

# When Words Collide: Network Simulation as an Exercise in a CS2 Course

---

**A. Michael Berman**

Department of Computer Science  
Rowan College of New Jersey

berman@rowan.edu

---

## **Abstract**

---

We use a simplified simulation of the operation of an Ethernet, implemented in C++, to illustrate important concepts in the CS2 course, among them abstraction, queues, and the use of objects. As a side-benefit, students get an introduction to the principles of operation of a network protocol, and an example of the use of a state machine as a program design technique.

## **Introduction: Motivating the Queue ADT**

---

Most textbooks and instructors introduce the Queue ADT in the CS2 course; see, for example, [3] or [4]. In this paper we show how the queue can be used to implement a simulation. I have been using this simplified model of an Ethernet to show students a realistic application of the Queue. In addition to illustrating the use of the Queue, the network simulation provides a nice example of the power of object-oriented programming, and introduces the students to the basic principles of operation of an Ethernet.

## **Related Work**

Network simulators such as VNET [7] and Netsim [1] were designed primarily to teach students about networking. In contrast, instruction in the use and design of abstract data types motivated the work here, with an introduction to networking concepts providing an extra benefit. Similar techniques have also been used to design a machine simulator [6], but again, the motive in that case was helping students understand computer architecture concepts.

### The Queue ADT

Our Queue ADT is shown as ADT 1.

#### ADT 1 Queue

##### *Characteristics*

A Queue  $Q$  stores items of some type (`queueElementType`), with First-In, First-Out (FIFO) access.<sup>1</sup>

##### *Operations*

```
queueElementType Q.dequeue()
```

Precondition: `!Q.isEmpty()`

Postcondition: `Q = Q` with top removed

Returns: The item  $x$  such that `Q.enqueue(x)` was the least recent invocation of `enqueue` for any item in the list.

```
void Q.enqueue(queueElementType x)
```

Precondition: None

Postcondition: Item  $x$  is added to  $Q$ .

```
queueElementType Q.front()
```

Precondition: `!isEmpty()`

Postcondition: None

Returns: The item  $x$  such that `enqueue(x)` was the least recent invocation of `enqueue` for any item in the list.

```
bool Q.isEmpty()
```

Precondition: None

Postcondition: None

Returns: true if and only if  $Q$  is empty, i.e., contains no data items.

## An Introduction to Ethernet

---

Ethernet is an important standard for building computer networks. Ethernet, developed by Xerox in the early 1980's, is not really a particular set of hardware and software. Rather, it is a set of rules, or a *protocol*, that describes exactly how devices can communicate with each other over a wire (or via radio). The standard IEEE 802.3 defines the protocol, called Carrier Sense Multiple Access with Collision Detection (CSMA/CD). In this paper, we present a simplified description of the CSMA/CD protocol; for a more complete and accurate view, consult a computer networking reference such as [5].

---

1. To simplify the syntax in this paper, we use a macro to define `queueElementType`; a more flexible approach would use the C++ template syntax.

FIGURE 1.

A sample Ethernet configuration

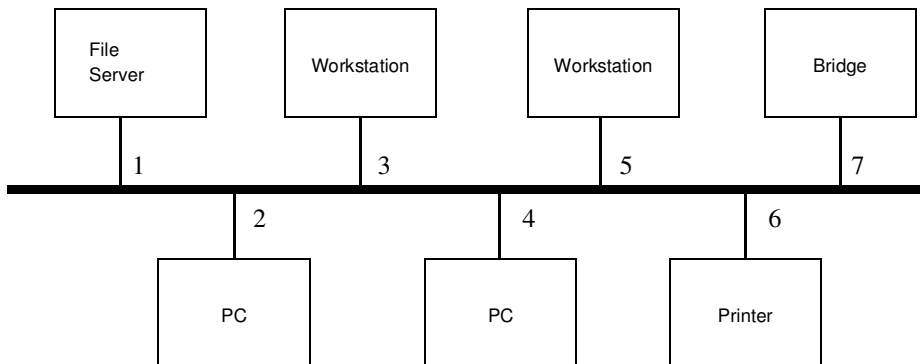
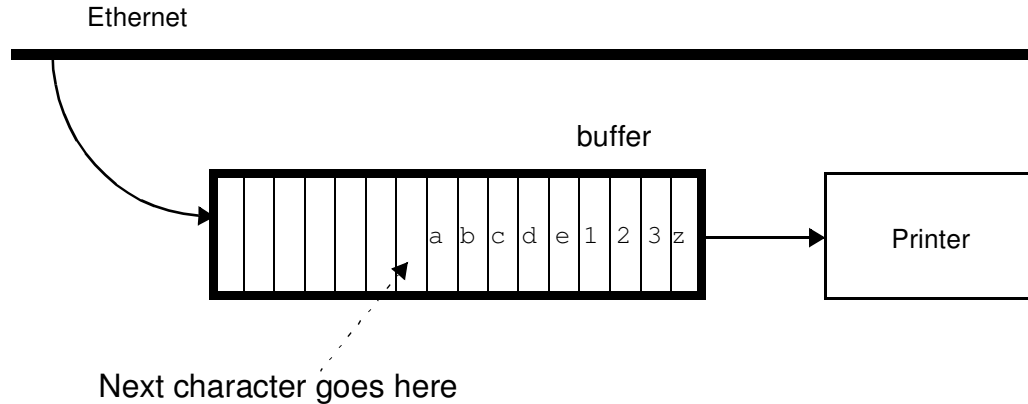


Figure 1 illustrates a typical Ethernet configuration. On an Ethernet, every device has a unique identification (ID) number. Actual Ethernet ID's consist of 6 2-digit hexadecimal numbers; to simplify things here, we'll just assign each device an integer, as shown in Figure 1. For example, the printer is device 6 and the file server is device 1.

Figure 1 shows the logical connections between devices on an Ethernet. Depending on the type of wiring used, the actual connections might look quite different, but the algorithm is always the same. A device communicates with another device by putting messages out onto the Ethernet, preceded by the ID number of the device the message is intended for. For example, if a PC — device 2 — wants to send a document to the printer — device 6 — it puts the characters that make up the document out onto the network, preceded by a header indicating the data is intended for device 6. All the other devices on the network can look at the stream of characters that represents the document, but since they have ID numbers other than 6, they will simply ignore them. The printer, seeing a message for device 6, reads the characters and prints the document.

Let's develop the example of device 2 talking to device 6 in more detail. We can envision the participants in the message as three: the PC, the printer, and the Ethernet itself. Conceptually, what happens is the PC is passing the message to the Ethernet, and the Ethernet is passing the message to the printer. What complicates matters is that these three players may each operate at different speeds. The Ethernet itself is very fast — 10 million bits per second — so it can always keep up with the devices connected to it, but the converse is often not true. Let's suppose that the PC puts the document onto the Ethernet at 100,000 characters per second, but the printer can only process 10,000 characters per second. How can the printer make sure it doesn't miss any characters? At the interface between the Ethernet and the printer will be a buffer. The operation of the buffer is illustrated in Figure 2. The next character received via the Ethernet will be inserted next to the 'a'; the next character read by the printer will be 'z'. This doesn't solve the whole problem — eventually, if nothing else is done, the buffer will overflow — but it is a critical piece of the puzzle. The Queue ADT closely models the operation of a buffer.

FIGURE 2. Operation of a buffer on an Ethernet



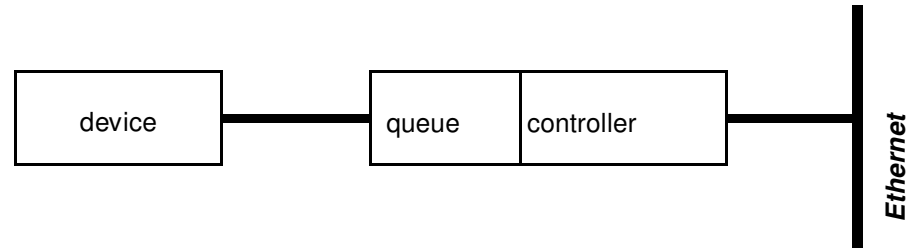
Each device on the Ethernet is connected to the net via special hardware called a controller. We are going to model the performance of the Ethernet when two or more controllers try to send a message at the same time. The network consists, conceptually, of a single cable, to which multiple devices are attached. You can think of it like an old-fashioned speaking tube, in which people in different parts of a building or a ship communicate by shouting into the tube. Before you try to call someone, you first listen to make sure no one is speaking. But there's always a chance that two people listen at the same time, hear nothing, and then shout at the same time.

Likewise, two (or more) controllers on the Ethernet may attempt to send a message at the same time. The network can only transmit one message at a time, so when two controllers start sending simultaneously, the message is garbled. The controllers listening to the network are able to detect the problem, so they ignore the garbled message; likewise, the two controllers trying to send will each notice that the message didn't go through. This condition is called a *collision*. When a controller detects a collision, it will back off and try again later. Making the back off time random makes it unlikely that the two controllers will try to send again at the same time.

A device connected to the network, such as a PC or a printer, is not expected to know anything about how the Ethernet works. All it should have to know is that it can send and receive data on the network. When a device sends data to the controller, the controller may or may not be able to send it right out to the network — after all, some other controller may be talking. Likewise, when information is being captured by the controller, the device may not be able to keep up with the speed of the Ethernet. So as Figure 3 shows, the controller will have a queue at its interface with the device. (Actually, it will need two queues, one for each direction.)

FIGURE 3.

Device/Controller Interface



For the purposes of our simulation, we'll ignore the process of reading information off the network, and instead focus on sending. This will be sufficient to illustrate network collisions. By simulating the performance of the network under various assumptions about the number of devices, the frequency of messages, and the size of messages, we can estimate the frequency of network collisions. One performance characteristic of a network is its *effective bandwidth ratio*. This is a measure of the ratio between the amount of data actually transmitted and the maximum possible data. For example, if the network is transmitting data half the time and is idle half the time, its effective bandwidth ratio is 1/2. As the number of messages goes up on an Ethernet, the effective bandwidth ratio will go up; but when too many devices are trying to communicate at once, collisions will occur, and the effective bandwidth will decline — after all, each time a collision occurs, data is not transmitted.

We will implement controllers and devices as C++ classes. The controller class will use the Queue ADT. “Real” network controllers can, of course, send and receive simultaneously, but to simulate the controllers, we divide the process into two phases. First, we check with each controller to see whether it has data to send. If no controllers are sending, the network is quiet; if exactly one controller sends, the network contains the data sent by that controller. When two or more controllers try to send, a collision occurs. In the second phase, each controller receives the contents of the network. The values `netQuiet` and `netCollide` are special constants defined by an enum in `control.h`. The controllers either receive the data sent by one of the controllers, or else one of the special values. The code for the `main` function of the simulation is shown below. Note that the connections illustrated in Figure 3 are mirrored by the code — the top level, which can be thought of as representing the whole network, knows only about the controllers. As we'll see in a moment, the code for the controllers calls the code for the devices.

---

### Code Example 1

### Main driver for the simulation

```
#include <iostream.h>
#include "control.h"
int main()
{
    const int ctrl_cnt = 100;
```

```
Controller ctrl[ctrl_cnt];
int collisions = 0, periods, step;
cout << "Enter number of periods: ";
cin >> periods;
// loop once for each time period
for (int step = 0; step < periods; step++) {
    int senders = 0;
    int netChar = netQuiet;
    int c;
    // loop through each controller and execute
    // its write phase
    for (c = 0; c < ctrl_cnt; c++) {
        int ch = ctrl[c].phase1();
        if (ch != 0) {
            cout << c << ":" << ch << '\n';
            senders++;
            netChar = ch;
        }
    }
    // check for collisions
    if (senders > 1) {
        netChar = netCollide;
        collisions++;
    }
    // report the results to each controller
    for (c = 0; c < ctrl_cnt; c++)
        ctrl[c].phase2(netChar);
}
cout << "\nCollision count: " <<
        collisions << "\n\n";
return 0;
}
```

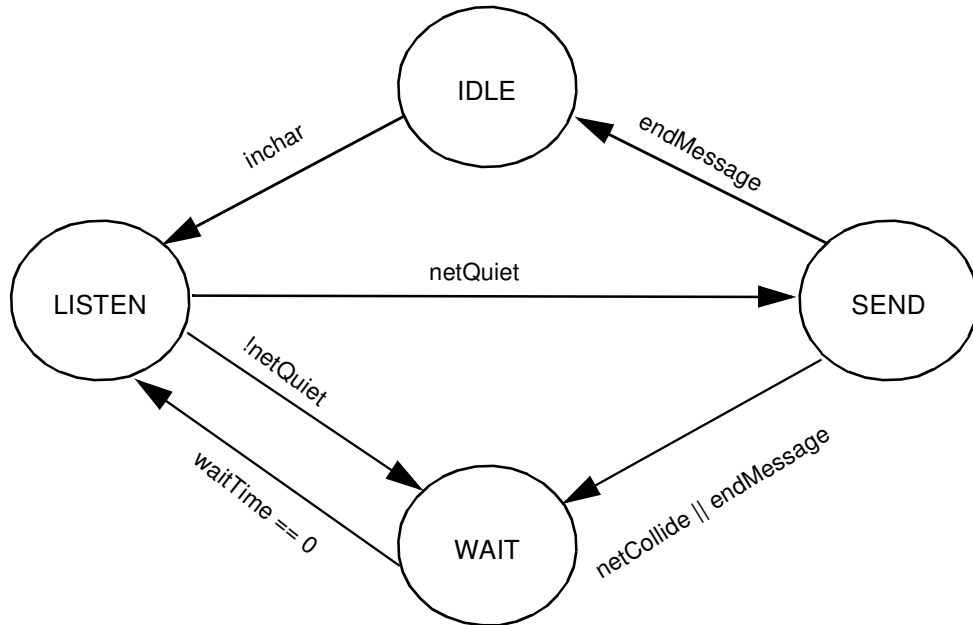
This code declares an array `c` of 100 `Controller` objects. Recall that the way C++ classes work, each controller will have its own values for its variables. Thus different controllers will be in various states, but all this gets taken care of by C++. In other words, each controller has the same logic, but each operates independently. This illustrates one of the advantages of an object-oriented language.

Now let's consider the controller. In phase 1, the controller starts by calling the update function for its associated device — if the device is trying to send data to the network, a value will be returned and stored in `inchar`; otherwise, 0 will be returned. This isn't too realistic, because it forces that controller and the device to operate synchronously, i.e., at exactly the same rate, but it's good enough for our simulation. If a character arrives from the device, it is enqueued in `outQ`, which corresponds to the queue in Figure 3.

The operation of the controller is described by a set of states: Idle, Listen, Wait, Send. We'll call each pair of calls to `phase1` and `phase2` a *step*. Each step begins in one of the four states, and, depending on the results of calling the device and upon the contents of

the queue, may stay in the same state or transit to another state. The operation of the controller can be illustrated by the state diagram in Figure 4. The circles represent the four states, and the arrows represent the possible transitions. The text along the arrows summarizes the conditions under which a transition occurs. Initially each controller begins in the Idle state.

**FIGURE 4.** State diagram for network controller



The only transition that can occur in phase 1 is from the Idle state to the Listen state — if the controller is idle, and the device sends a character, the controller should now listen to the state of the network. Phase 1 communicates with the device, and also sends characters out to the network. A character is sent to the network only from the Send state; the character always comes from the `outQ`. In phase 2, the network sends to the controller its current contents — a character, or the one of the special values `netQuiet` or `netCollide`. Note that in phase 2, the controller does not have any output. Another subtle point is that phase 1 looks at the front of the queue using the front function and doesn't dequeue the character, since phase 2 needs to examine the same character to determine whether it has reached the end of a message. (`endMessage` is a special value defined in `device.h`.) The rest of the details can be found in Code Example 2.

---

**Code Example 2**

**The Controller class**

```

// control.h
#include "queue.h"
#include "device.h"
enum {netQuiet = 0, netCollide = -100 };
  
```

```
const maxWait = 10;
class Controller {
private:
    enum state { Idle, Listen, Wait, Send };
    int currentState;
    QUEUE outQ;
    Device dev;
    int waitTime;
public:
    Controller();
    int phase1();
    void phase2(int netchar);
};

// control.cpp -- implementation for controller
#include "control.h"
Controller::Controller()
{
    currentState = Idle;
}
int Controller::phase1()
{
    int inchar = dev.update();
    int outchar = 0;
    if (inchar)
        outQ.enqueue(inchar);
    switch(currentState) {
        case Idle:
            if (inchar)
                currentState = Listen;
            break;
        case Listen:
        case Wait:
            break;
        case Send:
            if (outQ.isEmpty())
                currentState = Idle;
            else
                outchar = outQ.front();
            break;
    }
    return outchar;
}
void Controller::phase2(int netchar)
{
    int outchar;
    switch(currentState) {
        case Idle:
            break;
        case Listen:
```

```
        currentState =
            (netchar == netQuiet)? Send : Wait;
        break;
    case Wait:
        if (waitTime > 0)
            waitTime--;
        else
            currentState = Listen;
        break;
    case Send:
        if (netchar == netCollide) {
            waitTime = random(maxWait) + 1;
            currentState = Wait;
        }
        else {
            outchar = outQ.dequeue();
            if (outchar == endMessage)
                currentState =
                    outQ.isEmpty()? Idle : Wait;
        }
        break;
    }
}
```

Finally, we look at the code for the devices. Note that since each controller declares a device, there will be multiple device objects operating simultaneously, each using the same logic but potentially in different states. The device is either Idle or Sending. An Idle device picks a random number between 0 and the constant `avDelay - 1`. Each time it chooses 0, the device moves to the Sending state and returns nonzero values to the controller, ending with the special value `endMessage`, and then returning to the Idle state.

---

**Code Example 3****The Device class**

```
// device.h
#include <stdlib.h>
const int avDelay = 5000;
const int avMsg = 50;
const int endMessage = -1;
class Device {
private:
    enum state { Idle, Sending };
    state currentState;
    int messageLength;
public:
    Device();
    int update();
};
```

---

## Sample Simulation Results

---

```
// device.cpp
#include "device.h"
Device::Device()
{
    currentState = Idle;
}
int Device::update() {
    int outChar = 0;
    // if we're sending, put a character out
    // to the controller
    if (currentState == Sending) {
        if (messageLength-- > 0)
            outChar = messageLength + 32;
        else {
            currentState = Idle;
            outChar = -1;
        }
    }
    // if not, pick random number to decide whether
    // to start sending
    else // CurrentState == Idle
        if (random(avDelay) == 0) {
            messageLength = random(avMsg);
            currentState = Sending;
        }
    return(outChar);
}
```

## Sample Simulation Results

---

In Figure 5 on page 11, we show sample data from running the network simulation. To collect the data shown, we set the average delay between messages to 5000 and the average message length to 10. For each controller number, a trial of 1,000,000 steps was run, and the number of transmissions and collisions are shown. As expected, the data show that as the number of controllers is increased, the amount of data (as indicated by the effective bandwidth ratio) goes up, until the growing number of collisions eats into the available bandwidth.

There are a number of modifications and improvements that can be made to this simulation in order to model more closely the real operation of an Ethernet. The instructor should help the student understand that a simulation is by definition an abstraction from reality, and that whenever possible the results of simulation must be confirmed by comparing with real-world data. The choice of features to include in a simulation is quite difficult, and requires judgement based upon experience.

Another interesting direction to extend the example is in the area of object-oriented programming. The Device class can be redesigned as a abstract base class, with various child device classes inheriting their interface from the parent Device class, but each

---

## Conclusion

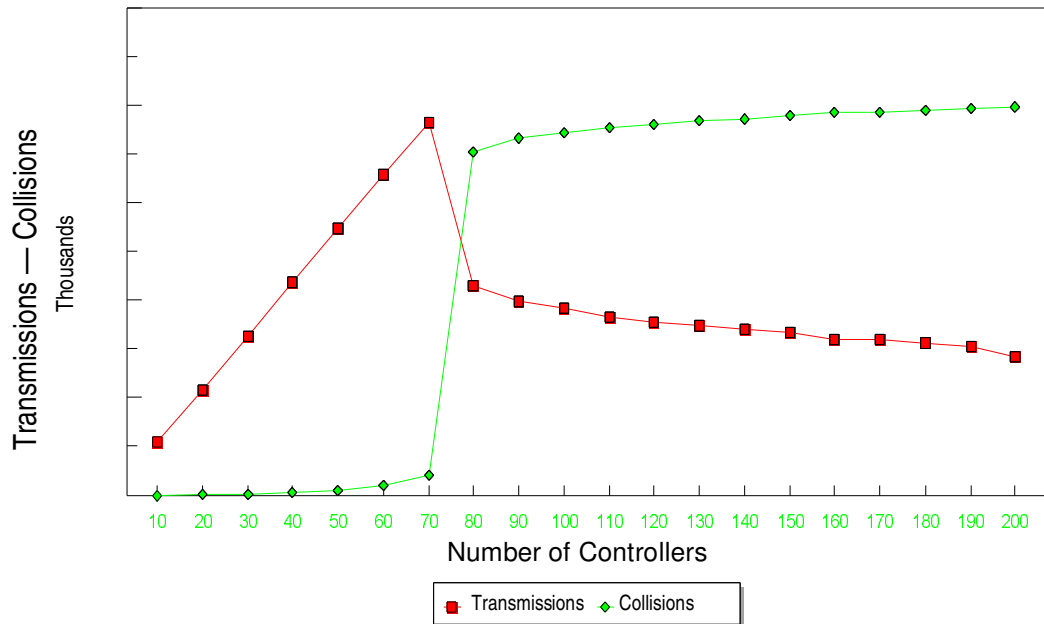
---

exhibiting different behavior. Using polymorphism, different Controllers could interface with different devices, with only a trivial change to the code for the Controllers. (C++ requires polymorphic references to be made via a pointer.)

---

**FIGURE 5.**

Sample data from network simulation



---

## Conclusion

---

This relatively simple network simulation has many attractive features: it provides a “real world” example of Queues, it illustrates the power of class-based programming, it gives an opportunity to introduce the Finite-State Machine as a programming technique, and it familiarizes the student with the fundamental features of an Ethernet. Students can spend many worthwhile hours running the simulation under different conditions, improving the realism of the simulation, and instrumenting the code to collect additional information. This exercise appears in an upcoming textbook by the author [2].

---

## References

---

- [1] Barnett, B. Lewis, III, “An Ethernet Performance Simulator for Undergraduate Networking”, *Papers of the Twenty-Fourth SIGCSE Technical Symposium on Computer Science Education*, 1993.
- [2] Berman, A. Michael. *Data Structures via C++: Objects by Evolution*. Oxford Univer-

---

## References

---

- sity Press, 1996.
- [3] Budd, Timothy A. *Classic Data Structures in C++*. Addison-Wesley, 1994
  - [4] Headington, Mark R. and David R. Riley. *Data Abstraction and Structures Using C++*. D.C. Heath and Company, 1994.
  - [5] Martin, James with Kathleen Kavanagh Chapman. *Local Area Networks: Architectures and Implementations*. Prentice-Hall, 1989.
  - [6] Reid, Richard J., “Object-Oriented Simulation of Computer Architectures Using C++”, *Papers of the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education*, 1995.
  - [7] Tymann, Paul, “VNET: A Tool for Teaching Computer Networking to Undergraduates”, *Papers of the Twenty-Second SIGCSE Technical Symposium on Computer Science Education*, 1991.