Arduino Support to Quorum Programming Language Through Cloud Computing

Karan Sandam [#1], Sanjyot Satvi [*2], Oren D'cunha[#3]

[#] *Department of Information Technology, St. Francis Institute of Technology, Mumbai India*

[1] sandamkaran3030@student.sfit.ac.in

[2] sanjyot.satvi242@student.sfit.ac.in

***Abstract*— Quorum programming language started out as a project aimed towards simplifying syntax to reduce complexity in writing codes. Over the years it has evolved and now it supports audio processing, basic game development, LEGO robotics and more. As a result, quorum can now be used to teach programming in high schools and colleges like. The aim of this paper is to allow students to program Arduino boards using this language along with learning the fundamentals of programming. This would result in adding a new application area to quorum. Our system transpiles (converts) user submitted quorum code into logically equivalent Arduino code. It incorporates a total of three web services developed using NodeJS and Tornado, deployed separately as containerized applications over Heroku. Apart from this, MongoDB and Firebase Cloud Storage are used for storing user details and files respectively. This design allows us with an asynchronous and scalable system entirely on cloud. Thus users can program Arduino boards using Quorum right in their browser.**

Keywords: arduino; cloud transpiler; cloud compiler; docker; programming; quorum language; web-app

I. INTRODUCTION

Quorum is an evidence-based programming language, mainly designed to be accessible by everyone, in particular to visually-impaired students. It aims to simplify syntax while writing codes. Its applications include audio processing, basic game development etc. Academic curriculum laid out by Quo- rum can be used by teachers to teach programming concepts to students. [1] [2] As such we believe that the simplicity of quorum can be used to teach other technologies in collabo- ration while learning the language itself. This encouraged us to think about various possible areas which can be benefited from quorum.

Arduino boards are one of the most widely used microcontrollers in embedded systems. It offers inherent advantages like being inexpensive, supports cross platform operation, is open-source and has extensible software and hardware. All these advantages allow Arduino to be used extensively in personal and industrial projects. Hence, Arduino support for the Quorum programming language was decided upon.

Cloud computing generally means storing and accessing data or programs over the internet. More specifically delivering computer services can be called cloud computing. Recent years have seen availability of all different types of compilers that are deployed online. Such compilers are widely found in online programming contests, recruitment platforms and learning platforms alike. Basically these compilers work by spawning a compile command in a certain sandboxed environ- ment returning output to the user. Different organizations have incorporated different architectures for their needs keeping in mind the number of users that will be using the compiler, available resources, response time and many other factors. Deploying compilers over cloud increases the range of devices through which people can write programs while reducing the overhead of compiler maintenance. [10] [4]

Transpilation is a process of converting a code in one high level language to another high level language. Eg. Converting a code written in Python to Java. This process involves converting the original source code into some intermediary data structure which captures all the essential features of the input code followed by conversion of this intermediary data structure to target language. Abstract Syntax Trees(AST) are widely used as intermediary data structures due to their small size and easier interpretation than parse trees. [8]

In this paper we present a transpiler for converting Quorum into Arduino that functions within a browser. In section II we provide a detailed description of all the components that make up the system followed by section III which depicts how these components are related and work in cohesion. System implementation is elaborated in section IV. Section V reviews the performance of the system and finally we conclude the paper along with possible technical implementations for increasing the performance and features.
of section headings, document margins, column width, column spacing and other features.

II. SYSTEM DESCRIPTION

A. *Quorum Compiler*

The Quorum compiler is packed as a jar file that needs to be invoked every time to compile a Quorum file. Just like other programming language files, quorum files end with extension .quorum. Apart from the compiler's jar file, path to the standard library along with the user file and custom sensor libraries need to be passed in order to compile a quorum file.

Thus the command to compile looks like:

java -jar Quorum.jar -library path/to/standard/libraries - compile userfile.quorum LED.quorum IRED.quorum

Quorum files after userfile.quorum are all sensor based libraries written to help quorum identify new functions and classes added to support arduino.

B. *Quorum to Arduino Transpiler*

The quorum to Arduino transpiler consists of 4 main files viz. lexer, parser, listener and actual transpiler itself with additional handler file to facilitate the process of transpilation. The lexer reads the quorum files and generates a stream of tokens which the parser reads to generate a parse tree. Listener contains functions that are invoked when traversing the parse tree. These functions are modified so as to generate another tree that stores all the minimal information needed for transpilation. The transpiler file then reads this newly generated tree and converts them by combining values of the nodes with a predefined template of statements. Detailed explanation of the transpilation is provided below.

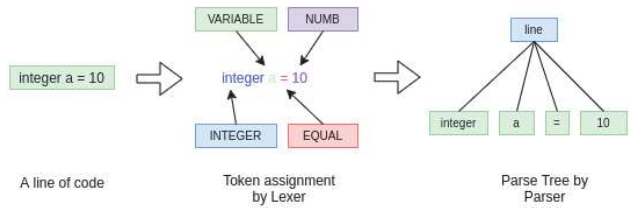


Fig. 1. Working of Lexer and Parser

The job of lexer is to identify words in the input stream and assign appropriate Tokens to them. After assigning tokens, the parser reads this sequence of tokens and generates a parse tree containing all the information of the input line of code.

Tokens and parser rules are specified in the grammar file. ANTLR4 reads this file and generates lexer, Parser and other necessary files.

Another important file that is generated is the listener. It is used to traverse the parse tree. While traversing the parse tree, a new tree is generated which serves as an intermediate data structure for code conversion. This new tree captures only essential information and is more compact than a parse tree.

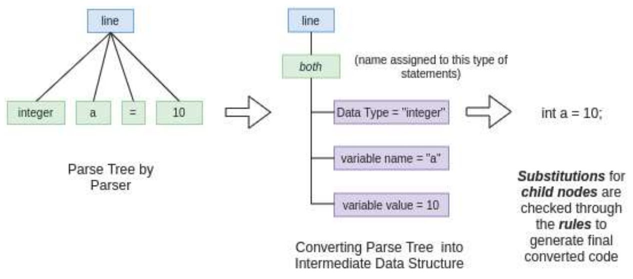

line
integer
a
=
10
Parse Tree by Parser
line
both
(name assigned to this type of statements)
Data Type = "integer"
variable name = "a"
variable value = 10
Converting Parse Tree into Intermediate Data Structure
int a = 10;
Substitutions for child nodes are checked through the rules to generate final converted code

Fig. 2. Conversion of Parse Tree into Transpiled code

For each of the syntax, keywords and possible statements of quorum, corresponding mapping is written in Arduino. The transpiler looks up these mappings while reading the new tree and applies appropriate substitution to all the nodes to generate the converted code.

C. *Docker Container*

Containers can be thought of as a package which contains the app itself bundled with all its dependencies and its execution environment. This allows greater portability, as these containers can be executed on any machine with a Docker engine. Containers are lightweight as compared to traditional virtual machines and are more secure than VMs due to tighter and more sandboxed environments. Our docker containers for both compiler and transpiler are derived from base ubuntu image. With the help of Dockerfile Python, Java and other dependencies are installed followed by copying of all the required source files while building the image.

D. *Heroku*

Heroku is a Platform-as-a-Service(PaaS) cloud service provider. With support for multiple languages, a wide range of add-ons, ease in deploying and scaling are some of the features that make Heroku a top choice among organizations and individual developers. Its free tier provides all the necessary requirements for deploying all our apps viz. Ui handler, compiler and transpiler. The UI handler is deployed simply as a webapp while the other two are containerized.

To ease the process of development and deployment, both transpiler and compiler are part of the same production pipeline. Corresponding github repositories are attached to them and environment variables are configured separately. Each app runs on a standard web dyno.

III. SYSTEM ARCHITECTURE

The system follows the Client-Server architecture and is heavily inspired by the designs presented in papers[4] and [5]. Login, UI and file storing functionalities are handled by the NodeJS server. For Signup and Login, user data is updated and fetched from the MongoDB database. Post signup/login user is presented with a text-editor with certain other features that aid him in writing code. After clicking on save, the file is saved to Firebase Cloud storage. Afterwards the user can click Compile. The NodeJS server then sends a request to the compiler server asking for the particular file to be compiled. The compile server downloads the file and compiles it. Results are sent back to the user. Similarly, after clicking Transpile, a request is made to the transpile server. It downloads and proceeds to transpilation. Errors if any or converted code are then sent back to calling NodeJS server. Both the compile and transpile servers delete the file on completion of the request. All the communication between the servers takes place in JSON.

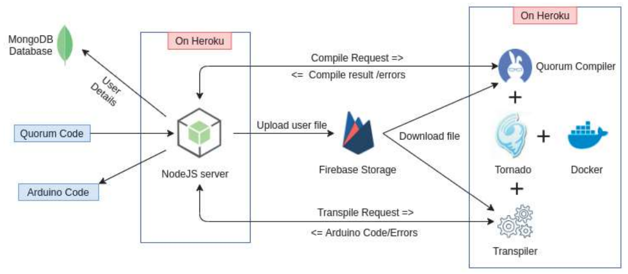


Fig. 3. System Diagram

IV. IMPLEMENTATION

A. *User Interface*

User interface consists of a general purpose text editor. An option to upload files is also provided. Before saving it is mandatory that the user sets the name of the file without any extension. Finally at the bottom there is a result area which shows errors or converted code accordingly.

B. *Quorum Compiler*

For each compile request a new subprocess is spawned. The terminal output of this process is checked and formatted to be sent back to the user. Sensitive information if any is removed from the response. If the compilation process is successful the compiler outputs Build Successful, else a list of errors along with line number and description is displayed.

C. *Quorum Transpiler*

Unlike the compiler, the transpiler is designed to only send results. If any errors are encountered they are simply ignored. This may lead to erroneous output. But the possibilities of errors arising during transpilation are quite low, since the compiler has already checked the files.

All the quorum files need to follow the below structure for successful and correct compilation.

- Names of sensor libraries that will be used followed by class declaration
- Inside the class begins with instantiation of libraries that will be used
  .
- Proceed with action Main() and configure pins using respective sensor methods.

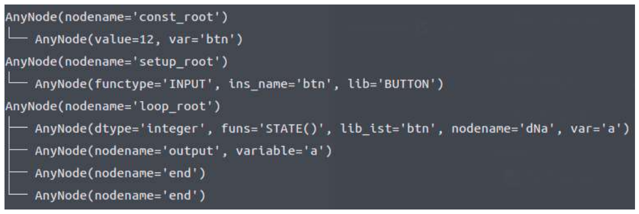
- End the method and class with keyboard “end”

Consider the below code snippet as an example.

```
Use BUTTON
class MyClass
.       BUTTON btn
        action Main()
                btn : pinBTN(12)
                integer a = btn : STATE()
                output a
        end
end
```

It uses the custom Button library. An instance of button is connected at pin 12, and its value is read into integer a.

For the above provided quorum code, the transpiler generates the following Intermediate Tree:



```
AnyNode(nodename='const_root')
└── AnyNode(value=12, var='btn')
AnyNode(nodename='setup_root')
└── AnyNode(functype='INPUT', ins_name='btn', lib='BUTTON')
AnyNode(nodename='loop_root')
├── AnyNode(dtype='integer', funs='STATE()', lib_ist='btn', nodename='dNa', var='a')
├── AnyNode(nodename='output', variable='a')
├── AnyNode(nodename='end')
└── AnyNode(nodename='end')
```

Fig. 4. Intermediate Tree Data Structure

Above figure has three different trees

- Tree that holds all constant values
- Tree that holds setup part
- Tree that holds loop part

The Constant tree has one node, which means in the input code one variable was identified which could be made a constant in arduino code. That variable has a value of 12 and its name is “btn”.

In the Setup tree there is a node which makes use of the library “Button”. Instance associated with it is “btn” and serves as an “input” to the arduino.

Similarly in the Loop tree there are 2 important nodes. The last two nodes represent end keyword mark end of Main function and class respectively. The first node is named as “dNa” which in code means Declaration and Assignment. Furthermore this node tells that there is a variable named “a” of integer datatype who is assigned a value which is returned from a function named STATE, associated with instance btn. Second node has its name set to output which tells it to print out a value. Variable that needs to be printed is a.

Internally, different types of node names are assigned as per various syntax available in quorum. Each line is represented as a node and different nodes hold different properties as per their needs. These nodename properties assigned to nodes help in identifying appropriate Quorum-to-Arduino statements. The statements are like templates in which the transpiler finds and fills values from input code.

For the mentioned quorum code, the transpiler produced following output:

```
const int btn = 12
 void setup ( ) {
\tpin Mode( btn , INPUT ) ;
\tSerial.begin( 9600 ) ;
}void loop ( ) {
\tint a =   digitalRead ( btn ) ;
\ t Serial.println( a ) ;
}
```

The transpiler prepends tabs according to the level of blocks. When using conditionals, the block level is increased by one, hence the number of tabs are incremented by 1 as compared to the previous line. When moving out of conditionals the block level is decreased and the number of tabs decremented by one. For- matting code on transpiler side, reduces overhead on UI side. Hence the result can be displayed directly.

V. RESULTS AND ANALYSIS

For measuring the performance of the system the compilation and transpilation time were measured on a local computer and system after it was deployed completely on cloud. Fifteen different files of varying complexity which involved all types of syntaxes which the transpiler is able to convert were used as input. The graphs have file number on x-axis and time taken for compilation/transpilation of that file on y-axis.

Dyno provided by Heroku for both these apps is Free type and is allocated a total of 512MB of RAM and CPU share of 1x each. The Dyno sleeps after 30 minutes of inactivity. Time taken to restart the dyno after it has been asleep is not considered here.

TABLE I

SUMMARY OF INPUT FILES

File	Lines	If..else blocks	Libraries Used	Libraries Instantiated	Level
1	**8**	1	0	0	Basic
2	**9**	0	1	1	Basic
3	**9**	0	1	1	Basic
4	**10**	0	1	1	Basic
5	**9**	0	1	1	Basic
6	**11**	1	0	0	Medium
7	**17**	2	2	2	Medium
8	**13**	1	1	1	Medium
9	**19**	1	1	2	Medium
10	**17**	1	2	2	Medium
11	**31**	2	3	4	Advanced
12	**30**	2	3	4	Advanced
13	**30**	2	2	4	Advanced
14	**36**	3	3	5	Advanced
15	**42**	4	3	5	Advanced

Table 1 gives a summary of the input files used for benchmarking the system, followed by plots. Each of the files was processed 5 times and corresponding times were noted. Average of these 5 times was taken and plotted.

The graphs for average of local and online compile time look similar in nature but differ in scale. While on the local computer, all files were compiled in less than 1 second, the same files took more than 6 seconds online. There seems to be no clear relation between time for compilation and complexity of lines but the resources allocated during compilation seem to be the factor determining time for compilation.

Similar to the compilation case, the nature of graphs for transpilation on local and online seem to close. Interestingly the time difference between them is also not huge. Thing to note is that as the number of lines, libraries and complexity of the code increases the time taken to transpile increases, which is evident from both the graphs. So, alinear relation between transpilation time and complexity of code exists.

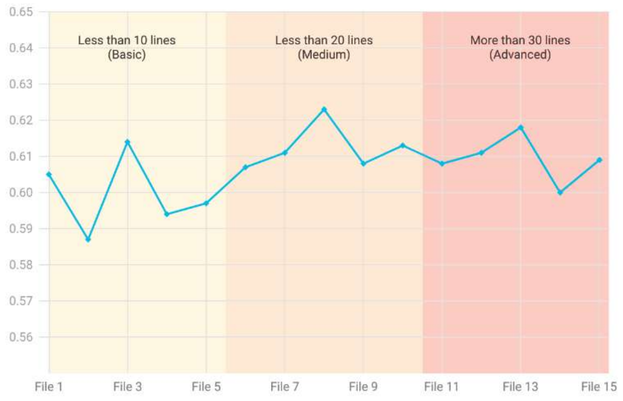


Fig. 5. Average Local Compile Time(in seconds)

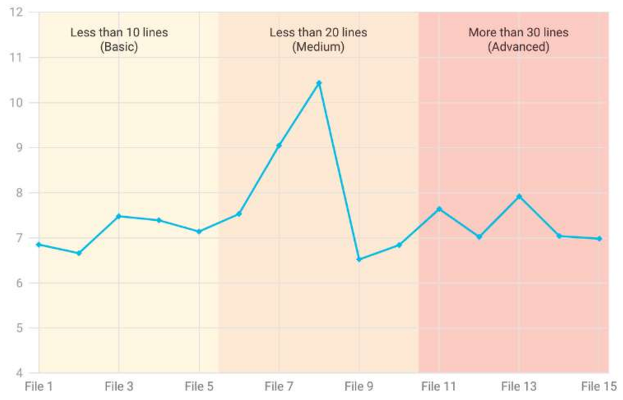


Fig. 6. Average Online Compile Time(in seconds)

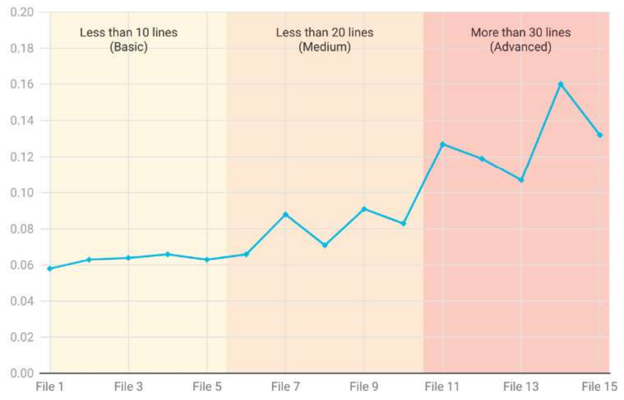


Fig.7. Average Local Transpile Time(in seconds)

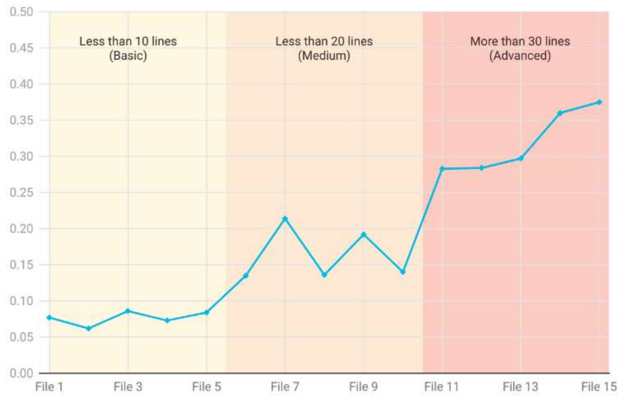


Fig. 8. Average Online Transpile Time(in seconds)

The difference in scales of graphs and linear relation between transpile time and complexity suggests that there are rooms for improvement of the system. Following things can be further incorporated to do so:

- Upgrading to better Dyno type on Heroku
- Switching to another Cloud Service Provider that provides more resources
- Making grammar rules more efficient to remove redundancies and eliminating rules that will never be used
- Writing the transpiler in some other language faster than python
- Modifying the way in which the transpiler is deployed over cloud. Other methods of deployment can be studied and implemented.

VI. CONCLUSION AND FUTURE WORK

We developed the system with an aim to bridge the gap between quorum and Arduino by allowing students to program Arduino boards. Deploying it over cloud allows everyone to access it and can be easily incorporated in teaching methodologies. Modular separation of storage, UI handler and actual compiler/transpiler allows a system that is easily scalable as per requirements. It also increases system reliability as any failures in the system are localized and contained within that module only. The limited flexibility in the quorum syntax that we have implemented can be increased to support more data types and control structures. At the same time transpiler efficiency can be increased and libraries to handle other sensors can be added. Responsiveness of the system in case of multiple users can be decreased by following a more asynchronous approach in which multiple operations like file fetching and formatting, compiling, etc belonging to different requests can be processed parallelly. Finally cloud security services can be placed throughout the system to prevent cloud software abuse.

REFERENCES

[1] *Programming Languages and Learning – Quorum Program- ming Language, Quorum Language. [Online]. Available: https://quorumlanguage.com/evidence.html. [Accessed:10- Jan- 2020].*

[2] *Andreas Stefik, Susanna Siebert, Melissa Stefik, Kim Slattery, ”An Empirical Comparison of the Accuracy Rates of Novices using the Quorum, Perl, and Randomo Programming Languages”, Online ISSN: ”Arduino - Introduction,” Arduino.cc. [Online]*Available:*https:https://www.arduino.cc/en/Main/FAQ#toc1*

[3] *”Arduino - Introduction,” Arduino.cc. [Online] Available: https:https://www.arduino.cc/en/Main/FAQ#toc1*

[4] *Ketan Kardile, Pradnya Punde, Kritika Pareek, ”A Survey on Online Cloud based Compiler”, International Journal of Innovative Research in Computer and Communication Engineering, Online ISSN: 2320-9801, 15 MAY 2017*

[5] *Taher Ahmed Ghaleb, ”Toward open-source compilers in a cloud-based environment: the need and current challenges”, Open Source Software Computing (OSSCOM) 2015 International Conference,pp. 1-6, 2015.*

[6] *T. Mohammed and M. Hamada, ”A cloud-based Java compiler for smart devices”, 15th International Conference on Information Technol- ogy Based Higher Education and Training (ITHET), 2016,Available: 10.1109/ithet.2016.7760742*

[7] *M. Varma Nandimandalam and E. Choi, ”Comparison of Open-Source PAAS Architectural Components”, Computer Science & Information Technology (CS & IT), 2016,Available: 10.5121/csit.2016.60206*

[8] *J. Kim and Y. Lee, ”A Study on Abstract Syntax Tree for Develop- ment of a JavaScript Compiler”, International Journal of Grid and Distributed Computing , vol. 11, no. 6, pp. 37-48, 2018. Available: http://dx.doi.org/*

[9] *M. Hossain, R. Rahman, M. Islam and M. Azam, ”A Study on Language Processing Policies in Compiler Design”, American Journal of Engineering Research (AJER) , vol. 8, no. 12, 20*