1:(length(time) - 1))) {
  delays[i] = time[i+1] - time[i]
}

index = 1
bitlen = length(raw)*8
encrpt <- numeric(length(raw)*8)
for (i in (0:(length(raw)-1))) {
  for (j in 1:8) {
    if (num[i*8+j] == 0) {
      encrpt[index] = 0.25
    }
    else {
      encrpt[index] = 0.75
    }
    index = index+1
    j = j-1
  }
}

delays2 = delays
for (i in (1:bitlen)) {
  delays2[i] = encrpt[i]
}
```

Question 2

```
hist(delays, col='grey', xlab = 'Delay Time',
     main = 'Histogram of Overt Packet Stream')
```




```
hist(delays2, col='grey', xlab = 'Delay Time',
     main = 'Histogram of Convert Packet Stream')
```



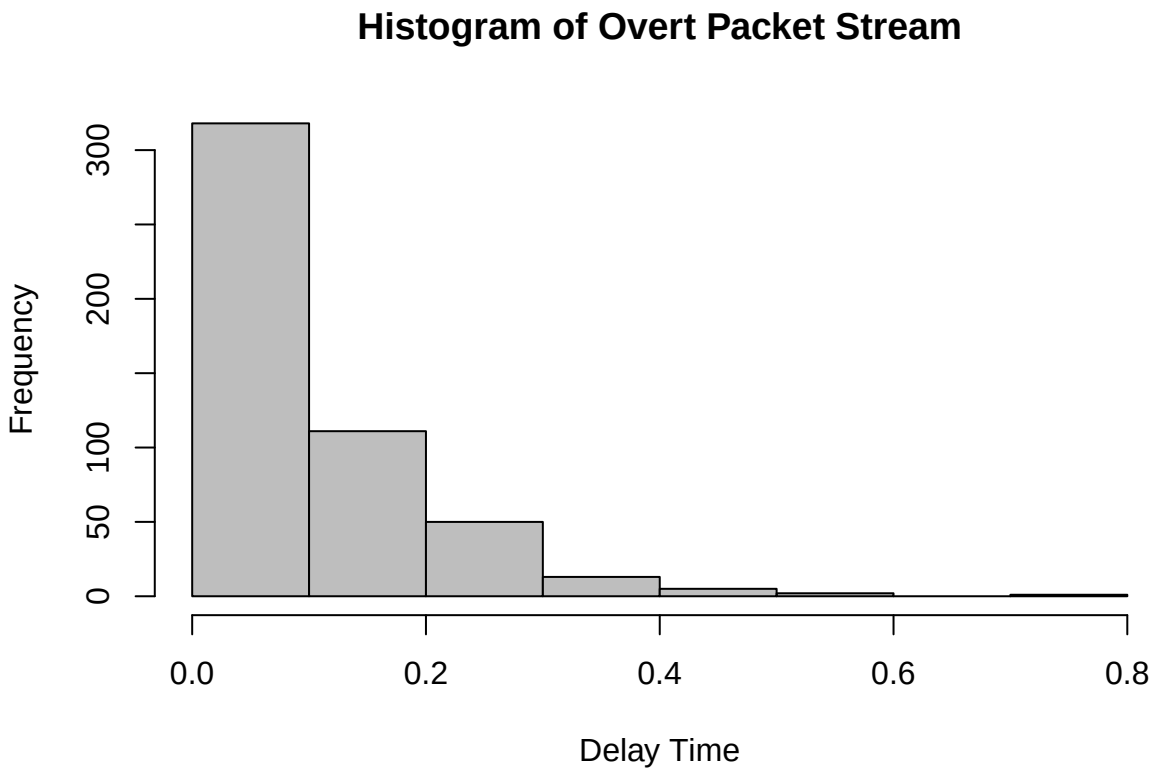
Yes, Eve will be suspicious because it is obvious that the distribution changed.The Frequence of delaytime of 0.25 and 0.75 have increased dramatically compared to the histogram of Overt Packet Stream.

Question 3

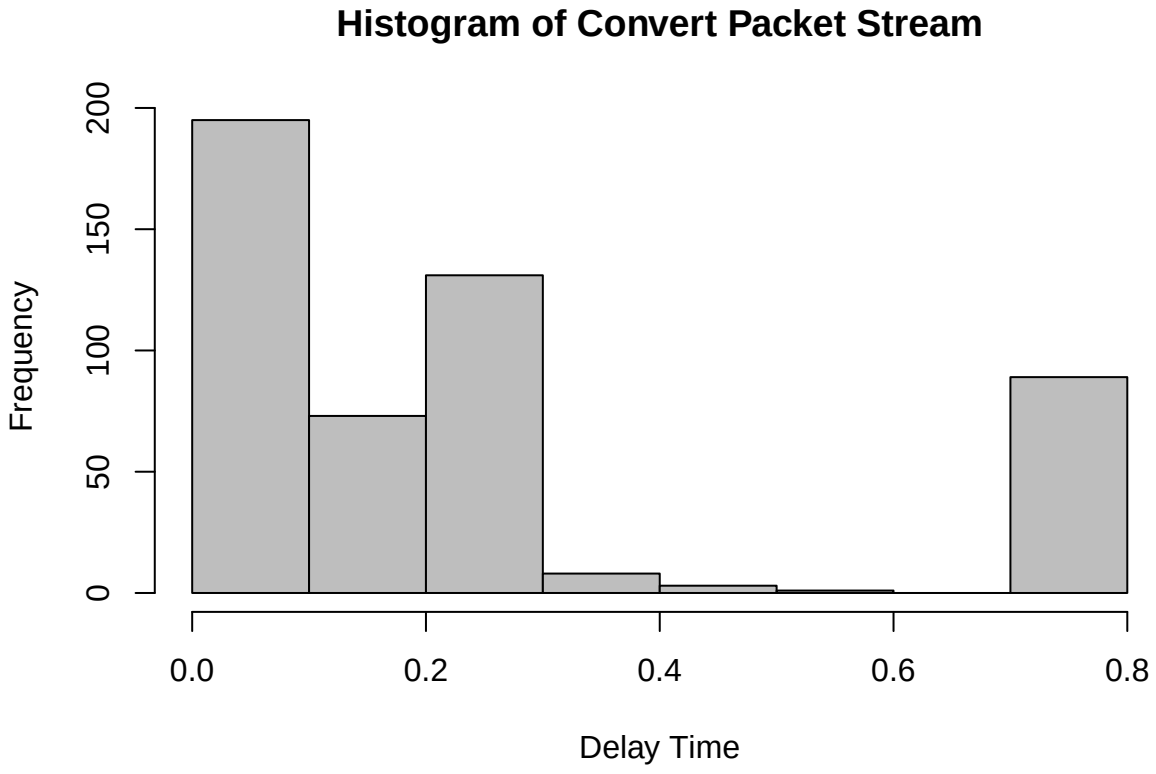
```
Traffic_data_orig <- read.csv("Traffic_data_orig.csv", header=TRUE)
message <- "this is a secret message"
raw <- charToRaw(message)
time = Traffic_data_orig$Time
num = as.integer(rawToBits(raw))
delays = numeric(length(time) - 1)
for (i in (1:(length(time) - 1))) {
  delays[i] = time[i+1] - time[i]
}
m = median(delays)
max = max(delays)
min = min(delays)

index = 1
bitlen = length(raw)*8
encrpt <- numeric(length(raw)*8)
for (i in (0:(length(raw)-1))) {
  for (j in 1:8) {
    if (num[i * 8 + j] == 0) {
      encrpt[index] = runif(1, min, m)
    }
    else {
      encrpt[index] = runif(1, m, max)
    }
    index = index + 1
    j = j - 1
  }
}

delays3 = delays
for (i in (1:bitlen)) {
  delays3[i] = encrpt[i]
}
```

Question 4

```
hist(delays, col='grey', xlab = 'Delay Time',
     main = 'Histogram of Overt Packet Stream')
```




```
hist(delays2, col='grey', xlab = 'Delay Time',
     main = 'Histogram of Convert Packet Stream')
```



I think Eva will not be suspicious because the trend of the Convert Packet Stream is similar to the Overt one.

Question 5

1. Instead of generating random number from m to max, and min to m, we can choose one of the existing one from m to max, and min to m.

```
Traffic_data_orig <- read.csv("Traffic_data_orig.csv", header=TRUE)
message <- "this is a secret message"
raw <- charToRaw(message)
time = Traffic_data_orig$Time
num = as.integer(rawToBits(raw))
delays = numeric(length(time) - 1)
for (i in (1:(length(time) - 1))) {
  delays[i] = time[i+1] - time[i]
}
m = median(delays)
max = max(delays)
min = min(delays)

index = 1
bitlen = length(raw)*8
encrpt <- numeric(length(raw)*8)
for (i in (0:(length(raw)-1))) {
  for (j in 1:8) {
    if (num[i * 8 + j] == 0) {
      encrpt[index] = sample(delays[which(delays >= min && delays <= m)])[1]
    }
    else {
      encrpt[index] = sample(delays[which(delays >= m && delays <= max)])[1]
    }
    index = index + 1
    j = j - 1
  }
}

delays4 = delays
for (i in (1:bitlen)) {
  delays4[i] = encrpt[i]
}
```

2. The packet stream is not predictable because the time that each packet is sent will not be a constant. Hence, this is not realistic for Alice to buffer up the packets.

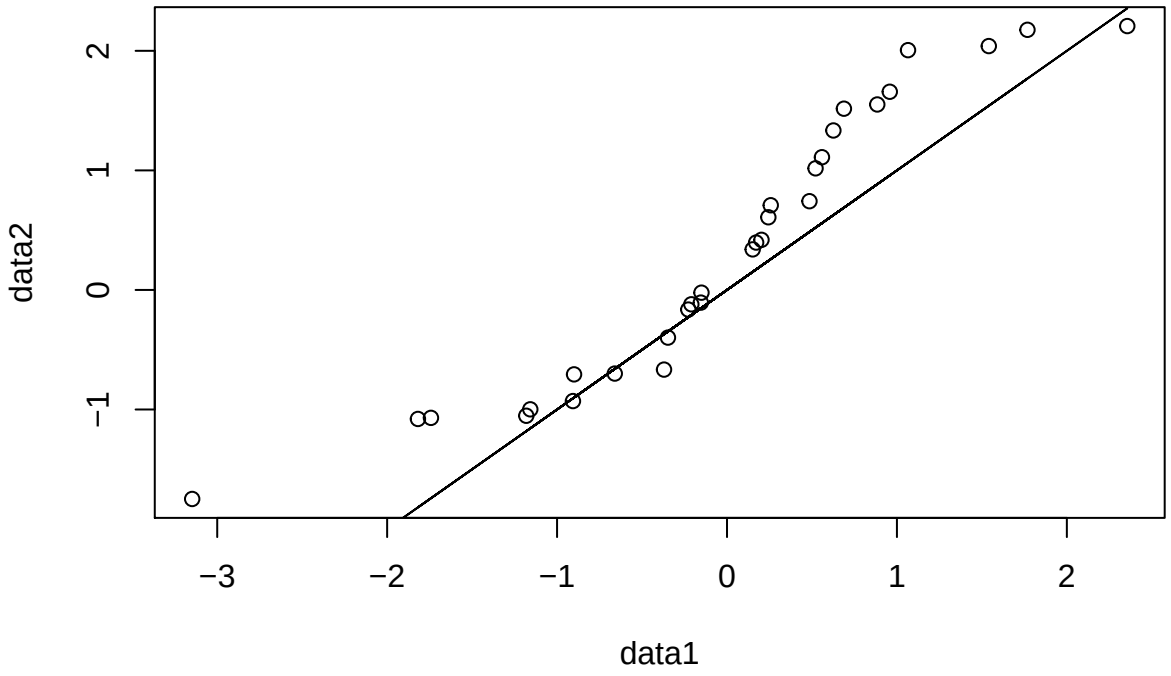
3. We assume that network has no effect so that we can use the given data as a standard to compare the results. Therefore, if the network does alter the inter-packet delays, then the algorithms and graphs we have worked will be slightly inaccurate. We can mitigate the effect of the changes of the interpacket delay by collecting a set of data, and using the average delaytime for each packet as the new standard.

Detection

Step 1

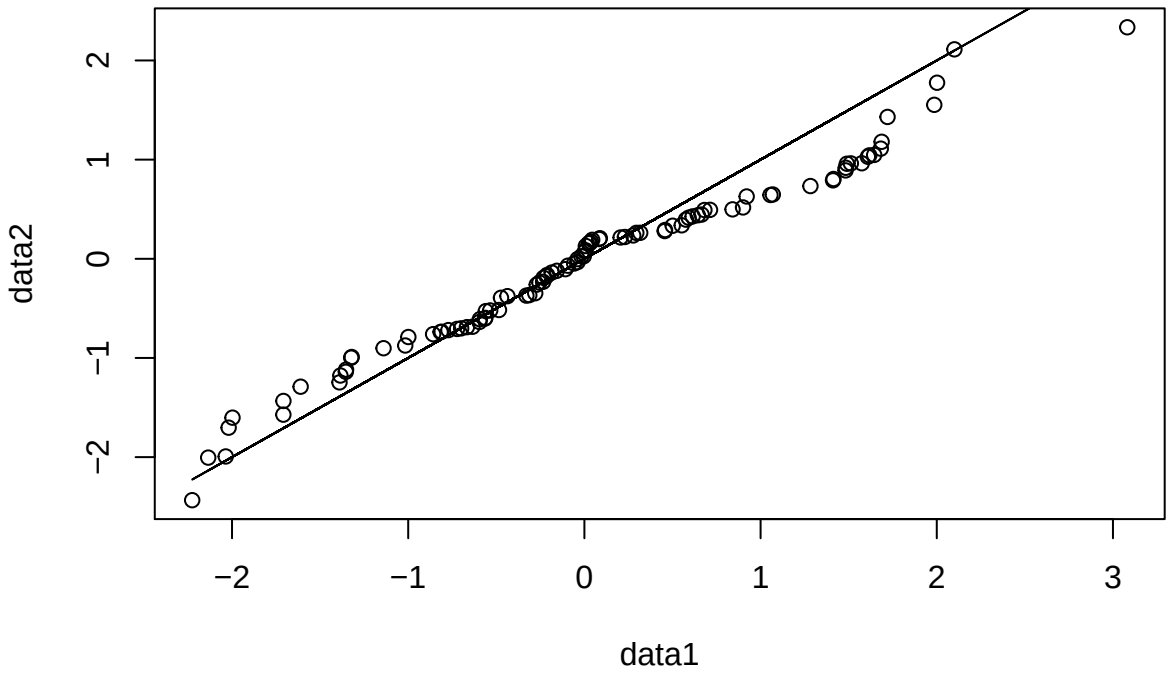
```
data1 <- rnorm(30)
data2 <- rnorm(30)
```

```
qqplot(data1, data2)
lines(data1, data1, type = 'l')
```

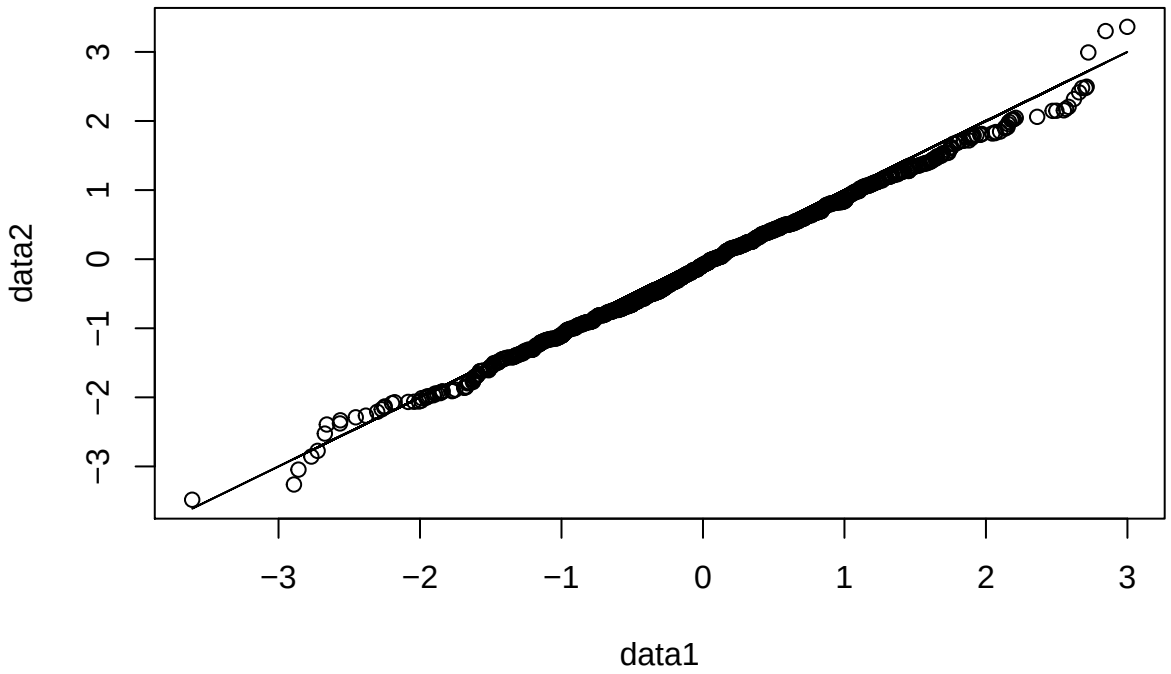


Step 2

```
data1 <- rnorm(100)
data2 <- rnorm(100)
qqplot(data1, data2)
lines(data1, data1, type = 'l')
```



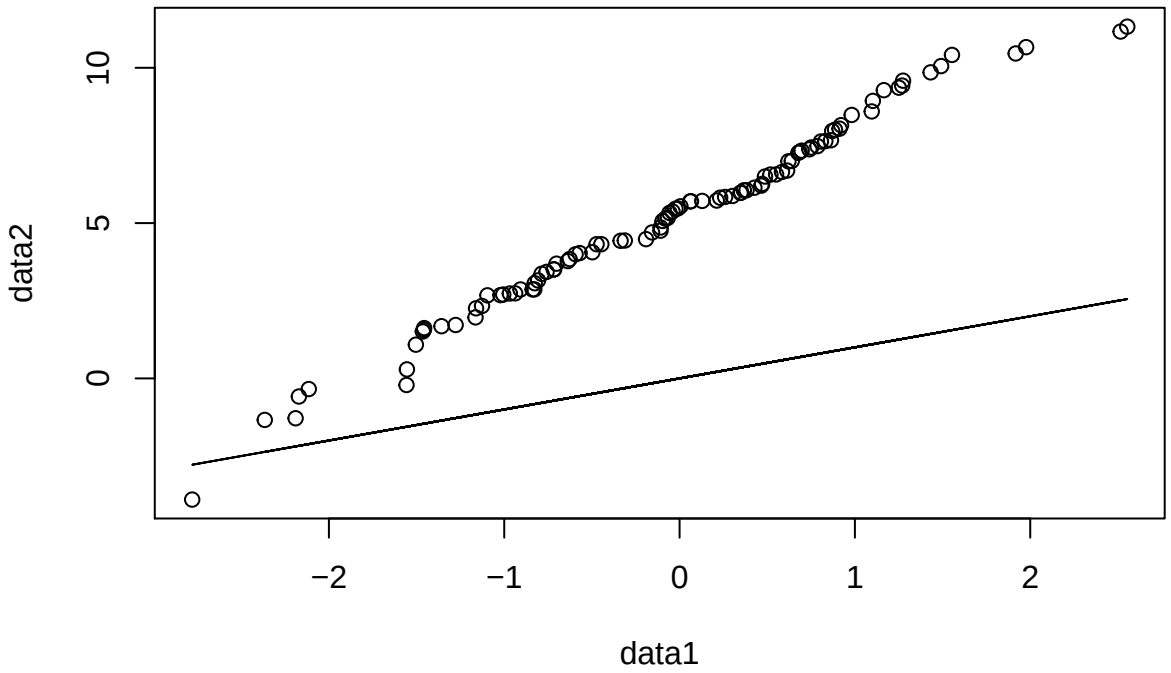
```
data1 <- rnorm(1000)
data2 <- rnorm(1000)
qqplot(data1, data2)
lines(data1, data1, type = 'l')
```



We have added line, y = x to represent the ideal case to see the trend clearly. When size n = 30, the plot has few points. The scattering points seems to be placed in a pattern that has the same trend with the ideal line. When size n = 100, more points are added comparing to n = 30. There is the same trend as the one with size n = 30. When size n = 1000, a more straight line is observed. The plot is almost the same with the ideal line. Comparing three data plots, it is obvious that the more points one graph has, the more clear how the trend is like. All plots share a similar slope which is close to the ideal line.

Step 3

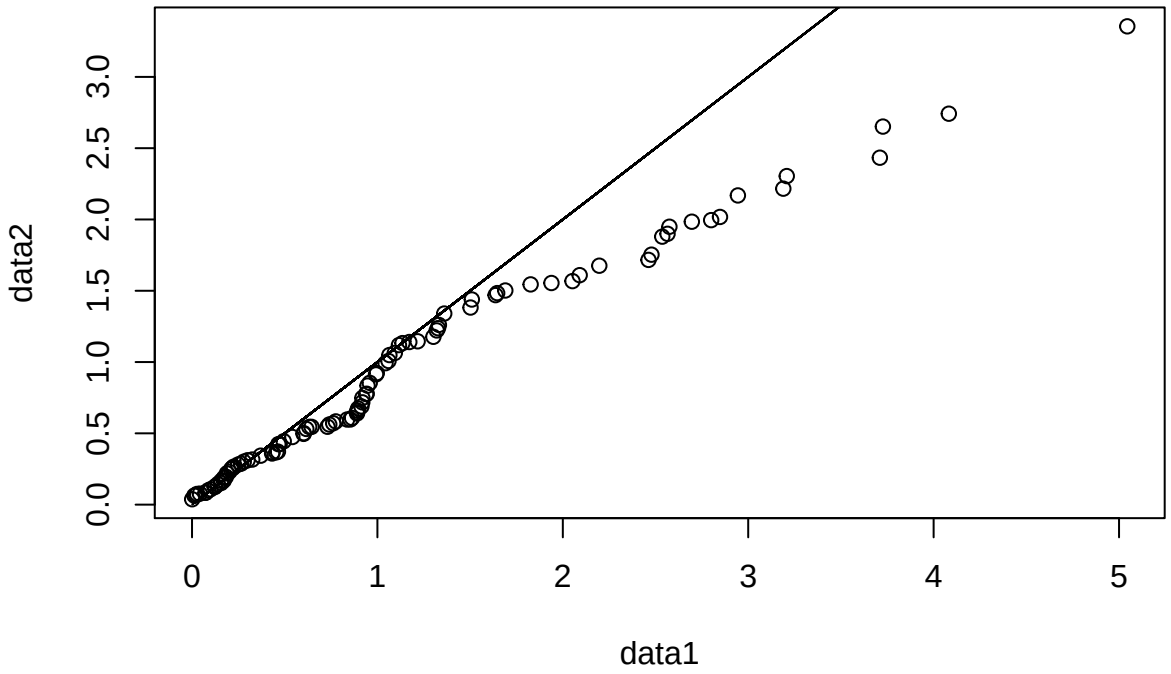
```
data1 <- rnorm(100)
data2 <- rnorm(100, mean = 5, sd = 3)
qqplot(data1, data2)
lines(data1, data1, type = 'l')
```



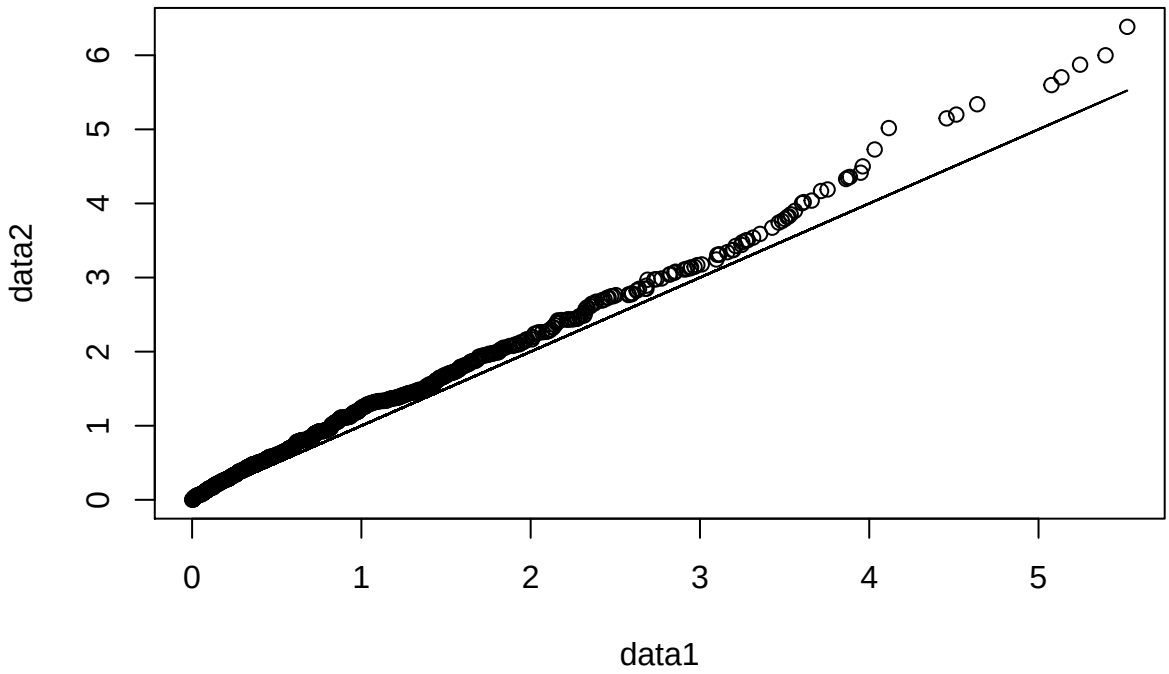
According to the graph, same trend is observed comparing to the plot from previous step. The slope is still positive. In previous plot, an increasing of 1 in x-axis results an increasing of 1 in y-axis. The slope for previous plot is close to 1 and slope for this graph is greater.

Step 4

```
data1 <- rexp(100)
data2 <- rexp(100)
qqplot(data1, data2)
lines(data1, data1, type = 'l')
```



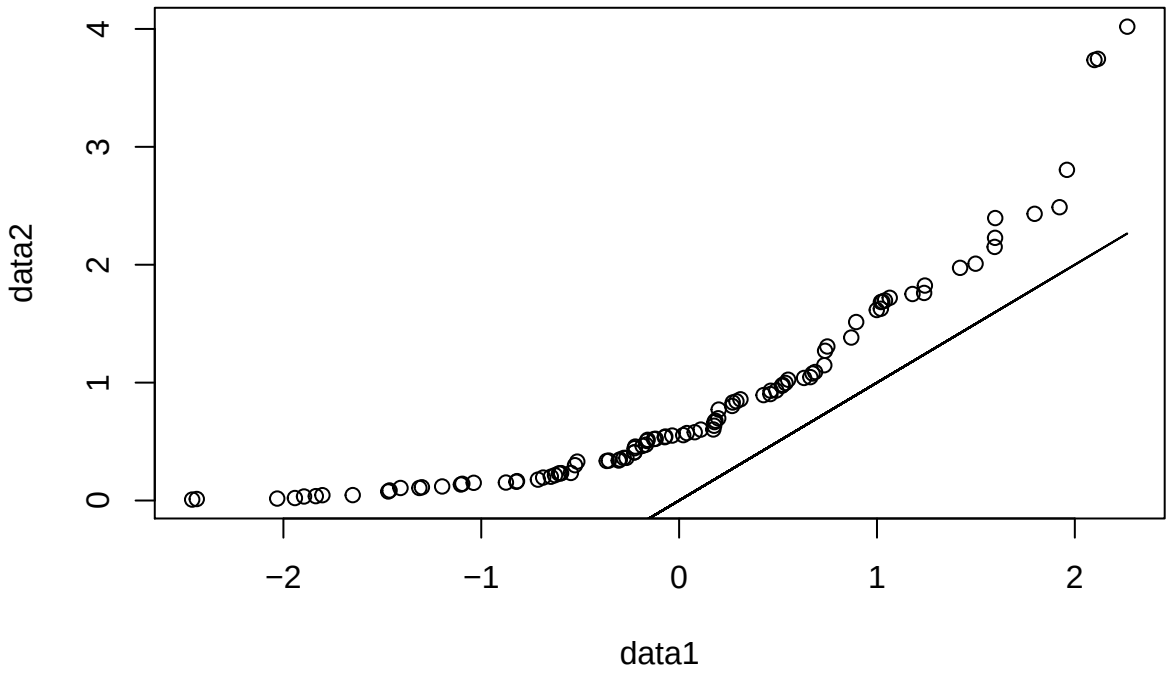
```
data1 <- rexp(1000)
data2 <- rexp(1000)
qqplot(data1, data2)
lines(data1, data1, type = 'l')
```



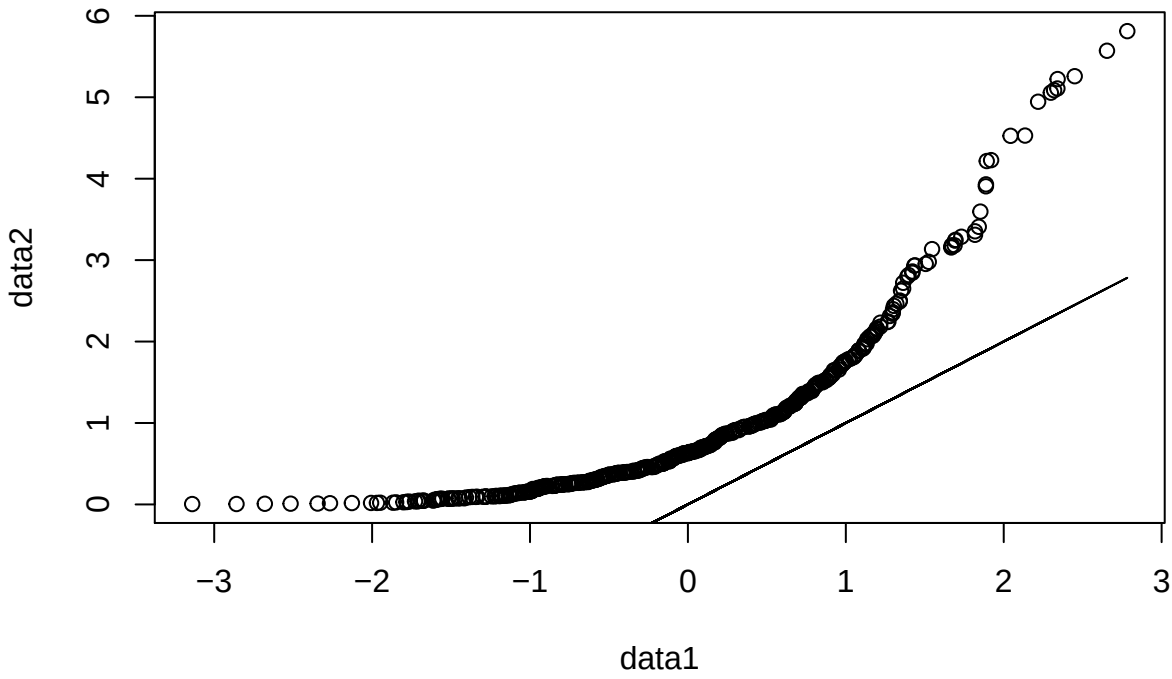
When n = 100, all the points follow the ideal line roughly. When n = 1000, the trend is easier to observe from the plot. Comparing both graph, they have the similar trend and similar slope; however, with a bigger size, two data sets are more consistent.

Step 5

```
data1 <- rnorm(100)
data2 <- rexp(100)
qqplot(data1, data2)
lines(data1, data1, type = 'l')
```



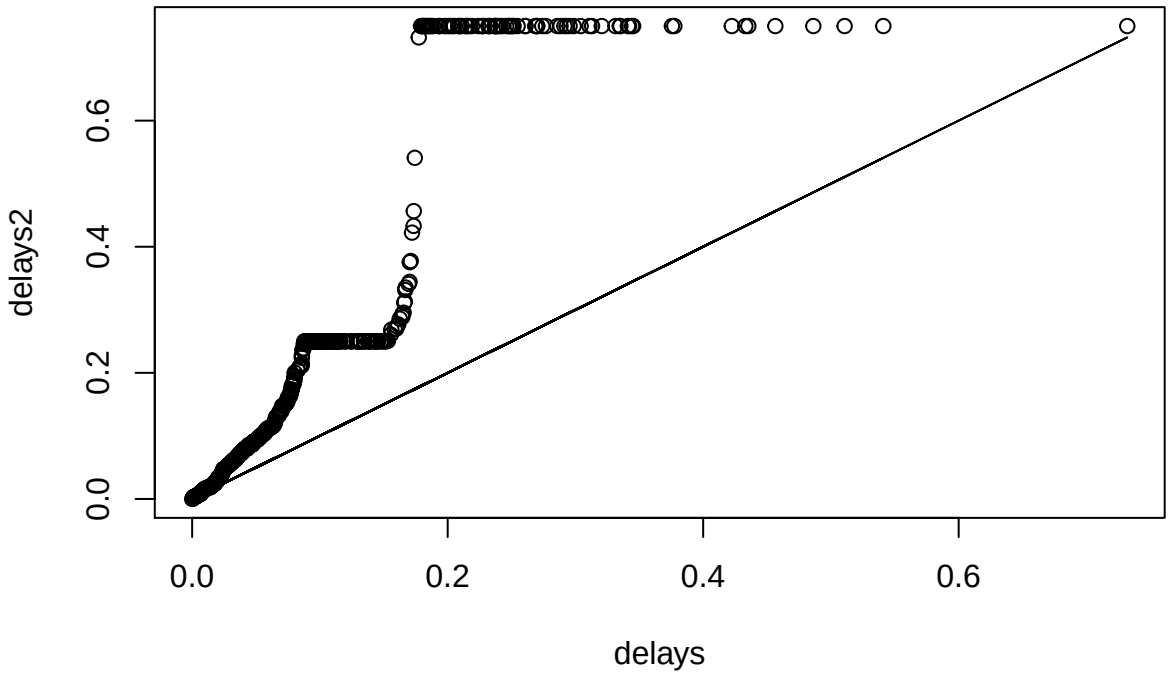

```
data1 <- rnorm(500)
data2 <- rexp(500)
qqplot(data1, data2)
lines(data1, data1, type = 'l')
```



For n = 100, the scattering points formed a curve. This curve tells that two data sets are exponentially related to each other. For n = 500, there is a more clear curve. Both plots, the relationship is still exponential. With different data sized, the trends are roughly the same. Two data sets are not identical because they are far away from the ideal line.

Step 6

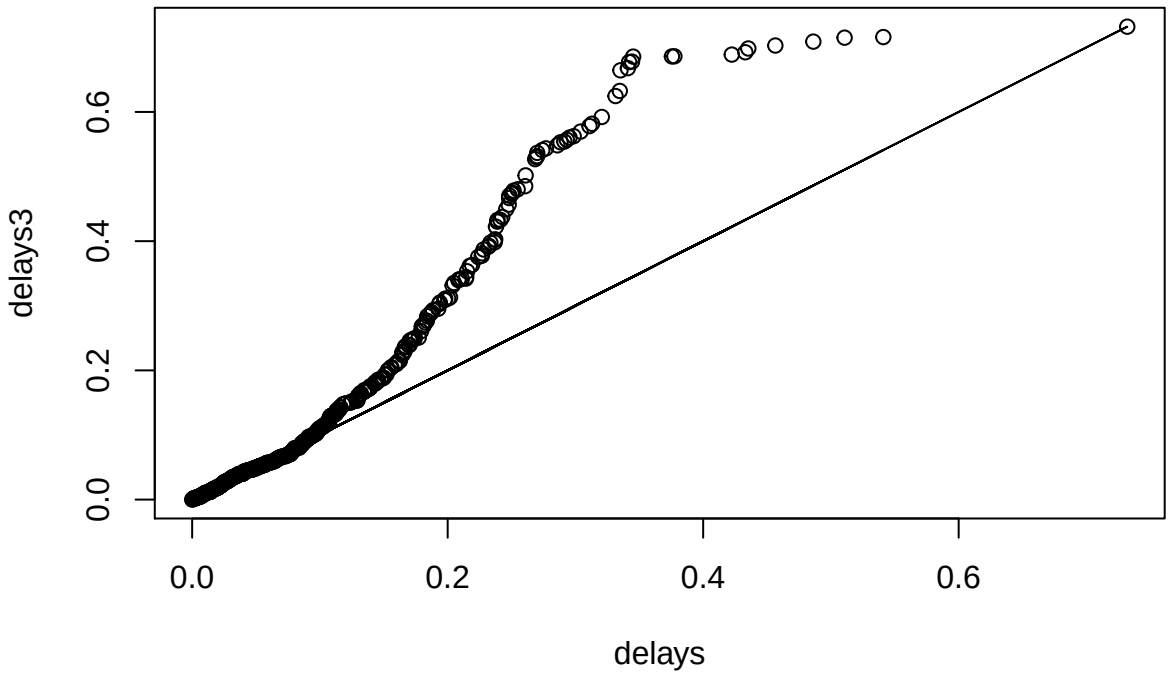
```
qqplot(delays, delays2)
lines(delays, delays, type = 'l')
```



A y = x line is displayed as an ideal line. There are two horizontal parts at y = 0.25 and 0.75. This plot indicates that the distribution of overt and covert package streams are not similar because the points are not far away from ideal line.

Step 7

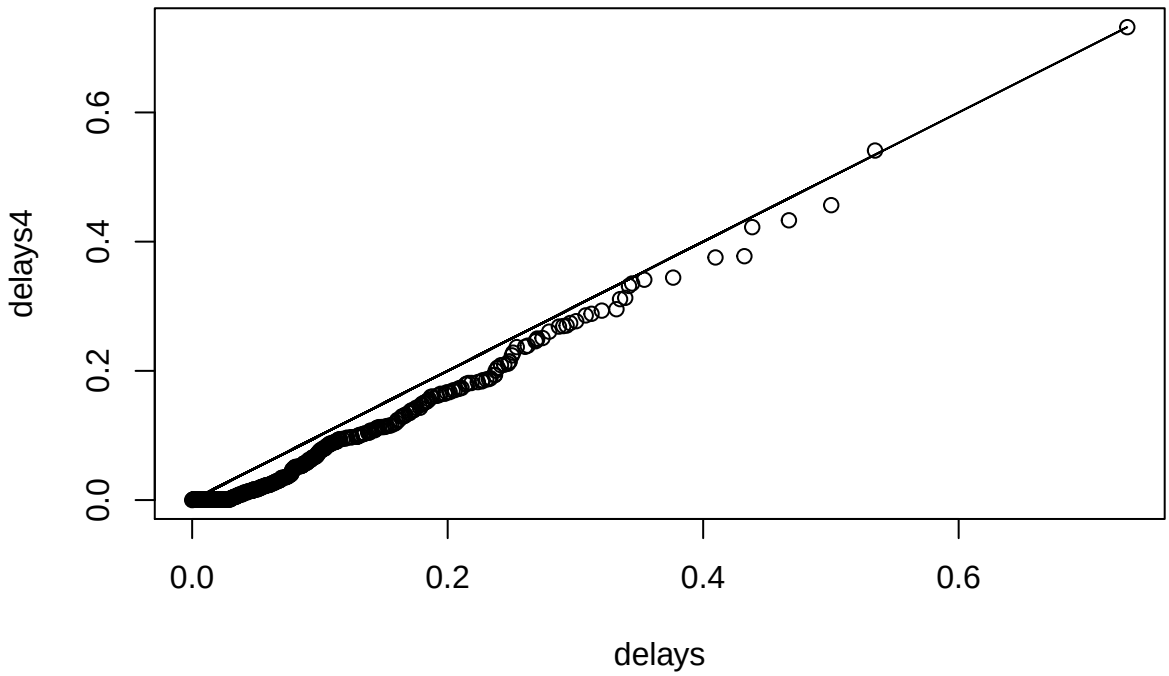
```
qqplot(delays, delays3)
lines(delays, delays, type = "l")
```



The tread of this plot is roughly a linear line. Only with this line, two data sets seem alike. However, it is far away from the ideal line, it still indicates that two data sets have a clear difference in distribution.

Step 8

```
qqplot(delays, delays4)
lines(delays, delays, type = 'l')
```



The scattering points forms a relatively straight line and most the points lies on line y = x. This plot indicates that the method we developed meets the distributions of delays.

Implementation

Implementation 1

```
generateMessage <- function (len) {
  message <- numeric(len)
  for (i in (1:len)){
    message[i] = sample(c(0,1), 1)
  }
  return(message)
}

generateTime <- function(ipd, len) {
  time <- numeric(len)
  time[1] = 0
  for (i in (2:len)){
    time[i] = time[i-1] + ipd[i-1]
  }
  return(time)
}

generateProb <- function (mlen, bufferNum) {
  bufferSize = 20
  currbuffer = bufferNum
  message <- generateMessage(mlen) # Generate the random bit pattern
  ipdSource <- rexp(50)
  ipdSend <- rexp(50)
  ipdEncrypt <- ipdSend
  ipdTime <- generateTime(ipdSource, 51)
  currTime = ipdTime[bufferNum]
  min = min(ipdSend)
  max = max(ipdSend)
  med = median(ipdSend)
  underflow = 0
  overflow = 0

  index = bufferNum + 1

    for (i in (1:mlen)) {
      # Generate a delay
      if (message[i] == 0) {
        delay = runif(1, min, med)
        ipdEncrypt[i] = delay
      } else {
        delay = runif(1, med, max)
        ipdEncrypt[i] = delay
      }
      currTime = currTime + delay # update time
      # Update the state of the buffer depending on the number of arrivals during that time.
      if (currTime <= ipdTime[index]) {
        currbuffer = currbuffer - 1
        if (currbuffer < 1){
          underflow = 1
```

```
          break
        }
      } else {
        currbuffer = currbuffer + 1
        if (currbuffer > bufferSize){
          overflow = 1
          break
        }
      }
      index = index + 1
    }
  return(c(underflow,overflow))
}

simulate <- function (m, i) {
  probsU <- numeric(1000)
  probsO <- numeric(1000)
  for (t in (1:1000)) {
    underflow = 0
    overflow = 0
    count = 0
    output <- generateProb(m, i)
    underflow = underflow + output[1]
    overflow = overflow + output[2]
    count = count + 1
    probsU[t] = underflow/count
    probsO[t] = overflow/count
  }
  return(c(mean(probsU), mean(probsO)))
}

simulate(16,2)
```

```
## [1] 0.393 0.000
```

```
simulate(16,6)
```

```
## [1] 0.203 0.310
```

```
simulate(16,10)
```

```
## [1] 0.101 0.583
```

```
simulate(16,14)
```

```
## [1] 0.070 0.676
```

```
simulate(16,18)
```

```
## [1] 0.000 0.814
```

```
simulate(32,2)
```

```
## [1] 0.423 0.557
```

```
simulate(32,6)
```

```
## [1] 0.251 0.711
```

```
simulate(32,10)
```

```
## [1] 0.180 0.785
```

```
simulate(32,14)
```

```
## [1] 0.142 0.818
```

```
simulate(32,18)
```

```
## [1] 0.079 0.888
```

Implementation 2

```
generateMessage <- function (len) {
  message <- numeric(len)
  for (i in (1:len)){
    message[i] = sample(c(0,1), 1)
  }
  return(message)
}

generateTime <- function(ipd, len) {
  time <- numeric(len)
  time[1] = 0
  for (i in (2:len)){
    time[i] = time[i-1] + ipd[i-1]
  }
  return(time)
}

generateProb <- function (mlen, bufferNum) {
  bufferSize = 20
  currbuffer = bufferNum
  message <- generateMessage(mlen) # Generate the random bit pattern
  ipdSource <- runif(50, 0, 1)
  ipdSend <- runif(50, 0, 1)
  ipdEncrypt <- ipdSend
  ipdTime <- generateTime(ipdSource, 51)
  currTime = ipdTime[bufferNum]
  min = min(ipdSend)
  max = max(ipdSend)
  med = median(ipdSend)
  underflow = 0
  overflow = 0

  index = bufferNum + 1

    for (i in (1:mlen)) {
      # Generate a delay
      if (message[i] == 0) {
        delay = runif(1, min, med)
        ipdEncrypt[i] = delay
      } else {
```

```
        delay = runif(1, med, max)
        ipdEncrypt[i] = delay
      }
      currTime = currTime + delay # update time
      # Update the state of the buffer depending on the number of arrivals during that time.
      if (currTime <= ipdTime[index]) {
        currbuffer = currbuffer - 1
        if (currbuffer < 1){
          underflow = 1
          break
        }
      } else {
        currbuffer = currbuffer + 1
        if (currbuffer > bufferSize){
          overflow = 1
          break
        }
      }
      index = index + 1
    }
  return(c(underflow,overflow))
}

simulate <- function (m, i) {
  probsU <- numeric(1000)
  probsO <- numeric(1000)
  for (t in (1:1000)) {
    underflow = 0
    overflow = 0
    count = 0
    output <- generateProb(m, i)
    underflow = underflow + output[1]
    overflow = overflow + output[2]
    count = count + 1
    probsU[t] = underflow/count
    probsO[t] = overflow/count
  }
  return(c(mean(probsU), mean(probsO)))
}

simulate(16,2)
```

```
## [1] 0.649 0.000
```

```
simulate(16,6)
```

```
## [1] 0.449 0.164
```

```
simulate(16,10)
```

```
## [1] 0.330 0.305
```

```
simulate(16,14)
```

```
## [1] 0.236 0.406
```

```
simulate(16,18)
```

```
## [1] 0.000 0.562
```

```
simulate(32,2)
```

```
## [1] 0.691 0.259
```

```
simulate(32,6)
```

```
## [1] 0.528 0.377
```

```
simulate(32,10)
```

```
## [1] 0.464 0.451
```

```
simulate(32,14)
```

```
## [1] 0.409 0.522
```

```
simulate(32,18)
```

```
## [1] 0.295 0.637
```

Implementation 3

```
generateTime <- function(ipd, len) {
  time <- numeric(len)
  time[1] = 0
  for (i in (2:len)){
    time[i] = time[i-1] + ipd[i-1]
  }
  return(time)
}

generateProb1 <- function (bufferNum) {
  bufferSize = 20
  currbuffer = bufferNum
  ipdSource <- rexp(50)
  ipdSend <- rexp(50)
  ipdTime <- generateTime(ipdSource, 51)
  currTime = ipdTime[bufferNum]
  min = min(ipdSend)
  max = max(ipdSend)
  med = median(ipdSend)
  underflow = 0
  overflow = 0

  index = bufferNum + 1

    for (i in (1:32)) {
      currTime = currTime + ipdSend[i] # update time
      # Update the state of the buffer depending on the number of arrivals during that time.
      if (currTime <= ipdTime[index]) {
        currbuffer = currbuffer - 1
        if (currbuffer < 1){
```

```
          underflow = 1
          break
        }
      } else {
        currbuffer = currbuffer + 1
        if (currbuffer > bufferSize){
          overflow = 1
          break
        }
      }
      index = index + 1
    }
  return(c(underflow,overflow))
}

generateProb2 <- function (bufferNum) {
  bufferSize = 20
  currbuffer = bufferNum
  ipdSource <- runif(50, 0, 1)
  ipdSend <- runif(50, 0, 1)
  ipdTime <- generateTime(ipdSource, 51)
  currTime = ipdTime[bufferNum]
  min = min(ipdSend)
  max = max(ipdSend)
  med = median(ipdSend)
  underflow = 0
  overflow = 0

  index = bufferNum + 1

    for (i in (1:32)) {
      currTime = currTime + ipdSend[i] # update time
      # Update the state of the buffer depending on the number of arrivals during that time.
      if (currTime <= ipdTime[index]) {
        currbuffer = currbuffer - 1
        if (currbuffer < 1){
          underflow = 1
          break
        }
      } else {
        currbuffer = currbuffer + 1
        if (currbuffer > bufferSize){
          overflow = 1
          break
        }
      }
      index = index + 1
    }
  return(c(underflow,overflow))
}

simulate <- function (i) {
  probsUe <- numeric(1000)
```

```
  probsOe <- numeric(1000)

  for (t in (1:1000)) {
    underflow = 0
    overflow = 0
    count = 0
    output <- generateProb1(i)
    underflow = underflow + output[1]
    overflow = overflow + output[2]
    count = count + 1
    probsUe[t] = underflow/count
    probsOe[t] = overflow/count
  }
  return(c(mean(probsUe), mean(probsOe)))
}

simulate2 <- function (i) {
  probsUu <- numeric(1000)
  probsOu <- numeric(1000)
  for (t in (1:1000)) {
    underflow = 0
    overflow = 0
    count = 0
    output <- generateProb2(i)
    underflow = underflow + output[1]
    overflow = overflow + output[2]
    count = count + 1
    probsUu[t] = underflow/count
    probsOu[t] = overflow/count
  }
  return(c(mean(probsUu), mean(probsOu)))
}

overexp <- numeric(5)
underexp <- numeric(5)
overunif <- numeric(5)
underunif <- numeric(5)
temp = simulate(2)
underexp[1] = temp[1]
overexp[1] = temp[2]
temp = simulate(6)
underexp[2] = temp[1]
overexp[2] = temp[2]
temp = simulate(10)
underexp[3] = temp[1]
overexp[3] = temp[2]
temp = simulate(14)
underexp[4] = temp[1]
overexp[4] = temp[2]
temp = simulate(18)
underexp[5] = temp[1]
overexp[5] = temp[2]
```

```
temp = simulate2(2)
underunif[1] = temp[1]
overunif[1] = temp[2]
temp = simulate2(6)
underunif[2] = temp[1]
overunif[2] = temp[2]
temp = simulate2(10)
underunif[3] = temp[1]
overunif[3] = temp[2]
temp = simulate2(14)
underunif[4] = temp[1]
overunif[4] = temp[2]
temp = simulate2(18)
underunif[5] = temp[1]
overunif[5] = temp[2]

#exponential
#underflow bounds
c(min(underexp),max(underexp))
```

```
## [1] 0.302 0.692
```

```
#overflow bounds
c(min(overexp),max(overexp))
```

```
## [1] 0.264 0.641
```

```
#uniform
#underflow bounds
c(min(underunif),max(underunif))
```

```
## [1] 0.328 0.702
```

```
#overflow bounds
c(min(overunif),max(overunif))
```

```
## [1] 0.240 0.597
```

Implementation 4

One way to deal with overflows and underflows is to prevent it from happening. Using the gambler's ruin and the IPD distribution, we can find the bounds that both overflows and underflows are not likely happening.