You Can Teach Computer Networking in High School

Brian Broll, Hamid Zare, Dung Nguyen Do, Mohini Misra and Akos Ledeczi
Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

***Abstract*—NetsBlox is a visual blocks-based programming language extending Snap! with a few carefully selected abstractions to support distributed programming. In this paper, we argue that with the help of NetsBlox, some of the fundamental concepts of computer networking can be included in the high school computer science curriculum. Specifically, we describe a set of curricular units that would fit nicely in the AP Computer Science Principles course. High school students in two short-term studies showed great engagement and were able to complete simple computer networking tasks.**

***Index Terms*—visual programming, distributed programming, computer science education, computer networking**

I. Introduction

More and more countries, states and cities realize that computer literacy is one of the most important skills of the 21st century workforce. Consequently, more and more K12 schools teach computer science one way or another. However, there is not enough attention being paid to the topic of computer networking and distributed computation even though the vast majority of the most popular computer applications are networked. The web, text messaging, Twitter, Facebook and other social networks, multi-player games, Spotify, Netflix, Amazon Echo, Siri, Bing and YouTube are just a few of the most popular examples. While some courses like the new AP Computer Science Principles[1] high school course in the US have curricular units related to the internet, there is a lack of educational tool support to teach these topics effectively. Currently, concurrency, computer networking and distributed programming are considered advanced topics that are only taught in college to computer science majors. We argue that with the help of a carefully designed visual representation, an intuitive user interface and a sophisticated cloud-based infrastructure, it will be possible to teach some of the key underlying concepts of distributed computation and computer networking to high school students.

To this end, we have created NetsBlox[2], [3], a visual programming environment that extends Snap![4] with a few carefully selected abstractions that enables user to create distributed applications. Remote Procedure Calls (RPC) provide access to a set of online data sources such as maps, weather, seismic information, astronomy imagery, etc. An RPC is just like a custom block in Snap!; it just happens to run on the server. Note that an RPC is more than just a call to a web API. Related RPCs are grouped together into services that have state information. For example, the map service returns an image from Google Maps but maintains information for subsequent coordinate transformations from screen coordinates to latitude and longitude and vice versa. It also caches maps to avoid unnecessary API calls.

NetsBlox also incorporates message passing. Messages are similar to events already present in Snap! but they can also contain data payload and can be sent across the network to other NetsBlox programs running on different computers. RPCs and messages enable the creation of a wide range of distributed applications and consequently, they open the door to teaching some of the fundamental concepts of distributed computing.

The original goals of NetsBlox were as follows:

- By building on Snap!, NetsBlox provides natural progression to students who take one of the popular AP CSP courses, the Beauty and Joy of Computing (BJC)[5] class and consequently, novel curricular units can be easily incorporated into BJC.
- Providing access to vast arrays of data on the Internet right from the visual programming environment in a uniform manner will empower students to create innovative science projects and bring STEM concepts into CS education at the same time.
- The ability to create multi-player games will provide increased motivation for many students making them creators and not just consumers of digital entertainment.

After teaching a few high school classes in various settings, we realized that several engaging curricular units can be created with NetsBlox specifically targeting computer networking. Each introduces a fundamental concept in networking such as addressing, latency, message loss, protocols, etc., and explains their importance through hands-on exercises. Throughout these units, the students remain engaged because they are solving real networking problems while collaborating with their peers and having fun at the same time.

A. Background

A NetsBlox project consists of one or more subprojects called *roles*. Each role has its own sprites and scripts. Each role runs on a separate computer (or browser tab during debugging). For example, a chess game would have two roles, black and white. A blackjack game might have one role for the bank and a number of additional roles for the regular

978-1-5386-2480-7/17/$31.00 ©2017 IEEE

players. There are two types of addressing in NetsBlox, local and global. Local addressing applies to roles. You can send a message to any other role by using its name as the address. You can also broadcast a message to everybody including yourself or to everybody but you. Local addressing makes it possible to have two or more instances of the same game played simultaneously by different users. The NetsBlox server routes the messages to the roles of the correct instance of the project. The only drawback of roles is that an application can only have a static number of roles determined at design time.

Global addressing makes it possible to communicate between separate NetsBlox projects. Since a user name, project name, role name triple uniquely identifies a running instance of a NetsBlox project/role, the following pattern can be used as a global address: "rolename@projectname@username." In NetsBlox terminology, this is called a public role id and there is an RPC that returns this string in the given context of a running program.

To introduce networking into the high school CS curriculum, we developed a set of units that rely on message passing using global addressing only. The rest of the paper briefly describes some of these curricular units and then concludes with a discussion.

II. Text Messaging

At the beginning of this unit, the instructor explains what message passing and addressing mean in general and how to send a message to any other project in NetsBlox in particular. In explaining why the address needs to be globally unique and why hierarchy helps in this regard, phone numbers and mailing addresses can be used as analogies. This is also a perfect opportunity to introduce IP addresses. During the hands on portion of the unit, the students are asked to write a simple client program that sends a message to a master program that runs on the instructors computer and displayed on a projector. NetsBlox comes with one predefined message type called *message* that has a single variable as a payload called *msg* as shown below. In this exercise, students simply use this built-in message type.

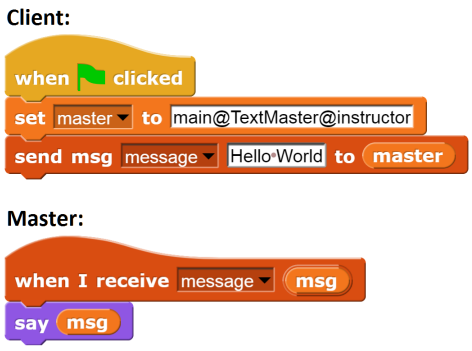

Fig. 1. A client program sends a text message to another program called *TextMaster* with a role called *main* and being run by user *instructor*.

In our experience, students have no difficulty completing this simple task. In fact, every time we delivered this unit, one or more students put the message sending block into an infinite loop causing other students messages to be delayed and/or show up for only a fraction of a second on the screen. This proved to be a great opportunity to talk about spam and denial of service attacks.

III. Chat Room

The next unit builds on the Text Messaging exercise. The goal is to build a chat room in which every student participates by sending messages to the chat room program, which, in turn, sends the messages back to everybody. The students' programs, the clients, need the display the messages received from the chat room. The instructor explains that for a more complicated distributed application like this, there needs to be a mutually agreed upon protocol that each participant needs to follow. In this particular case, a client needs to register with the chat room providing it with its own global address, that is, public role id. Then it needs to be able to send and receive text messages to and from the chat room application. This unit also introduces message types as part of a protocol. The first message type can be called *connect* and it contains the public role id of the sender. The second message type called *text* needs to contain two data fields: the sender and the text message itself. The sender field can simply be agreed upon to be the first name of the sender and it is to be displayed upon receiving a message along with the actual text message itself.

Fig. 2. Registering with the chat room server.

The instructor can develop the chat room program while the students are watching. First the new message types must be defined. Upon receiving a connect message, the chat room needs to save the received public role id in a list of clients unless it is already in the list. Additionally, upon receiving a text message, it can simply send it out to the list of clients with the two fields unmodified. Then it is the students' turn to implement their clients.

They need to define the exact same message types paying careful attention to the spelling the name of the message type and field names. Upon starting the program, the client needs to send a connect message to the chat room. It also needs a message handler for receiving text messages that can simply display the received message along with the sender, e.g., "John: Hello Jane." It also needs to have code to send messages that the user types in. Note that they do not need to display their own messages separately, as these will be sent back to them by the server.

This unit emphasizes the need for a protocol that each participant in a distributed program needs to follow carefully. A protocol includes the different kind of messages and who is sending and receiving what and when. In this example, the

client code can of course send a fake name as the sender. This offers the opportunity to talk to students about spoofing.

IV. Mesh Networking

Mesh networking can be introduced by explaining how the previous two units, text messaging and the chat room, were using a centralized server which can be a single point of failure and/or not always available. Then the instructor can describe how mesh networks work, that is, each node is connected to only a few neighbors and messages to nodes other than these neighbors need to propagate through the network where intermediate nodes forward messages along.

As a first hands-on exercise, the students in each row are put into a group (preferable no more than 4 or 5 members) and they are asked to form a loop where everybody is connected only to their right-hand neighbor, except for the last student in the row who is connected to the first one. Each student needs to write a program that can 1) send messages to anybody else in the group, 2) receive and display messages intended for them and 3) forward messages not intended for them. A simple example implementation is shown below.

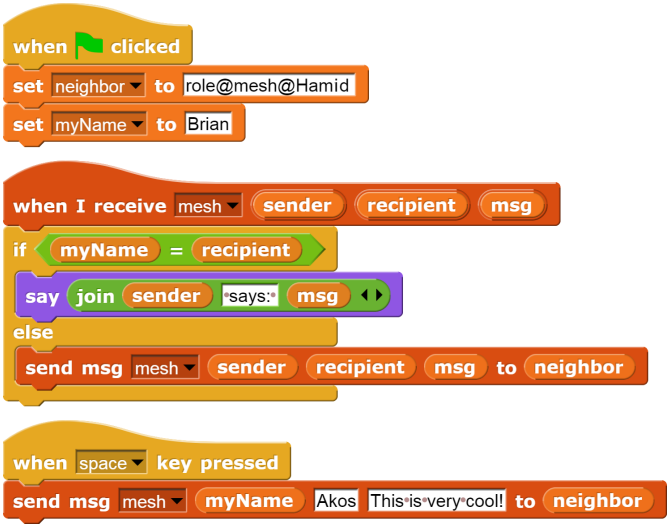

Fig. 3. Simple text messaging with mesh networking. If the intended recipient is not the current user, the message is simply forwarded unchanged.

This exercise quickly turns into a group project. Since any one of the students' programs could prevent the messaging from working, students typically start working together trying to find where messages get stuck. They come up with innovative ways to debug the distributed application. When they find that one particular program is not working properly, they change the addresses so they skip that person while some of them help that students to correct the code.

Once they are ready, there are a number of ways to proceed. If every group's project is working, the entire class can be turned into a giant loop and messages would still be delivered as expected. The instructor may ask what happens when somebody mistypes the addressee's name. Students typically figure out that the given message would just go around the network infinitely. They can be asked to figure out a way to prevent this. In this loop example, the solution can be that the original sender checks whether they ever receive a message they sent. In a more general network, messages may need to have a time to live or hop limit associated with them, so that old messages can be discarded. Finally, every node in this loop example is a single point of failure as the students experienced first hand while debugging this distributed application. A discussion can be had to devise different network structures with redundancy, so that the network becomes robust. Of course, the application code then needs to be able to handle receiving the same message multiple times and handle it correctly.

V. Multi-Player Games

Nothing gets most students more excited than games, especially multi-player games. NetsBlox makes it possible to create turn-based and strategy games, but it can also support games that involve a moderate amount of animation. This unit introduces students how to create such an application where a server program needs to maintain the shared state of all the distributed actors and send them periodic updates about it.

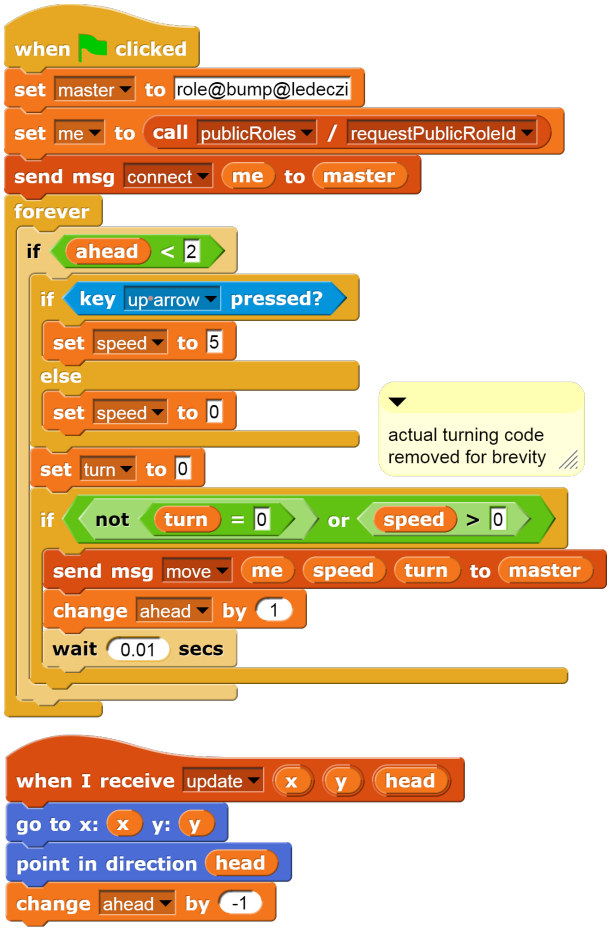

Fig. 4. Bumper car client sending move requests to the server, but only updating its own position upon receiving an update message from the server.

A simple example we use is that of bumper cars. Each student needs to write their own client that drives a bumper car around a shared stage. They get points by bumping other cars

driven by other students from their own computers. The server program is provided which follows a predefined protocol that the clients need to abide by as well. A client needs to register with its own address. Then it gets a message from the server with its initial position. Subsequently, it can send requests to the server to move and turn. In turn, it receives an update of its position. In this simple example, the clients do not get notification about the position of the other players to keep the client code short. The server program displays the game on its own stage that should be shown on the projector in the classroom.

The instructor needs explain how they cannot simply update their own position after a move since they may bump into another car and get bounced back. They have no information about the position of the other players, only the server does, so they need to wait for an update. Also, they are explained how they should not send move messages at a high rate, because if the server is unable to keep up, there will be an increasing amount of delay and they may make driving decisions based on potentially old information.

In the code above, the variable *ahead* handles flow control, another important concept in networked programs. It enforces that only two move requests can be made before the server had a chance to respond with an update message. The role of the wait block near the bottom is to let the scheduler run other scripts, i.e., it is basically a yield command.

The server keeps track of all the connected clients, processes all move commands, carries out collision detection, keeps the score, updates the stage, and notifies all clients of their most recent position. A snapshot is shown below.

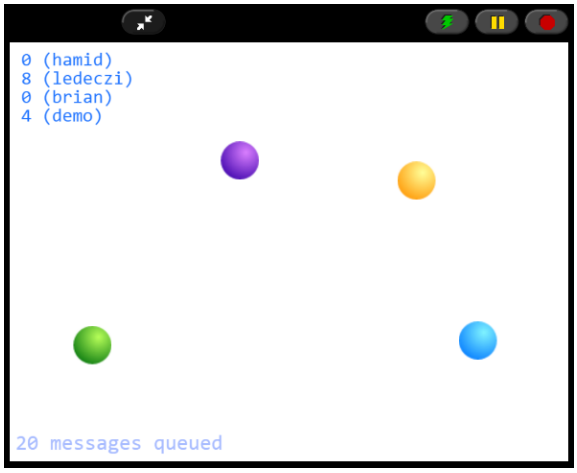

Fig. 5. Bumper car server stage. Note that the cars are circles to simplify collision detection. The current message buffer length is displayed to illustrate the need to limit the rate of messages sent to the server.

VI. Conclusions

The four curricular units presented can form the basis of a gentle introduction to computer networking to high school students. Notice how the actual programs the students write are short and simple. That makes it possible to focus on the fundamental networking concepts. Furthermore, these exercises are fun and collaborative. The students have to work with either each other (mesh networking) or with the instructor's master/server programs (text messaging, chat room) or both (bumper car). The units proved to be highly engaging which should not be surprising given that they involve two of today's teenagers' favorite activities, texting and video games.

Two more units build on top of the ones presented here. A shared whiteboard application introduces how to maintain distributed state information without a server. This is a somewhat more complicated program at a length of about 50 blocks. A slightly more advanced version may also serve to introduce a classical problem, distributed consensus, by requiring all participants to agree when to erase the board. The second, by far the most advanced unit introduces volunteer computing via prime factorization. The server code is quite complicated and beyond the scope of a high school class, but its major functions can be explained and demonstrated. The client code tests whether a number is a factor of another, both received via a message from the server. This is certainly within the grasp of an average AP CSP student. This unit introduces parallel computing, the need for robustness by keeping track of outstanding requests and watchdog timers catching and handling missing responses.

Obviously, there are very few high school teachers who are experts in computer networking. However, we believe that NetsBlox, as illustrated by these units, is able to distill the essence of computer networking down to a few core concepts that are easy enough to grasp and experiment with using the message passing functionality built into the environment. We believe that existing professional development efforts can be extended to cover computer networking. As NetsBlox is built on top of Snap!, the BJC course is the most obvious target to try these ideas out. We are working with the developers of BJC toward this goal.

Acknowledgment

Funding from the Trans-institutional Programs (TIPs) of Vanderbilt University made possible to start the development of the tool. This material is also based in part upon work supported by the National Science Foundation under Grant Numbers CNS-1644848 and DRL-1640199. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] O. Astrachan and R. B. Osborne, "Advanced placement computer science principles (apcsp): A report from teachers," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, 2016, pp. 681–682.

[2] B. Broll, A. Lédeczi, P. Volgyesi, J. Sallai, M. Maroti, A. Carrillo, S. L. Weeden-Wright, C. Vanags, J. D. Swartz, and M. Lu, "A visual programming environment for learning distributed programming," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, 2017, pp. 81–86.

[3] "NetsBlox," https://netsblox.org, cited 2017 July 25.

[4] "Snap!: a visual, drag-and-drop programming language," http://snap.berkeley.edu/snapsource/snap.html, cited 2016 March 16.

[5] "The Beauty and Joy of Computing," http://bjc.berkeley.edu/, cited 2016 May 14.