A Visual Programming Environment for Learning Distributed Programming

Brian Broll
Vanderbilt University
Nashville, TN, USA

Ákos Lédeczi
Vanderbilt University
Nashville, TN, USA
akos.ledeczi@vanderbilt.edu

Péter Völgyesi
Vanderbilt University
Nashville, TN, USA

János Sallai
Vanderbilt University
Nashville, TN, USA

Miklós Maróti
Vanderbilt University
Nashville, TN, USA

Alexia Carrillo
Vanderbilt University
Nashville, TN, USA

ABSTRACT

This paper introduces NetsBlox, a visual programming environment for learning distributed programming principles. Extending both the visual formalism and open source code base of Snap!, NetsBlox provides two accessible distributed programming abstractions to simplify the process of creating networked applications: message passing and Remote Procedure Calls (RPC). Messaging passing allows NetsBlox applications to send data to other connected NetsBlox clients. Remote Procedure Calls enable seamless integration of third party services, such as Google Maps, weather, traffic and other public domain data sources, into NetsBlox applications. Other RPCs help coordinating distributed clients which may be difficult for novice programmers allowing the user to more quickly create captivating and sophisticated applications. These abstractions empower users to develop networked programs, including multi-player games and client-server applications. By providing networking support, NetsBlox not only allows users to learn distribute programming concepts but also makes programming more engaging by incorporating diverse services available on the web.

Keywords

visual programming, distributed programming, computer science education, Snap!

1. INTRODUCTION

Visual programming has proven to be an effective environment for teaching computer programming at the introductory level and a number of different educational tools [8, 3, 5] have been developed. Some have even leveraged computational modeling to teach science [7, 2]. While many of the currently available tools are quite effective, they all lack the concept of networking. While it is understandable that students would need to learn the basics of programming on a single machine before they are able to develop distributed applications, the prevalence of networking in day-to-day interactions is nearly ubiquitous to the point where most people interact with network-driven applications such as Facebook, Google, Twitter, Snapchat, Skype and Netflix on a daily basis. The recent emphasis on self-driving cars and home automation suggests that the importance and pervasiveness of distributed computation will only increase in the coming years. Given the growing significance of networking in today's world, it is a difficult topic to ignore when teaching introductory computer science.

The prevalence of networking in today's world also provides a significant opportunity for teaching computer programming by making it more relevant and engaging. It can provide access to resources that are unavailable when simply creating local applications. Integrating networking into CS curriculum provides the opportunity for beginners to create applications that interact with networked resources, including scientific datasets, and can provide more relevance and timeliness to the exercises and assignments.

At the college level, the ACM IEEE Computer Science curriculum (2013) [9] advocates introducing the following topics to CS students: asynchronous and synchronous communication, reliable and unreliable protocols, and the need for concurrency in operating systems. We strongly believe that through the use of carefully designed distributed computing abstractions, an intuitive, familiar user interface and a sophisticated and scalable cloud-based infrastructure, it is possible to teach the basics of distributed programming principles to students who are still in high school. To this end, we have created NetsBlox, a visual programming environment designed to allow users to create distributed applications, giving them a first hand experience in network programming. NetsBlox extends the visual paradigm of Scratch [6] and Snap! [8] and, consequently, provides a natural progression for students already familiar with these environments.

The rest of the paper is structured as follows. First we introduce the distributed programming primitives at the heart of NetsBlox. Then we show a few example applications that can be created using these simple abstractions. Then we present the virtual overlay network model followed by the description of the cloud-based architecture of the environment. Finally, we present the results from a small-scale pilot study we conducted with 30 high school students.

SIGCSE '17, March 08-11, 2017, Seattle, WA, USA

DOI: http://dx.doi.org/10.1145/3017680.3017741

2. DISTRIBUTED PRIMITIVES

One main strength of NetsBlox lies in its choice of simple, accessible abstractions for distributed programming. These abstractions have been designed to hide unnecessary complexity while providing powerful, flexible primitives from which users can develop complex networked applications. The distributed abstractions provided by NetsBlox are *Remote Procedure Calls (RPC)*, *Messages* and the concept of a *Room*.

Remote Procedure Calls are the simplest, yet most diverse, distributed abstraction for NetsBlox users. As the name implies, RPC allows for invoking code that will be executed at a remote location, and then (optionally) getting back the results of the computation. The semantics of RPC is as expected: multiple input arguments, single output argument, pass-by-value and blocking call. Syntactically, RPCs are represented as another block in the NetsBlox environment and are similar to custom blocks in Snap! RPCs provide two major benefits to NetsBlox users: they provide access to external resources in a NetsBlox-friendly way and provide scaffolding for the user to create more complex applications by handling the difficult computation remotely.

Third party API access increases the relevancy of networked programming thereby empowering students to interact with familiar, existing web applications such as Google Maps. The support for weather, traffic, air quality and other APIs provides easy access to data, enabling users to better integrate their applications into the real world. This allows NetsBlox users to not only think about their single application but how their application interacts with other networked resources to serve its purpose.

NetsBlox RPCs simplify tasks that may be more difficult for novice programmers enabling them to build more complex and intriguing applications more quickly. For example, the current NetsBlox prototype provides RPCs for Tic Tac Toe and Battleship which manage the more complex aspects of the game, allowing the users to design their games at a higher level of abstractions. The user is therefore able to focus on when the applications are communicating with each other and how the application will handle events (such as a "hit" in Battleship) rather than struggling with the best way to represent the state of the game, such as the board and the boats on the board. Especially in a classroom setting, this provides a natural progression for creating the given application. Students can first build the game using the RPCs then be asked to remove the RPCs and solve some of the more difficult aspects of the given application in an iterative fashion. As they will build a fully functioning application more quickly, this scaffolding procedure will build student confidence and allow them to focus on a subset of the problem at a time rather than being potentially overwhelmed when trying to tackle all the challenges at once.

Another important network abstraction in NetsBlox is the concept of ***Messages***. Messages are very similar to Scratch events. Individual event handlers can be created for the given messages and receiving a message will trigger all message handlers concurrently. Unlike Scratch and Snap!, NetsBlox messages are sent to remote clients and contain a structured data payload. Also, NetsBlox messages support peer to peer communication in which users can send messages to a single specific client (rather than simply broadcasting).

In order to support structured data payloads, NetsBlox introduces the concept of a ***Message Type***. A Message Type defines the data that comprises the given type of message. Defining Message Types allows the different clients to agree upon a communication protocol; a necessity when creating a distributed application. Note that the send/receive visual blocks dynamically restructure themselves based on the selected message type. This provides a clear and concise way to interact with structured data as the user is shown only the necessary inputs or outputs of the given block. An example of these dynamic messaging blocks can be seen in Figure 1.

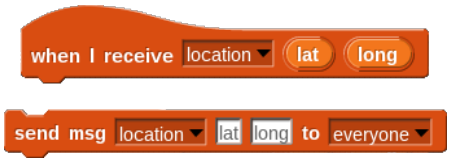

Figure 1: Messages with data in NetsBlox

In Figure 1, the "location" Message Type has been defined with two fields: "lat" and "long". Given this Message Type schema, the "`send msg`" block has created two corresponding inputs with hint text over each (which specify the name of the given field). The message handler is updated similarly where variable blocks have been created for each defined field in the "location" Message Type. Figure 2 shows another message handler block for a "chat" Message Type containing three fields: "sender", "time" and "message".

Figure 2: Chat Message Handler Block

The third distributed programming abstraction is the concept of the ***Room***. Much like the Stage in Snap!, a room is defined for every NetsBlox project. A room defines the virtual local network for the given project and is composed of a number of roles, or named NetsBlox clients. For example, a Tic Tac Toe game would have two roles, "X" and "O." These roles can then be occupied by NetsBlox users who want to play the given roles in the Tic Tac Toe game. As the room is associated with the NetsBlox project, the project owner is also the maintainer of the room. The room simplifies network complexities such as client discovery and network addressing. In NetsBlox, clients sharing a room can see each other and can send messages to one another using their client names (unique within the given room).

Roles are similar to projects in Snap! where each role has its own sprites and stage which all have their own associated scripts. Roles can also have their own custom blocks and custom message types. When occupying a role, the user can modify the scripts owned by the given role and can interact with the users occupying the other roles in the room. As the project owner also owns the associated room, he or she manages the creation, modification and removal of roles from a room as well as the users currently occupying roles in the room (by inviting or evicting users).

Figure 3 shows the room editor for a NetsBlox project titled "TicTacToe." The sections of the room in the room editor represent each role in the room. The bolded text displays the given role's name while the italicized text (below

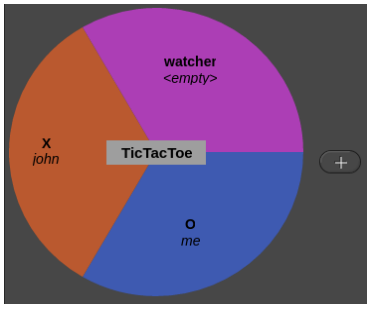

Figure 3: NetsBlox Room

the role name) shows the username of the NetsBlox user currently occupying the given role. The project name is also displayed in the grey box in the center of the room editor and is editable by the project owner on click. The owner of the project can edit visible roles by clicking on them and can create new roles with the button to the right of the room.

This room contains three roles: "X", "O" and "watcher." The current user is occupying the "O" role and a user with the name "john" is occupying the "X" role. The last role, "watcher", is left unoccupied. As this room defines the visibility and addressing of the given roles, the "`send msg`" block is automatically updated to contain the defined roles in the "target" dropdown menu. That is, the "`send msg`" block now provides "X" and "watcher" as options for the target of the message sent from "O."

3. EXAMPLE PROJECTS

Figure 4 shows an example application which displays historical earthquake data. It incorporates the Maps RPC to retrieve a dynamic map of the desired region (using Google Maps). The application then uses the earthquake RPC to request information about all earthquakes in the region the user navigates to with the arrow keys and +/- for zooming. The RPC request triggers messages from the NetsBlox server, one per earthquake event; the application then uses a message handler which displays each earthquake on the map.

This process is relatively easy to see in the code in Figure 4. The middle script is an event handler which is triggered when it receives the local "showEarthquakes" event triggered by a mouse click. It calls the earthquake RPC which, in turn, initiates the messages from the server about the earthquakes in the currently visible map region. The bottom script is the corresponding message handler. It contains the logic for drawing each earthquake on the map. This is done by first moving the sprite to the given x and y coordinate (converted from the latitude and longitude using another NetsBlox RPC[1]). The sprite then draws the given circle for the earthquake with a radius corresponding to its magnitude. Finally, the top script stops further messages and clears the map when its position or zoom is updated.

The earthquake application exemplifies both RPCs and remote messages and highlights that they can be combined to create a powerful application relatively simply. In this case, messages provide users with a simple way to manage structured data in an effective and concise manner within a visual programming environment. As the earthquake RPC is providing a list of structured data, the use of messages facilitates "unpacking" of the data into its composed fields while managing the complexity introduced due to the request for a potentially large number of these structured elements simultaneously.

Figure 5 shows an example of a Battleship multi-player game implemented in NetsBlox including a portion of the script of one of its sprites. This application uses RPCs to coordinate the game by maintaining the game state, i.e., the players are still placing ships or are already firing upon one another, etc., and sending the "hit" and "miss" events after a player makes a move. This process simplifies the development of the battleship game, providing users with the necessary resources to quickly advance their distributed programming skills.

[1]This conversion is implemented as an RPC due to the complexity of handling Mercator projections.

4. NETWORK MODEL

NetsBlox requires a powerful yet intuitive network model. It is important to note that the technical details described here are completely hidden from users. The core design principles are: (1) simplified distributed programming through the elimination of tedious and error prone tasks to shield programmers from distracting technical details and common pitfalls that include firewalls, routing, address resolution, automatic reconnection, etc., (2) uniform addressing and discovery scheme for distributed artifacts, and (3) support for both synchronous and asynchronous communication.

NetsBlox provides a ***virtual overlay network abstraction*** to eliminate many of these accidental complexities resulting from network programming. This virtual overlay network abstraction facilitates bidirectional, peer-to-peer communication between NetsBlox clients without the need for explicit message routing from the NetsBlox programmer. This overlay network was built on top of existing network technologies and does not have a one-to-one mapping between virtualized primitives and actual network packets and connections. For example, NetsBlox utilizes a centralized server to route all messages within NetsBlox and maintains a persistent connection between the server and the connected clients. This centralized approach enables NetsBlox to provide effective management and instrumentation capabilities.

These abstractions are presented to the user in the form of two distributed primitives: ***Rooms*** and ***Roles***. As introduced in Section 2, a room is associated with each NetsBlox project and roles are the NetsBlox clients associated with the given room. That is, the room represents the virtual overlay network abstraction associated with the given NetsBlox project and the roles represent the clients in this overlay network. The use of a centralized approach to the overlay network simplifies the challenge of discovering the available clients as the NetsBlox server keeps record of the connected clients and which role the given client is occupying.

Containers: To extend past web browsers and into more diverse, heterogeneous execution environments, NetsBlox introduces the idea of a container. A container is a NetsBlox hosting environment which is capable of running NetsBlox projects and providing the necessary services which are required for the given project. Currently, containers are available for the web browser and Android; in the future, containers will be developed for more platforms such as iOS, networked embedded devices, such as the Raspberry PI, and running as a service in the cloud.

Containers use a common JavaScript and HTML5 code

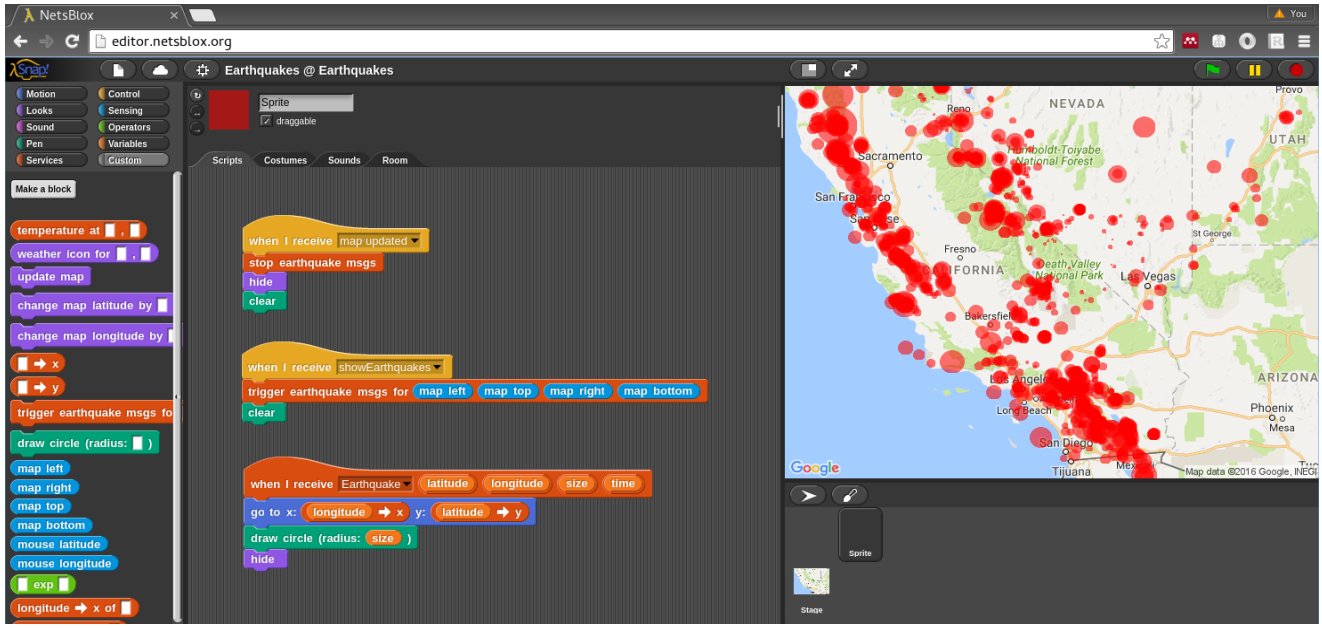

Figure 4: The Earthquake application

base and provide the required services, such as network connectivity, to the hosted environment. Mobile platforms are targeted using Apache Cordova which supports deployment on a number of mobile platforms. The hosting environment can be executed on-demand or registered as a background service to allow remote application deployment.

Another main component of the NetsBlox network model is the concept of ***coordination***. While coordination among a few clients in a small networked application can be simple; coordination becomes much more challenging as the application and number of clients grow. For example, a chat application requires very little coordination between the given roles; if a client wants to send a message he/she simply enters the message (there are no rules about the order or frequency of the messages). However, in the case of a two player game such as Battleship, much more coordination between the players is required. The game first expects the users to place their ships and neither player can fire upon the other until this placement period is over. After placing the ships, the game still enforces typical turn-based rules; that is, the players cannot fire upon one another until the previous player has successfully made his/her move. Additionally, NetsBlox programmers may want to develop applications which support a flexible number of clients. In this scenario, coordinating the discovery and routing of messages becomes more complex.

NetsBlox addresses both of these accidental complexities via the use of RPCs. As was introduced in Section 2, RPCs trigger the execution of some remote functionality. They also have the ability to store state information on the server. This functionality enables them to house information (e.g., current turn), and manage the coordination of the clients by enforcing rules (e.g., requiring roles to alternate turns). Upon the start of a given role's turn, the NetsBlox server sends a notification to the respective role and initiates the given role's turn. This simplifies the coordination of roles in the room by providing the user with the appropriate events rather than leaving the management of the coordination of the distributed roles to the user.

As RPCs can save state and send messages to NetsBlox clients, they are powerful enough to solve the challenge of creating applications with a flexible number of clients. Because of this, NetsBlox RPCs can easily be used to provide publish-subscribe style messaging capabilities between NetsBlox clients from different rooms.

5. ARCHITECTURE

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, (4) authentication and run-time user association. 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.

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, e.g., authentication, project access, table/role management, are also implemented by REST primitives.

Bi-directional NetsBlox communication, 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

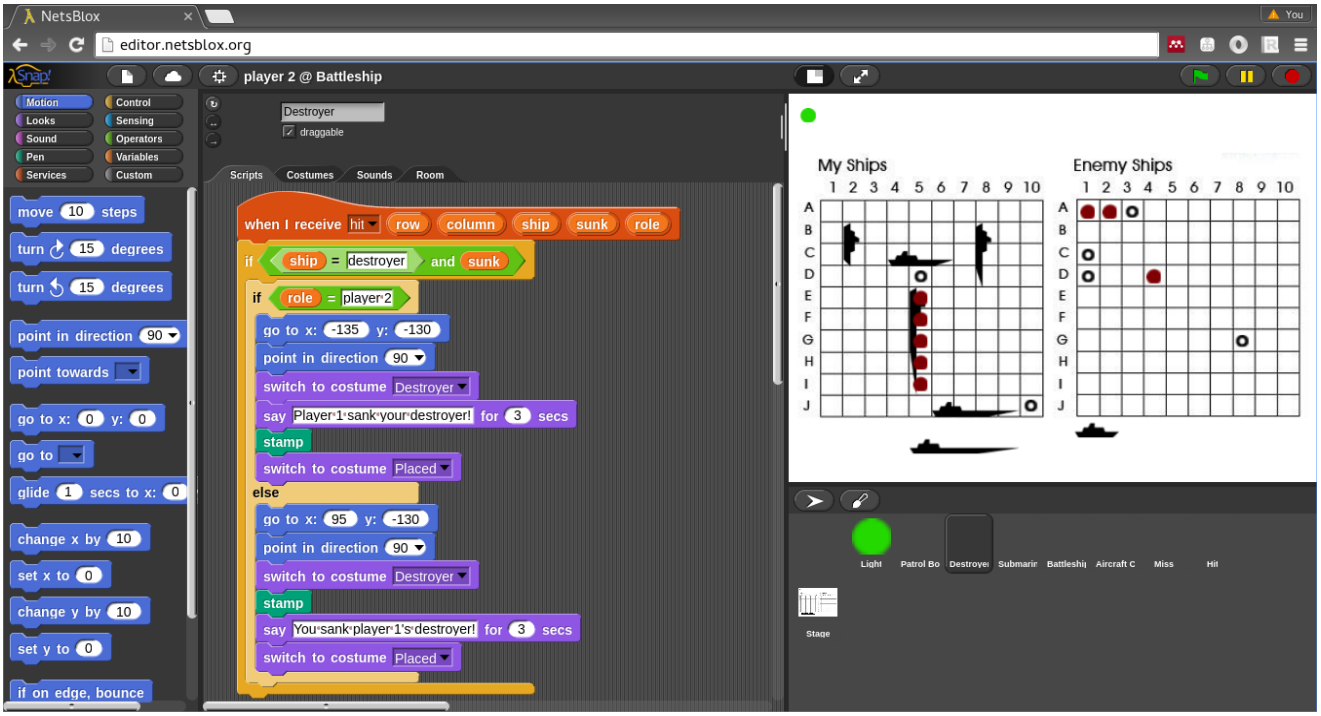

Figure 5: The Battleship multi-player game

within the same room, we have implemented a virtual network abstraction. 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).

The high-level architecture of NetsBlox is shown in Figure 6. The server —implemented with server-side JavaScript

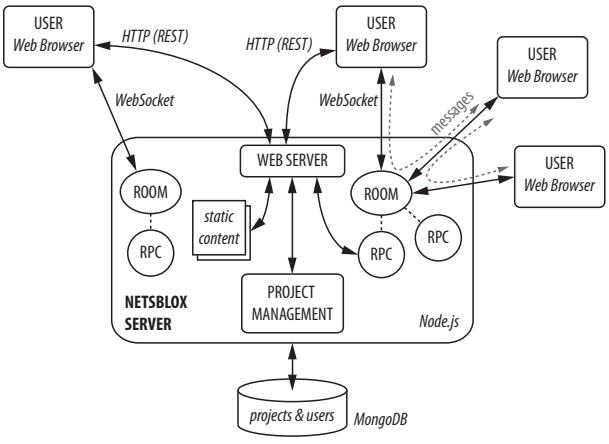

Figure 6: Network Architecture

technologies—provides web services and hosts the web application (e.g., the 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.

6. PRELIMINARY EVALUATION

We have recently completed a small-scale pilot study with 30 students at Hillsboro High School as part of the Interdisciplinary Science and Research Program [4], a partnership between the Center for Science Outreach (CSO) at Vanderbilt University and Metropolitan Nashville Public Schools. This pilot study assumed no prior programming experience and taught programming concepts ranging from basic data structures to complex networking.

The first half of the 9-week long curriculum introduced computer programming using Snap!. After establishing an understanding of basic programming concepts, the students moved to NetsBlox and spent the second half of the course learning basic distributed programming. The distributed programming curriculum started by first introducing students to RPCs using a developed weather map application. After presenting the weather map application, students were tasked with extending the project to show the weather of three randomly selected locations currently visible on the map (the map could move all around the world).

Following the introduction to RPCs, students were introduced to more complex networked programming using

the earthquake application (see Figure 4). This example required students to use RPCs and messaging to retrieve the relevant information. The students' final assignment was to use the network messaging functionality to develop a chatroom in which different NetsBlox clients could communicate with one another.

The students responded positively to the networking functionality. Shortly after introducing the concept of remote messaging (and providing a simple demo), the students were given time to work in small groups and create applications using messaging. We found that students were quite excited to see the ability to send, even sometimes trivial, messages from their computer to another student in the course. One group in particular lined up all their computers and created an application in which clients simply forwarded the message along to other roles at the room (much like a game of telephone). The students then demonstrated how they could send a message and see it show up on each screen before it was passed along to the next role in the room. The program finished with the message being received by the intended target, the last role in room.

Although the level of enthusiasm can be difficult to quantify, we found that high school students enjoyed the social aspect of developing networked applications. Networked applications also allowed the demonstrations to be more interactive as students were able to partake in the examples.

We conducted a brief survey of the students at the conclusion of the program to assess their attitudes towards the small scale pilot study. Although students enjoyed the additional network functionality, it was clear that the glitches and bugs in the early prototype made it difficult for students (and teaching staff) to focus on the new concepts while trying to troubleshoot software errors.

It was clear that students with previous programming experience were much quicker to grasp the distributed programming concepts and apply them successfully. These students also had a more favorable view of the tool itself. While these lessons are not surprising, it does emphasize the need to develop a general foundation in programming and an applied knowledge of Snap! before attempting to teach them distributed programming.

7. CONCLUSIONS

This paper presented NetsBlox, a scalable, visual programming platform for learning distributed programming principles. NetsBlox extends the well-known and widely-used Snap! environment, providing a natural progression into distributed programming for the expanding population of students familiar with blocks-based programming environments. NetsBlox is well equipped to support many of the big ideas and computational thinking practices that the AP CS Principles curriculum emphasizes due to its support for distributed programming.

Furthermore, NetsBlox leverages powerful RPCs to provide access to third party web services, including APIs serving scientific, geographic and social databases within the visual programming environment. This seamless integration of third party datasets empowers early programmers to build applications which interact with the external world and leverage this information to potentially enhance their understanding of STEM concepts while developing computer programming skills simultaneously.

Nevertheless, NetsBlox is in its infancy. Our future work involves adding a lot of new services and data sources to NetsBlox in the form of a large library of RPCs. Equally important is the creation of new curricular modules that can be incorporated to existing courses such as the Beauty and Joy of Computing (BJC) [1]. Finally, a robust classroom evaluation strategy will need to be developed and executed to steer the ongoing development of NetsBlox in the right direction.

8. ACKNOWLEDGMENTS

This work was supported by a Vanderbilt University TIPs grant and 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 authors and do not necessarily reflect the views of the National Science Foundation.

9. ADDITIONAL AUTHORS

Additional authors: Stephanie L. Weeden-Wright (Vanderbilt University), Chris Vanags (Vanderbilt University), Joshua D. Swartz (Hillsboro Comp High School), and Melvin Lu (Vanderbilt University),

10. REFERENCES

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

[2] P. Blikstein and 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):81–119, 2009.

[3] N. Fraser. Google blockly-a visual programming editor. *URL: http://code. google. com/p/blockly. Accessed Aug*, 2014.

[4] Interdisciplinary Science and Research Program. http://www.vanderbilt.edu/cso/isr/. Cited 2016 August 14.

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

[6] J. Maloney, L. Burd, Y. Kafai, N. Rusk, B. Silverman, and M. Resnick. Scratch: A sneak preview. In *Proceedings of the Second International Conference on Creating, Connecting and Collaborating through Computing*, C5 '04, pages 104–109, Washington, DC, USA, 2004. IEEE Computer Society.

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

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

[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. http://www.acm.org/education/CS2013-final-report.pdf, 2013.