A Visual Programming Environment for Introducing Distributed Computing to Secondary Education

Brian Broll, Ákos Lédeczi*, Hamid Zare, Dung Nguyen Do, János Sallai, Péter Völgyesi, Miklós Maróti, Lesa Brown, Chris Vanags

Vanderbilt University, Nashville, TN, USA.

Abstract

The paper introduces a visual programming language and corresponding web and cloud-based development environment called NetsBlox. NetsBlox is an extension of Snap! and builds upon its visual formalism as well as its open source code base. NetsBlox adds distributed programming capabilities by introducing two well-known abstractions to block-based programming: message passing and Remote Procedure Calls (RPC). Messages containing data can be exchanged by two or more NetsBlox programs running on different computers connected to the Internet. RPCs are called on a client program and are executed on the NetsBlox server. These two abstractions make it possible to create distributed programs such as multi-player games or client-server applications. We believe that NetsBlox not only teaches basic distributed programming concepts but also provides increased motivation for high-school students to become creators and not just consumers of technology.

Keywords: visual programming, distributed programming, computer science education

*Corresponding author

Email address: akos.ledeczi@vanderbilt.edu (Ákos Lédeczi)

1. Introduction

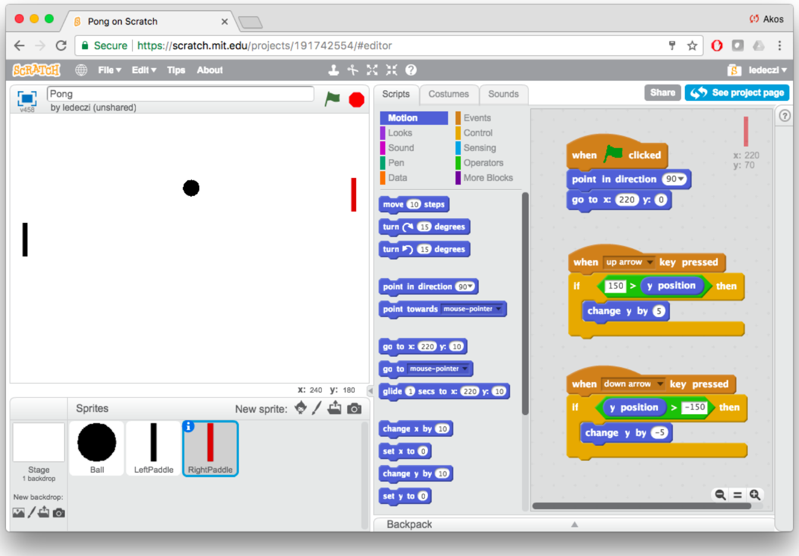
Computational thinking (CT) has been described as a general analytic approach to problem-solving, designing systems, and understanding human behaviors [1, 2]. The integration of CT within the K12 curriculum has also been argued for by the ACM committee on K12 education [3] and has become a focus of several researchers on educational computing [4, 5].

There are many efforts around the world aimed at introducing children to computer science and programming, such as code.org, Khan Academy, or the Raspberry Pi Foundation. Visual programming languages have come to play a prominent role in this movement and have been used to teach children programming [6, 7] as well as computational modeling [4, 8]. However, most of these efforts focus exclusively on the ***computer*** and neglect an equally important concept, the ***network***. This is, of course, completely understandable: you need to learn how to program a computer before you can create networked/distributed applications. Nevertheless, the majority of computer applications we and our children interact with daily rely on the network to provide their functionality. The world-wide web, messaging apps, online games, social networks (e.g., Twitter and Facebook), streaming music and video services (e.g., Pandora, Netflix, and YouTube), home assistants (e.g., Amazon Echo), and massive open online courses are just a few of the most popular examples. Even embedded systems are becoming networked at a rapid pace, with cars and home automation being the prime examples. These advances have created a demonstrated need to teach distributed computing as a part of basic computer literacy. Fortunately, the ubiquity and utility of the applications will make the adoption by a novice much more attractive.

We believe that it is not enough to introduce computer programming into the K12 curriculum — it is also necessary to teach distributed computing concepts to young learners. At the college level, the ACM IEEE Computer Science curriculum (2013) [9] advocates introducing the following topics to CS majors: asynchronous and synchronous communication, reliable and unreliable protocols, and the need for concurrency in operating systems. 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 to high school students. To this end, we have developed a new learning environment called NetsBlox, an extension of the visual programming paradigm of Snap! [10, 11], which, itself, is an extension of Scratch [6]. NetsBlox introduces a few carefully selected abstractions that enable novice programmers to create distributed computing applications.

The literature on educational computing is rife with observations of the impediments that keep young students from learning the basic constructs of programming. Programming languages tend to have only a few components which are combined in many different ways, and students find it challenging to understand the semantic results of different combinations, especially to achieve particular goals [12, 13]. When students try to assemble language elements, they often get confused with syntax problems as they struggle to understand semantic ones [13]. By alleviating syntax issues, students can focus on the semantic problems [14]. In fact, research comparing learning in a more and a less syntactically strict language, Java and Python, respectively, attributed the greater success of students in Python to reduced syntactic complexity [15]. Another documented programming challenge in the literature is students' lack of understanding of computational processes. Many students do not understand how traditional programming languages are interpreted by the computer, e.g., how control flows and how variables get updated [16]. Some research claims that visual programming languages can make these processes more accessible by making control flow constructs more natural.

Alleviating syntactic complexity is an important pedagogical affordance of a visual programming paradigm. In such an environment, students construct programs using graphical objects on a drag-and-drop interface [7]. This significantly reduces students' challenges in learning the language syntax (compared to text-based programming), and thus makes programming more accessible to novices. Examples of some visual programming environments are Scratch [6], Snap! [10, 11], AgentSheets [17], StarLogo TNG [18], ViMAP [19] and Alice [20].



```
when [flag] clicked
point in direction 90
go to x: 220 y: 0

when up arrow key pressed
if 150 > y position then
    change y by 5

when down arrow key pressed
if y position > -150 then
    change y by -5
```

Figure 1: Scratch example

We start by introducing Scratch [6] because it is one of the most mature and widely used approaches and Snap! is based on its visual formalism. Figure 1 shows the Scratch web-based editor with a simple example Pong program loaded. A Scratch program consists of one or more sprites that can have multiple visual representations ("Costumes") and one or more scripts. The example Pong program has three sprites (a ball and two paddles shown in the bottom left window). The currently selected paddle sprite has three scripts (shown in the rightmost window). As can be easily seen from the code, the top script handles what happens at startup (when the green flag is clicked, the orientation and position of the paddle is initialized), and the other two respond to the up or down arrow keys, respectively (the paddle moves up or down, but it is never allowed to reach the border). The program is executed on the "stage" in the top left window. The area in the middle shows the available computing blocks (instructions, operators, etc.) grouped by various color coded tabs. The shape of the block is based on the role. For example, one can only insert hexagons as a condition into an ‘if’ block header; no other shapes are accepted. This prevents syntax errors altogether.

1.1. Understanding Concurrency

Although researchers have been investigating approaches for teaching and learning computer programming in K12 classrooms, very few studies have looked at how K16 students can learn about concurrency. A few researchers have pointed out that Scratch can be used productively to introduce basic ideas of concurrency to novices [21, 22]. Meerbaum-Salant *et al.* [21] investigated and classified two types of difficulties experienced by students in understanding concurrency using Scratch. Type I concurrency occurred when several sprites were executing scripts simultaneously, such as sprites representing two dancers. Type II concurrency occurred when a single sprite executed more than one script simultaneously; for example, a Pac-Man sprite moving through a maze while opening and closing its mouth. They found that Type I concurrency seemed to be much more intuitive for students and easier to grasp.

Maloney, *et al.* [22] reported on an experiment where Scratch was used by students in an after-school club. These students were self-selected and self-paced, receiving no formal instruction. An analysis of the projects revealed that the majority of the student who actually constructed executable scripts used both sequential and concurrent execution. However, as the authors themselves note: “ without realizing it, most Scratch users make use of multiple threads.” The researchers did not investigate if the use of concurrency demonstrated an understanding of this concept. The internalization of concepts was measured by counting the portion of projects using them. These measures were higher (about 50%) for user interaction and loops, lower for conditional statements and for communications and synchronization (about 25%), and much lower for Boolean logic, variables and random numbers (about 10%).

Visual programming environments have also been used to simplify parallel programming as demonstrated by Feng *et al.*. Their research introduced an extension of Snap! [11] with explicit parallel programming concepts [23]. The system also added the ability to generate textual parallel code from visual blocks. Snap! blocks are translated to OpenMP [24] code, enabling execution on different HPC platforms. Developing parallel code in a popular visual programming environment enabled parallel programming concepts to be accessible to a wide range of audiences ranging from sixth graders to domain scientists.

In Snap! multiple scripts may appear to be running concurrently but are simply context switching rapidly. This extension added true parallelism by adding three native blocks: *parallelMap*, *parallelForEach* and *MapReduce*, which were processed in parallel using HTML5 web workers and the Paralleljs [25] library. This enabled the given concurrent scripts to run in parallel rather than simply context switching. Where Feng *et al.*[23] targeted concurrent programming education on individual machines, the aim of NetsBlox is distributed computing and computer networking.

1.2. Paper Organization

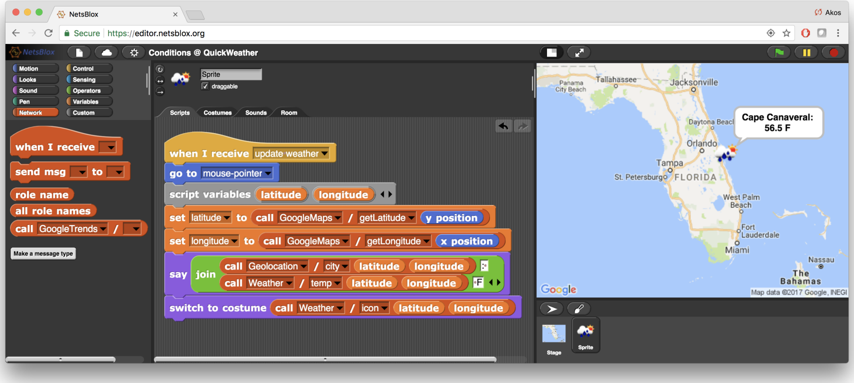
The rest of the paper is organized as follows. The next section presents a brief overview of NetsBlox followed by a detailed description of the distributed programming primitives employed. Their utility is illustrated through a few example applications in the subsequent section. Section 5 describes how an example high level distributed computing pattern can be implemented and taught using NetsBlox. It is followed by an overview of the network model and software architecture of the tool in Sections 6 and 7. The paper concludes with a description of two classroom studies involving middle and high school students.

2. NetsBlox

While Scratch is implemented in Flash, Snap! is an open source extension of Scratch written in JavaScript. It is the tool of choice for the popular Beauty and Joy of Computing high school course that originated at UC Berkeley [26], which is why it was chosen to be the foundation for NetsBlox. Snap! constitutes an excellent starting point because (like Scratch) it also supports concurrency. Sprites run in parallel, and each script runs in its own thread. The keyboard and the mouse generate events that scripts can handle, and scripts can generate and handle custom events. NetsBlox builds on these concepts to supply primitives for synchronization and communication ***across computers***, providing a gentle introduction to distributed computing. Introducing two powerful abstractions, *Messages* and *Remote Procedure Calls*, NetsBlox makes distributed computing accessible to novice programmers. The message abstraction is very similar to the Event concept of Snap!, but it adds the capability to carry data and can be sent to NetBlox programs running on different computers.

We believe that the main appeal of NetsBlox is the accessibility and relevance. It enables young learners to create new classes of programs which were previously out of reach. For example, multi-player video games are very popular with children, and message passing supports the creation of non-trivial distributed gaming programs. While real-time games with 3D scene rendering are obviously beyond the realm of possibilities, strategy games, turn-based board games and games that include slower paced animation are quite feasible. In addition, NetsBlox applications can be hosted on phones and tablets. We imagine a world where the average high school student can create a multi-player game, run it on his/her phone and play against a friend over the Internet after just a few weeks of instruction. This vision is fully achievable with the current capabilities of NetsBlox.

Furthermore, there are a large number of publicly available, interesting data sets on the web. Examples include the weather [27], air pollution [28], seismic data [29], real-time traffic information [30] and many others. Typically the data is visualized on a given website, but in many cases, a public API is available to access the data programmatically. The NetsBlox server provides access to a select set of interesting data sources. These are available for NetsBlox programs via a simple abstraction called Remote Procedure Call (RPC). Essentially, an



```
when I receive update weather
go to mouse-pointer
script variables latitude longitude
set latitude to call GoogleMaps / getLatitude y position
set longitude to call GoogleMaps / getLongitude x position
say join call Geolocation / city latitude longitude :
         call Weather / temp latitude longitude F
switch to costume call Weather / icon latitude longitude
```

Figure 2: Weather application in NetsBlox utilizing 1) the GoogleMaps service to get a map image for the stage (not shown) and provide coordinate transformation from screen (stage) coordinates to latitude and longitude, 2) the Weather service to get temperature data and current conditions represented by an icon, and 3) the Geolocation service to get the city name.

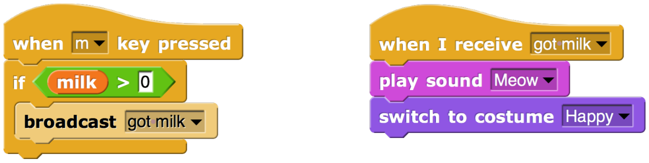
RPC provides a mapping between the NetsBlox call block and the corresponding API of the public data service. A set of related RPCs are grouped together to form a *Service*. For example, the NetsBlox Weather Service has an RPC called "*temp*" that takes arguments for the location and returns the corresponding temperature. A second RPC, "icon," returns a weather icon representing the current conditions at the given location. On the NetsBlox server, they silently invoke the proper calls on the OpenWeatherMap API to get the data and then send it back in a format expected by the NetsBlox programming environment.

With NetsBlox, students are able to create imaginative applications that utilize the wealth of information available on the web using a single, simple abstraction. One potential difficulty is the use of geospatial data. To help students make use of it, NetsBlox integrates Google Maps as an interactive background, again, using RPCs (see Figure 2). Displaying real-time data on an interactive map using a Scratch-like easy-to-use visual programming language is one of the most attractive features of NetsBlox.

3. Distributed Programming Primitives

The key design decision for NetsBlox was the selection of distributed programming primitives manifesting themselves as visual abstractions. In order for the students to engage with the technology and be able to learn the basics of distributed computation, these needed to be intuitive, easy-to-grasp and show the essence of important concepts while hiding unnecessary complexity. The two main distributed programming primitives NetsBlox supports are *Messages* and *Remote procedure Calls (RPC)*.

Peer to peer communication is supported by ***Messages***. Messages are very similar to Events already present in Snap!. Basically, a separate event handler script can be defined in any sprite of the application that will be invoked when the event is generated (see Figure 3).

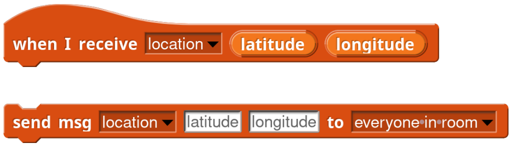


```
when m key pressed
if milk > 0
  broadcast got milk
```

```
when I receive got milk
play sound Meow
switch to costume Happy
```

Figure 3: Snap! event example

In NetsBlox, a Message is an Event that contains data payload. Users are able to type in values or drag and drop one or more variables on the "`send msg`" block (called `broadcast` for events in Snap!). On the receiver side, when they pick the given message from the list of available ones, these data items will appear in the "`when I receive`" block header as variables with the appropriate names, as shown in Figure 4.



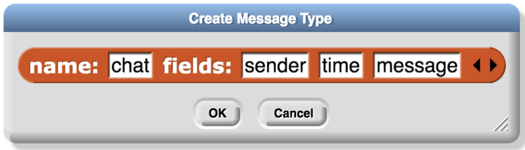
```
when I receive location latitude longitude
```

```
send msg location latitude longitude to everyone in room
```

Figure 4: Sending and receiving messages with data in NetsBlox

In order to support complex data payloads, NetsBlox messages follow a schema specified by their given *message type*. A message type is composed of a name and a list of fields defined for the given messages. Message blocks, as shown in Figure 4, provide a dropdown of all the currently defined message types; upon selecting a given message type, the block is updated to reflect the name and the fields of the given type. The message type in Figure 4 has the name "location" which contains two fields: "latitude" and "longitude."

As the creation of different distributed applications will typically require unique messaging protocols, including unique message types, it is important that users are able to define their own custom messages. NetsBlox supports the creation and management of message types, similar to how variables and custom blocks are defined in Snap!. The message type editor is shown in Figure 5.

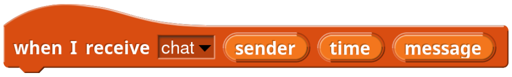


Create Message Type

name: chat fields: sender time message

Figure 5: Custom Message Creation

After a new message type has been created, the send and receive message blocks are dynamically reconfigured to reflect the structure of the message. An example of this is shown in Figure 6; the receipt of the message type defined in Figure 5 is handled by the given "`when I receive`" block.



```
when I receive chat sender time message
```

Figure 6: Chat Message Handler Block

Another important distributed programming primitive is the concept of a ***Room***. A Room defines the virtual network for the project and consists of ***Roles*** which are named NetsBlox clients. That is, a Room defines the NetsBlox clients which share a network and provides names for each client to facilitate messaging between them. Each of these Roles have their own stage, sprites and scripts. In essence, a NetsBlox project consists of one or more sub-projects called Roles, each of which runs on a different computer (or browser tab). This facilitates the development of networked applications for novices and allows them to avoid the challenge of node discovery over the network. Note, however, that communication is not restricted only to the Room, and more dynamic networked applications can be developed that leverage inter-room communication. Examples of this can be found in Section 5.

A NetsBlox project automatically has a single associated Room. The project owner manages the Room and its Roles. This includes creating, removing, renaming and duplicating Roles. Along with building the structure of the project and its Room, the owner also has the ability to invite other users to specific Roles in the project to run a distributed application together, e.g., to play a multi-player game. An example of a NetsBlox Room can be seen in Figure 7.

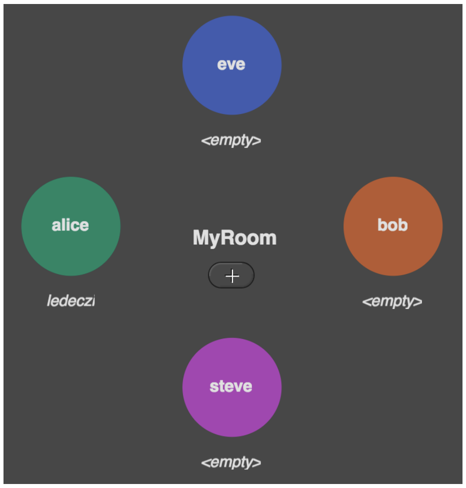


Figure 7: NetsBlox Room

Figure 7 shows the Room for a project called *MyRoom* which contains 4 Roles: “alice,” “bob,” “eve,” and “steve.” The current user is occupying the “alice” Role; the other three Roles currently are unoccupied. The + button in the middle allows the owner to add new Roles to the Room; this will result in another client being added to the project. If the user clicks on any of the given colored Roles he/she will be able to edit the given Role (i.e., rename, clone or remove it) or invite a peer to the given Role to join the given project. Another example for a Room could be that of a Tic-Tac-Toe game with exactly two

Roles: “X” and “O.”

When sending NetsBlox messages, the “target” field of the message is populated with the other Roles present in the given Room as well as two broadcasting options: “others in room” and “everyone in room.” Both broadcast options will send the message to all other Roles in the Room, but “everyone in room” will also trigger the given message handlers in the origin Role. Figure 8 shows one example for sending a simple message in the context of the Room in Figure 7. The items in the addressee pull-down menu are dynamically populated given the Roles currently defined in the Room. This simplifies the process of sending messages and reduces the likelihood of simple routing errors.

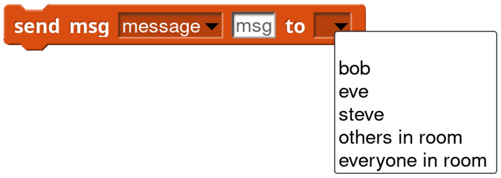


Figure 8: Sending messages to other NetsBlox clients

The semantics of Messages in NetsBlox are based on the semantics of Events in Snap!. Multiple handlers can be defined for the same kind of message and all of them will be invoked when a message of the given type arrives, each in its own thread, but the order of execution is not specified. However, two messages sent from the same script are guaranteed to be delivered in the same order as they were sent. Furthermore, when two Roles send messages, the order of delivery is guaranteed to be consistent. That is, if Roles A and B send one message each to every other Role at the same time, all Roles will get these in the same order. The order is decided by the message arrival time on the server.

Message passing is asynchronous, hence, the sender is not blocked and no acknowledgements are returned. Note that if a message handler is still executing when a new message of the same type arrives, the message handler will queue the message to be executed once the current execution completes.

It is interesting to note that messages are addressed to one or more Roles of the Room, that is, nodes participating in the virtual network defined by the application. Within a Role, i.e., the NetsBlox program running on one host (computer or browser tab), messages are broadcast just like events. This means that any sprite, and the stage as well, can receive and handle any and all message types.

The ***Remote Procedure Call (RPC)*** is the highest level of distributed abstraction NetsBlox employs. As the name implies, RPC allows for invoking code that will be executed at a remote location, and then (optionally) receiving the results of the computation. This includes both predefined functions provided by NetsBlox as well as user defined blocks[1]. The semantics of RPC is as expected: multiple input arguments, single output argument, pass-by-value and blocking call. RPCs are very similar to the concept of Custom Blocks (functions) in Snap!, therefore, it should be familiar to students who are already acquainted with Snap!. In fact, one of our major goals with supporting the RPC primitive was to provide an easy transition for students from using local function execution to invoking remote functionality. This is necessary for getting access to public online databases, and it is also very helpful in facilitating more complex multi-player games.

NetsBlox RPCs also have the ability to maintain state. This state can be shared either globally or just among the users in the given NetsBlox Room. Examples of RPCs using a global context can be found in the RPCs exposing 3rd party endpoints, such as Google Maps, as they often cache their results to minimize redundant requests to the given external API.

One specific example of a stateful RPC can be found in a Battleship game application. The Battleship helper RPCs provide assistance for turn coordination and event detection in applications implementing the game. This includes maintaining whether the players are still placing ships or have already proceeded to shooting at one another as well as enforcing the turn-based nature of the game. In order to provide this support, the Battleship helpers must save state between procedure calls (e.g., placing a ship or shooting at the opponent's

[1] Custom user defined RPC support is currently under development.

ships) and share this state only among the clients in the current virtual network, that is, the Room. This means that multiple concurrent instances of the same game can be executed on the NetsBlox server.

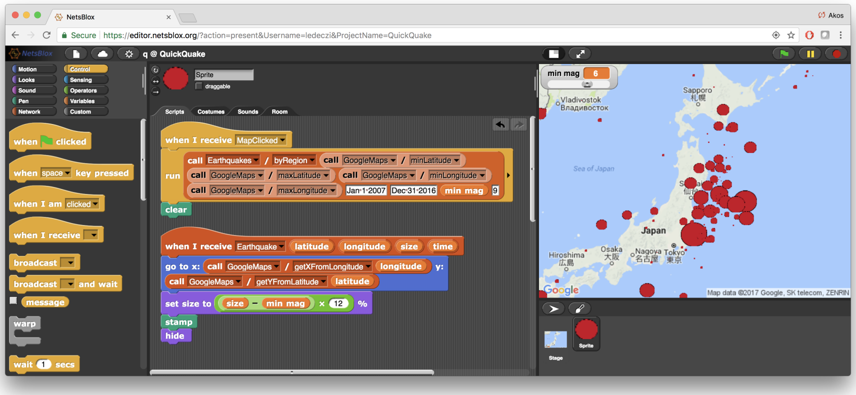
Currently, NetsBlox supports a fixed set of RPCs that are implemented and executed in the NetsBlox network infrastructure (on the NetsBlox server). From the user's perspective, RPCs are executing "in the cloud." We are planning support for exporting user-defined custom blocks as RPCs and hosting them on the server in an upcoming release.

In the future, we will also provide an additional facility: a script-local variable called "`error`" represented by a red block. If the RPC was successful, the value of `error` will be the string "OK." Other values will indicate various problems with the call, such as "timeout" or "bad argument." In a curriculum using NetsBlox, the recommended way of using RPCs needs to be explained: after a call, an 'if' statement should check whether the value of the error variable is OK or not. If not, the user program is expected to handle the error according to the value of the `error` variable.

4. Illustrative Examples

To illustrate some of the concepts introduced in the previous section, let us consider a simple example that displays historical earthquake data where the program applies a combination of RPCs and messages (see Figure 9).

The application utilizes the GoogleMaps service just like the weather example in Figure 2. Once the user navigates to the desired region using the arrow keys and +/- for zooming (stage scripts implementing this map handling is not shown) and clicks on the background, the stage broadcasts a *MapClicked* event. The corresponding event handler in the sprite invokes the "byRegion" RPC of the "Earthquake" service as shown in Figure 9. The coordinate mapping RPCs of the GoogleMaps service are used to specify the desired region determined by the limits of the currently shown, i.e., last requested, map. The date arguments specify ten years, and the magnitude parameters request the interval between 6



```
when I receive MapClicked
run (call Earthquakes / byRegion) (call GoogleMaps / minLatitude)
    (call GoogleMaps / maxLatitude) (call GoogleMaps / minLongitude)
    (call GoogleMaps / maxLongitude) Jan 1 2007 Dec 31 2016 (min mag) 9
clear

when I receive Earthquake (latitude) (longitude) (size) (time)
go to x: (call GoogleMaps / getXFromLongitude (longitude)) y:
    (call GoogleMaps / getYFromLatitude (latitude))
set size to ((size) − (min mag)) × 12 %
stamp
hide
```

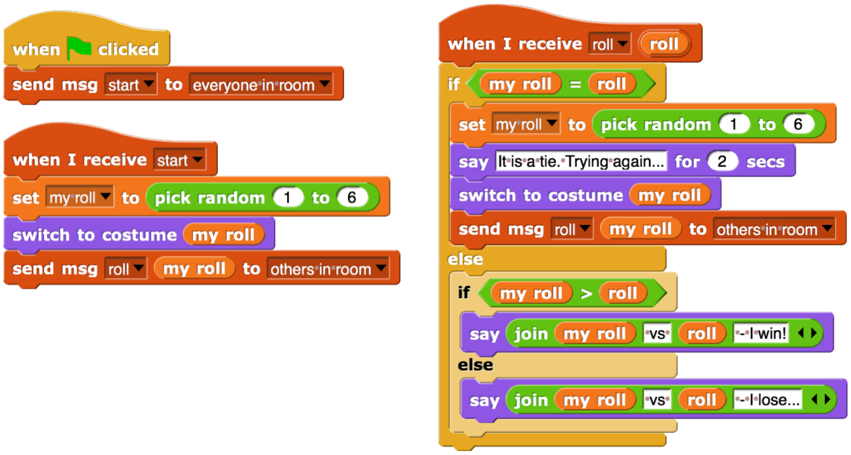
Figure 9: The Earthquake application

and 9 (only strong to catastrophic events are to be included). Instead of returning the potentially large dataset as the output argument of the RPC, the server collects the historical earthquake data from the web for the given geographic area and starts sending messages back to the client (one message sent per earthquake). The script that handles the Earthquake message simply displays the location with a red dot (again using coordinate mapping), where the size of the dot is proportional to the magnitude of the given earthquake.

As a second example, Figure 10 shows the scripts for a 2-person, simplistic dice game. In this scenario, two players both roll their dice, and whoever has a higher number wins. In the case of a tie, they roll again.

The program is symmetrical in that both Roles use the exact same scripts. The game is started by one player clicking the green flag. The script corresponding to this event simply broadcasts a “start” message to everyone in the Room (including itself). Upon receiving a “start” event, each client rolls the dice by picking a random number between one and six and sends a “roll” message to the other player containing the generated number.

The script with the “when I receive roll” header runs when the “roll” message arrives, and it supplies the data in the payload as the variable called “roll.” The code then simply compares the two values, “roll” and “my roll.” The interesting case is when the two are equal. In this case, each player rolls again and sends



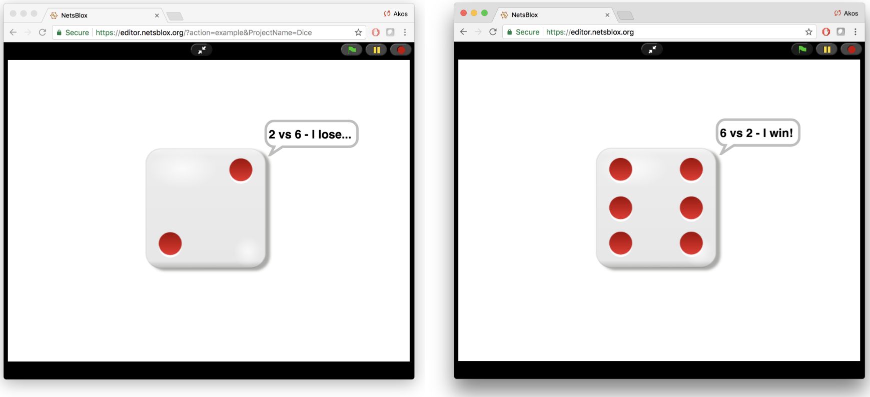
```
when ⚑ clicked
send msg start to everyone in room

when I receive start
set my roll to pick random 1 to 6
switch to costume my roll
send msg roll my roll to others in room
```

```
when I receive roll roll
if my roll = roll
    set my roll to pick random 1 to 6
    say It is a tie. Trying again... for 2 secs
    switch to costume my roll
    send msg roll my roll to others in room
else
    if my roll > roll
        say join my roll vs roll - I win!
    else
        say join my roll vs roll - I lose...
```

Figure 10: The scripts of the Dice game

the new dice value to the other side using another "roll" message. Otherwise, the players are notified by a text displayed on the stage as shown in Figure 11.



2 vs 6 - I lose...

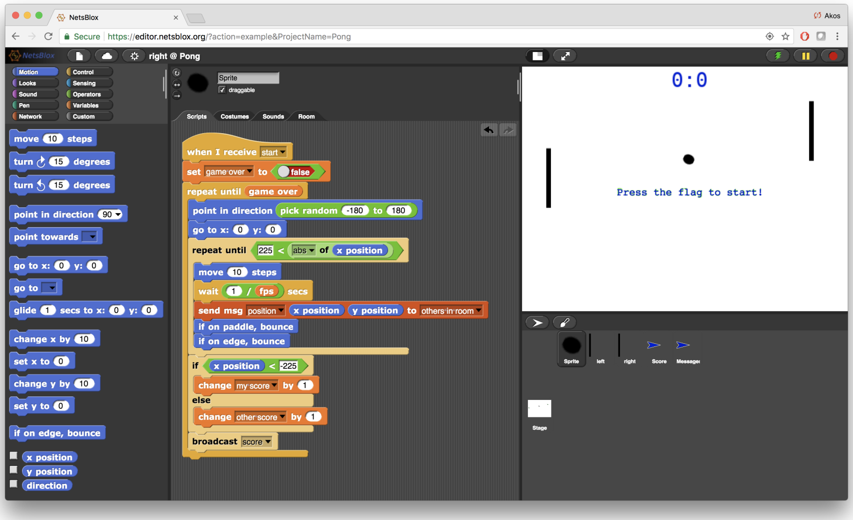
6 vs 2 - I win!

NetsBlox
Secure
https://editor.netsblox.org/?action=example&ProjectName=Dice
Akos
NetsBlox
Secure
https://editor.netsblox.org
Akos

Figure 11: Dice game running in two separate browser tabs

It is interesting to note the message addressing in this example. The roll send blocks use "others in room" as the addressee, meaning every other Role (player) in the current Room (game). In a two-player game such as this, it simply means the single other player. Alternatively, one could select the name of the other Role, but that would require slightly different scripts for the two players because of the different Role names they would need to use as addresses.

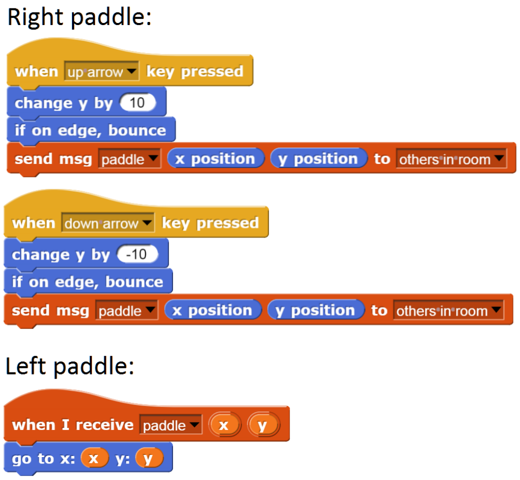
For a more complicated example, consider the classic Pong game. This is interesting as it involves animation and a need to keep the game shown on two stages on different computers in sync. Keeping the distributed state consistent across the two computers is not trivial for novices. Figure 12 shows the program for the right-hand side player. The three main sprites are the ball and the left and right paddles. The other two sprites are for keeping and displaying the score. The code for the ball is shown. It takes care of the motion of the ball and sends update messages to the other player. The code for the other player's ball (not shown) does not do any of this; it simply receives the update messages and moves to the received coordinates. So, the state of the ball is maintained by the right-hand side player only!



```
when I receive start
set game over to false
repeat until game over
  point in direction pick random -180 to 180
  go to x: 0 y: 0
  repeat until 225 < abs of x position
    move 10 steps
    wait 1 / fps secs
    send msg position x position y position to others in room
    if on paddle, bounce
    if on edge, bounce
  if x position < -225
    change my score by 1
  else
    change other score by 1
  broadcast score
```

Figure 12: Pong game with the program for the ball shown

Figure 13 shows the code for the two paddles of the right-hand side player. The right paddle handles the keyboard arrow keys and moves itself accordingly. It sends an update to the other player after each such move. The left paddle simply receives the messages from the other player and updates the position of the sprite. The code for the left-hand side player simply swaps the code of the left- and right paddles of the right-hand side player. Essentially, each player maintains the state of its own paddle and updates the other player with any changes.



```
Right paddle:
when up arrow key pressed
change y by 10
if on edge, bounce
send msg paddle x position y position to others in room

when down arrow key pressed
change y by -10
if on edge, bounce
send msg paddle x position y position to others in room

Left paddle:
when I receive paddle x y
go to x: x y: y
```

Figure 13: Pong paddle code of the right-hand side player

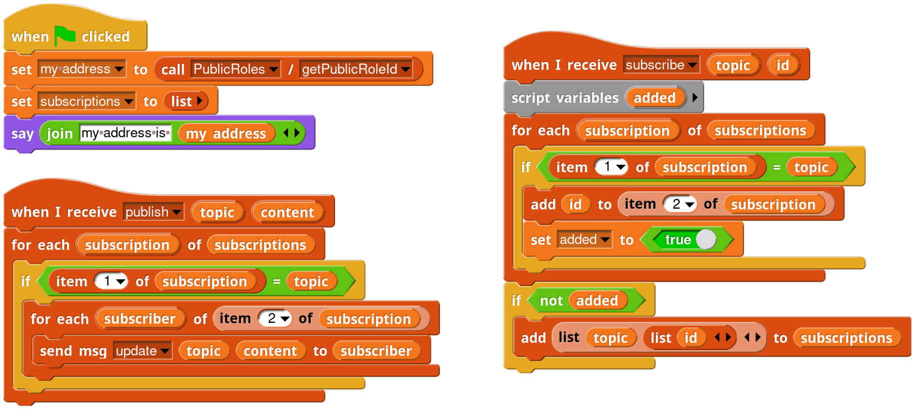
There are many other interesting applications of NetsBlox publicly available from the website. For brief video demonstrations of some of them, visit `http://netsblox.org/tutorials`.

5. Distributed Programming Patterns

The Room facilitates the development of distributed applications by simplifying challenges such as node discovery over the network as well as detecting and handling disconnected nodes in the application. Although this can be very beneficial for novice programmers, it can be somewhat restrictive as the number of clients (Roles) is fixed at design time. More advanced applications may want to manually handle connections in a more dynamic and flexible network. To facilitate such scenarios, NetsBlox supports inter-room communication by providing a simple node addressing scheme which consists of the node's Role name, project name, and the project owner name joined by the @ symbol[2]. Using this address as the target of the "send message" block enables users to send messages to other users outside of the given Room and facilitates the development of more advanced networked applications. In this section, we present and discuss an example of a higher level distributed computing abstraction created within NetsBlox: the Publish-Subscribe pattern.

5.1. Publish-Subscribe

Publish-Subscribe is a programming paradigm for distributed computing in which nodes can subscribe to various types of messages and can publish messages. When messages are published, they are forwarded to any nodes which have subscribed to the given type of message. In practice, there typically is a message broker that coordinates the sending of the published messages to the relevant subscribers. An example of a Publish-Subscribe message broker implemented in NetsBlox can be found in Figure 14.



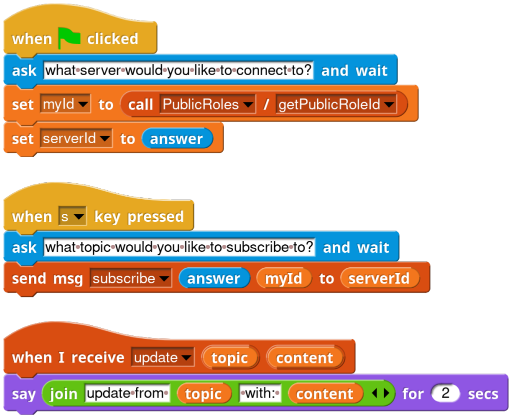
```
when ⚑ clicked
set my address to call PublicRoles / getPublicRoleId
set subscriptions to list
say join my address is  my address

when I receive publish topic content
for each subscription of subscriptions
    if item 1 of subscription = topic
        for each subscriber of item 2 of subscription
            send msg update topic content to subscriber

when I receive subscribe topic id
script variables added
for each subscription of subscriptions
    if item 1 of subscription = topic
        add id to item 2 of subscription
        set added to true
if not added
    add list topic list id to subscriptions
```

Figure 14: Publish-Subscribe Broker in NetsBlox

[2] NetsBlox provides a simple helper RPC for requesting the public address of a Role programmatically.



```
when ⚑ clicked
ask [what server would you like to connect to?] and wait
set [myId ▾] to (call [PublicRoles ▾] / [getPublicRoleId ▾])
set [serverId ▾] to (answer)

when [s ▾] key pressed
ask [what topic would you like to subscribe to?] and wait
send msg [subscribe ▾] (answer) (myId) to (serverId)

when I receive [update ▾] (topic) (content)
say (join [update from ] (topic) [ with: ] (content) ◂▸) for (2) secs
```

Figure 15: Subscribe Client in NetsBlox

The example NetsBlox Publish-Subscribe broker has defined 4 different message types: *publish*, *subscribe*, *unsubscribe* and *update*. The broker then maintains a variable called “subscriptions” which contains a list of topics and the associated subscribers. On *subscribe* and *unsubscribe* events, the broker will add or remove the requester from its internal record of subscriptions. On a *publish* event, the broker will send an *update* message with the topic and the content to all of the Roles subscribed to the given topic.

A simple example of a subscriber is shown in Figure 15. The user is first prompted about which Publish-Subscribe broker to connect to, and a public role id is requested. On pressing the “s” key, the user is prompted about which topic he/she would like to subscribe to, and the broker is sent a *subscribe* message with the topic and the client’s public role id. When data for this topic is published to the broker, the client will receive the *update* message (as shown in Figure 14) which will simply display the message to the user.

6. Network Model

NetsBlox requires a powerful yet easy-to-understand network model. It is important to note that the technical details described here are completely hidden from users. The core design principles of the network model are: (1) a reduction of accidental complexities of distributed programming to shield the programmers from distracting technical details and common pitfalls such as firewalls, routing, address resolution, and automatic reconnection, (2) a uniform addressing and discovery scheme for distributed artifacts, and (3) support for both synchronous and asynchronous communication.

We opted for a ***virtual overlay network abstraction*** to achieve these goals. This facilitates direct, bidirectional, peer-to-peer communication between NetsBlox software artifacts, as if they were part of a local area network, without the need for explicit message routing. This overlay network is built on top of existing network technologies, but without one-to-one mapping between virtualized primitives and actual network packets and connections. For example, the NetsBlox server infrastructure emulates peer-to-peer message exchange between two remote scripts using reliable connections between several nodes (i.e., two clients and the server) to provide robust communication and enable instrumentation and centralized management capabilities.

NetsBlox provides two primary concepts to the programmer to manage the network: ***Rooms*** and ***Roles*** that were introduced in Section 3. The Room defines the virtual network for the associated client. Each NetsBlox project consists of a single Room which, in turn, consists of a number of uniquely named clients (Roles). These Roles are automatically discovered by the NetsBlox environment, and the names are provided automatically when performing operations such as direct messaging within the network.

Containers: To extend the platforms supported beyond just web browsers, NetsBlox introduces the concept of a container. A container is a hosting environment for applications that provide a well-defined set of services, including network connectivity. Currently, NetsBlox only provides containers for webbrowsers. In the future, the container concept will be extended to support Android, iOS and networked embedded devices (e.g., Raspberry Pi), as well as server containers running in the cloud. In contrast to web-based or mobile clients that may come and go, cloud containers could be always on, making them an ideal choice to implement user-defined RPCs in the future.

Coordination: While it is fairly straightforward to write a simple two-player game such as Dice using only the *Message* primitive (see Section 4), games with more complicated rules, such as chess, or with more than two players, such as bridge, can quickly become too complicated for the budding NetsBlox programmer to design and implement without help or structure. In addition, some applications may require a flexible network which accepts a variety of clients which may not all be defined in the Room. An example of this is creating a chat application where the number of users (consequently, the number of Roles in the Room) is unknown during development. In applications such as these, coordination becomes more challenging as the interacting clients should not be restricted solely to the Roles defined in the Room; students should be able to create their own chat client and join the chat freely.

NetsBlox addresses these challenges by providing a public addressing scheme and powerful RPCs. Public addresses enable inter-room communication with NetsBlox clients. This enables users to implement their own coordination protocol, as demonstrated by the Publish-Subscribe broker shown in Section 5. This also enables users to build applications with a dynamic number of clients, including chat applications and mesh networks.

NetsBlox RPCs can also provide scaffolding for coordinating a complex distributed application. As discussed in Section 3, NetsBlox RPCs can maintain the state for each connected Room, allowing them to manage more complex parts of applications (such as managing board state in the games mentioned in the previous sections). RPCs can also send messages to the NetsBlox clients. This ability to both maintain a context for the associated Room and send messages to these connected clients makes them ideal options for simplifying coordination for more complex games. For example, using the Battleship game service, one Role calls an RPC to make a move and in turn, the other Role will receive a “your turn” message from the server following the successful execution of the opponent’s turn.

7. NetsBlox Infrastructure

To support the network abstractions described above, as well as the distributed programming primitives and the overall application life-cycle, we have developed and deployed a cloud-based infrastructure and an easy-to-use web application. The web application runs in the client browsers and communicates with the NetsBlox server via HTTP and WebSocket interfaces.

The core server-side services include (1) hosting and serving the web application artifacts, (2) project and user information persistence, (3) RPC and message delivery services, and (4) authentication and run-time user association (tables). We host the server-side infrastructure on the Amazon Web Services cloud computing platform and provide all software components and deployment know-how on GitHub with MIT open source licensing. Note that access to our server is free, so students, teachers and schools only need web browsers to use NetsBlox. Nevertheless, the server-side program is also open source for people who want to run their own server or wish to extend NetsBlox.

NetsBlox RPCs are implemented as REST endpoints hosted on the originating server. REST provides a simple endpoint that naturally supports synchronous requests from the client, enabling the creation of simpler RPCs that provide additional functionality (such as Google Maps integration) seamlessly. Project management tasks such as authentication, project access, and table membership management are also implemented by REST primitives.

Bi-directional NetsBlox communication services such as message delivery and asynchronous project notifications are implemented with WebSocket connections between the clients and the servers. For supporting message passing among clients within the same Room, we have implemented a virtual network abstraction (introduced in Section 6). In this model, each node initiates a WebSocket connection to the server and registers itself with one of the active Rooms. Messages from the client are sent through this connection to the server, which takes care of the routing and fan-out based on the current registrations. Note that no direct communication channels are created between browser clients, which would be extremely problematic through discrete and OS-level firewalls. Also, the server-based message delivery mechanism makes it possible to record accurate and ordered message traces and to emulate arbitrary network effects (packet loss, latency, integrity) for educational purposes.

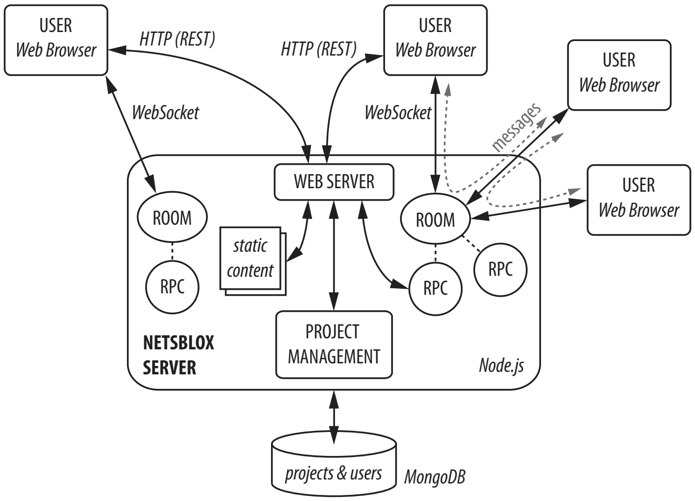


Figure 16: Network Architecture

The high-level architecture of NetsBlox is shown in Figure 16. The server (implemented with server-side JavaScript technologies) provides web services and hosts the web application (e.g., the NetsBlox development environment), which can be accessed by browser-based clients. Projects and user information are stored in a MongoDB database. As the first step, a new client uses HTTP requests for downloading the static artifacts of the web application. Once initialized, this client application creates a permanent WebSocket link towards the server for asynchronous two-way communication. Finally, HTTP/REST connections are created on-demand for accessing projects, joining Rooms and invoking RPC services from the users' applications.

8. Classroom Studies

We conducted two small, week-long studies where we taught students computer programming using NetsBlox. In both studies, we could not assume prior programming experience and thus started with two days of basic programming curriculum. We then introduced the concept of networking by spending one day teaching remote procedure calls and another day teaching messaging. Finally, the last day was a work day where students worked on a project of their choosing.

For each of these studies, students were given both a survey and a pre- and post-test. The survey collected demographic information as well as general questions about disposition towards computer programming and networking; the pre- and post-tests measured computational thinking, networking and concurrency competency. These tests included questions about the Two Generals' Problem, messaging in a mesh network, performance improvements of parallelization and potential outcomes in the presence of a race condition in concurrent programming.

The first study was conducted with 24 high school students who attend the School for Science and Math at Vanderbilt (SSMV) [31]. The program teaches research skills to high school students and is a partnership between Vanderbilt University and Metro Nashville Public Schools. The second study was conducted with 16 students ranging from 6th to 11th grade attending a summer camp in Budapest, Hungary. These students were self-selected by their interest in computer science and not necessarily representative of general high school populations. A week-long Computer Science curriculum was developed which focused on concepts from basic programming constructs, such as control flow, events, and lists, to networking concepts including RPCs and coordination over the network.

Both studies showed a significant enhancement in students' computational thinking and networking understanding with p-values at or below 1%. The SSMV students showed an average overall improvement of about 21 percentage

	Pre-Test				Post-Test			
	CT	Networking	Concurrency	n	CT	Networking	Concurrency	n
SSMV	78	59	12	50	93	78	42	71
Budapest	67	52	11	43	88	71	25	62

Table 1: Comparison of pre- and post-test scores (0 to 100-point scale).

points (pp) on the post-test with about 19 pp improvement on the networking section and a 15 pp improvement on the computational thinking section (see Table I). The second study showed similar results with an 19 pp overall, and 19 pp and 21 pp improvement in networking and computational thinking, respectively. Student scores in concurrency also showed progress. However, they struggled more with these questions on both the pre- and post-tests in both studies. The average scores in concurrency on the post-test were 41% and 25% with an average score of about 12% and 11% on the pre-test for the first and second studies, respectively. The low scores on the concurrency portion of the post-test suggest that the questions may not have been aligned well with the curriculum; as time was a limiting factor in the studies, some of the topics of concurrency were not explicitly discussed. For example, theoretical performance improvements due to task parallelization was not discussed during the course, yet one out of the two concurrency questions focused on this concept.

For their final projects, students developed a number of interesting projects which leveraged distributed computing. These diverse projects ranged from an interactive map interface for learning about country demographics around the world to a multi-player "Tron" clone. One particularly motivated pair of students decided to create their own encrypted chat client (inspired from a mesh networking exercise from class) in which they developed their own simple encryption and decryption techniques within NetsBlox.

Overall, these studies showed promising results with using NetsBlox to introduce distributed computing to high school students. Despite the short duration of the studies, students showed significant improvement in both computational thinking and networking assessments. The creative uses of networking in the students' final projects also support the idea that networking can make programming more collaborative and engaging. The assumption that opening the Internet to students' programs will prove to be motivating was validated informally based on the anonymous feedback we received at the end of each day. A few quotes are included below.

- "I liked this because it was cool to see how you can communicate with people across the globe using a few lines of code."
- "As we continue to learn more about the program, the easier and more natural the coding seems. This has been true throughout our learning with NetsBlox."
- "I really enjoyed getting into multi-computer messages and games because it is my first experience making anything close to an online program."
- "We learned how to do this in chat rooms, as well as across servers. I thought this function was really neat, and was the most interesting part about coding to learn. I really enjoyed being able to expand beyond just one computer and be able to work through multiple places."
- "Even though this week is over, I'm still going to continue working on coding to improve my skills."

A minority of the students, however, did struggle: "Today, we started off yet again with three consecutive hours of programming that, for me, went just as bad as the first time we programmed. When they explain it up on the board I understand completely, but when I must replicate it on my own I do not even know where to start."

These studies introduced a number of open questions. How well could these concepts be taught in a more traditional setting such as a high school classroom? How could we further improve the students' learning given more time for a less condensed course on computer programming and networking? We are planning our next study to answer these questions.

9. Conclusions

The paper presented NetsBlox, a web- and cloud-based visual programming environment that enables users to create distributed applications. NetsBlox extends the well-known and widely used Snap! environment, and hence, it provides a natural progression to students who take the Beauty and Joy of Computing (BJC) class. Consequently, novel curricular units can be easily incorporated into BJC, one of the new AP CS Principles courses [32]. NetsBlox is an ideal vehicle to support some of the big ideas and computational thinking practices that are emphasized in the AP CS Principles curriculum, including the Internet, communication, collaboration, cybersecurity and global impact. NetsBlox also supports collaborative editing from multiple computers, allowing students to work together on shared projects from their own computers.

Furthermore, providing access to vast arrays of data on the Internet directly in the visual programming environment will empower the students to create innovative science projects and simultaneously bring STEM concepts into CS education. The ability to create multi-player games will provide increased motivation for a large number of students and encourage them to be creators and not just consumers of digital entertainment.

10. Acknowledgements

We thank Pratim Sengupta for his contributions during the initial discussions about NetsBlox. We are also grateful to Györgyi Dallos, Hassan Charaf, Gábor Recski and the Budapest University of Technology and Economics for organizing the NetsBlox coding camp. The development of the tool was made possible through Vanderbilt University's Trans-institutional Programs (TIPs). This material is also based in part upon work supported by the National Science Foundation under grants CNS-1644848 and DRL-1640199. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

[1] J. M. Wing, Computational thinking, Communications of the ACM, Viewpoint 49 (3) (2006) 33–35.

[2] Committee for the Workshops on Computational Thinking; National Research Council, Report of a Workshop on The Scope and Nature of Computational Thinking, The National Academies Press, 2010.

[3] S. Hambrusch, C. Hoffmann, J. T. Korb, M. Haugan, A. L. Hosking, A multidisciplinary approach towards computational thinking for science majors, in: Proceedings of the 40th ACM technical symposium on Computer science education, SIGCSE '09, ACM, New York, NY, USA, 2009, pp. 183–187. doi:10.1145/1508865.1508931.

[4] P. Sengupta, J. Kinnebrew, S. Basu, G. Biswas, D. Clark, Integrating computational thinking with k-12 science education using agent-based computation: A theoretical framework, Education and Information Technologies 18 (2) (2013) 351–380. doi:10.1007/s10639-012-9240-x.

[5] M. Guzdial, Software-realized scaffolding to facilitate programming for science learning, Interactive Learning Environments 4 (1) (1994) 001–044. doi:10.1080/1049482940040101.

[6] J. Maloney, M. Resnick, N. Rusk, B. Silverman, E. Eastmond, The Scratch programming language and environment, ACM Transactions on Computing Education (TOCE) 10 (4) (2010) 16.

[7] C. Kelleher, R. Pausch, Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers, ACM Comput. Surv. 37 (2) (2005) 83–137. doi:10.1145/1089733.1089734.

[8] P. Blikstein, U. Wilensky, An atom is known by the company it keeps: A constructionist learning environment for materials science using agent-based modeling, International Journal of Computers for Mathematical Learning 14 (2) (2009) 81–119. doi:10.1007/s10758-009-9148-8.

[9] I. C. S. The Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM), Computer science curricula 2013: Curriculum guidelines for undergraduate degree programs in computer science, `https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf` (2013).

[10] B. Harvey, J. Mönig, Bringing no ceiling to Scratch: can one language serve kids and computer scientists, in Proc. of Constructionism (2010) 1–10.

[11] Snap!: a visual, drag-and-drop programming language, `http://snap.berkeley.edu/snapsource/snap.html`, cited 2018 February 2.

[12] J. C. Spohrer, E. Soloway, Novice mistakes: are the folk wisdoms correct?, Commun. ACM 29 (7) (1986) 624–632. `doi:10.1145/6138.6145`.

[13] D. N. Perkins, R. Simmons, Patterns of misunderstanding: An integrative model for science, math, and programming, Review of Educational Research 58 (3) (1988) pp. 303–326.

[14] L. Hohmann, M. Guzdial, E. Soloway, Soda: A computer-aided design environment for the doing and learning of software design, in: I. Tomek (Ed.), Computer Assisted Learning, Vol. 602 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1992, pp. 307–319. `doi:10.1007/3-540-55578-1_78`.

[15] L. Grandell, M. Peltomäki, R.-J. Back, T. Salakoski, Why complicate things?: introducing programming in high school using python, in: Proceedings of the 8th Australasian Conference on Computing Education - Volume 52, ACE '06, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2006, pp. 71–80.

[16] B. D. Boulay, T. O'Shea, J. Monk, The black box inside the glass box: presenting computing concepts to novices, Int. J. Hum.-Comput. Stud. 51 (2) (1999) 265–277. `doi:10.1006/ijhc.1981.0309`.

[17] A. Repenning, Agentsheets: a tool for building domain-oriented visual programming environments., in: S. Ashlund, K. Mullet, A. Henderson, E. Hollnagel, T. N. White (Eds.), INTERCHI, ACM, 1993, pp. 142–143.

[18] E. Klopfer, S. Yoon, T. Um, Teaching complex dynamic systems to young students with starlogo, Journal of Computers in Mathematics and Science Teaching 24 (2) (2005) 157–178.

[19] P. Sengupta, A. V. Farris, M. Wright, From agents to continuous change via aesthetics: Learning mechanics with visual agent-based computational modeling, Technology, Knowledge and Learning 17 (1-2) (2012) 23–42.

[20] M. J. Conway, Alice: Easy–to–Learn 3D Scripting for Novices, Master's thesis, University of Virginia, Faculty of the School of Engineering and Applied Science (December 1997).

[21] O. Meerbaum-Salant, M. Armoni, M. Ben-Ari, Learning computer science concepts with Scratch, Computer Science Education 23 (3) (2013) 239–264.

[22] J. H. Maloney, K. Peppler, Y. Kafai, M. Resnick, N. Rusk, Programming by choice: urban youth learning programming with Scratch, in: ACM SIGCSE Bulletin, Vol. 40, ACM, 2008, pp. 367–371.

[23] A. Feng, M. Gardner, W. Feng, Parallel programming with pictures is a Snap!, Journal of Parallel and Distributed Computing 105 (2017) 150 – 162.

[24] G. G. Abraham Silberschatz, Peter B. Galvin, Operating System Concepts, 9th Edition, Wiley, 2012.

[25] S. M. A. Savitzky, Parallel.js - easy multi-core processing with javascript, cited 2018 February 2.
URL `https://parallel.js.org/`

[26] The Beauty and Joy of Computing, `http://bjc.berkeley.edu/`, cited 2018 February 2.

[27] Weather Underground: meteorological datasource, http://www.wunderground.com/weather/api/, cited 2018 February 2.

[28] AIRNow: National air quality information website, http://airnow.gov/, cited 2018 February 2.

[29] EarthScope: Complementary seismic data sets, http://www.earthscope.org/research/data, cited 2018 February 2.

[30] MapQuest Traffic API, http://developer.mapquest.com/web/products/dev-services/traffic-ws, cited 2018 February 2.

[31] School for Science and Math at Vanderbilt, http://theschool.vanderbilt.edu/, cited 2018 February 2.

[32] O. Astrachan, A. Briggs, The CS principles project, ACM Inroads 3 (2) (2012) 38–42.

[33] Interdisciplinary Science and Research Program, http://www.vanderbilt.edu/cso/isr/, cited 2018 February 2.

11. Authors

Brian Broll is a Ph.D student at the Department of Electrical Engineering and Computer Science at Vanderbilt University, and a Research Assistant with the Institute for Software Integrated Systems at Vanderbilt. He holds a B.Sc. from Buena Vista University, majoring in mathematics education. His research interests include model integrated computing and computer science education.

Ákos Lédeczi is a Professor of Computer Engineering and a Senior Research Scientist at the Institute for Software Integrated Systems at Vanderbilt University. He has an M.Sc. from the Technical University of Budapest and a Ph.D. from Vanderbilt, both in electrical engineering. His research interests include model integrated computing, wireless sensor networks and computer science education.

Hamid Zare is a Ph.D student majoring in Computer Science at Vanderbilt University, and a Graduate Research Assistant with the Institute for Software Integrated Systems. He holds a B.Sc. from Tehran Azad University in Information Technology Engineering. His research interests include computer science education and web technologies.

Dung Nguyen Do T. is a sophomore studying Computer Science and Math at Vanderbilt. She is an intern at the Institute of Software Integrated Systems and contributor to NetsBlox. In her free time, she works as a member of the development team for VietAbroader Organization and a photographer for Vanderbilt Programming Board.

János Sallai is a research scientist with the Institute for Software Integrated Systems and holds an appointment of Adjunct Assistant Professor with the Department of Electrical Engineering and Computer Science in the School of Engineering at Vanderbilt University. He has 10 years of research experience in scientific computing, low-power wireless networks, networked sensing, localization, time synchronization and software development technologies for low-power platforms. Dr. Sallai holds a Ph.D. in Computer Science from Vanderbilt University (2008), and an M.Sc. from the Technical University of Budapest, Hungary (2001).

Péter Völgyesi is a Research Scientist at the Institute for Software Integrated Systems at Vanderbilt University. He is one of the architects of the Generic Modeling Environment, a widely used metaprogrammable visual modeling tool, and WebGME - its modern web-based variant. As PI on two NSF funded projects Mr. Volgyesi and his team recently developed a low-power software-defined radio platform (MarmotE) and a component-based development toolchain targeting multicore SoC architectures for wireless cyber-physical systems. His research interests include wireless sensor networks, model-based engineering and signal processing.

Miklós Maróti is an Associate Professor of Mathematics at the Bolyai Institute of the University of Szeged, Hungary, and a Visiting Professor at the Institute for Software Integrated Systems at Vanderbilt University. His research interests include universal algebra, logic, computational complexity, distributed algorithms, programming languages, wireless communication and signal processing.

Lesa Brown is a postdoctoral fellow with an instructor position at the School for Science and Math at Vanderbilt, a partnership between Metro Nashville Public Schools and Vanderbilt University. She holds a Ph.D. and M.S. in Civil Engineering from Vanderbilt University and a B.S. in Civil Engineering from Oklahoma State University. Her research interests include the multi-scale characterization and modeling of cementitious materials, the chemo-mechanical behavior of fiber reinforced composites, and the integration of hands-on engineering curriculum into K12 classrooms to better communicate STEM career pathways to high school students.

Chris Vanags is the Director of Research Initiatives in the Peabody College of Education and a Research Assistant Professor in the Department of Earth and Environmental Sciences. Dr. Vanags is one of the founding faculty of the School for Science and Math at Vanderbilt (SSMV), which was later adapted to become the Interdisciplinary Science and Research Program (ISR) [33], a partnership between Vanderbilt University and Metro Nashville Public Schools (MNPS). Dr. Vanags is the co-founder and Associate Editor of the journal *Young Scientist*. Dr. Vanags received his Ph.D. in Hydrology and Catchment Management from the University of Sydney.