Санкт-Петербургский государственный политехнический университет

Отчет по курсовой работе № 1

по курсу «Компьютерные сети»

«Моделирование канального уровня сетевой модели OSI»

Студент: Руцкий В. В.
Группа: 5057/2
Преподаватель: Баженов А. Н.

Санкт-Петербург 2010

Содержание

1 Постановка задачи

Необходимо реализовать модель канального уровня сетевой модели OSI и исследовать эффективность используемого в модели протокола передачи данных при наличии ошибок при передаче.

Для модели необходимо использовать протокол, использующий идею плавающего окна.

Передача ведётся по полнодуплексному каналу. Данные должны передаваться с гарантией доставки.

2 Выбранный метод решения

Для реализации был выбран протокол выборочного повтора (*selective repeat*), описанный в [1].

Протокол использует идею плавающего окна: отправитель отправляет кадры в пределах некоторого «окна» — интервал ограниченной длины последовательно идущих кадров; получатель ожидает кадров также в пределах своего рабочего окна.

При получении кадра получатель отправляет уведомление о доставке отправителю. При получении уведомления о доставке отправитель сдвигает рабочее окно таким образом, чтобы для первого кадра окна ещё не было получено уведомления о доставке. Аналогичным образом получатель поддерживает своё рабочее окно.

Для каждого отправленного кадра отправитель создаёт таймер, который подсчитывает сколько времени кадр находится в состоянии ожидания уведомления о доставке. В случае превышения времени ожидания для кадра устанавливается флаг ошибки доставки.

При обнаружении кадра с ошибкой доставки производится его повторная отправка.

3 Детали реализации

3.1 Физический уровень передачи данных

Модель физического уровня передачи данных представлена классом узла *FullDuplexNode* (см. приложение A.1). Узлы конструируются связанными парами. Каждый узел предоставляет два метода: *write* и *read*, позволяющие отправлять связанному узлу непрерывный поток байтов.

В реализации физического уровня предусмотрено внесение ошибок в передаваемые данные. Возможно внесение ошибок трёх типов: подмена передаваемого байта, добавление лишнего байта, удаление передаваемого байта.

3.2 Передача данных низкоуровневыми кадрами

Для удобства контроля передачи данных непрерывный поток байтов делится на группы байтов переменной длины — низкоуровневые кадры (см. приложение A.2). Деление на низкоуровневые кадры делается по аналогии с делением в протоколе SLIP.[1]

3.3 Передача данных кадрами с гарантией доставки

Поверх уровня передачи низкоуровневыми кадрами реализован протокол передачи кадров с выборочным повтором, описанный в разделе 2 (см. реализацию в приложении A.3).

Передаваемые данные делятся на последовательные блоки небольшого размера и упаковываются в кадры, далее кадры передаются смежному узлу через нижележащий уровень сети.

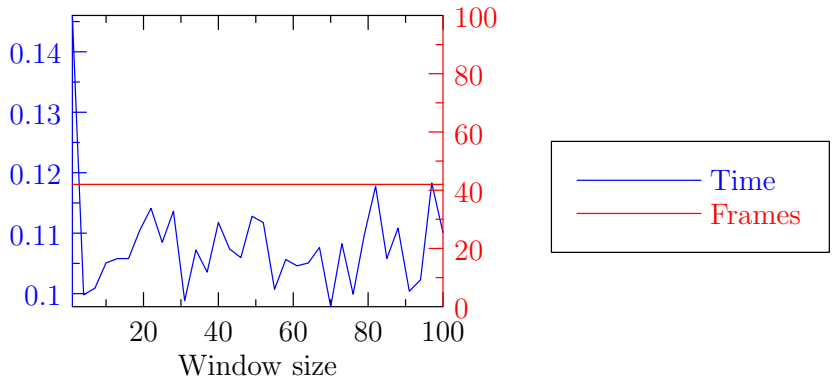
Используется кадр следующего формата (см. таблицу 1):

type	id	last	len	data	CRC32

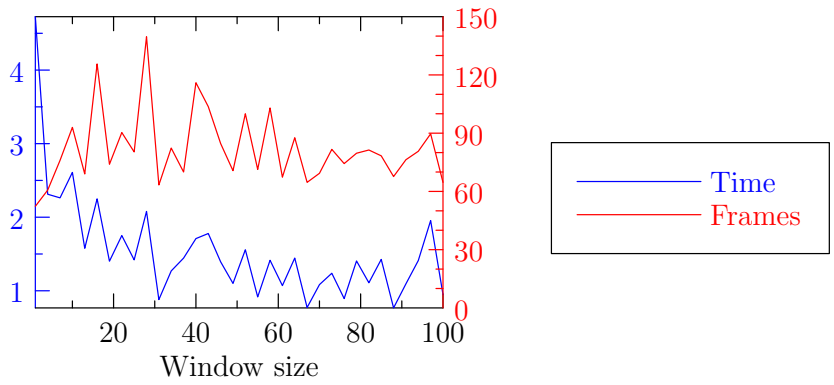
Таблица 1: Формат кадра.

- **type** (1 байт) — тип кадра: данные или подтверждение полученных данных;

[1] *Serial Line Internet Protocol*, описан в RFC 1055.



(a) Передача без потерь.



(b) Передача с потерями.

Рис. 1: Зависимость времени передачи данных и количества переданных низкоуровневых кадров от размера окна передачи.

- **id** (2 байта) — идентификатор (номер) передаваемого кадра;
- **last** (1 байт) — является ли кадр последней частью передаваемых данных;
- **len** (4 байта) — длина поля данных;
- **data** (**len** байт) — данные;
- **CRC32** (4 байта) — контрольная сумма пакета (CRC32[2]).

4 Исследования алгоритма

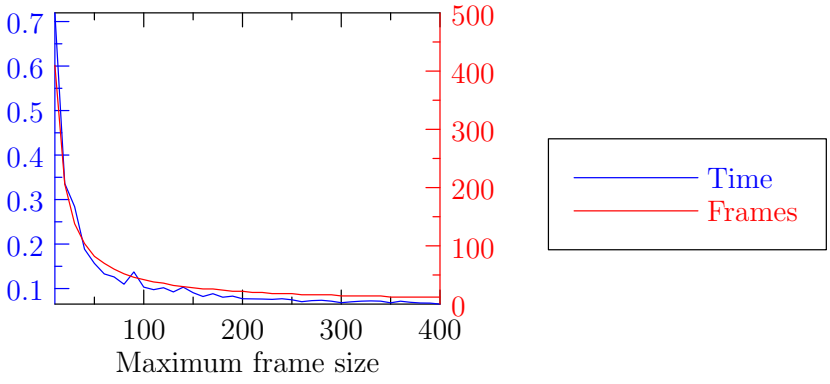
В результате проведённых исследований были получены следующие результаты.

На рис. 1a и рис. 1b представлены полученные зависимости времени передачи сообщений и количества переданных кадров от размера окна передачи.

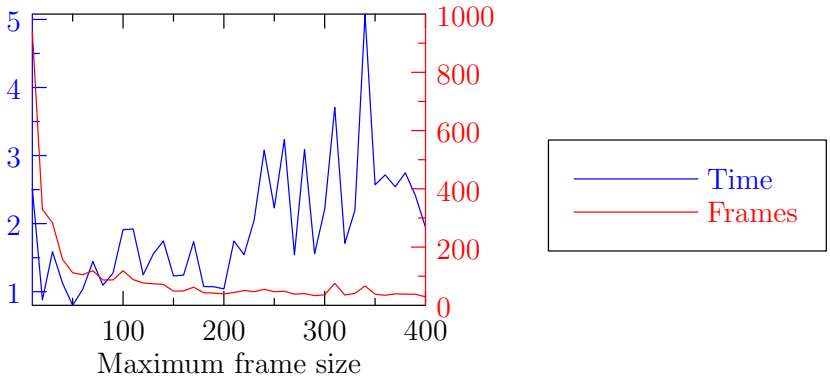
На рис. 2a и рис. 2b представлены полученные зависимости времени передачи сообщений и количества переданных кадров от максимального размера низкоуровнего кадра.

На рис. 3 представлена полученная зависимость времени передачи сообщений и количества переданных кадров от вероятности потери данных при передаче.

[2] *Cyclic redundancy check — циклический избыточный код*, описан в [2].



(a) Передача без потерь.



(b) Передача с потерями.

Рис. 2: Зависимость времени передачи данных и количества переданных низкоуровневых кадров от максимального размера низкоуровнего кадра.

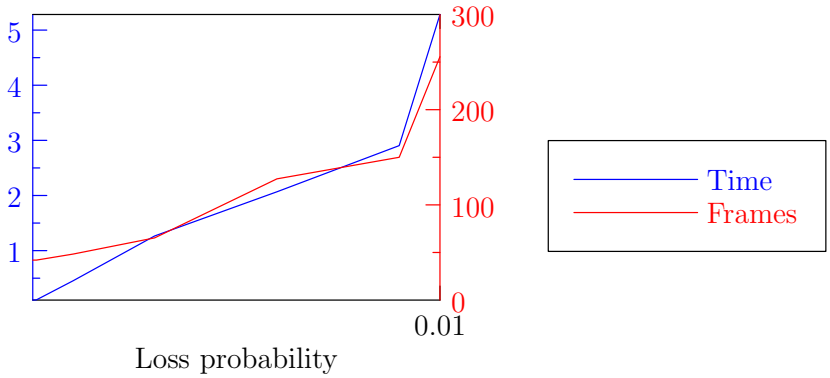


Рис. 3: Зависимость времени передачи данных и количества переданных низкоуровневых кадров от вероятности потери данных.

5 Результат работы

Исследование поведения реализованной модели в различных условиях показало:

- Малые размеры окна передачи сильно замедляют работу сети. При увеличении размера окна, с некоторого момента увеличение не улучшает скорость работы сети.
- В случае наличия потерь при передаче, при увеличении размера кадра увеличивается процент недоставленных кадров, что замедляет работу сети. При уменьшении размера кадра процент недоставленных кадров уменьшается, но растёт общее количество передаваемых кадров, что с некоторого момента также сильно замедляет работу сети.
- Время передачи данных и количество посылаемых низкоуровневых кадров экспоненциально зависит от вероятности потери данных.

A Исходный код

A.1 Полнодуплексная передача данных

```
#  This file is part of network emulation test model.
#
#  Copyright (C) 2010, 2011  Vladimir Rutsky <altsysrq@gmail.com>
#
#  This program is free software: you can redistribute it and/or modify
#  it under the terms of the GNU General Public License as published by
#  the Free Software Foundation, either version 3 of the License, or
#  (at your option) any later version.
#
#  This program is distributed in the hope that it will be useful,
#  but WITHOUT ANY WARRANTY; without even the implied warranty of
#  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#  GNU General Public License for more details.
#
#  You should have received a copy of the GNU General Public License
#  along with this program.  If not, see <http://www.gnu.org/licenses/>.

__author__  = "Vladimir Rutsky <altsysrq@gmail.com>"
__license__ = "GPL"

__all__ = ["ReceivingNode", "SendingNode", "FullDuplexNode",
    "FullDuplexLink", "LossFunc"]

"""Byte channel implementation.
"""

import Queue
import bisect
import random
import StringIO

# TODO: Maybe loss function will loose bits, not bytes?
# TODO: Maybe rename 'node_a'/'node_b'? 'Source' and 'destination' are not
# really good due to symmetric relation between link ends.

class SendingNode(object):
    def __init__(self, **kwds):
        self.send_queue = kwds.pop('send_queue')
        super(SendingNode, self).__init__(**kwds)

    def _write_ch(self, ch):
        assert isinstance(ch, str)
        assert len(ch) == 1
        self.send_queue.put(ch)

    def write(self, string):
        assert isinstance(string, str)
        for ch in string:
            self._write_ch(ch)

class ReceivingNode(object):
    def __init__(self, **kwds):
        self.receive_queue = kwds.pop('receive_queue')
```

```
54          super(ReceivingNode, self).__init__(**kwds)
55
56      def read(self, size=0, block=True):
57          """Read bytes from input queue.
58          If 'size' equal to zero it reads all available in queue characters.
59          Assumed that this function is the only reader from queue.
60
61          Otherwise if 'block' is True it reads exactly 'size' elements.
62          If 'block' is False it reads not more that 'size' elements depending
63          on how much is currently available in queue.
64          """
65
66          assert size >= 0
67
68          in_str = StringIO.StringIO()
69
70          while size == 0 or len(in_str.getvalue()) < size:
71              try:
72                  in_str.write(self.receive_queue.get(block and (size > 0)))
73              except Queue.Empty:
74                  break
75
76          return in_str.getvalue()
77
78  # TODO: May be not "loss" but "noise"?
79  class LossFunc(object):
80      def __init__(self, skip_ch_prob, modify_ch_prob, new_ch_prob):
81          super(LossFunc, self).__init__()
82          self.configure(skip_ch_prob, modify_ch_prob, new_ch_prob)
83
84      def configure(self, skip_ch_prob, modify_ch_prob, new_ch_prob):
85          assert 0 <= skip_ch_prob <= 1
86          assert 0 <= modify_ch_prob <= 1
87          assert 0 <= new_ch_prob <= 1
88          assert 0 <= skip_ch_prob + modify_ch_prob + new_ch_prob <= 1
89
90          # Construct array so that by finding upper bound of random number in it
91          # will be possible to select random modification.
92          self._selection_array = [0, skip_ch_prob, modify_ch_prob, new_ch_prob]
93          for i, v in enumerate(self._selection_array[1:]):
94              self._selection_array[i + 1] = self._selection_array[i] + v
95          self._selection_array.append(1)
96
97      def __call__(self, ch):
98          """Takes single character and returns transformed string."""
99          choice = bisect.bisect_right(self._selection_array, random.random())
100
101         random_ch = lambda: chr(random.randint(0, 255))
102
103         if choice == 1:
104             # Skip character.
105             return ""
106         elif choice == 2:
107             # Modify character.
108             return random_ch()
109         elif choice == 3:
110             # Append new character.
```

```
111             return ch + random_ch()
112         else:
113             assert choice == 4
114             # Don't modify character.
115             return ch
116 
117 class SendingWithLossNode(SendingNode):
118     def __init__(self, **kwds):
119         self._loss_func = kwds.pop('loss_func', None)
120         if self._loss_func is None:
121             self._loss_func = LossFunc(0, 0, 0)
122         super(SendingWithLossNode, self).__init__(**kwds)
123 
124     def _write_ch(self, ch):
125         for new_ch in self._loss_func(ch):
126             super(SendingWithLossNode, self)._write_ch(new_ch)
127 
128 class FullDuplexNode(SendingWithLossNode, ReceivingNode):
129     def __init__(self, **kwds):
130         super(FullDuplexNode, self).__init__(**kwds)
131 
132 def FullDuplexLink(a_to_b_queue=None, b_to_a_queue=None, loss_func=None):
133     queue1 = a_to_b_queue if a_to_b_queue is not None else Queue.Queue()
134     queue2 = b_to_a_queue if b_to_a_queue is not None else Queue.Queue()
135 
136     node_a = FullDuplexNode(
137         send_queue   =queue1,
138         receive_queue=queue2,
139         loss_func=loss_func)
140     node_b = FullDuplexNode(
141         send_queue   =queue2,
142         receive_queue=queue1,
143         loss_func=loss_func)
144     return node_a, node_b
```

A.2 Передача данных низкоуровневыми кадрами

```
1  #  This file is part of network emulation test model.
2  #
3  #  Copyright (C) 2010, 2011  Vladimir Rutsky <altsysrq@gmail.com>
4  #
5  #  This program is free software: you can redistribute it and/or modify
6  #  it under the terms of the GNU General Public License as published by
7  #  the Free Software Foundation, either version 3 of the License, or
8  #  (at your option) any later version.
9  #
10 #  This program is distributed in the hope that it will be useful,
11 #  but WITHOUT ANY WARRANTY; without even the implied warranty of
12 #  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13 #  GNU General Public License for more details.
14 #
15 #  You should have received a copy of the GNU General Public License
16 #  along with this program.  If not, see <http://www.gnu.org/licenses/>.
17 
18 __author__  = "Vladimir Rutsky <altsysrq@gmail.com>"
19 __license__ = "GPL"
20 
```

```
__all__ = ["SimpleFrameTransmitter"]

"""Transmit raw frame between two connected hosts without any acknowledge.
"""

import StringIO

class SimpleFrameTransmitter(object):
    # Similar to SLIP.
    frame_end       = "\xC0"
    esc_char        = "\xDB"
    frame_end_subst = esc_char + "\xDC"
    esc_subst       = esc_char + "\xDD"

    def __init__(self, *args, **kwargs):
        self.node = kwargs.pop('node')
        super(SimpleFrameTransmitter, self).__init__(*args, **kwargs)
        self._read_buffer = StringIO.StringIO()

        self._read_frames_count = 0
        self._write_frames_count = 0

    @property
    def read_frames_count(self):
        return self._read_frames_count

    @property
    def write_frames_count(self):
        return self._write_frames_count

    def write_frame(self, frame):
        raw_data = (
            # Replace escape characters.
            frame.replace(self.esc_char, self.esc_subst)
            # Replace frame end characters inside frame.
                .replace(self.frame_end, self.frame_end_subst) +
            # Append frame end at end.
            self.frame_end)
        self.node.write(raw_data)

        self._write_frames_count += 1

    def read_frame(self, block=True):
        """Read single frame from input channel.
        Assumed that this is the only reader from channel.
        """
        while True:
            ch = self.node.read(1, block=block)
            if ch == "":
                # No more characters for now.
                return None
            elif ch == self.frame_end:
                # Read till frame end. Decode and return it.

                # Obtain encoded frame.
                encoded_frame = self._read_buffer.getvalue()

```

```
78                  # Reset input buffer.
79                  self._read_buffer = StringIO.StringIO()
80
81                  # Decode frame.
82                  frame = encoded_frame.replace(self.frame_end_subst,
83                                                self.frame_end).\
84                      replace(self.esc_subst, self.esc_char)
85
86                  self._read_frames_count += 1
87
88                  return frame
89              else:
90                  self._read_buffer.write(ch)
```

A.3 Передача данных кадрами с гарантией доставки

```
1   #  This file is part of network emulation test model.
2   #
3   #  Copyright (C) 2010, 2011  Vladimir Rutsky <altsysrq@gmail.com>
4   #
5   #  This program is free software: you can redistribute it and/or modify
6   #  it under the terms of the GNU General Public License as published by
7   #  the Free Software Foundation, either version 3 of the License, or
8   #  (at your option) any later version.
9   #
10  #  This program is distributed in the hope that it will be useful,
11  #  but WITHOUT ANY WARRANTY; without even the implied warranty of
12  #  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13  #  GNU General Public License for more details.
14  #
15  #  You should have received a copy of the GNU General Public License
16  #  along with this program.  If not, see <http://www.gnu.org/licenses/>.
17
18  __author__  = "Vladimir Rutsky <altsysrq@gmail.com>"
19  __license__ = "GPL"
20
21  __all__ = ["FrameTransmitter", "FrameTransmitterWorker", "worker"]
22
23  """Transmit frame between two connected hosts with acknowledge.
24  """
25
26  import itertools
27  import struct
28  import binascii
29  import threading
30  import time
31  import logging
32  import Queue
33  from collections import deque
34  from recordtype import recordtype
35
36  import config
37  from frame import SimpleFrameTransmitter
38
39  class FrameType(object):
40      data = 1
41      ack  = 2
```

```
42
43  class InvalidFrameException(Exception):
44      def __init__(self, *args, **kwargs):
45          super(InvalidFrameException, self).__init__(*args, **kwargs)
46
47  # TODO: Inherit from recordtype.
48  class Frame(object):
49      # Frame:
50      #    1     2     1      4             4     - field size
51      # *------*----*------*-----*--  --*-------*
52      # | type | id | last | len | data | CRC32 |
53      # *------*----*------*-----*--  --*-------*
54
55      format_string = '<BHBL{0}sL'
56      empty_frame_size = struct.calcsize(format_string.format(0))
57
58      def __init__(self, *args, **kwargs):
59          self.type = kwargs.pop('type')
60          self.id = kwargs.pop('id')
61          if self.type == FrameType.data:
62              self.data    = kwargs.pop('data')
63              self.is_last = kwargs.pop('is_last')
64          else:
65              self.data = ""
66              if 'data' in kwargs:
67                  kwargs.pop('data')
68              self.is_last = False
69              if 'is_last' in kwargs:
70                  kwargs.pop('is_last')
71          super(Frame, self).__init__(*args, **kwargs)
72
73      def crc(self):
74          return binascii.crc32(self.serialize(0)) & 0xffffffff
75
76      def serialize(self, crc = None):
77          """Returns string representing frame."""
78
79          if crc is not None:
80              return struct.pack(
81                  self.format_string.format(len(self.data)),
82                  self.type, self.id, self.is_last, len(self.data),
83                  self.data, crc)
84          else:
85              return self.serialize(self.crc())
86
87      @staticmethod
88      def deserialize(frame_str):
89          # TODO: Add frame dump into InvalidFrameException error message.
90
91          data_len = len(frame_str) - Frame.empty_frame_size
92          if data_len < 0:
93              raise InvalidFrameException(
94                  "Frame too small, not enough fields")
95
96          frame_type, frame_id, is_last, read_data_len, frame_data, frame_crc = \
97                  struct.unpack(Frame.format_string.format(data_len), frame_str)
98
```

```
99          if frame_type not in [FrameType.data, FrameType.ack]:
100             raise InvalidFrameException(
101                 "Invalid frame type '{0}'".format(frame_type))
102
103         if read_data_len != data_len:
104             raise InvalidFrameException(
105                 "Invalid data length: {0}, expected {1}".format(
106                     read_data_len, data_len))
107
108         frame = Frame(type=frame_type, id=frame_id, is_last=is_last,
109             data=frame_data)
110
111         if frame_crc != frame.crc():
112             raise InvalidFrameException(
113                 "Invalid ckecksum: {0:04X}, correct one is {1:04X}".format(
114                     frame_crc, frame.crc()))
115
116         return frame
117
118     def __str__(self):
119         if self.type == FrameType.data:
120             return "Data({id}, is_last={is_last}, 0x{data})".format(
121                 id=self.id, is_last=self.is_last,
122                 data=self.data.encode('hex'))
123         elif self.type == FrameType.ack:
124             return "Ack({id})".format(id=self.id)
125         else:
126             assert False
127
128     # TODO: Implement __eq__(), __ne__().
129
130 def clear_queue(q):
131     while True:
132         try:
133             q.get(block=False)
134         except Queue.Empty:
135             break
136
137 class FrameTransmitterWorker(object):
138     def __init__(self):
139         super(FrameTransmitterWorker, self).__init__()
140
141         self._logger = logging.getLogger("FrameTransmitterWorker")
142
143         self._frame_transmitters = set()
144         self._frame_transmitters_lock = threading.RLock()
145
146         self._working_thread = None
147         self._exit_lock = threading.RLock()
148
149     def add_frame_transmitter(self, frame_transmitter):
150         with self._frame_transmitters_lock:
151             if self._working_thread is None:
152                 # If working thread will be able to acquire the lock, then it
153                 # should terminate himself.
154                 self._exit_lock.acquire()
155
```

```
156                 self._working_thread = threading.Thread(target=self._work)
157                 self._working_thread.start()
158 
159             self._frame_transmitters.add(frame_transmitter)
160 
161     def remove_frame_transmitter(self, frame_transmitter):
162         with self._frame_transmitters_lock:
163             self._frame_transmitters.remove(frame_transmitter)
164 
165             if not self._frame_transmitters:
166                 # Release exit lock and wait until working thread will not
167                 # terminate.
168 
169                 # If _exit_lock already released then somebody already called
170                 # terminate().
171                 self._exit_lock.release()
172                 self._working_thread.join()
173 
174                 self._working_thread = None
175 
176     def _work(self):
177         self._logger.info("Working thread started")
178 
179         while True:
180             if self._exit_lock.acquire(False):
181                 # Obtained exit lock. Terminate.
182 
183                 self._exit_lock.release()
184                 self._logger.info("Exit working thread")
185                 return
186 
187             with self._frame_transmitters_lock:
188                 for frame_transmitter in self._frame_transmitters:
189                     frame_transmitter.update()
190 
191             time.sleep(config.frame_transmitter_thread_sleep_time)
192 
193 worker = FrameTransmitterWorker()
194 
195 class FrameTransmitter(object):
196     _frame_id_period = 32768
197 
198     class _SendWindow(object):
199         SendItem = recordtype('SendItem', 'id time frame ack_received')
200 
201         def __init__(self, logger, maxlen, frame_id_it, timeout):
202             super(FrameTransmitter._SendWindow, self).__init__()
203 
204             self._logger = logger
205             self.maxlen = maxlen
206             self.queue = deque(maxlen=maxlen)
207             self.frame_id_it = frame_id_it
208             self.timeout = timeout
209 
210         def can_add_next(self):
211             return len(self.queue) < self.maxlen
212 
```

```
213         def add_next(self, is_last, data, curtime=None):
214             assert self.can_add_next()
215 
216             using_curtime = curtime if curtime is not None else time.time()
217 
218             frame_id = self.frame_id_it.next()
219             p = Frame(type=FrameType.data, id=frame_id, is_last=is_last,
220                 data=data)
221             item = FrameTransmitter._SendWindow.SendItem(
222                 frame_id, using_curtime, p, False)
223             self.queue.append(item)
224 
225             return item
226 
227         def timeout_items(self, curtime=None):
228             using_curtime = curtime if curtime is not None else time.time()
229             for item in self.queue:
230                 if item.time + self.timeout < using_curtime:
231                     yield item
232 
233         def ack_received(self, frame_id):
234             # TODO: Performance issue.
235             for item in self.queue:
236                 if item.id == frame_id:
237                     item.ack_received = True
238                     break
239             else:
240                 self._logger.warning(
241                     "Received ack for frame outside working window: {0}".
242                         format(frame_id))
243 
244             while len(self.queue) > 0 and self.queue[0].ack_received:
245                 self.queue.popleft()
246 
247     class _ReceiveWindow(object):
248         ReceiveItem = recordtype('ReceiveItem', 'id frame')
249 
250         def __init__(self, logger, maxlen, frame_id_it):
251             super(FrameTransmitter._ReceiveWindow, self).__init__()
252 
253             self._logger = logger
254             self.queue = deque(maxlen=maxlen)
255             self.frame_id_it = frame_id_it
256 
257             # Fill receive window with placeholders for receiving frames.
258             for idx in xrange(maxlen):
259                 item = FrameTransmitter._ReceiveWindow.ReceiveItem(
260                     self.frame_id_it.next(), None)
261                 self.queue.append(item)
262 
263         def receive_frame(self, frame):
264             # TODO: Performance issue.
265             for item in self.queue:
266                 if item.id == frame.id:
267                     item.frame = frame
268                     break
269             else:
```

```
                self._logger.warning(
                    "Received frame outside working window: {0}".
                        format(frame))

            while self.queue[0].frame is not None:
                yield self.queue[0].frame

                self.queue.popleft()
                new_item = FrameTransmitter._ReceiveWindow.ReceiveItem(
                    self.frame_id_it.next(), None)
                self.queue.append(new_item)

    def __init__(self, *args, **kwargs):
        self._simple_frame_transmitter = kwargs.pop('simple_frame_transmitter')
        self._max_frame_data = kwargs.pop('max_frame_data', 100)
        self._window_size = kwargs.pop('window_size', 100)
        self._ack_timeout = kwargs.pop('ack_timeout', 0.5)

        global worker
        self._worker = kwargs.pop('worker', worker)

        self._debug_src = kwargs.pop('debug_src', '?')
        self._debug_dest = kwargs.pop('debug_dest', '?')
        super(FrameTransmitter, self).__init__(*args, **kwargs)

        self._logger = logging.getLogger("FrameTransmitter.{0}->{1}".format(
            self._debug_src, self._debug_dest))

        # Queue of tuples (is_last, frame_data).
        self._frames_data_to_send = Queue.Queue()
        # Queue of tuples (is_last, frame_data).
        self._received_data = Queue.Queue()

        self._received_frames_buffer = []

        self._send_window = FrameTransmitter._SendWindow(
            self._logger, self._window_size,
            itertools.cycle(xrange(self._frame_id_period)),
            self._ack_timeout)
        self._receive_window = FrameTransmitter._ReceiveWindow(
            self._logger, self._window_size,
            itertools.cycle(xrange(self._frame_id_period)))

        self._enabled = True
        self._enabled_lock = threading.RLock()

        self._worker.add_frame_transmitter(self)

    @property
    def enabled(self):
        return self._enabled

    @enabled.setter
    def enabled(self, value):
        with self._enabled_lock:
            if self._enabled != bool(value):
                self._enabled = bool(value)
```

```
327
328                 if self._enabled:
329                     # Link up.
330                     self._link_up()
331                 else:
332                     # Link down.
333                     self._link_down()
334
335     def _link_up(self):
336         pass
337
338     def _link_down(self):
339         clear_queue(self._frames_data_to_send)
340         clear_queue(self._received_data)
341
342     # TODO
343     def terminate(self):
344         self._worker.remove_frame_transmitter(self)
345
346     def send(self, data_string):
347         with self._enabled_lock:
348             if self._enabled:
349                 # Subdivide data string on frames and put them into working
350                 # queue.
351                 frame_data_parts = \
352                     [data_string[i:i + self._max_frame_data]
353                         for i in xrange(
354                             0, len(data_string), self._max_frame_data)]
355                 for frame_data_part in frame_data_parts[:-1]:
356                     self._frames_data_to_send.put((False, frame_data_part))
357                 self._frames_data_to_send.put((True, frame_data_parts[-1]))
358             else:
359                 # Link is down.
360                 pass
361
362     def receive(self, block=True):
363         with self._enabled_lock:
364             if self._enabled:
365                 while True:
366                     try:
367                         is_last, frame = self._received_data.get(block)
368                         self._received_frames_buffer.append(frame)
369                         if is_last:
370                             data_string = "".join(self._received_frames_buffer)
371                             self._received_frames_buffer = []
372                             return data_string
373                     except Queue.Empty:
374                         break
375             else:
376                 # Link is down.
377                 assert self._received_data.empty()
378                 assert not self._received_frames_buffer
379                 return None
380
381     def update(self):
382         # Send frames.
383         if (not self._frames_data_to_send.empty() and
```

```
                self._send_window.can_add_next()):
            # Have frame for sending in queue and free space in send
            # window. Send frame.

            is_last, frame_data = self._frames_data_to_send.get()

            item = self._send_window.add_next(is_last, frame_data)

            self._logger.debug("Sending:\n  {0}".format(str(item.frame)))
            self._simple_frame_transmitter.write_frame(
                item.frame.serialize())

        # Handle timeouts.
        curtime = time.time()
        for item in self._send_window.timeout_items(curtime):
            # TODO: Currently it is selective repeat.

            self._logger.warning("Resending due to timeout:\n  {0}".format(
                str(item.frame)))
            self._simple_frame_transmitter.write_frame(
                item.frame.serialize())
            item.time = curtime
        assert len(list(self._send_window.timeout_items(curtime))) == 0

        # Handle receiving data.
        frame = self._simple_frame_transmitter.read_frame(block=False)
        if frame is not None:
            # Received frame.

            try:
                p = Frame.deserialize(frame)
            except InvalidFrameException as ex:
                self._logger.warning("Received invalid frame: {0}".format(
                    str(ex)))
            else:
                self._logger.debug("Received:\n  {0}".format(p))

                if p.type == FrameType.data:
                    # Received data.

                    # Send ACK (even if frame already received before).
                    ack = Frame(type=FrameType.ack, id=p.id, data="")
                    self._logger.debug("Sending acknowledge:\n  {0}".format(ack))
                    self._simple_frame_transmitter.write_frame(
                        ack.serialize())

                    for frame in self._receive_window.receive_frame(p):
                        self._received_data.put((frame.is_last, frame.data))

                elif p.type == FrameType.ack:
                    # Received ACK.

                    self._send_window.ack_received(p.id)

                else:
                    assert False
```

Список литературы

[1] Э. Таненбаум. Компьютерные сети. 2003.

[2] WW Peterson and DT Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235, 1961.