Declarative and Reactive Serverless Web Applications (Final Project Plan)

Team #2: Donald Pinckney

November 12, 2021

To the reader: apologies for the somewhat verbose project plan. You can safely skip Section 1 if needed.

1 Overview

1.1 Introduction

Serverless functions are a recently introduced cloud computing abstraction which allows developers to upload functions to a serverless platform (e.g., AWS Lambda), and later invoke them on-demand. The platform handles all work of dynamically allocating server resources for running the function. A potential application is writing *serverless Web applications*, in which progrmmers implement backend logic code in serverless functions, and frontend code invokes these serverless functions as needed over HTTP. Such an architecture can free programmers from managing server resources, increasing availability, and lowering costs.

Unfortunately, programming solely using serverless functions is currently difficult, because function composition is not well-supported for vanilla serverless functions. Baldini et al. [1] discuss that unless serverless platforms provide special support for composition, composition of functions is inherently flawed. The main platform that supports serverless function composition via additional primitives is AWS Lambda Step Functions which which allows programmers to specify a sequence of serverless functions. Unfortunately, the specification format is an ad-hoc language nearly unreadable to programmers, and it is unclear if it provides sufficiently flexibility.

1.2 An Alternative Approach

Rather than serverless platforms treating serverless functions alone as the primitive, and composition as an afterthought, I propose that serverless platforms should allow programmers to deploy function composition pipelines as one unit, and the serverless platform does the work of managing the individual component functions.

Second, by allowing the client and serverless platform to have rich information about the compositional structure, the client and server can decide where each function should be placed (e.g. in geographic server region or on the client itself) and with what CPU and RAM resources so as to minimize total latency and financial cost.

Specifically, the contributions of this work will be:

- A programming model which is convenient and expressive for programmers to write serverless Web applications in, and allows them to make easy use of serverless function composition.
- A new serverless platform which allows both functions and static composition structure data to be deployed in the format from the above programming model.
- An optimization algorithm which allows the serverless platform to optimize deployment and provisioning of serverless function resources to minimize latency and cost.

1.3 Background and Related Work

Arrows and Functional Reactive Programming A natural programming model for writing restricted forms of functional composition is Hughes's Arrow model [3], and has been used for writing interactive applications such as in the work on Arrowized Functional Reactive Programming [6]. In a related vein, Functional Reactive Programming has also been used as an abstraction for writing interactive Web applications (client-side only) [5]. A highly limited version of using Hughes's Arrows for serverless composition was explored in my previous OOPSLA 2019 paper on serverless computing [4].

In this project I aim to extend the compositions to cross the client-cloud boundary so that programmers don't need to separately manage their serverless functions from their frontend code. I believe that my programming model will make use of, or at least be heavily influence by, Arrowized Functional Reactive Programming.

Automatic Partitioning of Web Applications Prior work has explored automatic placement of code onto either the client or server [2], to minimize a given cost function. In many ways I am trying to explore a version of this line of work but for a serverless world, which requires a more restrictive programming model.

2 Goals

2.1 System Goals

While I have lots of thoughts for this work, for the purposes of a one month long final project I want to keep my goals attainable. My specific goals are:

1. Design a (prototype) programming model for writing function pipelines that can then be deployed to my serverless platform. This programming model will be an embedded DSL in Javascript, and will likely be a version of Hughes's Arrows / Arrowized FRP. As a bare minimum I will support function composition in this DSL, but if time allows I would try to add more features such as persistent storage and event reactivity.

2. Implement a (prototype) serverless platform which will take as input a program written in the embedded DSL, extract out the individual functions, and setup the functions to run in sequence to implement to correct semantics of the function composition. My project will focus on getting the semantics correct, rather than on handling high throughput, so for this (first) prototype I will *not* implement a serverless platform that runs the individual serverless functions on separate compute node, rather all will run on a single compute node, though likely in separate threads. I will run this on a single Khoury VDI machine.

3. Develop an optimization algorithm that can let the serverless platform decide if a single serverless function should be run in the cloud, or if the client itself should actually run it. Namely, functions should be placed in the cloud / client so as to minimize bytes transfered over the

Internet. Following the work in [2] this may be able to be cast as an instance of a graph min-cut problem. I plan to explore and develop this algorithm conceptually for my final project, but I am unsure that I will have the time to implement it fully as part of the serverless platform.

3 System Design

3.1 Assumptions

For this project, I want to focus on developing a) the programming model, and b) developing the corresponding serverless platform with correct semantics. Thus, I will assume:

1. Failures: I assume that failures of the serverless platform and / or network errors may occurr. In these cases the client code should react gracefully, and ideally resume working when connectivity is restored. That being said, I am unsure how much time I will have to get to implementing failure cases, but I would at least like to keep them in mind in terms of design.

2. Workload: I assume that there will be relatively light workload on the serverless platform, since I want to focus on a simpler and more correct implementation. However, I will allow for concurrent execution of serverless functions, since that is a really important aspect of semantics in practice.

3. Users: I assume that users are non-malicious with regards to serverless functions they deploy on the platform. In particular I don't want to spend time working on execution isolation, since this would require lots of engineering work and isn't very relevant to the semantics.

4. Supported Programs: To keep the project simple, I will only support small subset of a potentially more advanced programming model. In particular, I will only support pure functions without side effects or persistent state.

3.2 Modules of the System

3.2.1 Programming Model DSL

- The programming model for writing serverless function compositions will be a DSL embedded in Javascript.
- Serverless function compositions will be defined in the client-side Javascript code, and then deployed as-needed to the serverless platform.
- At a bare minimum the DSL will include the primitives `arr` (lift a JS function to a serverless function) and `f >>> g` (compose f followed by g). Depending on how exactly I handle function arguments there may be additional primitives, but that will be worked out later.
- A serverless function composition (e.g. `f >>> g >>> h`) will also have a serialization / deserialization implementation, so that these compositions can be sent over the network to the serverless platform.

3.2.2 Clients (Web application Frontends)

- The front-end code of web applications will aim to implement the bulk of their functionality inside serverless function compositions defined in the DSL outlined above. The front-end code will be able to attach *local* event listeners to know when the final step in a composition has executed, and can then appropriately updated the DOM, etc.
- The client will be responsible for deploying the serverless function composition to the serverless platform orchestrator. This will be automatically managed by a runtime library in the client so it doesn't get in the way of the Web application logic.
- The client will then communicate with the serverless platform orchestrator over the Internet to invoke a serverless function composition, and receive response notifications back.

3.2.3 Serverless Platform Orchestrator

- The serverless platform orchestrator manages a thread pool of serverless platform workers.

- The serverless platform orchestrator will wait for clients to connect and upload serverless function compositions.
- Upon receiving a serverless function composition from a client, the serverless platform orchestrator will setup the necessary data structures to ensure correct execution order of the functions in the composition.
- Upon receiving a request from a client to invoke a serverless platform composition, the serverless platform orchestrator will send the task for the first function in the composition to one of the workers in the thread pool.
- When a worker finishes executing a function, the orchestrator will update its data structures and then schedule the next function in the composition to run appropriately.
- *Optional: the serverless platform orchestrator may instead decide to ask the client to execute a serverless function task rather than a worker in the cloud, so as to allow for minimize bytes transfered over the network. The decision would be based on the analysis from the Orchestration Optimizer (see below). This is a stretch goal and might not be implemented!*
- When the final function in a composition completes, the orchestrator will send the result to the client.

3.2.4 Serverless Platform Worker

- A serverless platform worker thread waits for the orchestrator to assign it a task to execute, which consists of the function definition to run, and input arguments.
- A worker then executes the function, and when finished notifies the orchestrator of the result.

3.2.5 Orchestration Optimizer (stretch goal, might not be implemented)

- The orchestration optimizer will routinely ask the serverless platform orchestrator to collect dynamic trace information (e.g. # of bytes sent between functions).

- The orchestration optimizer will use this data to solve a minimization problem to decide whether each serverless function should be run in the cloud, or on the client. This minimization problem would be solved via a MAX-SMT encoding with a solver such as Z3.
- The orchestration optimizer will notify the serverless platform orchestrator of the new, optimal placements.

4 Evaluation

The evaluation will focus on correctness rather than performance. For all evaluation, the serverless platform would be hosted on a VDI machine.

4.1 Serverless Function Composition Unit Tests

The following serverless function compositions should be a) expressible in the Javascript DSL, and b) should be able to be deployed to the serverless platform and invoked, with correct behavior.

1. Execute functions A, B, C, and D in sequence, passing the return value of one to the parameter of the next (sequential DAG).
2. Execute function A, pass its return value to B and C, execute B and C in parallel, then execute D passing the return values of B and C (diamond DAG).

In addition, the above serverless function compositions should be able to handle concurrent invokations.

For running these serverless function composition unit tests, they could be deployed and invoked from any either from a browser or via command line, on any machine as long as it has connectivity to VDI.

4.2 Stretch goal: Unit Test of Optimization

If the orchestration optimizer is also completed, then the following function composition will be used to test it:

$$A \ggg B \ggg C$$

where:

- A takes as input only a few bytes
- A returns a value which is also only a few bytes
- A is constrained to run on the cloud (e.g. by a programmer-preference in the DSL)
- B takes as input the output of A (a few bytes)
- B then returns a value which is a large number of bytes (e.g. a couple of megabytes), perhaps an image, video, etc.
- B is able to run on either the cloud or the client
- C takes as input the output of B (a large amount of data)
- C is constrained to run on the client (e.g. so it can access the DOM)

By default, the orchestrator will assume that B should be run on the cloud. However, this is non-optimal since a large amount of data is being transferred over the network. The orchestration optimizer should be able to place B on the client instead.

4.3 Building Web Applications Using the Serverless Platform

To check how viable the programming model is for building actual Web applications, I will develop a small but fully-functional Web application which utilizes the serverless platform to perform some of the computation in the cloud.

An example Web application I expect I could build is: a Web application in which the user can type in a URL, and the Web app will fetch data from that URL, analyze it and see what the top 20 most frequent words are. The fetching and analysis should be able to be run in the cloud via my serverless platform.

For running the Web application, the Web application HTML / JS can loaded simply from a local file on any machine that has connectivity to VDI.

5 Project Timeline

1. Develop the Javascript DSL in which to express *basic* serverless function compositions. At this point, unit tests #1 and #2 should be expressible, as well as any arbitrary static DAG. However, these function composition programs will not yet be runnable. Estimated completion date: Nov 16.

2. Implement the serverless platform. At this point, the unit tests #1 and #2 should be able to be run by deploying them to the serverless platform, and invoking them. Estimated completion date: Nov 21.

3. Debugging time! Estimated completion date: Nov 24.

4. Implement the example Web application. Estimated completion date: Nov 26.

5. Nov 26: Start writing the final report & planning video presentation. Should take a few days.

6. Flex time to handle project taking longer than expected, or time to implement the orchestration optimizer.

References

[1] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. The serverless trilemma: Function composition for serverless computing. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2017.

[2] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 31–44, New York, NY, USA, 2007. Association for Computing Machinery.

[3] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, May 2000.

[4] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. Formal foundations of serverless computing. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[5] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for ajax applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, page 1–20, New York, NY, USA, 2009. Association for Computing Machinery.

[6] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, page 51–64, New York, NY, USA, 2002. Association for Computing Machinery.