# COMPUTER NETWORKS

Faculty of Engineering
Ain Shams University
Computer department

SUBMITTED BY:
Pierre Nabil  (16E0056) Sec(1)
Girgis Michael (1600446) Sec(1)
John Bahaa (1600459) Sec(1)
Hazem Mohammed(1600469)Sec(1)
Ahmed Taha (1600108) Sec(2)

Data Link Layer Assignment

## Contents

## Introduction:

In the making of this project, we decided to add some extra features to make it more useful and not just some other assignment. We worked really hard on this assignment and we hope that you enjoy it as much as we did making it.

## Implementation Details:

We started by implementing Protocol 5 (Go Back n) from the reference in a class called "DataLink" in "datalink.py". Our code runs in discrete timesteps instead of in continuous time like the one in the reference which proved challenging for some implementation details. Another detail is that each device (sender and/or receiver) can only work at specific times; to be more exact, the device "$C_0$" can only work for even numbered timesteps, and the device "$C_1$" can only work for odd numbered timesteps.

We noticed that we needed to implement the timers and some form of dummy network layer. We implemented the timers in the Data Link layer first, then we chose to implement the network layer as a list of tuples "$[(\mathrm{t}_1, \mathrm{p}_1), (\mathrm{t}_2, \mathrm{p}_2), \dots, (\mathrm{t}_\mathrm{n}, \mathrm{p}_\mathrm{n})]$"where each tuple contains the time ($t_i$) which the network layers wants to send the packet ($p_i$) for any timestep ($i$).

We started by writing a test case in "testcase.py" (which can be easily modified) where we specify the network layer input structures of each device, and we also specify the times which an error can occur in the simulation (again, even numbered timesteps affect $C_0$ while odd numbered timesteps affect $C_1$).

Each Datalink Layer Proceeds as normal. For each timestep ($t$) the device "$C_i$" (where $i = t \bmod 2$ as stated before) checks its state for what events are occurring right now. It then decides what to do for each given event. (each case is treated independently as in the given pseudocode). The simulation runs for exactly 100 timesteps.

You can see the output of each timestep by running the script "main.py" which has a CLI-type interface.

### GUI:

We decided to take this project one step further and try to design a GUI for this simulation. We used PyQt to design the main window in "mainwindow.py" and used the simulation output from before as a basis to create a Graphical Simulation for each timestep.

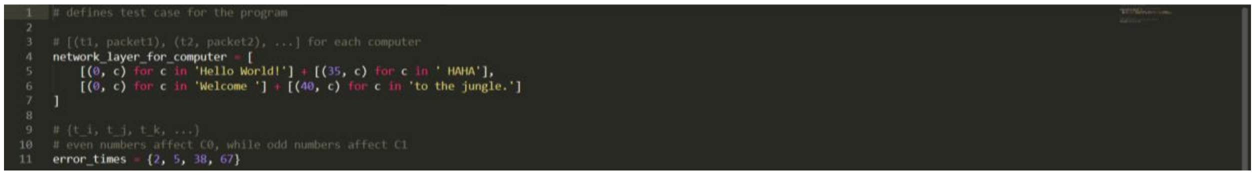
We added 2 buttons to show the next and previous timesteps of the simulation. We were planning to add an auto play feature which runs the whole simulation from the beginning without the need to keep clicking on the next timestep button. However, we couldn't make it work in time, so we decided to disable the button instead of redesigning the whole GUI.

The GUI can be run by running the script "mainwindow.py" or by running "mainwindow.exe".

## Snapshots:

### Code:

First of all, Here is the code for the test case (which can be modified) from “testcase.py”:



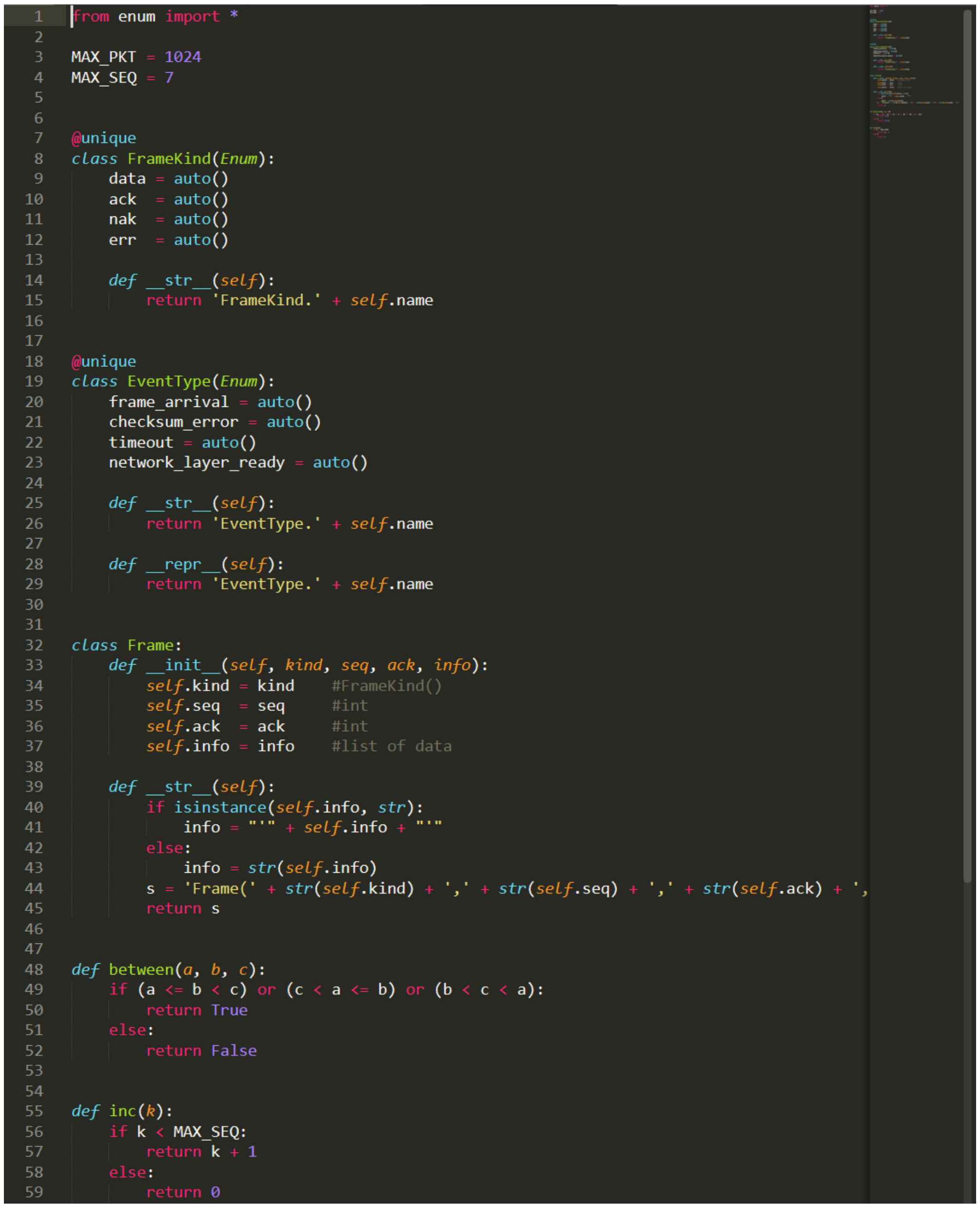
```python
# defines test case for the program

# [(t1, packet1), (t2, packet2), ...] for each computer
network_layer_for_computer = [
    [(0, c) for c in 'Hello World!'] + [(35, c) for c in ' HAHA'],
    [(0, c) for c in 'Welcome '] + [(40, c) for c in 'to the jungle.']
]

# {t_i, t_j, t_k, ...}
# even numbers affect C0, while odd numbers affect C1
error_times = {2, 5, 38, 67}
```

The network layer of “$C_0$” will want to send the string “Hello World!” character -by-character starting from timestep 0, and “ HAHA” starting from timestep 35.

The network layer of “$C_1$” will want to send the string “Welcome ” character -by-character starting from timestep 0, and “to the jungle.” starting from timestep 40.

There will be errors affecting the messages sent at times 2 and 38 affecting $C_0$ and times 5 and 67 affecting $C_1$.

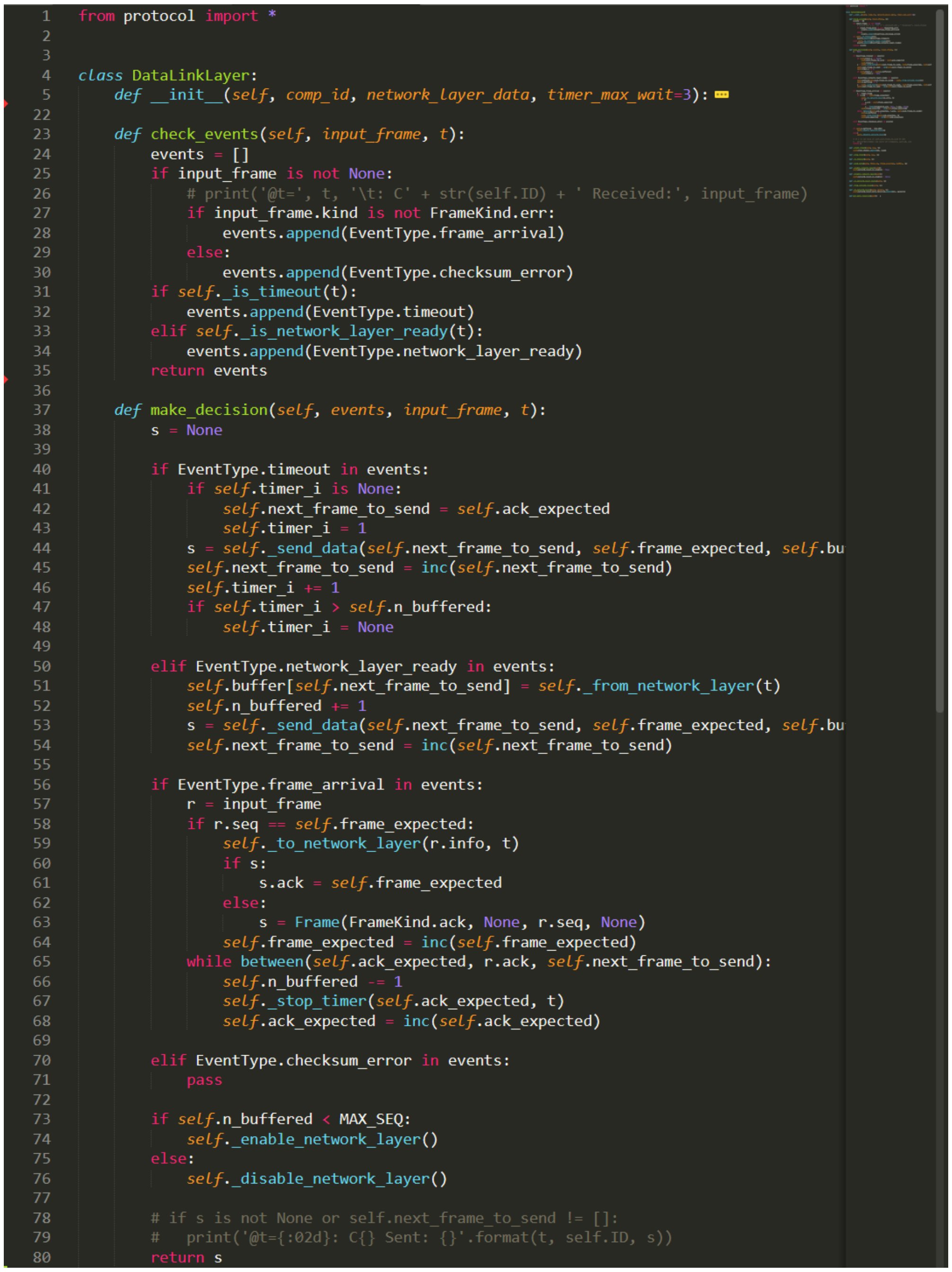
Next, we have the file “protocol.py” with the basic helper classes and functions:



```python
from enum import *

MAX_PKT = 1024
MAX_SEQ = 7


@unique
class FrameKind(Enum):
    data = auto()
    ack  = auto()
    nak  = auto()
    err  = auto()

    def __str__(self):
        return 'FrameKind.' + self.name


@unique
class EventType(Enum):
    frame_arrival = auto()
    checksum_error = auto()
    timeout = auto()
    network_layer_ready = auto()

    def __str__(self):
        return 'EventType.' + self.name

    def __repr__(self):
        return 'EventType.' + self.name


class Frame:
    def __init__(self, kind, seq, ack, info):
        self.kind = kind    #FrameKind()
        self.seq  = seq     #int
        self.ack  = ack     #int
        self.info = info    #list of data

    def __str__(self):
        if isinstance(self.info, str):
            info = "'" + self.info + "'"
        else:
            info = str(self.info)
        s = 'Frame(' + str(self.kind) + ',' + str(self.seq) + ',' + str(self.ack) + ',
        return s


def between(a, b, c):
    if (a <= b < c) or (c < a <= b) or (b < c < a):
        return True
    else:
        return False


def inc(k):
    if k < MAX_SEQ:
        return k + 1
    else:
        return 0
```

Here we define the Enumerations FrameKind and EventType, the constants MAX_PKT and MAX_SEQ, the 2 helper functions between() and inc(), and finally the class Frame.

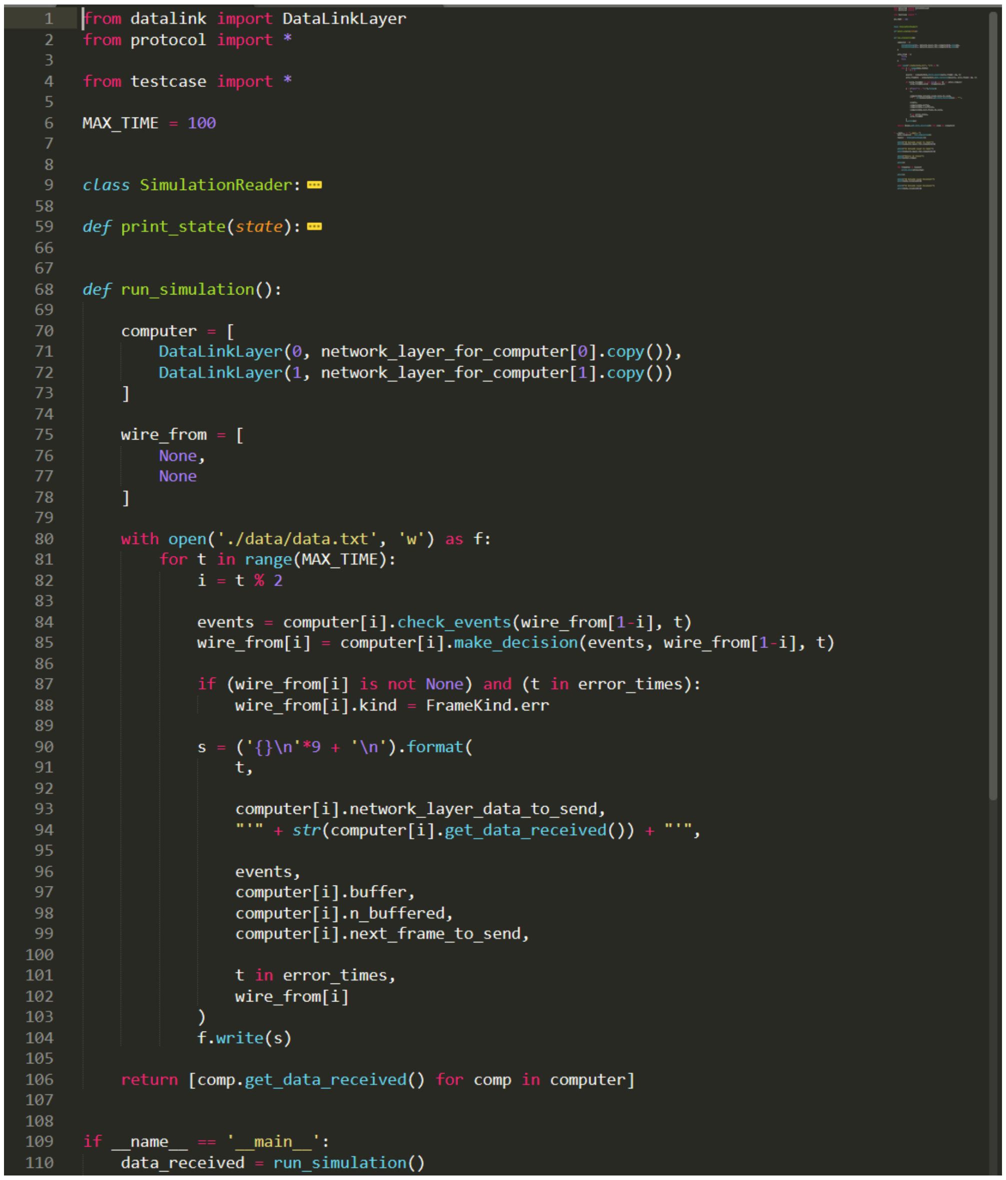
Next this is the DataLinkLayer Class implementation:



```python
from protocol import *


class DataLinkLayer:
    def __init__(self, comp_id, network_layer_data, timer_max_wait=3):

    def check_events(self, input_frame, t):
        events = []
        if input_frame is not None:
            # print('@t=', t, '\t: C' + str(self.ID) + ' Received:', input_frame)
            if input_frame.kind is not FrameKind.err:
                events.append(EventType.frame_arrival)
            else:
                events.append(EventType.checksum_error)
        if self._is_timeout(t):
            events.append(EventType.timeout)
        elif self._is_network_layer_ready(t):
            events.append(EventType.network_layer_ready)
        return events

    def make_decision(self, events, input_frame, t):
        s = None

        if EventType.timeout in events:
            if self.timer_i is None:
                self.next_frame_to_send = self.ack_expected
                self.timer_i = 1
            s = self._send_data(self.next_frame_to_send, self.frame_expected, self.bu
            self.next_frame_to_send = inc(self.next_frame_to_send)
            self.timer_i += 1
            if self.timer_i > self.n_buffered:
                self.timer_i = None

        elif EventType.network_layer_ready in events:
            self.buffer[self.next_frame_to_send] = self._from_network_layer(t)
            self.n_buffered += 1
            s = self._send_data(self.next_frame_to_send, self.frame_expected, self.bu
            self.next_frame_to_send = inc(self.next_frame_to_send)

        if EventType.frame_arrival in events:
            r = input_frame
            if r.seq == self.frame_expected:
                self._to_network_layer(r.info, t)
                if s:
                    s.ack = self.frame_expected
                else:
                    s = Frame(FrameKind.ack, None, r.seq, None)
                self.frame_expected = inc(self.frame_expected)
            while between(self.ack_expected, r.ack, self.next_frame_to_send):
                self.n_buffered -= 1
                self._stop_timer(self.ack_expected, t)
                self.ack_expected = inc(self.ack_expected)

        elif EventType.checksum_error in events:
            pass

        if self.n_buffered < MAX_SEQ:
            self._enable_network_layer()
        else:
            self._disable_network_layer()

        # if s is not None or self.next_frame_to_send != []:
        #   print('@t={:02d}: C{} Sent: {}'.format(t, self.ID, s))
        return s
```

I decided to split the given code into 2 functions: check_events() and make_decision() for code clarity. Any difference between this code and the given pseudocode is due to either the discrete implementation or the implementation details of the timers and network layers (which are not shown in this screenshot).

Finally we have the code for main.py:



```python
from datalink import DataLinkLayer
from protocol import *

from testcase import *

MAX_TIME = 100


class SimulationReader: ⋯

def print_state(state): ⋯


def run_simulation():

    computer = [
        DataLinkLayer(0, network_layer_for_computer[0].copy()),
        DataLinkLayer(1, network_layer_for_computer[1].copy())
    ]

    wire_from = [
        None,
        None
    ]

    with open('./data/data.txt', 'w') as f:
        for t in range(MAX_TIME):
            i = t % 2

            events = computer[i].check_events(wire_from[1-i], t)
            wire_from[i] = computer[i].make_decision(events, wire_from[1-i], t)

            if (wire_from[i] is not None) and (t in error_times):
                wire_from[i].kind = FrameKind.err

            s = ('{}\n'*9 + '\n').format(
                t,

                computer[i].network_layer_data_to_send,
                "'" + str(computer[i].get_data_received()) + "'",

                events,
                computer[i].buffer,
                computer[i].n_buffered,
                computer[i].next_frame_to_send,

                t in error_times,
                wire_from[i]
            )
            f.write(s)

    return [comp.get_data_received() for comp in computer]


if __name__ == '__main__':
    data_received = run_simulation()
```

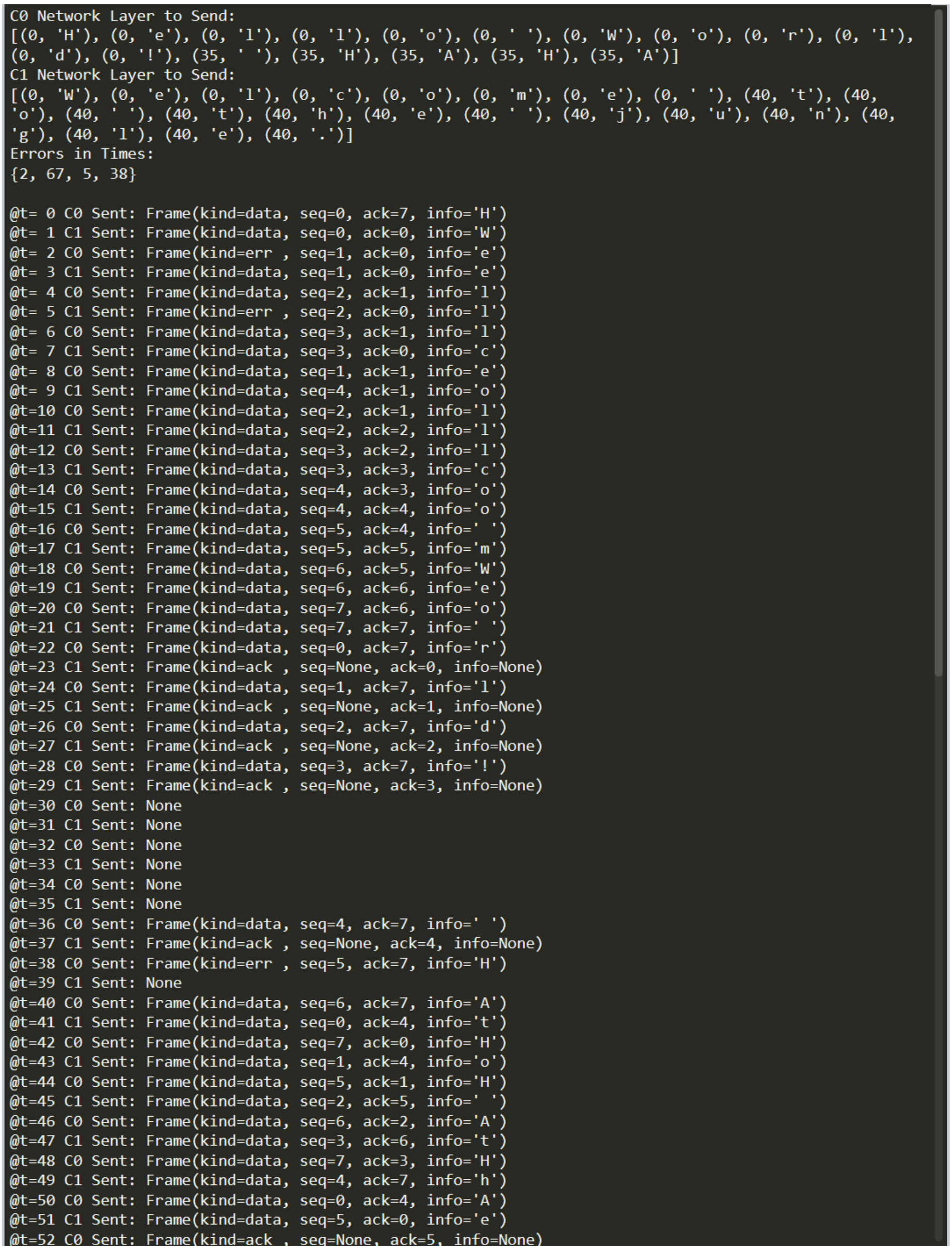
Here, we create to objects from the DatalinkLayer class and define the 2 wires going to and from each computer.

Then, for each timestep $t$, $C_i$ (where $i = t\ mod\ 2$ as stated before) checks it's state and the coming wire for events and then takes the actions required for the specified events. If an error is supposed to happen at time $t$, it is reflected by changing the sent frame's kind to be an error frame.

The System then writes the state of the system in a file for future reference.

## CLI Interface by running main.py:

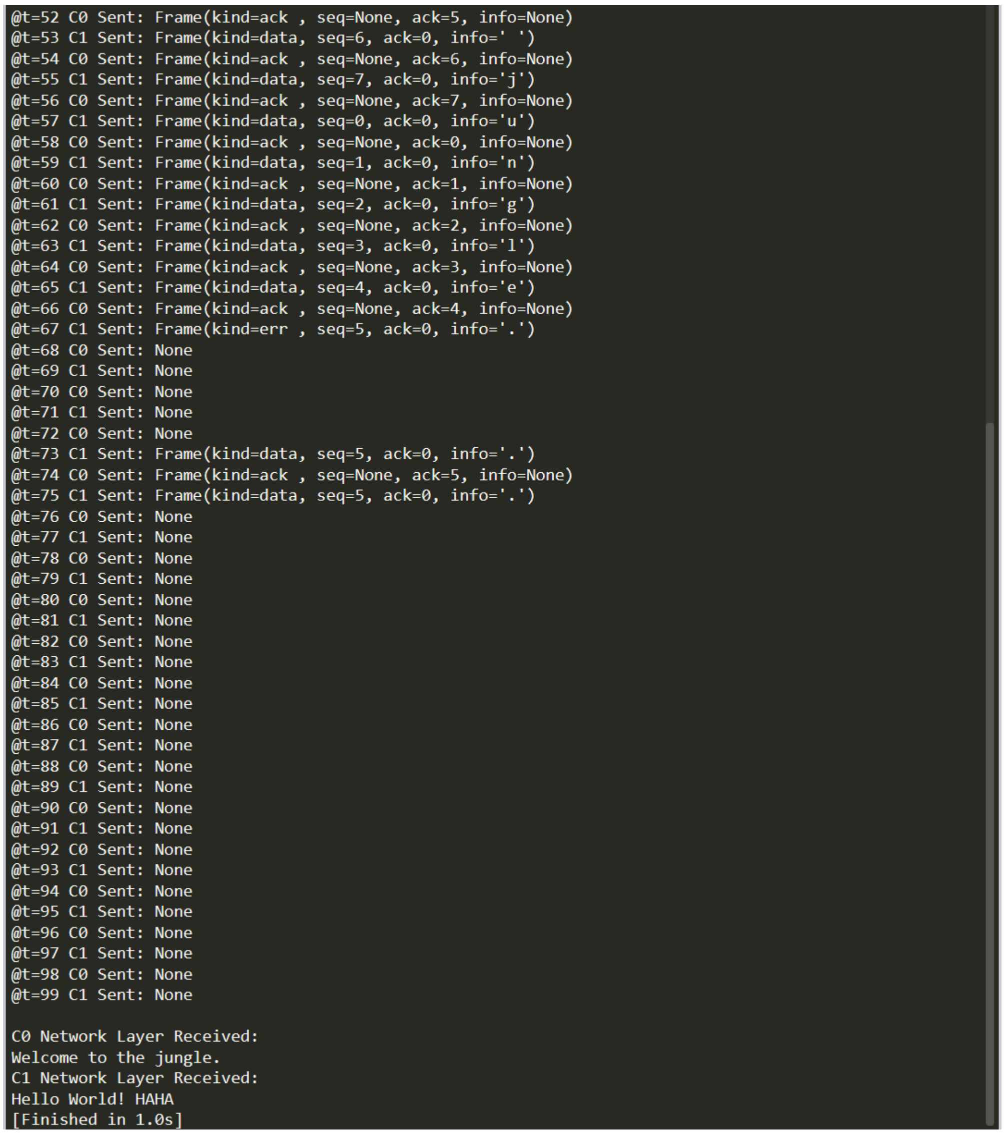
Running “main.py” for the given test case in “testcase.py” returns the following output:


```
C0 Network Layer to Send:
[(0, 'H'), (0, 'e'), (0, 'l'), (0, 'l'), (0, 'o'), (0, ' '), (0, 'W'), (0, 'o'), (0, 'r'), (0, 'l'),
(0, 'd'), (0, '!'), (35, ' '), (35, 'H'), (35, 'A'), (35, 'H'), (35, 'A')]
C1 Network Layer to Send:
[(0, 'W'), (0, 'e'), (0, 'l'), (0, 'c'), (0, 'o'), (0, 'm'), (0, 'e'), (0, ' '), (40, 't'), (40,
'o'), (40, ' '), (40, 't'), (40, 'h'), (40, 'e'), (40, ' '), (40, 'j'), (40, 'u'), (40, 'n'), (40,
'g'), (40, 'l'), (40, 'e'), (40, '.')]
Errors in Times:
{2, 67, 5, 38}

@t= 0 C0 Sent: Frame(kind=data, seq=0, ack=7, info='H')
@t= 1 C1 Sent: Frame(kind=data, seq=0, ack=0, info='W')
@t= 2 C0 Sent: Frame(kind=err , seq=1, ack=0, info='e')
@t= 3 C1 Sent: Frame(kind=data, seq=1, ack=0, info='e')
@t= 4 C0 Sent: Frame(kind=data, seq=2, ack=1, info='l')
@t= 5 C1 Sent: Frame(kind=err , seq=2, ack=0, info='l')
@t= 6 C0 Sent: Frame(kind=data, seq=3, ack=1, info='l')
@t= 7 C1 Sent: Frame(kind=data, seq=3, ack=0, info='c')
@t= 8 C0 Sent: Frame(kind=data, seq=1, ack=1, info='e')
@t= 9 C1 Sent: Frame(kind=data, seq=4, ack=1, info='o')
@t=10 C0 Sent: Frame(kind=data, seq=2, ack=1, info='l')
@t=11 C1 Sent: Frame(kind=data, seq=2, ack=2, info='l')
@t=12 C0 Sent: Frame(kind=data, seq=3, ack=2, info='l')
@t=13 C1 Sent: Frame(kind=data, seq=3, ack=3, info='c')
@t=14 C0 Sent: Frame(kind=data, seq=4, ack=3, info='o')
@t=15 C1 Sent: Frame(kind=data, seq=4, ack=4, info='o')
@t=16 C0 Sent: Frame(kind=data, seq=5, ack=4, info=' ')
@t=17 C1 Sent: Frame(kind=data, seq=5, ack=5, info='m')
@t=18 C0 Sent: Frame(kind=data, seq=6, ack=5, info='W')
@t=19 C1 Sent: Frame(kind=data, seq=6, ack=6, info='e')
@t=20 C0 Sent: Frame(kind=data, seq=7, ack=6, info='o')
@t=21 C1 Sent: Frame(kind=data, seq=7, ack=7, info=' ')
@t=22 C0 Sent: Frame(kind=data, seq=0, ack=7, info='r')
@t=23 C1 Sent: Frame(kind=ack , seq=None, ack=0, info=None)
@t=24 C0 Sent: Frame(kind=data, seq=1, ack=7, info='l')
@t=25 C1 Sent: Frame(kind=ack , seq=None, ack=1, info=None)
@t=26 C0 Sent: Frame(kind=data, seq=2, ack=7, info='d')
@t=27 C1 Sent: Frame(kind=ack , seq=None, ack=2, info=None)
@t=28 C0 Sent: Frame(kind=data, seq=3, ack=7, info='!')
@t=29 C1 Sent: Frame(kind=ack , seq=None, ack=3, info=None)
@t=30 C0 Sent: None
@t=31 C1 Sent: None
@t=32 C0 Sent: None
@t=33 C1 Sent: None
@t=34 C0 Sent: None
@t=35 C1 Sent: None
@t=36 C0 Sent: Frame(kind=data, seq=4, ack=7, info=' ')
@t=37 C1 Sent: Frame(kind=ack , seq=None, ack=4, info=None)
@t=38 C0 Sent: Frame(kind=err , seq=5, ack=7, info='H')
@t=39 C1 Sent: None
@t=40 C0 Sent: Frame(kind=data, seq=6, ack=7, info='A')
@t=41 C1 Sent: Frame(kind=data, seq=0, ack=4, info='t')
@t=42 C0 Sent: Frame(kind=data, seq=7, ack=0, info='H')
@t=43 C1 Sent: Frame(kind=data, seq=1, ack=4, info='o')
@t=44 C0 Sent: Frame(kind=data, seq=5, ack=1, info='H')
@t=45 C1 Sent: Frame(kind=data, seq=2, ack=5, info=' ')
@t=46 C0 Sent: Frame(kind=data, seq=6, ack=2, info='A')
@t=47 C1 Sent: Frame(kind=data, seq=3, ack=6, info='t')
@t=48 C0 Sent: Frame(kind=data, seq=7, ack=3, info='H')
@t=49 C1 Sent: Frame(kind=data, seq=4, ack=7, info='h')
@t=50 C0 Sent: Frame(kind=data, seq=0, ack=4, info='A')
@t=51 C1 Sent: Frame(kind=data, seq=5, ack=0, info='e')
@t=52 C0 Sent: Frame(kind=ack , seq=None, ack=5, info=None)
```

Firstly we print the network layer start state for each DataLinkLayer object, and show at which times will there be an error.

After that outputs of each device for all 100 timesteps are shown in the 2 given screenshots.

Finally, We show the received data for each device.

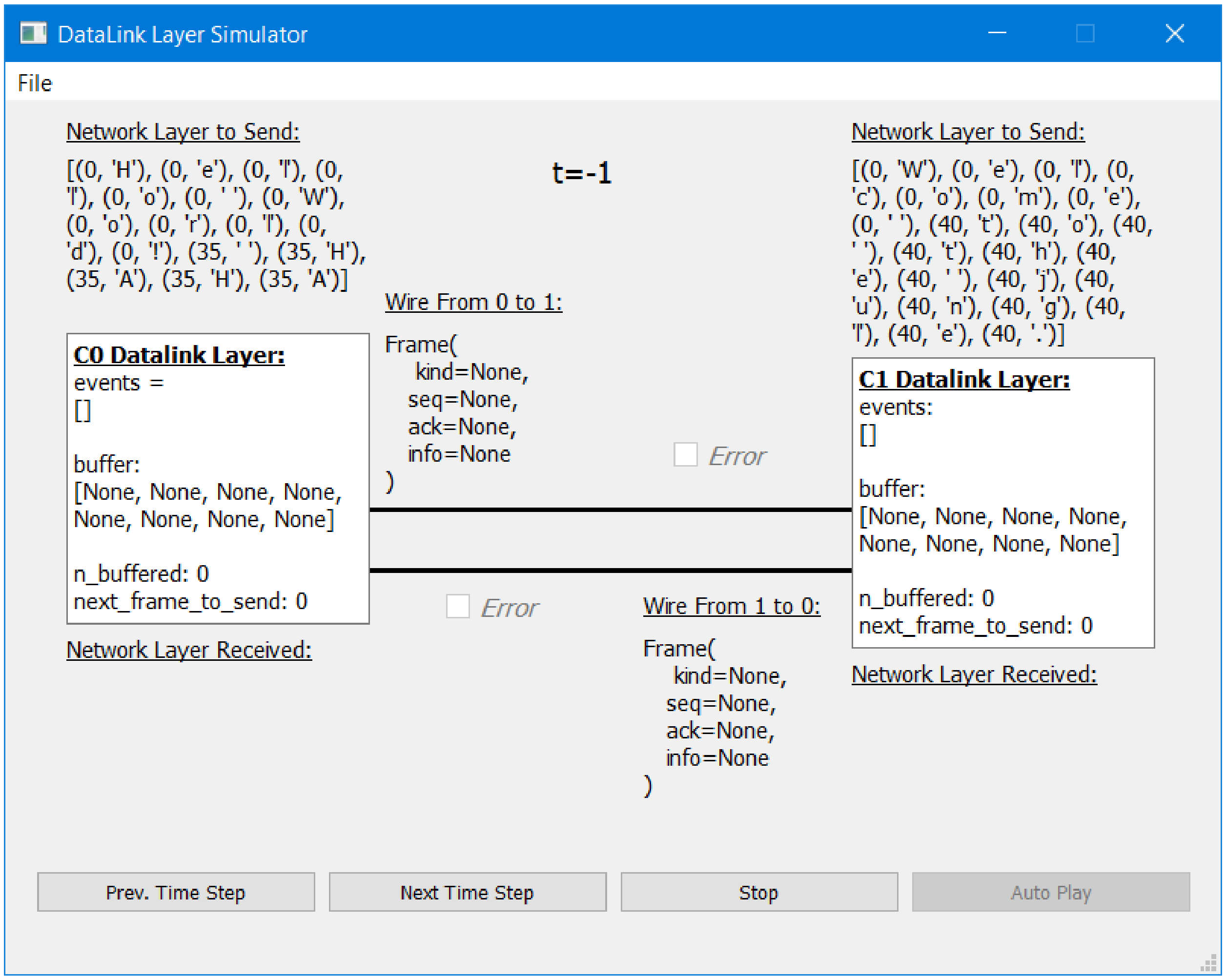


```
@t=52 C0 Sent: Frame(kind=ack , seq=None, ack=5, info=None)
@t=53 C1 Sent: Frame(kind=data, seq=6, ack=0, info=' ')
@t=54 C0 Sent: Frame(kind=ack , seq=None, ack=6, info=None)
@t=55 C1 Sent: Frame(kind=data, seq=7, ack=0, info='j')
@t=56 C0 Sent: Frame(kind=ack , seq=None, ack=7, info=None)
@t=57 C1 Sent: Frame(kind=data, seq=0, ack=0, info='u')
@t=58 C0 Sent: Frame(kind=ack , seq=None, ack=0, info=None)
@t=59 C1 Sent: Frame(kind=data, seq=1, ack=0, info='n')
@t=60 C0 Sent: Frame(kind=ack , seq=None, ack=1, info=None)
@t=61 C1 Sent: Frame(kind=data, seq=2, ack=0, info='g')
@t=62 C0 Sent: Frame(kind=ack , seq=None, ack=2, info=None)
@t=63 C1 Sent: Frame(kind=data, seq=3, ack=0, info='l')
@t=64 C0 Sent: Frame(kind=ack , seq=None, ack=3, info=None)
@t=65 C1 Sent: Frame(kind=data, seq=4, ack=0, info='e')
@t=66 C0 Sent: Frame(kind=ack , seq=None, ack=4, info=None)
@t=67 C1 Sent: Frame(kind=err , seq=5, ack=0, info='.')
@t=68 C0 Sent: None
@t=69 C1 Sent: None
@t=70 C0 Sent: None
@t=71 C1 Sent: None
@t=72 C0 Sent: None
@t=73 C1 Sent: Frame(kind=data, seq=5, ack=0, info='.')
@t=74 C0 Sent: Frame(kind=ack , seq=None, ack=5, info=None)
@t=75 C1 Sent: Frame(kind=data, seq=5, ack=0, info='.')
@t=76 C0 Sent: None
@t=77 C1 Sent: None
@t=78 C0 Sent: None
@t=79 C1 Sent: None
@t=80 C0 Sent: None
@t=81 C1 Sent: None
@t=82 C0 Sent: None
@t=83 C1 Sent: None
@t=84 C0 Sent: None
@t=85 C1 Sent: None
@t=86 C0 Sent: None
@t=87 C1 Sent: None
@t=88 C0 Sent: None
@t=89 C1 Sent: None
@t=90 C0 Sent: None
@t=91 C1 Sent: None
@t=92 C0 Sent: None
@t=93 C1 Sent: None
@t=94 C0 Sent: None
@t=95 C1 Sent: None
@t=96 C0 Sent: None
@t=97 C1 Sent: None
@t=98 C0 Sent: None
@t=99 C1 Sent: None

C0 Network Layer Received:
Welcome to the jungle.
C1 Network Layer Received:
Hello World! HAHA
[Finished in 1.0s]
```

## GUI Interface by running mainwindow.exe:

Showing the Simulation in CLI is a big hassle and can be hard to visualize (and/or debug) what is happening. That is why we created a GUI that shows what happens exactly for each timestep; not only for the objects' input and output, but also for some elements of the inner state of each device.

Starting "mainwindow.py" (or "mainwindow.exe") gives the following:

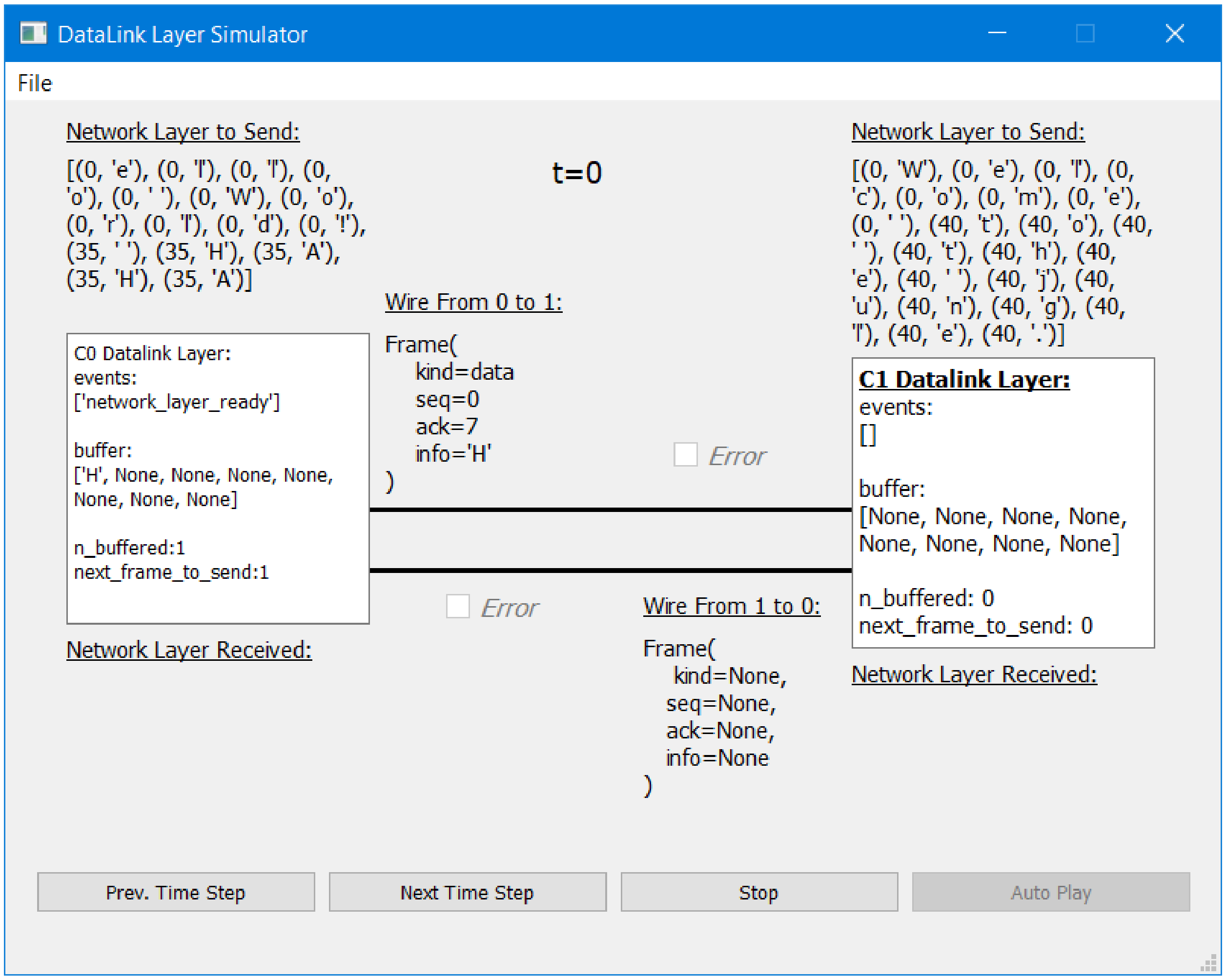


```
Network Layer to Send:
[(0, 'H'), (0, 'e'), (0, 'l'), (0, 'l'), (0, 'o'), (0, ' '), (0, 'W'), (0, 'o'), (0, 'r'), (0, 'l'), (0, 'd'), (0, '!'), (35, ' '), (35, 'H'), (35, 'A'), (35, 'H'), (35, 'A')]

t=-1

Network Layer to Send:
[(0, 'W'), (0, 'e'), (0, 'l'), (0, 'c'), (0, 'o'), (0, 'm'), (0, 'e'), (0, ' '), (40, 't'), (40, 'o'), (40, ' '), (40, 't'), (40, 'h'), (40, 'e'), (40, ' '), (40, 'j'), (40, 'u'), (40, 'n'), (40, 'g'), (40, 'l'), (40, 'e'), (40, '.')]

C0 Datalink Layer:
events =
[]

buffer:
[None, None, None, None, None, None, None, None]

n_buffered: 0
next_frame_to_send: 0

Wire From 0 to 1:
Frame(
    kind=None,
   seq=None,
   ack=None,
   info=None
)

C1 Datalink Layer:
events:
[]

buffer:
[None, None, None, None, None, None, None, None]

n_buffered: 0
next_frame_to_send: 0

Wire From 1 to 0:
Frame(
    kind=None,
   seq=None,
   ack=None,
   info=None
)

Network Layer Received:

Network Layer Received:
```

Here we can see the initial state of everything from the network layers of each device to their internal states, to the Frames being sent in each wire to whether an error is happening right now in each wire.

From here we can Press the "Next Time Step Button" to proceed to the next Timestep:

Note that we can press the "Prev. Time Step Button" to show the previous timestep. Also, the "Stop" and "Auto Play" buttons do not do anything because we could not make them work in time. However, we intend to make them work as an extracurricular project in the near future.



Network Layer to Send:
[(0, 'e'), (0, 'l'), (0, 'l'), (0, 'o'), (0, ' '), (0, 'W'), (0, 'o'), (0, 'r'), (0, 'l'), (0, 'd'), (0, '!'), (35, ' '), (35, 'H'), (35, 'A'), (35, 'H'), (35, 'A')]

t=0

Network Layer to Send:
[(0, 'W'), (0, 'e'), (0, 'l'), (0, 'c'), (0, 'o'), (0, 'm'), (0, 'e'), (0, ' '), (40, 't'), (40, 'o'), (40, ' '), (40, 't'), (40, 'h'), (40, 'e'), (40, ' '), (40, 'j'), (40, 'u'), (40, 'n'), (40, 'g'), (40, 'l'), (40, 'e'), (40, '.')]

Wire From 0 to 1:
```
Frame(
    kind=data
    seq=0
    ack=7
    info='H'
)
```

C0 Datalink Layer:
events:
['network_layer_ready']

buffer:
['H', None, None, None, None, None, None, None]

n_buffered:1
next_frame_to_send:1

**C1 Datalink Layer:**
events:
[]

buffer:
[None, None, None, None, None, None, None, None]

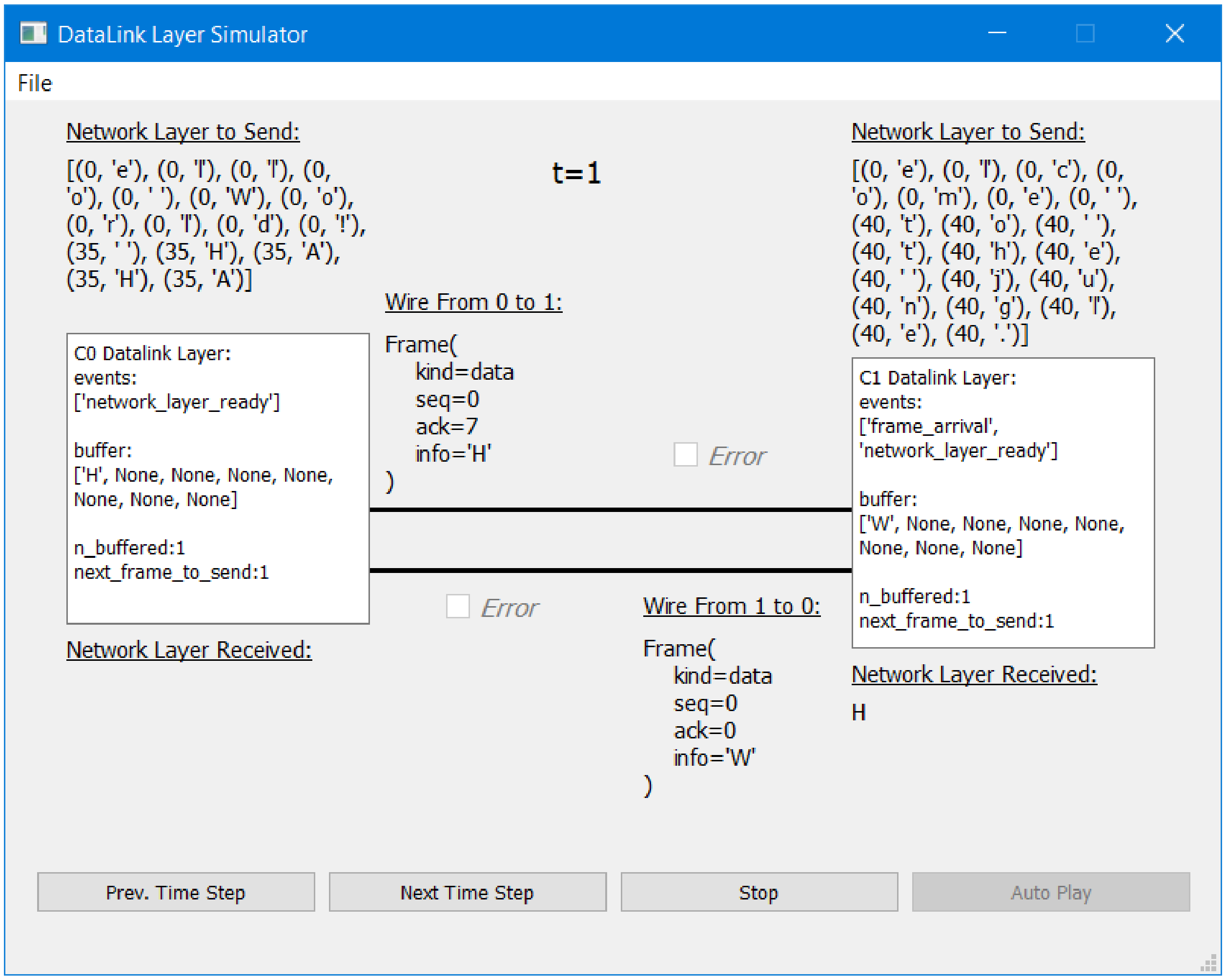
n_buffered: 0
next_frame_to_send: 0

Wire From 1 to 0:
```
Frame(
    kind=None,
    seq=None,
    ack=None,
    info=None
)
```

Network Layer Received:

Network Layer Received:

We can see that $C_0$ has found the event “network_layer_ready” and taken the first packet from the network layer to put it in the buffer and send it as a frame to the other device.



t=1

Network Layer to Send:

[(0, 'e'), (0, 'l'), (0, 'l'), (0, 'o'), (0, ' '), (0, 'W'), (0, 'o'), (0, 'r'), (0, 'l'), (0, 'd'), (0, '!'), (35, ' '), (35, 'H'), (35, 'A'), (35, 'H'), (35, 'A')]

```
C0 Datalink Layer:
events:
['network_layer_ready']

buffer:
['H', None, None, None, None,
None, None, None]

n_buffered:1
next_frame_to_send:1
```

Network Layer Received:

Wire From 0 to 1:

```
Frame(
    kind=data
    seq=0
    ack=7
    info='H'
)
```

Wire From 1 to 0:

```
Frame(
    kind=data
    seq=0
    ack=0
    info='W'
)
```

Network Layer to Send:

[(0, 'e'), (0, 'l'), (0, 'c'), (0, 'o'), (0, 'm'), (0, 'e'), (0, ' '), (40, 't'), (40, 'o'), (40, ' '), (40, 't'), (40, 'h'), (40, 'e'), (40, ' '), (40, 'j'), (40, 'u'), (40, 'n'), (40, 'g'), (40, 'l'), (40, 'e'), (40, '.')]

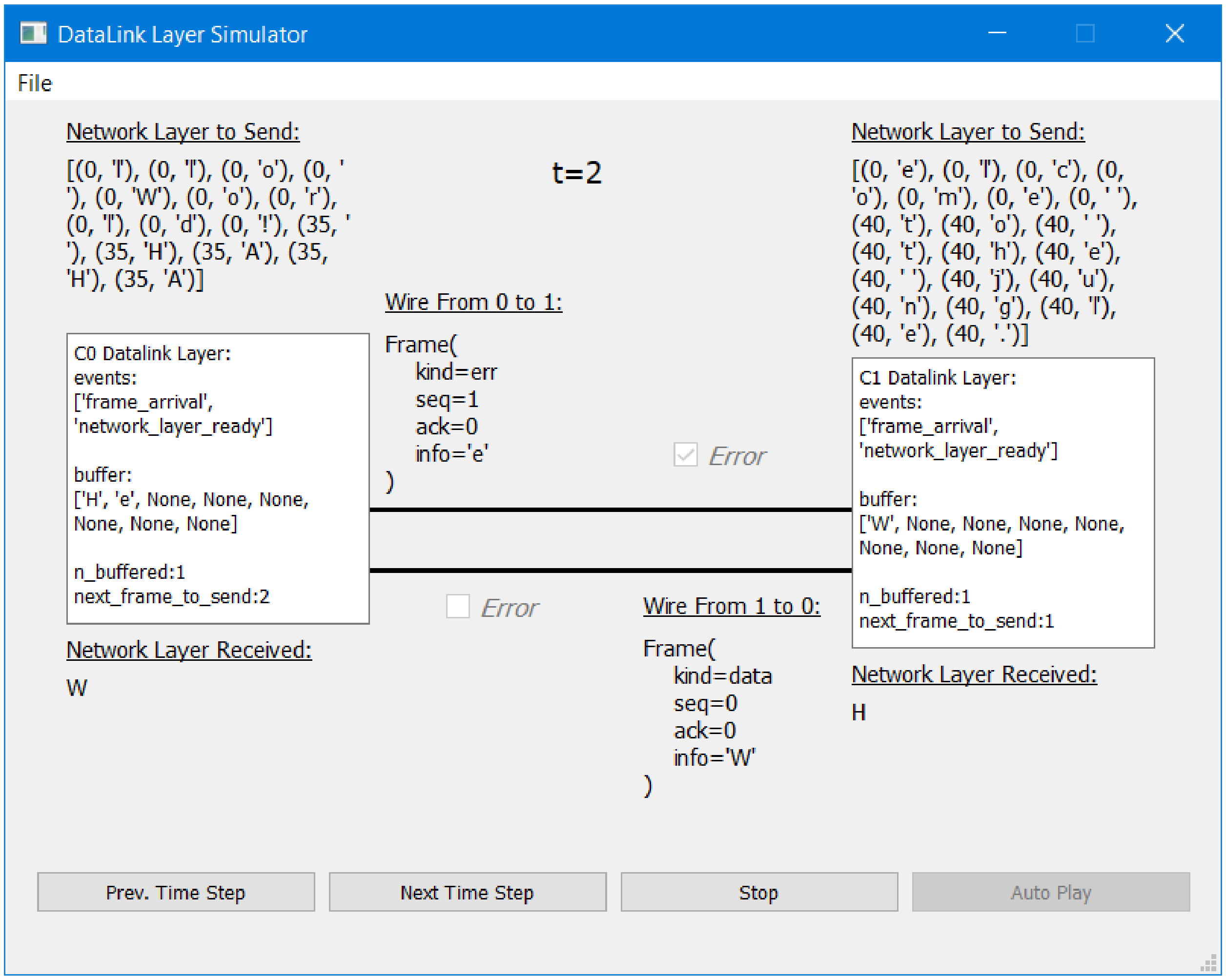
```
C1 Datalink Layer:
events:
['frame_arrival',
'network_layer_ready']

buffer:
['W', None, None, None, None,
None, None, None]

n_buffered:1
next_frame_to_send:1
```

Network Layer Received:

H

Then at the next timestep, $C_1$ found the incoming frame and found its network layer to be ready to send so it set the events to “frame_arrival” and “network_layer_ready”.

It then took the first packet from the input layer, bufferd it and sent it in a new packet. It also received the first packet from the input wire “H” and sent it to the network layer (shown below the DataLinkLayer) and piggybacked the acknowledge for this packet in the already sent frame.



Network Layer to Send:

[(0, 'l'), (0, 'l'), (0, 'o'), (0, ' '), (0, 'W'), (0, 'o'), (0, 'r'), (0, 'l'), (0, 'd'), (0, '!'), (35, ' '), (35, 'H'), (35, 'A'), (35, 'H'), (35, 'A')]

```
C0 Datalink Layer:
events:
['frame_arrival',
'network_layer_ready']

buffer:
['H', 'e', None, None, None,
None, None, None]

n_buffered:1
next_frame_to_send:2
```

Network Layer Received:

W

Wire From 0 to 1:

```
Frame(
    kind=err
    seq=1
    ack=0
    info='e'
)
```

Wire From 1 to 0:

```
Frame(
    kind=data
    seq=0
    ack=0
    info='W'
)
```

Network Layer to Send:

[(0, 'e'), (0, 'l'), (0, 'c'), (0, 'o'), (0, 'm'), (0, 'e'), (0, ' '), (40, 't'), (40, 'o'), (40, ' '), (40, 't'), (40, 'h'), (40, 'e'), (40, ' '), (40, 'j'), (40, 'u'), (40, 'n'), (40, 'g'), (40, 'l'), (40, 'e'), (40, '.')]

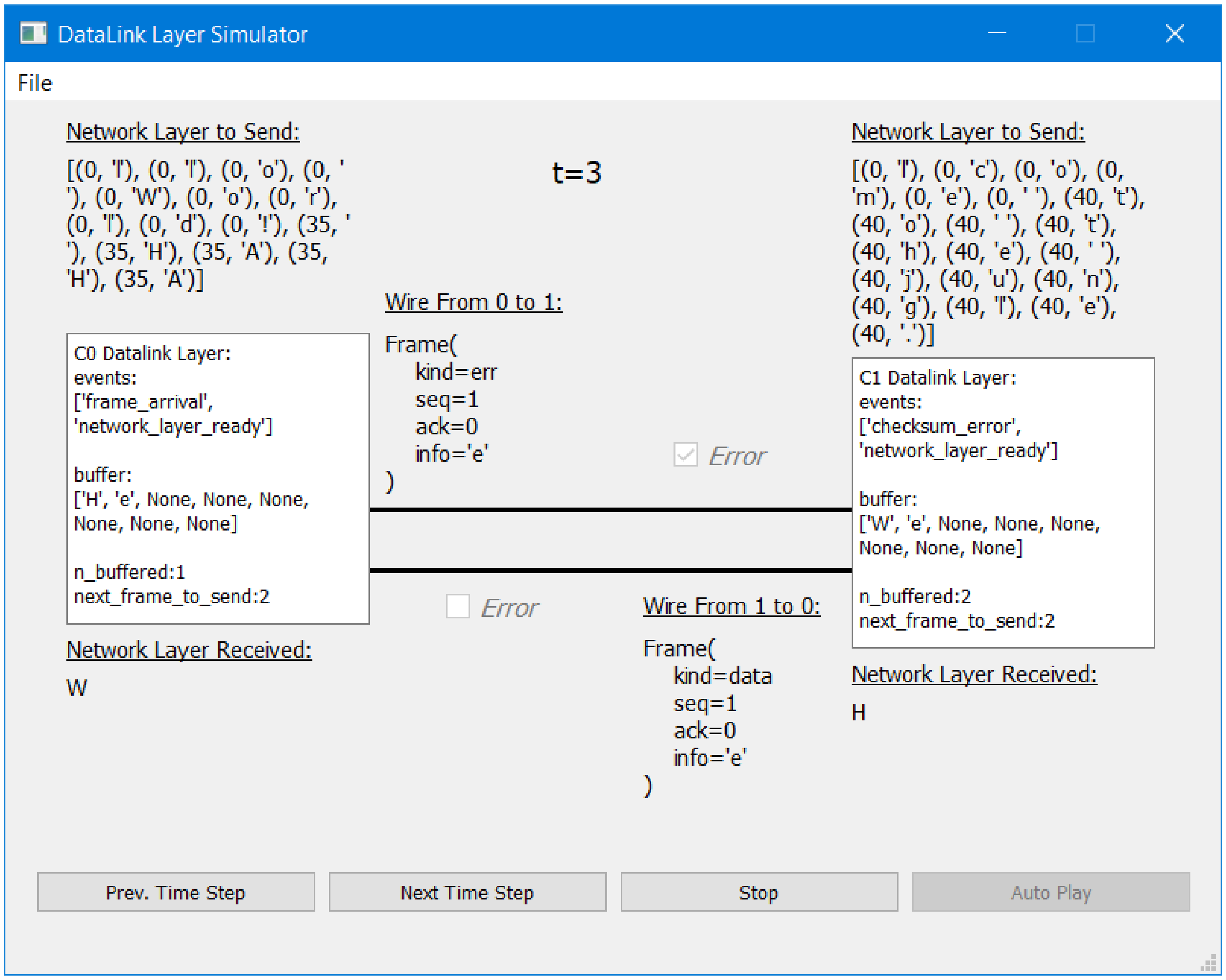
```
C1 Datalink Layer:
events:
['frame_arrival',
'network_layer_ready']

buffer:
['W', None, None, None, None,
None, None, None]

n_buffered:1
next_frame_to_send:1
```

Network Layer Received:

H

In timestep 2, $C_0$ proceeds as normal with receiving the incoming frame and sending the next frame from the network layer with acknowledge of the first received packet.

However, there is an error at time step 2 which causes the sent frame to be an error frame. (as shown by the faint tick in the Error checkbox next to the sent frame.



t=3

Network Layer to Send:

[(0, 'I'), (0, 'l'), (0, 'o'), (0, ' '), (0, 'W'), (0, 'o'), (0, 'r'), (0, 'l'), (0, 'd'), (0, '!'), (35, ' '), (35, 'H'), (35, 'A'), (35, 'H'), (35, 'A')]

```
C0 Datalink Layer:
events:
['frame_arrival',
'network_layer_ready']

buffer:
['H', 'e', None, None, None,
None, None, None]

n_buffered:1
next_frame_to_send:2
```

Network Layer Received:

W

Wire From 0 to 1:

```
Frame(
    kind=err
    seq=1
    ack=0
    info='e'
)
```

[✓ Error]

[ ] Error

Wire From 1 to 0:

```
Frame(
    kind=data
    seq=1
    ack=0
    info='e'
)
```

Network Layer to Send:

[(0, 'I'), (0, 'c'), (0, 'o'), (0, 'm'), (0, 'e'), (0, ' '), (40, 't'), (40, 'o'), (40, ' '), (40, 't'), (40, 'h'), (40, 'e'), (40, ' '), (40, 'j'), (40, 'u'), (40, 'n'), (40, 'g'), (40, 'l'), (40, 'e'), (40, '.')]

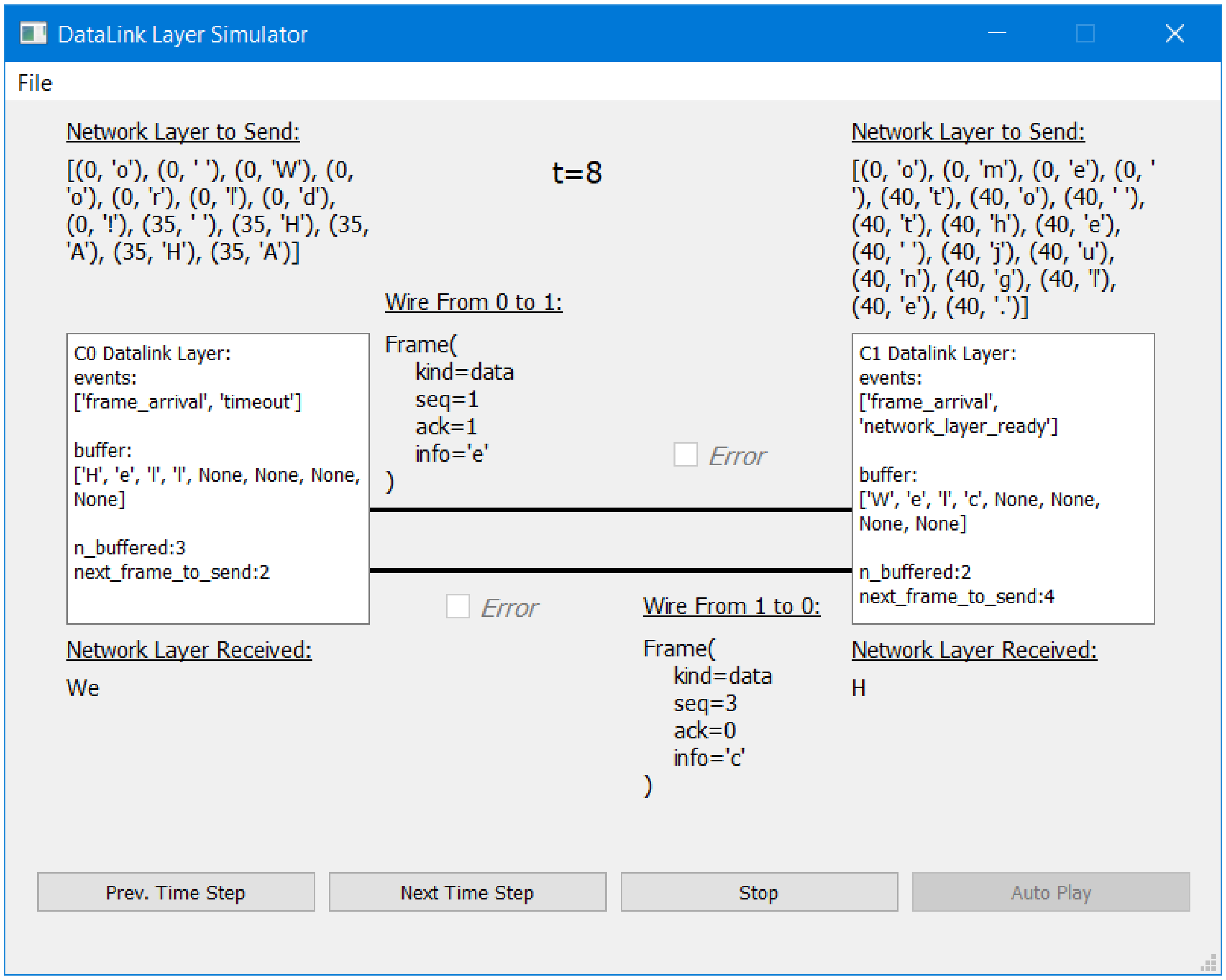
```
C1 Datalink Layer:
events:
['checksum_error',
'network_layer_ready']

buffer:
['W', 'e', None, None, None,
None, None, None]

n_buffered:2
next_frame_to_send:2
```

Network Layer Received:

H

$C_1$ doesn't receive the frame correctly and shows this with the "checksum_error" event. Therefore it doesn't acknowledge this frame (frame seq =1).

$C_1$ keeps ignoring $C_0$'s frames by sending the acknowledge to frame 0 only until $C_0$ detects a timeout event at timestep 8.



t=8

Network Layer to Send:

[(0, 'o'), (0, ' '), (0, 'W'), (0, 'o'), (0, 'r'), (0, 'l'), (0, 'd'), (0, '!'), (35, ' '), (35, 'H'), (35, 'A'), (35, 'H'), (35, 'A')]

```
C0 Datalink Layer:
events:
['frame_arrival', 'timeout']

buffer:
['H', 'e', 'l', 'l', None, None, None, None]

n_buffered:3
next_frame_to_send:2
```

Network Layer Received:

We

Wire From 0 to 1:

```
Frame(
    kind=data
    seq=1
    ack=1
    info='e'
)
```

Wire From 1 to 0:

```
Frame(
    kind=data
    seq=3
    ack=0
    info='c'
)
```

Network Layer to Send:

[(0, 'o'), (0, 'm'), (0, 'e'), (0, ' '), (40, 't'), (40, 'o'), (40, ' '), (40, 't'), (40, 'h'), (40, 'e'), (40, ' '), (40, 'j'), (40, 'u'), (40, 'n'), (40, 'g'), (40, 'l'), (40, 'e'), (40, '.')]

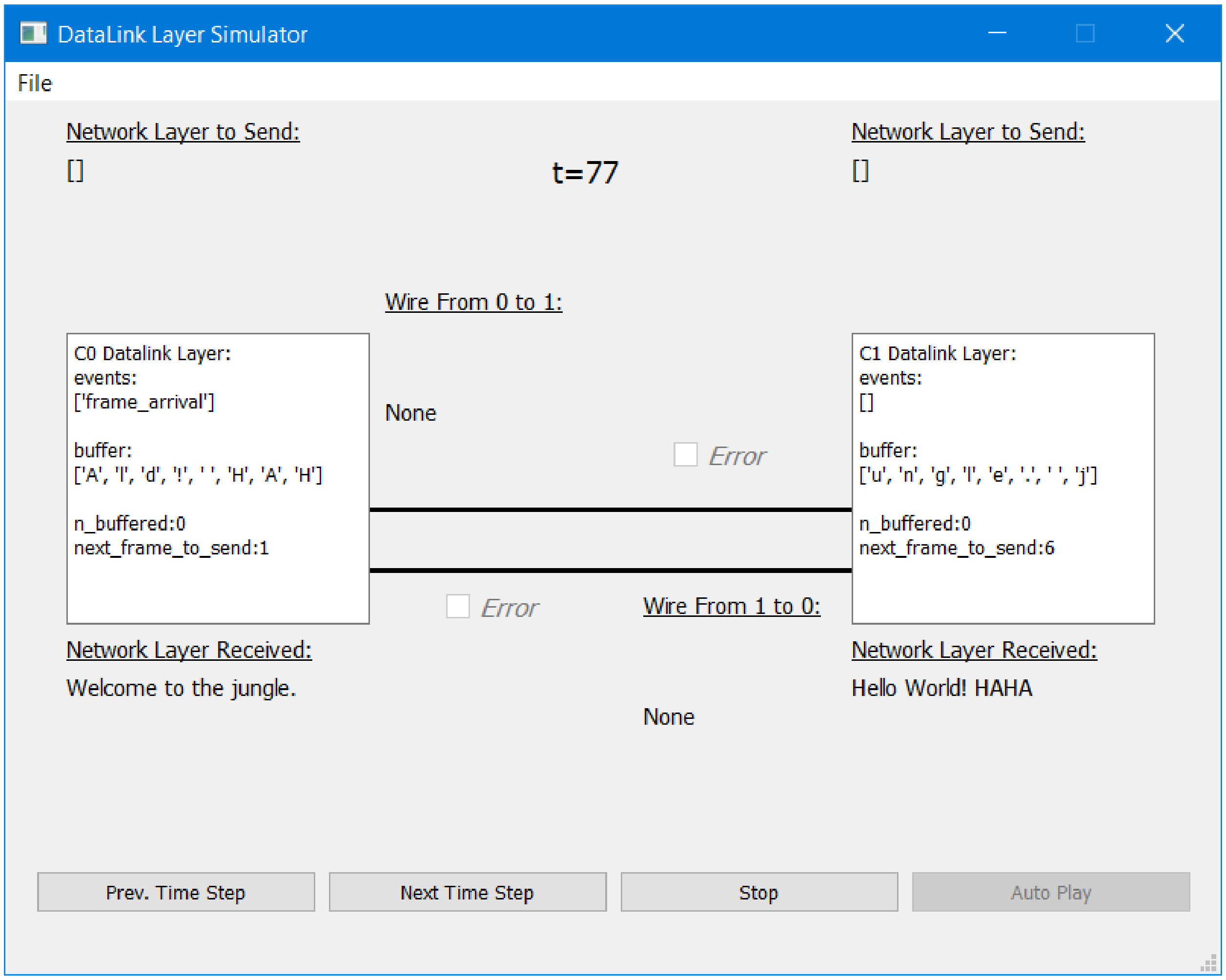
```
C1 Datalink Layer:
events:
['frame_arrival',
'network_layer_ready']

buffer:
['W', 'e', 'l', 'c', None, None,
None, None]

n_buffered:2
next_frame_to_send:4
```

Network Layer Received:

H

Here $C_0$ noticed the timeout event and sends frame 1 again to $C_1$ which then acknowledges it at timestep 9.

The simulation continues correctly until both datalink layers have no data to send from their network layers and have empty buffers (shown by n_buffered=0) at timestep 77.



Network Layer to Send:
[]

t=77

Network Layer to Send:
[]

Wire From 0 to 1:
None

```
C0 Datalink Layer:
events:
['frame_arrival']

buffer:
['A', 'l', 'd', '!', ' ', 'H', 'A', 'H']

n_buffered:0
next_frame_to_send:1
```

```
C1 Datalink Layer:
events:
[]

buffer:
['u', 'n', 'g', 'l', 'e', '.', ' ', 'j']

n_buffered:0
next_frame_to_send:6
```

Wire From 1 to 0:
None

Network Layer Received:
Welcome to the jungle.

Network Layer Received:
Hello World! HAHA

Notice that the example given is filled with special cases that were not shown here. It is recommended to try out the simulator for yourself and see all the coded edge cases turn out working correctly.