Implementação de Técnicas de Detecção de Erros

Guilherme Branco, Lucas Amaral, Daniella Angelos
Pedro Henrique Lima, Francisco Anderson

1 Introdução e objetivos

Na transmissão de dados, erros podem ocorrer. A camada de link de dados deve garantir uma transmissão livre de erros que, geralmente, ocorrem na camada física. Para isso, existem vários métodos.

No nosso trabalho, abordamos quatro desses método: MD5, SHA-1, Hamming code e CRC-8. O MD5 e o SHA-1 são funções de hash que usamos para a detecção de erros. Com elas, é possível identificar erros na transmissão, mas não é possível corrigir dados errados. O Hamming code permite a correção de 1 bit no dado transmitido, caso tenha detectado algum problema.

O objetivo do trabalho é utilizar os 4 algortimos citados acima para detecção de erros, onde a implementação do código de Hamming e do CRC-8 será feita por nós. Também é parte do objetivo mostrar as principais vantagens e desvantagens de cada um dos algoritmos em diferentes contextos (flipando bits ímpares, pares e de maneira aleatória).

2 Algoritmos de detecção

2.1 MD5

O MD5 (Message-Digest algorithm 5) é um algortimo de hash criptográfico que gera um valor de 128 bits. Ele já foi bastante usado para criptografia, porém, após algumas falhas serem descobertas, o MD5 foi considerado criptograficamente quebrado e impróprio para utilização.

Em nossa aplicação, utilizamos o MD5 para a detectar erros na transmissão de um dado. Por exemplo, um cliente A envia como payload o dado original concatenado com o hash do dado gerado pelo MD5. O cliente B, ao receber o payload, separa o dado do hash gerado, gera o hash do dado recebido com a função do MD5 e compara com o que foi recebido do cliente A. Se for diferente, houve erro na transmissão do dado. Uma grande desvantagem de usar o MD5 para a detecção de erros é o enorme tamanho do payload de cada dado transmitido.

2.2 SHA-1

O SHA-1 (Secure Hash Algorithm 1), assim como o MD5, é uma função de hash criptográfica, onde a saída gerada tem 160 bits. Esse algoritmo foi projetado pela NSA e foi considerado o sucessor do MD5. Após um tempo, também foram encontradas falhas no SHA-1.

O uso do SHA-1 em nossa aplicação é exatamente o mesmo do MD5, detecção de erros. O payload da mensagem será o dado que o cliente deseja enviar concatenado com a saída da função de hash. Do mesmo modo do MD5, o cliente que receber a mensagem separa o dado do hash gerado, aplica a função do SHA-1 no dado e compara a nova saída com o hash que veio com a mensagem. Assim como o MD5, o uso do SHA-1 para identificação de erros apresenta como desvantagem um payload muito grande.

2.3 Hamming

O código de Hamming, é um algoritmo de detecção e correção de erros, podendo detectar até dois bits e corrigir um. É muito usado em telecomunicações devido a sua simplicidade e eficiência, pois por mais que possa corrigir apenas um bit os canais normalmente não tem ruído o suficiente para precisar de muito mais correção.

O algoritmo utiliza uma quantidade p de bits de paridade que são obtidos a partir da quantidade de bits de dados que deverão ser codificados, a partir da fórmula: $2^p - 1$ podemos codificar até n bits de dados, portanto para dados de 32 bits são necessários 6 bits de paridade. A codificação em si é dada a partir da composição dos índices dos bits de dados por potências de 2. Após encontrar a composição é colocado nos bits de paridade alguma operação matemática sobre o conteúdo dos bits de dados, a mais simples seria verificar se a quantidade de bits é par ou ímpar.

2.4 CRC-8

O código CRC (cyclic redundancy check), conhecido também como código polinomial, é um método para identificação de erros utilizado em redes digitais para detectar alterações acidentais nos dados.

Um CRC de n-bit utiliza um polinômio de grau n que será o divisor na divisão polinomial. O dividendo será a mensagem, onde o polinômio representado por ela é dado pela sua cadeia de bits. Por exemplo, se a mensagem é um número inteiro igual a 17 (10001 em binário), o polinômio representado por ela é $1 \times x^4 + 0 \times x^3 + 0 \times x^2 + 0 \times x^1 + 1 \times x^0$. De maneira reversa, podemos tratar o polinômio divisor como uma cadeia de bits. O quociente desta divisão será descartado, e o resto será usado para a verificação de erros.

De maneira simplificada, ao usarmos um CRC-n, após fazermos a divisão polinomial do dado com o polinômio divisor, anexamos o resto da divisão (que terá n bits) ao fim do dado original. Isto será o nosso payload. Ao receber o payload, separamos o dado dos n bits que foram anexados após a divisão. Fazemos uma outra divisão do dado pelo polinômio

divisor. Por fim, comparamos o resto da divisão com os n bits anexados no payload. Se forem diferentes, então o dado contém um erro.

3 Detalhes de Implementação

3.1 Cliente e Servidor

Foi utilizado a linguagem Python, simplesmente devido a facilidade de utilização de sockets na linguagem. Como especificado pelo trabalho o servidor deve manter uma conexão aberta com todos os clientes que estão conectados a ele, para tal foi utilizado uma lista que guarda o IP e a porta de cada cliente. Além disso é feito uma fila FIFO para que se possa enviar os dados numa certa ordem para cada cliente. No servidor também tem a opção de flipagem de bits, onde pode-se não flipar nenhum bit do payload, flipar somente bits pares, somente bits impares ou flipar aleatoriamente baseado numa porcentagem p. Para que haja tanto o recebimento, envio e o "check-in"de clientes teve-se a necessidade de se utilizar três threads, uma para cada tarefa.

```
1  def esperar_clientes():
2      global num_clients
3      global client_list
4      global addr
5
6      while True:
7          c_aux, addr_aux = s.accept()
8          client_list.append( (True, c_aux) ) # True diz que o cliente ainda est
               á vivo
9          addr.append(addr_aux)
10
11         num_clients+=1
12         id_buf = struct.pack("@i", (num_clients - 1)) #Codifica o ID do
               cliente para mandar pelo socket
13         c_aux.send(id_buf)
```

```
1  def receber_dados():
2      global num_clients
3      global client_list
4      global addr
5      global to_send
6      data = b''
7      while True:
8          for i in range(len(client_list)):
9              mutex.acquire()
10             (client_online, com) = client_list[i]
11             mutex.release()
12             if client_online:   # Este IF e os Try/catch são tolerancia a
                   falhas, caso o cliente caia
13
14                 try:
```

```
15                      sz_buf = com.recv(4)
16                  except socket.error:
17                      #client_pair[0] = False
18                      mutex.acquire()
19                      client_list[i] = (False, com)
20                      mutex.release()
21                      continue
22
23                  size = 0
24
25                  try:
26                      size = struct.unpack("@i", sz_buf)[0]
27                  except struct.error as msg:
28                      continue
29
30                  if size == 0:
31                      continue
32
33                  try:
34                      data = com.recv(size)
35                  except socket.error:
36                      mutex.acquire()
37                      client_list[i] = (False, com)
38                      mutex.release()
39                      continue
40
41                  if data == b'':
42                      continue
43                  data_loaded = json.loads(data) #De-serializa o data recebido
44                  bitfield_msg = int_to_bit32(data_loaded['msg']) #passa o
                        inteiro da mensagem para bitfield
45                  # Passa o checksum pra bits:
46                  vetor_bits_check = int_to_bit32(data_loaded['check'])
47                  for bitfield in vetor_bits_check:
48                      bitfield_msg.append(bitfield)
49                      pass
50                  #print vetor_bits_check
51                  if flipar == 1:
52                      #como usar o de porcentagem aleatoria
53                      porcentagem = random.uniform(0,1)
54                      bitfield_msg = flipar_paleatorio(bitfield_msg,porcentagem)
55                  elif flipar == 2:
56                      #como usar o flip de pares
57                      bitfield_msg = flipar_pares(bitfield_msg)
58                  elif flipar == 3:
59                      #como usar o flip de impares
60                      bitfield_msg = flipar_impares(bitfield_msg)
61
62                  #Passa o bitfield para inteiro
63                  _mensagem = bitfield_msg[0:32]
64                  _check = bitfield_msg[32:]
65
```

```
66                  data_loaded['msg'] = bit32_to_int(_mensagem)
67                  data_loaded['check'] = bit32_to_int(_check)
68                  # Coloca o dado recebido em uma fila sincrona de dados para
                        serem enviados
69                  to_send.put(data_loaded)
70                  # FIM receber_dados
71
72
73  def enviar_dados():
74      global to_send
75      while True:
76          data_loaded = to_send.get()
77          data = json.dumps(data_loaded)
78          dest = int(data_loaded['dest'])
79
80          if len(client_list) > dest: # Dropa o pacote se o cliente nao existir
81              mutex.acquire()
82              client_pair = client_list[dest]
83              client_online, client = client_pair
84              mutex.release()
85
86              if client_online: # Este IF e os Try/catch são tolerancia a falhas
                    , caso o cliente caia
87                  print "Enviando para: " + str(dest)
88                  n = len(data)
89                  sz_buf = struct.pack("@i", n)  # converte N em 4 bytes, para
                        enviar pelo socket
90
91                  try:
92                      client.send(sz_buf) # envia o tamanho do dado antes
93                      client.send(data)
94                  except socket.error:
95                      mutex.acquire()
96                      client_list[dest] = (False, client)
97                      mutex.release()
98                      continue
99
100         to_send.task_done()
101         # FIM enviar_dados
```

Já o código do cliente utiliza somente duas threads, uma para recebimento de mensagens e uma para envio e é feito uma criptografia nas mensagens que pode ser através de quatro algoritmos, SHA1, MD5, Hamming ou CRC-8. E no recebimento é feito uma detecção de erro para qualquer um dos algoritmos, afim de checar se a mensagem foi recebida corretamente como esperado.

```
1   def receber():
2       global my_id
3       global s
4
5       while True:
```

```
6           sz_buf = 0
7
8           try:
9               sz_buf = s.recv(4)
10              size = struct.unpack("@i", sz_buf)[0]
11              data = s.recv(size)
12          except socket.error as msg:
13              print "Servidor não encontrado!"
14              return
15          except struct.error as msg:
16              return
17
18          if data == b'':
19              continue
20          data_loaded = json.loads(data)
21
22          # Deteccao de erros:
23          payload_recv = data_loaded['msg']
24          check_recv = data_loaded['check']
25          if(cript==0): #SHA1
26              check_sum = generate_sha1(payload_recv)
27          elif(cript==1): #MD5
28              check_sum = generate_md5(payload_recv)
29          elif(cript==2): #Hamming
30              _recv =  bin(check_recv)[2:].zfill(38)
31              #print _recv
32              dado_bin = disjunta(_recv)
33              dado_int = bin_to_int(dado_bin)
34              check_sum = generate_hamming(dado_int)
35              #check_sum = generate_hamming(payload_recv)
36          elif(cript==3):#CRC-8
37              check_sum = generate_crc8(payload_recv)
38              _recv =  bin(check_recv)[2:].zfill(32)
39              _recv = "00000000000000000000000" + _recv[24:]
40              check_recv = bin_to_int(_recv)
41
42
43          if check_recv == check_sum:
44              print str(data_loaded['dest']) + "-> Recebido de: " + str(
                    data_loaded['source']) + \
45              " : " + str(data_loaded['msg']) + " : " + str(check_sum)
46              pass
47          else:
48              print "Mensagem recebida com erro ", check_sum, " ", check_recv
49              pass
50
51
52
53
54  # FIM receber()
55  def enviar():
56      global num_clients
```

```
57      global destino
58      i = 0
59      check = ""
60      while True:
61          if(cript==0):#SHA1
62              check = generate_sha1(i)
63          elif(cript==1):#MD5
64              check = generate_md5(i)
65          elif(cript==2):#Hamming
66              check = generate_hamming(i)
67          elif(cript==3):#CRC-8
68              check = generate_crc8(i)
69          data = {'source': my_id, 'dest': destino, 'msg': i, 'check' : check}
70          data_string = json.dumps(data) # serialize data para mandar por socket
71          n = len(data_string)
72
73          sz_buf = struct.pack("@i", n)  # converte N em 4 bytes, para enviar
                pelo socket
74
75          time.sleep(0.01)
76
77          try:
78              s.send(sz_buf)
79              s.send(data_string)
80          except socket.error as msg:
81              print "Servidor não encontrado!"
82              return
83
84          # todo: gerar payload de envio
85          i += 1
86          pass
87
88  # FIM enviar()
```

3.2 Algoritmos

O algoritmo do código de Hamming foi implementado como descrito anteriormente, porém ao invés de utilizar a quantidade par ou impar como paridade foi feito um xor entre o conteudo dos índices que compõe cada bit de paridade.

```
1  def generate_hamming(payload):
2    dados = payload
3    dados = bin(dados)[2:].zfill(32)
4    qtd_paridade = 6
5    #qtd_paridade = int(math.ceil(math.log(len(dados)+1,2)))
6    hamming = (qtd_paridade+len(dados))*[0]##32 bits de dados + 6 bits de
         paridade
7    hamming = junta(dados,hamming)
8    i=0
9    while(i<len(hamming)):
10     if(not is_paridade(i)):
11       binario = bin(i+1)[2:].zfill(32)
12       lista = get_list(binario)
13       for elemento in lista:
14         hamming[elemento-1] = hamming[elemento-1] ^ hamming[i]
15     i+=1
16   return bit32_to_int(hamming)
```

Por fim, o CRC-8 foi implementando usando como divisor o polinômio $x^8 + x^7 + x^4 + x^3 + x + x^0$, que representa a seguinte cadeia de 9 bits: 110011011. Para fazer a divisão do dado (que tem 32 bits) pelo polinômio, primeiro concatenamos 8 bits ao final dos 32 bits do dado, depois fazemos o xor bit a bit entre o dado e o polinômio. Quando o quociente (os primeiros 32 bits) chegar a zero, ele é descartado e retornamos o resto da divisão (os últimos 8 bits) que será usado para a verificação de erros.

```
1  def generate_crc8(payload):
2    binario = bin(payload)[2:].zfill(32)
3    resto = "00000000"
4    quociente = binario + resto
5    # polinomio: x^8 + x^7 + x^4 + x^3 + x + 1
6    polinomio = "110011011"
7
8    i = 0
9    so_zero = bin(0)[2:].zfill(32)
10   while(quociente[0:32] != so_zero):
11     x = i
12     for j in range(0, 9):
13       x2 = ord(quociente[x])
14       j2 = ord(polinomio[j])
15       aux = (x2 ^ j2) + 48
16       quociente = quociente[:x] + chr(aux) + quociente[x+1:]
17
18       x+=1
19     while( (quociente[i] == '0') & (i < 32) ):
20       i+=1
```

```
21    resto = quociente[32:40]
22
23    _resto = []
24    for c in resto:
25      _resto.append(int(c))
26      pass
27
28    return bit32_to_int(_resto)
```

3.3 Bibliotecas utilizadas

Python traz diversas facilidades para implementação de soluções, entre elas a diversidade de bibliotecas prontas para serem utilizadas. Segue abaixo uma lista das que usamos no nosso projeto.

- *socket:* interface de comunicação
- *threading:* disparar threads de cada lado da comunicação com funções diferentes
- *md5:* biblioteca utilizada para gerar o md5
- *sha:* biblioteca utilizada para gerar o sha1
- *json:* funções de serialização e desserialização de dados
- *time:* utilizada função de *sleep()* que faz uma thread esperar um tempo determinado
- *random:* utilizadas funções de randomização

3.4 Executando a simulação

Criamos um arquivo de configuração, *config_file.json*, que contém detalhes importantes e instruções específicas de como cada indivíduo de uma conexão estabelecida deve se comportar. Abaixo, um exemplo de configuração:

```
{
    "server" : {
        "flip" : 0,
        "port" : 12345
    },
    "clients" : {
        "algoritmo" : 1,
        "destinos" : [2,1,0],
        "server_port" : 12345
    }
}
```

Onde:

- Flip: Modo de flipar os bits, pode ser:
  * 0 = Não flipar
  * 1 = Aleatório
  * 2 = Só os pares
  * 3 = Só os ímpares
- Port e server_port são o mesmo valor, pra dizer qual a porta do servidor
- Algoritmo é um número representando o algoritmo utilizado para se realizar o checksum, onde:
  * 0 = SHA1
  * 1 = MD5
  * 2 = Hamming
  * 3 = CRC
- Destino é o campo em que cada cliente irá procurar, por sua id, o cliente com quem deseja se comunicar

As partes "server"e "clients"são independentes e utilizadas somente pelo respectivo programa.

O cliente deve ser executado da seguinte maneira:

```
./client.py -f config_file.json
```

O servidor deve ser executado da seguinte maneira:

```
./server.py -f config_file.json
```

Onde config_file.json é o arquivo de configuração, como especificado acima.

Tabela 1: Tabela dos algoritmos em relação a flipagens de bits

Algortimo	Flipagem aleatória(%)	Flipagem de pares(%)	Flipagem de ímpares(%)
Hamming	2.56	24.9	62.4
CRC-8	1.38	0	0
MD5	0	0	0
SHA-1	0	0	0

4 Resultados e análise

A tabela acima mostra o comportamento dos algoritmos de detecção de erros em diferentes contextos: quando flipamos os bits do payload de forma aleatória, quando flipamos os bits pares e quando flipamos os bits ímpares do payload. Os números na tabela indicam a porcentagem de erros apresentados por cada algoritmo em cada uma dessas situações.

Para a construção da tabela, para cada algoritmo transmitimos números de 0 a 1000 entre 2 clientes, flipando os bits do payload de alguma das 3 formas indicadas na tabela. Caso o cliente destino não consiga detectar que houveram bits flipados durante a transmissão usando o algoritmo indicado, isso é considerado um falso positivo. Fazemos a soma de todos os falsos positivos resultantes, dividimos por 10 e como resultado temos uma porcentagem de erros do algoritmo em determinado contexto. No caso da flipagem de bits aleatória, fizemos 10 testes com cada algoritmo e tiramos a média disso.

Por exemplo, usando o Hamming code num certo dado e flipando os bits pares do payload, em 24.9% das vezes o código de Hamming não indica que houve erro na transmissão do dado. Logo, quando os bits pares do payload são flipados, o código de Hamming é o que apresenta o pior resultado dentre os quatro métodos testados.

Apesar do Hamming code apresentar o pior resultado em todas as situações, ele tem a vantagem de ser o algoritmo que menos usa bits adicionais para a checagem de erros (como os dados sempre tem 32 bits, a quantidade de bits de paridade será sempre 6), além de ser o único que consegue corrigir um bit errado na transmissão.

5 Considerações finais

Como resultado, vemos que o Hamming apresenta os piores resultados em todos os contextos testados. Mas como vimos, não podemos descartar seu uso apenas por isso. Ele apresenta vantagens pois é o único algoritmo dos quatro apresentados que corrige erro na transmissão (1 bit no máximo), além de ser o que usa menos bits adicionais para detecção de erro. Mesmo assim, ao fliparmos bits pares ou ímpares do payload, as taxas de erro do Hamming code sobem bastante e fica inviável o seu uso, principalmente flipando bits ímpares onde a taxa de erro passa de 50%.

Os dois algoritmos de hash criptográfico, o MD5 e o SHA-1, detectaram erros na transmissão em todos os testes e em todos os contextos de flipagens diferentes. Apesar disso, não é certo concluir que eles devem ser sempre usados para identificação de erros, pois uma das

grandes desvantagens deles é a grande quantidade de bits transmitidos para cada dado. Por exemplo, usando a função de MD5 em um dado, teremos um hash resultante de 128 bits, que concatenados ao dado transmitido (que no trabalho sempre tem 32 bits), teremos um payload de 160bits. Para várias aplicações, isso é inviável.

O crc-8 apresentou um bom desempenho por detectar erros na mensagem em todos os casos testados de flipagem de bits pares e ímpares, e apenas uma pequena faixa de falsos positivos na flipagem aleatória. Um adicional desse algoritmo, é que o checksum calculado possui tamanho fixo de 8 bits.