# Introducing Parallel and Distributed Computing to K12

Brian Broll, Ákos Lédeczi,
Péter Völgyesi, János Sallai, Miklós Maróti
*Institute for Software Integrated Systems*
*Vanderbilt University*
*Nashville, TN, USA*
*Email: akos.ledeczi@vanderbilt.edu*

Chris Vanags
*Center for Science Outreach*
*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 it builds upon its visual formalism as well as its open source code base. NetsBlox adds distributed programming capabilities to Snap! by introducing two simple abstractions: messages and NetsBlox services. Messages containing data can be exchanged by two or more NetsBlox programs running on different computers connected to the Internet. Services are called on a client program and are executed on the NetsBlox server. These two abstractions make it possible to create distributed programs, for example multi-player games or client-server applications. We believe that NetsBlox provides increased motivation to high-school students to become creators and not just consumers of technology. At the same time, it helps teach them basic distributed programming concepts.**

***Keywords*-visual programming, distributed programming, computer science education**

## I. 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].

There are many efforts around the world to introduce young learners to computer programming, such as code.org, Khan Academy, LEGO Mindstorms or the Raspberry Pi. Visual programming languages have come to play a prominent role in this movement and have been used to teach children programming [4], [5] as well as using computational modeling to teach and learn science [6], [7]. 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, some of the most widely used computer applications today rely on the network to provide their functionality. The web, texting, Twitter, Facebook and other social networks, multiplayer games, Pandora, Netflix, Amazon Echo, Siri, Google Maps and YouTube 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. Teaching distributed programming then constitutes both a necessity and a great opportunity. It is a necessity, because distributed computing is becoming part of basic computer literacy. And it is also an opportunity, because children already use the technology every day and their natural curiosity will provide excellent motivation for them to learn more about it.

We believe that it is not enough to introduce computer programming into the K12 curriculum, but it also necessary to teach distributed computing concepts to young learners. At the college level, the ACM IEEE Computer Science curriculum (2013) [8] 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 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 which extends the visual programming paradigm of Scratch [4]. NetsBlox introduces a few carefully selected abstractions that enables children to create distributed computing applications [9].

The literature on educational computing is rife with observations of children's difficulties with learning basic constructs of programming. 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 [5]. 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 Snap! [10], [11], which itself is an extension of Scratch [4], StarLogo TNG [12], and Alice [13].

We decided to base our research on Scratch [4] because it is one of the most mature and widely used approaches and we have significant experience using and teaching it. A Scratch program consists of one or more sprites that can

have multiple visual representation ("Costumes") and one or more scripts. The program is executed on the "stage". The available computing blocks (instructions, operators, etc.) are grouped by various color coded tabs according to their role, e.g., operators, program control, variables, etc. The shapes of the different blocks give a hint about their role, for example, you can only insert hexagons as a condition into an if block header. This prevents syntax errors altogether. As an illustration of the power of Scratch, see a 10-block program that plots the sound volume measured by the microphone of the computer continuously at http://tinyurl.com/loudnessmeter. Note that it is not only the intuitive and easy to learn visual language but also the superb user interface that make Scratch a great tool.

### A. 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 [14], [15]. Meerbaum-Salant et al. [14] investigated difficulties experienced by students in understanding concurrency using Scratch. They divided this concept in two categories: Type I concurrency occurs when several sprites are executing scripts simultaneously, such as sprites representing two dancers. Type II concurrency occurs when a single sprite executes 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 seems to be much more intuitive for students and easier to grasp.

Maloney, et al. [15] reported on an experiment where Scratch was used by students in an after-school clubhouse. These students were self-selected and self-paced, receiving no formal instruction. By analyzing the students projects, they found that the majority of the projects that 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 (p. 368, our emphasis). The researchers did not investigate if the use of concurrency demonstrates 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 % or less).

## II. NetsBlox

Scratch is implemented in Flash. However, Snap! [11] 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 [16]. Therefore, we have built NetsBlox upon Snap! [9] It constitutes an excellent starting point because just 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.

We believe that the main appeal of NetsBlox is the increased motivation it provides because young learners are able to create new classes of programs that are currently out of reach. For example, multi-player video games are very popular with children and NetsBlox supports the creation of non-trivial gaming programs. Real-time games with 3D scene rendering are obviously beyond the realm of possibilities. However, 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. Just imagine an average high school student creating a multi-player game, running it on her phone and playing against a friend over the Internet after just a few weeks of instruction. That is the promise of NetsBlox.

Furthermore, there are a large number of publicly available interesting data sets on the web. Examples include the weather, air pollution, seismic data, real-time traffic information 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 already provides access to a select set of interesting data sources. These are available from NetsBlox programs via a simple abstraction called NetsBlox Services. Essentially these services provide a mapping between NetsBlox service calls and the corresponding API of the public data service. For example, the NetsBlox Weather Service has a function called "*temp*" that takes arguments for the location and returns the corresponding temperature. A second function returns a weather icon representing the current conditions. On the NetsBlox server, they silently invoke the proper call on the OpenWeatherMap API to get the data.

The possibilities are quite literally limitless. With NetsBlox, children are able to create all kinds of imaginative applications that utilize the wealth of information available on the web provided to them using a single, simple abstraction. One potential difficulty is that much of the data are geospatial. To help students make use of it, NetsBlox integrates Google Maps as an interactive background, again, using services. See Figure 1. 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.

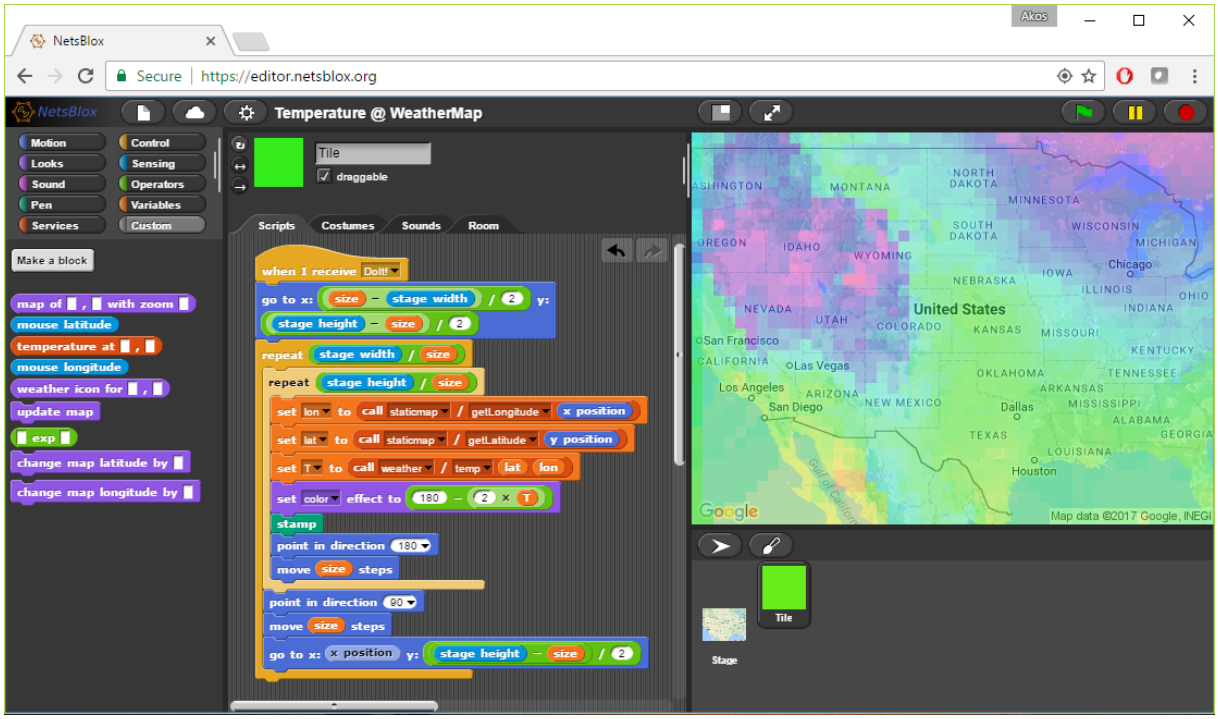


Figure 1. Weather Map application in NetsBlox showing temperatures in the continental US in the morning of January 29, 2017

## III. 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)*.

### A. Messages

Peer to peer communication is supported by ***Messages***. Messages are very similar to Events already present in Snap! and in Scratch. 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 2).


Figure 2. Scratch event example

In NetsBlox, a Message is an Event that contains data payload. Users are able to 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 3.

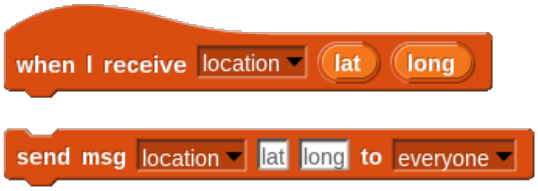


Figure 3. Sending and receiving messages with data in NetsBlox

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 3, provide a dropdown of all the currently defined message types; upon selecting a given message type, the block is updated to show the corresponding data fields. The message type in Figure 3 has the name "location" which contains two fields: "lat" and "long".

As the creation of different distributed applications will likely require unique messaging protocols, including unique message types, it is important that users are able to define their own custom message types. NetsBlox supports this creation and management of message types similarly to the creation and management of variables in Snap!. An example of creating a message type can be found in Figure 4 and the corresponding message handler is shown in Figure 4.


Figure 4. Custom Message Creation


Figure 5. 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 can communicate with each other using messages. For example, a chess game app would have two Roles, black and white.

Like the Stage in Scratch, every NetsBlox project automatically has a single associated Room. The project owner manages the Room and its Roles. This includes creating, removing, renaming and cloning 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 enabling collaboration with other users by delegating parts of the project to peers. Once a distributed program is ready, the owner can invite other users to run the program, e.g., to play the game.

Figure 6 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 on the right 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, 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 collaborate on 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

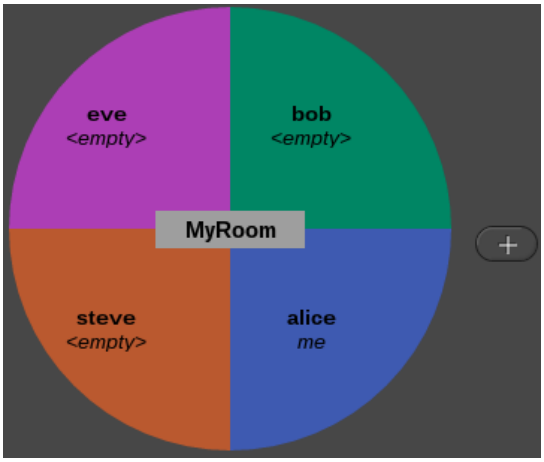


Figure 6. NetsBlox Room

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 Role of the sender. Figure 7 shows one example for sending a simple message in the context of the Room in Figure 6. 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 7. Sending messages to other NetsBlox clients

The semantics of Messages in NetsBlox are based on the semantics of Events in Scratch and Snap! [4]. 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 either. Note that if a message handler is still executing when a new message of the same type arrives, the new message is queued

and will execute once the current execution has completed. Multiple message handlers for the same message type will all be executed in parallel for every received message of the given type.

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, that is, 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.

To illustrate the concepts introduced above, let us consider a simple example. Figure 8 shows a 2-person dice game. The idea is that two players both roll their dice and whoever has a higher number wins. In case of a tie, they roll again.


Figure 8. The scripts of the Dice game

The program is symmetrical in that both players (Roles) use the exact same scripts. The game is started by one player clicking the green flag. The script corresponding to this event sends a message to every role to start the game. The corresponding message handler picks a random number between 1 and 6, shows the correct costume, i.e., side of the dice, and sends a "roll" message to the "others in room" that is, the other player. Not naming the other Role explicitly makes it possible to have the exact same code for both Roles.

The script with the "when I receive roll" header runs once 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, the local "dice" and the remote "roll". The interesting case is when the two are equal. In this case, each player rolls again and send the new dice value to the other side using another "roll" message. Otherwise, the players are notified about the outcome of the game by a text displayed on the stage.

### B. Remote Procedure Calls (RPC)

RPCs are the highest level of distributed abstraction NetsBlox employs. An 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 RPCs are as expected: multiple input arguments, single output argument, pass-by-value and blocking call. Syntactically, RPCs appear as a reporter block on the block palette in NetsBlox; however, they are often packaged in custom blocks as convenient, more user friendly libraries. This allows the blocks to be represented in a more intuitive way as custom blocks can be a different type of block (e.g., an RPC without a return value may use a command block) or can be assigned to a different color to make their functionality more apparent. The block for calling RPCs is shown in Figure 9.

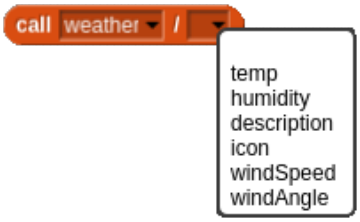


Figure 9. Calling a NetsBlox Service

Related RPCs are grouped into *Services*. As shown in Figure 9, the block for calling RPCs is a reporter block with two drop-down menus. The first drop-down is dynamically populated with the supported Services of the given NetsBlox server. The second drop-down is populated with the RPCs of the given Service. In the given example, the block is set to the *weather* Service and the second drop-down is populated with all weather-related RPCs. These include *temp*, *humidity*, *windSpeed*, etc. After selecting an RPC, the block is updated to provide named fields for each argument for the given function. An example of this is shown in Figure 10. Also see Figure 1 for an additional example of using the Map Service.


Figure 10. Getting the current temperature with the Weather Service

Note that a Service can also include message types and server initiated messages. For example, the Earthquake Service has a single RPC called "trigger earthquake messages" which is called with coordinates of the area of interest. In turn, the Service will gather historical earthquakes from the web and send one message per earthquake event to the user, i.e., the caller of the RPC. Each such message contains the location, magnitude and date of the given seismic event.

NetsBlox Services 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 services using a global context can be found in the services exposing 3rd party endpoints, such as Google Maps, as they often cache their results to minimize redundant requests to the given external API.

Similarly, for multi-player games, one of the main challenges can be maintaining the game state and enforcing the rules. For example, implementing a chess, poker or even a battleship application in a visual language is a non-trivial undertaking. The NetsBlox Service concept makes it possible to move the most challenging aspects of such games to the server. For example, the Battleship Helper Service provides assistance in turn coordination and game state management. This includes maintaining whether the players are still placing ships or have already proceeded to shooting at one another, storing the list of hits and misses, as well as enforcing the turn-based nature of the game. These kinds of Services generally send messages to notify the players (Roles) of important events, such as "your turn" or "game over" and expect RPC invocations about the users actions.

Currently, NetsBlox supports a fixed set of Services that run on a NetsBlox server. From the user's perspective, services are executing "in the cloud". In the near future, we will enable users to host one Role of their program on the server, in effect creating their own Service. Custom blocks defined for the given Role will become RPCs and they will be able to send messages as well. This will be one way to support the extensibility of NetsBlox.

### C. Dynamic Virtual Networks

The Room concept provides an easy-to-understand abstraction for students to implement their programs that include more than one computer. However, its inherently static nature can prove to be inflexible for various applications. What if we wanted to write a program with a varying number of users, i.e., Roles? What if we wanted to allow two different NetsBlox programs to message each other?

NetsBlox supports more dynamic networking with the *Public Role Id* Service that allows users to request a public id, a kind of address, which can be used to facilitate inter-room communication. That is, a user can request a public role id and share this id with other users via email or text message. The other users can then send messages to this id. The NetsBlox server will then resolve these public ids to the initial user and route the messages accordingly.

A simple illustration of using the Public Roles Service is shown in Figure 11. In this example, when the green flag is pressed the program request a public role id. It then records this id and announces it to the user so he/she can easily share it with others.

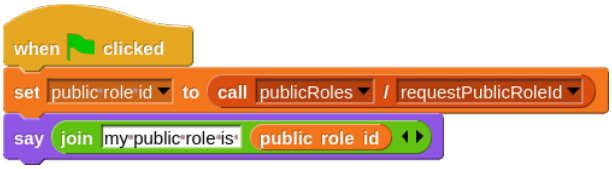


Figure 11. Requesting a Public Role Id

Figure 12 shows the other side. In this example, the user is first prompted for the public role id of the recipient. Then the program requests its own public role id so it can receive any responses. Finally, the program sends a message to the recipient that includes a message and its public role id. This example illustrates the simplicity of inter-room communication; rather than introducing an entirely new concept, public roles simply build on the existing concepts in an intuitive and natural way. This gradual progression should simplify the transition from building applications in a small, clearly defined network (the Room) to more flexible, scalable applications which include a higher degree of uncertainty and complexity.


Figure 12. Sending message to a public role

A good use of this facility is an illustration of the "massively" parallel, volunteer computing concept. For example, prime factorization is embarrassingly parallel and simple enough to implement in NetsBlox. A master program can request a public role id and wait for worker programs to connect. In turn, it can send messages to the workers to test possible factors one by one. The master distributes the work and gathers and combines the results. A brief video demonstration of this application can be found at [17].

The public role id concept, when used in conjunction with NetsBlox messages, also allows users to develop higher level messaging patterns, such as Publish-Subscribe shown in Figure 13. In this example, we have created a Publish-Subscribe broker which has defined 4 different message types: *publish*, *subscribe*, *unsubscribe* and *update*. The broker then maintains a variable called "subscriptions" which contains

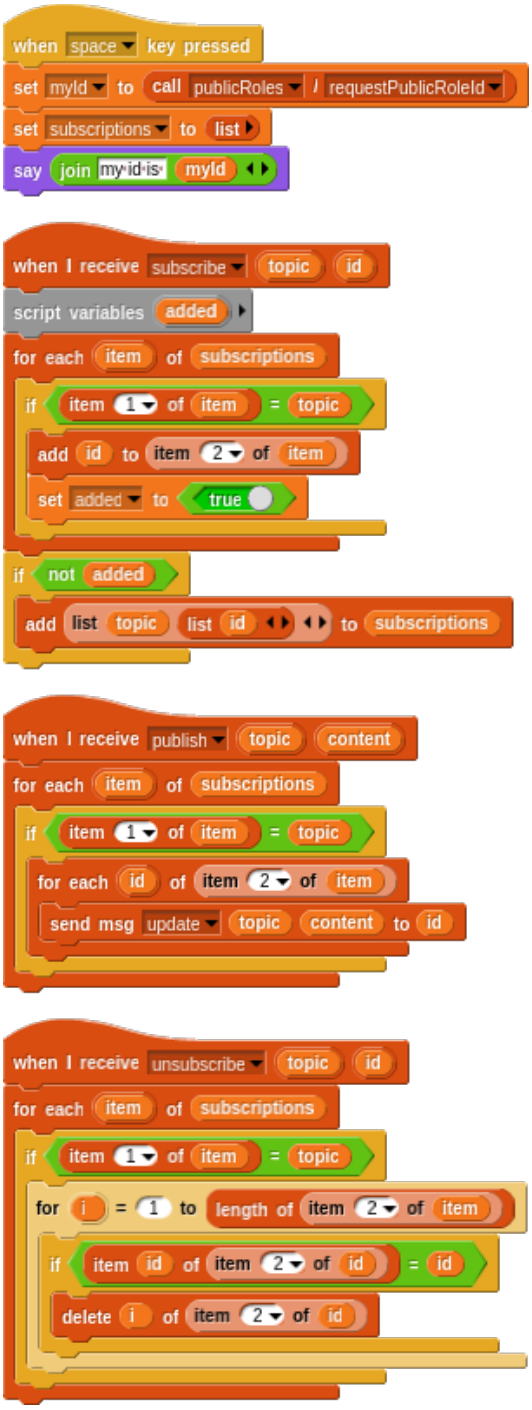


Figure 13. Publish-Subscribe Broker in NetsBlox

a list of topics and the associated subscribers. On *subscribe* and *unsubscribe* events, the broker will add or remove the requestor from it's 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.

An example of a subscriber is shown in Figure 14. 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 13) which will simply display the update to the user.

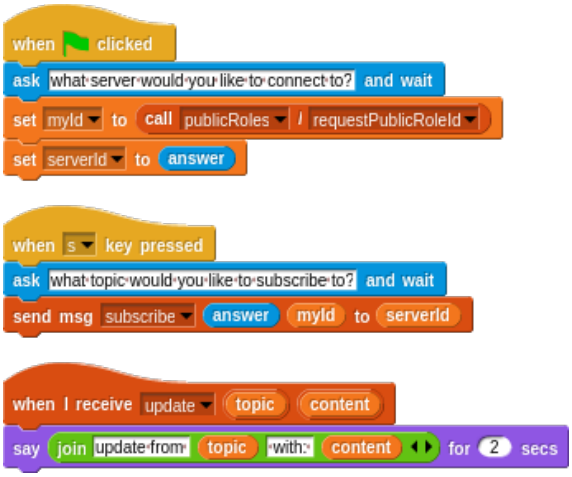


Figure 14. Subscribe Client in NetsBlox

Although this is a relatively simple example, it demonstrates the ability to compose higher level messaging patterns completely within NetsBlox. Building higher-level abstractions from the NetsBlox primitives allows users to not only use these messaging patterns but also to understand them on a lower level. Alternatively, providing these patterns as built-in concepts would allow users to understand how to use them but would not enable them to open the actual implementation and gain a deeper understanding of the concepts.

## IV. Conclusions and Future Work

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 natural progression to students who take the Beauty and Joy of Computing (BJC) class and consequently, novel curricular units can be easily incorporated into BJC, one of the new AP CS Principles courses [18]. NetsBlox is an ideal vehicle to support some of the big ideas and computational thinking practices that the AP CS Principles curriculum emphasizes. These include the Internet, communicating, collaborating, cybersecurity and global impact.

Furthermore, providing access to vast arrays of data on the Internet right from the visual programming environment in a uniform manner will empower the students to create innovative science projects and bring STEM concepts into CS education at the same time. The ability to create multi-player games will provide increased motivation for a large

number of students making them creators and not just consumers of digital entertainment.

Nevertheless, NetsBlox is in its infancy. While the distributed computing primitives are fully implemented, the robustness of the tool needs to be improved. The most promising feature we are working on is collaborative editing of a project by multiple students similarly to how Google Docs works. This will enable pair programming, group projects and other novel forms of collaboration even outside of the classroom. Our future work also involves adding a lot of new services and data sources to NetsBlox in the form of a large library of services. Equally important is to create new curricular modules that can be incorporated to existing courses such as the BJC. Finally, extensive classroom studies need to be developed and executed to steer the ongoing development of NetsBlox in the right direction.

## V. Acknowledgements

We thank Pratim Sengupta for his contributions during the initial discussions about NetsBlox. Funding from the Trans-institutional Programs (TIPs) of Vanderbilt University made possible to start the development of the tool. This material is also based in part upon work supported by the National Science Foundation under Grant Numbers CNS-1644848 and DRL-1640199. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

## References

[1] J. M. Wing, "Computational thinking," *Communications of the ACM, Viewpoint*, vol. 49, no. 3, pp. 33–35, Mar. 2006.

[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. [Online]. Available: http://www.nap.edu/openbook.php?record_id=12840

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

[4] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The scratch programming language and environment," *ACM Transactions on Computing Education (TOCE)*, vol. 10, no. 4, p. 16, 2010.

[5] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Comput. Surv.*, vol. 37, no. 2, pp. 83–137, Jun. 2005. [Online]. Available: http://doi.acm.org/10.1145/1089733.1089734

[6] 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*, vol. 18, no. 2, pp. 351–380, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10639-012-9240-x

[7] 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*, vol. 14, no. 2, pp. 81–119, 2009. [Online]. Available: http://dx.doi.org/10.1007/s10758-009-9148-8

[8] 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.

[9] B. Broll, A. Lédeczi, P. Völgyesi, J. Sallai, M. Maróti, S. Wieden-Wright, A. Melo, and C. Vanags, "A visual programming environment for learning distributed programming," in *Proceedings of the 48th ACM Technical Symposium on Computing Science Education*. ACM, 2017.

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

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

[12] E. Klopfer, S. Yoon, and T. Um, "Teaching complex dynamic systems to young students with starlogo," *Journal of Computers in Mathematics and Science Teaching*, vol. 24, no. 2, pp. 157–178, April 2005. [Online]. Available: http://www.editlib.org/p/5537

[13] 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.

[14] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari, "Learning computer science concepts with scratch," *Computer Science Education*, vol. 23, no. 3, pp. 239–264, 2013.

[15] J. H. Maloney, K. Peppler, Y. Kafai, M. Resnick, and N. Rusk, "Programming by choice: urban youth learning programming with scratch," in *ACM SIGCSE Bulletin*, vol. 40. ACM, 2008, pp. 367–371.

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

[17] "Prime Factorization in NetsBlox," https://www.youtube.com/watch?v=-qS7hGowQKQ, cited 2017 January 15.

[18] O. Astrachan and A. Briggs, "The CS principles project," *ACM Inroads*, vol. 3, no. 2, pp. 38–42, 2012.