Laminar: Dataflow Programming for Serverless IoT Applications

Tyler Ekaireb
University of California, Santa Barbara
Santa Barbara, CA, USA
tylerekaireb@ucsb.edu

Lukas Brand
HAW Landshut
Landshut, Germany
s-lbran1@haw-landshut.de

Nagarjun Avaraddy
University of California, Santa Barbara
Santa Barbara, CA, USA
nagarjun@ucsb.edu

Markus Mock
HAW Landshut
Landshut, Germany
mock@haw-landshut.de

Chandra Krintz
University of California, Santa Barbara
Santa Barbara, CA, USA
ckrintz@cs.ucsb.edu

Rich Wolski
University of California, Santa Barbara
Santa Barbara, CA, USA
rich@cs.ucsb.edu

ABSTRACT

Serverless computing has increased in popularity as a programming model for "Internet of Things" (IoT) applications that amalgamate IoT devices, edge-deployed computers and systems, and the cloud to interoperate. In this paper, we present Laminar – a dataflow program representation for distributed IoT application programming – and describe its implementation based on a network-transparent, event-driven, serverless computing infrastructure that uses append-only log storage to store all program state. We describe the initial implementation of Laminar, discuss some useful properties we obtained by leveraging log-based data structures and triggered computations of the underlying serverless runtime, and illustrate its performance and reliability characteristics using a set of benchmark applications.

KEYWORDS

IoT, serverless, dataflow programming

ACM Reference Format:
Tyler Ekaireb, Lukas Brand, Nagarjun Avaraddy, Markus Mock, Chandra Krintz, and Rich Wolski. 2023. Laminar: Dataflow Programming for Serverless IoT Applications. In *The 1st Workshop on SErverless Systems, Applications and MEthodologies (SESAME '23), May 8, 2023, Rome, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3592533.3592805

1 INTRODUCTION

Serverless computing has emerged as a model of application deployment and execution that separates infrastructure configuration concerns from application logic. In a "serverful" model, the application development and operations team (e.g., in a DevOps setting) must "program" both the application and the infrastructure (e.g., provision resources, establish an operational configuration) to meet the requirements of the application. In a serverless context, the infrastructure management is implemented via automation, leading to greater application portability and more reliable functionality.

Our work focuses on leveraging these benefits for "Internet of Things" (IoT) applications and deployments. In an IoT context, the infrastructure is more heterogeneous with respect to performance, reliability, capacities, and trustworthiness than in a datacenter or cloud computing context. Thus, a serverless approach to developing and deploying IoT applications should accelerate IoT innovation in the same way that it fosters innovation in the cloud.

Serverless IoT, however, requires a technological approach that extends the typical serverless platform in several fundamental ways. First, IoT is necessarily a distributed computing endeavor. Most serverless infrastructures do not include distributed computing abstractions. Secondly, IoT is multi-scale in terms of its infrastructure requirements. An end-to-end IoT application must be able to amalgamate functionality from resource-restricted devices, edge-based single-board computers, mobile devices, private clouds, and large-scale public clouds. Thirdly, the infrastructure available for an IoT deployment is typically bespoke both in terms of its architectural makeup and also in terms of its trust topology. Two different deployments of the same application may need to leverage vastly different resources with similarly heterogeneous trust characteristics.

Previous work has proposed a new serverless runtime system for IoT, called CSPOT [31], which is specifically designed to support multi-scale, distributed IoT applications as serverless microservices. Optionally, CSPOT can use CAPLets [6] to secure cross-service requests, and Ambience [2] for a set of common operating system abstractions, but it can also operate as a stand-alone, distributed language runtime system. Thus, using CSPOT, it is possible to achieve secure "write-once-run-anywhere" functionality across all devices, from the most resource restricted to the least, for IoT applications that are structured as communicating microservices.

In this work, we describe Laminar – a dataflow programming environment for developing these IoT applications. While CSPOT provides the fundamental runtime abstractions necessary to develop and deploy multi-scale IoT applications, writing and deploying these applications is laborious and error-prone. Thus, we propose a programming methodology that is designed to allow developers to code distributed IoT applications in a high-level functional language that uses CSPOT as a distributed, multi-scale, and serverless runtime system.

SESAME '23, May 8, 2023, Rome, Italy

ACM ISBN 979-8-4007-0185-6/23/05.
https://doi.org/10.1145/3592533.3592805

2 RELATED WORK

Laminar has several antecedents. Early functional languages, such as Id [19], FP [14], Haskell [15], ML [29], SISAL [10], and Lucid [30] all demonstrated the feasibility of using dataflow as a runtime system, either with hardware support such as i-structures [20] or purely in software. Modern functional languages (including Haskell), such as Miranda [28] and Purescript [24], can also use dataflow as a runtime system, although they often compile to a more imperative language variant, such as Javascript [13]. Laminar's visual programming component is inspired by languages such as Keysight VEE [1], KNIME [7], and Node-RED [8]. Laminar is distinct from these antecedents in that it implements a dataflow runtime using a log-based, append-only storage model and triggered execution. Log-based storage provides Laminar with single-assignment variable semantics, and triggered execution is the basis for firing a node.

While our work targets the CSPOT [31] runtime system, it is possible to implement log-based storage and triggered execution using other systems for multi-scale IoT such as CloudPath [17], tinyFaaS [23], AWS Greengrass [4], and Azure IoT Edge [18]. Finally, the use-case we envision for Laminar is one in which the vast heterogeneity and highly varying reliability of IoT devices and software makes unification with a single language that has good robustness properties attractive. It should be possible, however, to use Laminar with cloud-based or heavier-weight FaaS systems as well, including AWS Lambda [5], Azure Functions [9], Google Functions [12], OpenWhisk [25], and OpenFaaS [22].

3 CSPOT, DATAFLOW, AND LAMINAR

The Laminar prototype uses CSPOT as its language runtime system, leaving CSPOT to manage the security and device interfaces. Using CSPOT alone, the application must incorporate the use of CAPLets as an application-level protocol. In this work, we do not extend Laminar to incorporate CAPLets capabilities in this way. Thus, the current prototype relies on Ambience (and its use of CAPLets), and Ambience manifests to implement security. However, based on experimentation, it is our understanding that the CSPOT-Ambience integration is not yet ready for full-scale usage by non-expert users [3]. We plan to extend Laminar to permit the use of CAPLets without an Ambience dependency as part of our future work.

CSPOT implements an event-driven serverless computing model using an append-only log storage abstraction for all program state. The CSPOT developer writes stateless event handlers that can only be triggered when a data item is appended to some CSPOT log (called a *WooF*. CSPOT logs (WooFs) are persistent, so all program state changes (up to a configurable history length) are saved. This log-based approach makes it possible to write highly reliable applications [16], since the application's state at the time of a failure is recoverable from the logs. Also, because computation cannot be triggered without first appending a datum to a log, the runtime system tracks causal dependencies as a side effect, aiding debugging. Finally, the program logs are named via URNs. If an append operation refers to a locally hosted log, the runtime uses a fast path to complete the append transaction. Otherwise, it serializes the data for network transmission to a remote log.

The CSPOT API consists of a `Create` function to create a WooF, a `Put` function to append a datum to a WooF (and trigger a handler function if specified), a `Get` function to extract a datum from a WooF with a specific log sequence number, and a `Delete` function to remove a WooF. Because the semantics are a combination of append-only state update and asynchronous event-driven programming, writing CSPOT programs is notoriously challenging. Handlers execute concurrently, the only transaction being the assignment of a sequence number to a log append event once the append is stored. Because the storage abstraction is not random access, handlers (which do not persist state) must often scan from the current tail of the log to re-construct the internal state of their logic on each invocation. Further, the only synchronization possible between handlers is via the tail of some log. Thus, naïve versions of CSPOT programs often scan their logs repeatedly and "spin" polling the log tails when synchronizing.

However, from the perspective of an applicative functional language, CSPOT's logs and append-only semantics correspond to single-assignment variables. Each append to a log is associated with a unique sequence number, and the tuple of log name and sequence number occurs only once in a CSPOT program. Thus, an implementation of dataflow in which CSPOT handlers are stateless, and all program state is stored in CSPOT logs, layers applicative programming semantics atop CSPOT's distributed log-based runtime.

This correspondence provides three immediate benefits. First, dataflow programs can be represented graphically. This benefit has proven invaluable in developing Laminar itself (cf. Subsection 4.5). Secondly, in an IoT setting, the deployment architecture often varies substantially between different deployments of the same service mesh. The one-to-one correspondence between dataflow and applicative functional programming language semantics make it possible to "program" each deployment (of the same service mesh) using Laminar. Lastly, and as a result, it will be possible to use Laminar as an intermediate representation for higher-level applicative functional languages such as Scala [21] and ML [29]. In this way, a user of Laminar can develop an IoT service mesh and its deployment using either a graphical representation or a high-level applicative functional language and execute that service mesh using CSPOT as its runtime.

Finally, we chose CSPOT as a serverless target for Laminar because it incorporates an append-only storage abstraction, and because it is portable to microcontroller devices, edge computers, and clouds (it uses Linux containers on systems where it does not run natively). However, it should be possible to emulate CSPOT's functionality using cloud or edge-based FaaS systems and one or more storage services that can implement a persistent log. Thus we present Laminar as a way to develop distributed dataflow programs using serverless technologies as a runtime for IoT.

4 LAMINAR IMPLEMENTATION AND PROGRAMMING CONSTRUCTS

At an abstract level, a Laminar program comprises the following elements:

- **Sources**, which are external computations that introduce data into a Laminar program. These include sensor readings (in an IoT context), database reads, remote API calls,

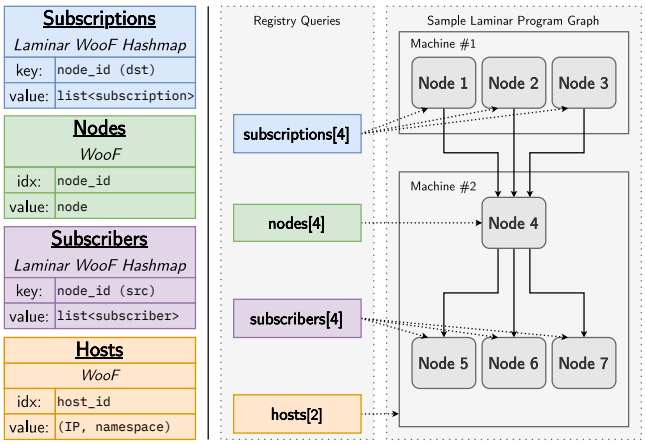

Figure 1: Data structures associated with a Laminar registry. The example indicates which program elements are referenced when each data structure is queried for information regarding node#4.

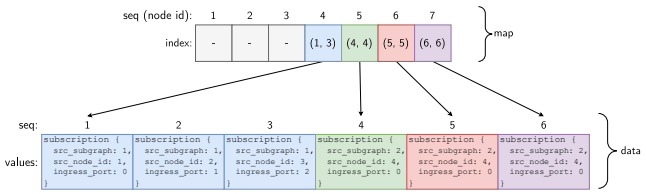

Figure 2: Detailed view of the Subscriptions structure of the sample Laminar program shown in Figure 1. Each consumer node identifier is mapped to a list of subscriptions. Each subscription associates a consumer node input port with the producer node's output.

or arbitrary program functions from a program capable of exercising Laminar's API.

- **Nodes**, which perform operations on data.
- **Edges**, which express data flow between nodes.
- **Sinks**, which transmit data outside of a Laminar program. These include database writes, remote API calls, or arbitrary program functions which consume data from Laminar 's API.

A node can have an arbitrary number of inputs and outputs (represented by unique "ports" on each node). One directed edge links each output to each input. Output ports can have fan-out, but input ports receive data from a unique output. Note that sources are a special case of nodes without incoming edges, as they pull external data into a Laminar program, and sinks are a special case of nodes without outgoing edges, as they send data outside of a Laminar program. Input and output ports are implemented using WooFs, which act as queues of data items passing between nodes. A Laminar edge is represented as a "subscription" by a node for an input to receive the datum from a specific output of a predecessor node.

An *advent* event is triggered for every input that arrives at a node (that is, every time a data item is appended to the WooF representing the *advent* event log of a node). The subscription handler for the node checks if all inputs have arrived. If they have not, the handler exits. Otherwise, when the last input has arrived, the handler executes the computation associated with the node and populates the output ports for the node. When a datum is appended to the WooF implementing an output port, an *advent* event is sent to each node subscribing to the output.

Note that Laminar creates all required CSPOT resources (creates the node WooFs, installs the handlers, etc.) statically before program execution begins. In this way, a Laminar application is "compiled" statically to use the CSPOT runtime abstractions. This runtime architecture can be categorized into two sections: a registry that is global to the program and node-specific data structures. The global registry stores indices of the program nodes, subscriptions, and subscribers. This information is encoded to be stored in logs (each implemented as a CSPOT WooF).

4.1 Global Registry

The Laminar program registry consists of four data structures that track *Nodes*, *Subscriptions*, *Subscribers*, and *Hosts* respectively, as shown in Figure 1. The *Nodes* WooF stores the host ID to identify the machine a node is running on (CSPOT uses RPC when a `Put` or a `Get` operation is non-local) and an opcode that is used to dispatch the node's computation when all inputs are available.

The *Subscriptions* data structure maps node identifiers to the identifiers of its incoming edge nodes, enabling fast access for a consumer when determining whether or not all input data is available. Subscription information is stored as a hashmap implemented using two WooFs, as shown in Figure 2. The *data* WooF stores all subscriptions for each node contiguously in the sequence number space of the the WooF. Each element of the *data* WooF contains a C++ `subscription` structure that associates an input port of the consumer node with the output of a producer node. The *map* WooF maps a range of sequence numbers in the data WooF to the node to which those subscriptions pertain. Each node's identifier is used as an index into the *map* WooF (CSPOT's `Get` function takes a sequence number indicating the specific log entry to return).

For the example in Figure 2, sequence number 4 in the *map* WooF (which correspond to node #4 in Figure 1) contains $(1, 3)$, indicating that *node*#4's input port edges can be found by scanning the `data` WooF sequentially from sequence numbers 1 through 3. The input ports for *node*#5 begin at sequence number 4 in the *data* WooF, and so on.

The *Subscribers* WooF hashmap uses the same two-level log encoding to access *subscriber* structures that represent the relationship of the output ports of each node to the nodes that consume that node's outputs. The *Hosts* WooF stores the CSPOT information necessary to locate a WooF remotely (e.g., the host network address and path to the CSPOT container storage). The Laminar runtime uses this WooF for host discovery and dataflow across hosts.

4.2 Subgraphs

Nodes can be grouped to implement scoping and modular composition. A subgraph represents a functional "subprogram" that acts as a node in any Laminar program in which it is embedded. That is, no node within a subgraph fires until all of the inputs to the

subgraph are available, and no outputs from the subgraph can be consumed as inputs by other nodes or subgraphs until all subgraph outputs have been produced.

In Laminar, a subgraph implements scope for identifiers and states. Multiple Laminar programs can be developed in isolation and run together on the same cluster by grouping nodes into subgraphs. Moreover, subgraphs can encapsulate implementation details and provide communication interfaces between programs without exposing graph internals (i.e., to support modular design). Finally, Laminar uses subgraphs to implement iteration (cf. Subsection 4.4).

4.3 Integrating Data Sources

A typical IoT application will need to consume data from various sources. Data is produced by *Source* nodes, which may be defined within the Laminar high-level language or by a C/C++ program on an IoT device. Typically, some native code is necessary in order to access registers with sensor values on a particular embedded device. Laminar provides an API and a library to integrate existing C and C++ programs as Laminar sources. A user can use this API to ingress data to a Laminar program as the output of a *Source*, e.g., `laminar::emit([SOURCE], [DATA])`.

In principle, data can be passed to a Laminar program from nearly any source. However, the current Laminar prototype supports arbitrary C and C++ functions.

4.4 Laminar Structured Programming Constructs

In addition to traditional dataflow, Laminar supports several constructs that facilitate the development of more complex programs with control flow and iteration.

Conditionals. Laminar supports two forms of conditional statements: `SELECT` and `FILTER`. A `SELECT` node uses its first port as a selector, whose value is used to index the remaining ports to be forwarded as output. The second construct, `FILTER`, accepts a boolean value on its first port, determining whether or not the data on the second port is forwarded.

Iteration. The asynchronous and event-driven nature of Laminar naturally supports powerful parallelism. Each node can operate on data independently so that Laminar can exploit the inherent parallelism of many applications. However, some algorithms are necessarily iterative, so Laminar supports iterative looping constructs. One such loop is inspired by the IF1 specification [27]. It consists of four subgraphs: **Initialization** runs once before the loop executes; **Body** performs the computation that typically resides within a traditional loop body; **Test** calculates a boolean value that determines whether or not to repeat the body, and finally, **Result** runs once after the loop finishes.

This looping construct and its variants can be used to achieve the same functionality as the typical `for` or `while` loop of traditional imperative programming languages; an example is shown in Figure 3. The node diagram shows an implementation of Heron's Methon for estimating square roots. An equivalent C++ program is provided in Listing 4.

4.5 Debugging

Distributed applications, particularly ones that use concurrent append-only persistence semantics, are notoriously difficult to debug. One advantage of dataflow, in this regard, is that there is a one-to-one correspondence between program elements (nodes and edges) and the components of a directed, acyclic graph (DAG).

We exploit this relationship as a debugging aid. Laminar includes the ability to autogenerate a graphical DAG representation of a Laminar program using DOT [11] to encode the graph, which can be rendered in various formats. Figure 8 shows an example of an autogenerated diagram for the quadratic formula $(-b + \sqrt{b^2 - 4ac}/2a)$ implemented as a single Laminar subgraph. This capability has proved invaluable in developing and debugging tests and benchmarking the applications used this paper (cf. Section 5).

Since every operation performed by Laminar is stored in an append-only log, extensive information for debugging is available. Dependencies between nodes are explicit; thus, a causal ordering can be reconstructed to analyze a program after execution. The nature of append-only logs and preservation of causality could be used to create powerful debugging tools which allow the program to pause, roll back, and replay execution, even when distributed across devices.

Finally, the combination of functional language semantics and a high-level, device-independent program representation makes it possible to use Laminar as a flexible development environment for IoT. Programs can be developed locally and then deployed in a variety of distributed configurations by placing the WooFs on different resources in the network. Furthermore, functional composition provides a number of optimization opportunities that can be exploited in conjunction with this deployment flexibility.

4.6 Fault Tolerance

Laminar runtime supports recovery from network partitions by retrying failed events with exponential backoff. Laminar is configured to retry at most 10 times with exponentially increased duration (capped at 32s) between retries. The output handler maintains an in-memory queue of failed events for retry. The handler that generates the node output ensures fault tolerance by generating *advent* events at all subscriber nodes with retries during network partition.

5 EVALUATION

We believe that Laminar affords the user with a high-level, flexible, and convenient programming modality for distributed IoT programs that must run in tiered deployments (device, edge, and cloud). Our goal in this work is to understand the performance and reliability properties that accompany these potential benefits.

For the following experiments, we run Laminar in containers hosted in virtual machines on a campus private cloud. The virtual machines have 4 dedicated cores that share 16 GB of memory. The underlying processor is an X86 architecture machine with a 2.8 GHz clock speed, the containers used by CSPOT are provisioned by Docker CE, and the Linux distribution is CentOS 7.4.

To illustrate the performance profile associated with Laminar, we have implemented a simple "online" linear regression program in C++, and two different Laminar encodings. Figure 5 shows the

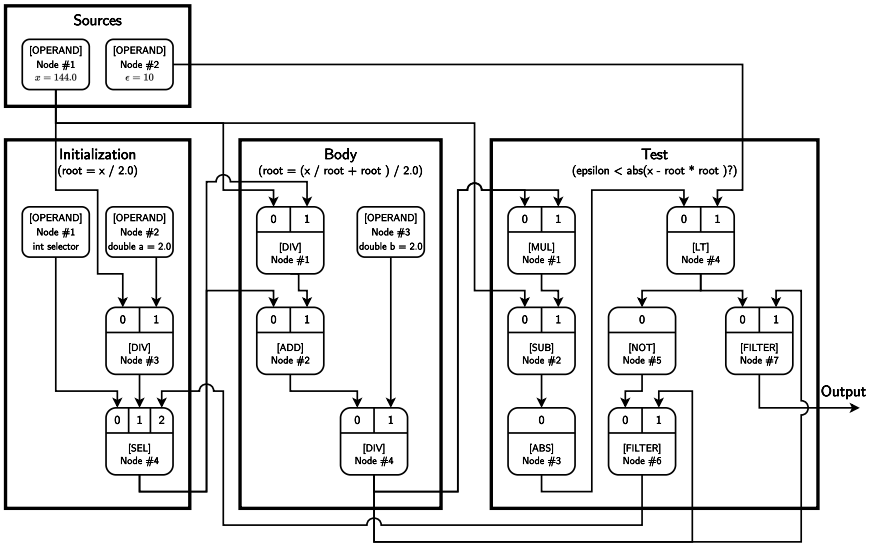

Figure 3: Node diagram of a Laminar program which implements Heron's Method to estimate square roots.

```cpp
#include <cmath>

double calc_sqrt(double x, double eps) {
    double root = x / 2.0;
    do {
        root = (x / root + root) / 2.0;
        cout << root << endl;
    } while (abs(x - root * root) >= eps);
    return root;
}
```

Figure 4: Heron's Method for square root estimation implemented in C++.

core C++ algorithm that performs the update to the regression coefficients when each new data item (i.e., x-y pair) is input.

In the first Laminar representation, we encode the regression as a single Laminar node embedded in a single Laminar subgraph (termed *Uninode*). Comparing this encoding to the native C++ implementation shows the minimum overhead incurred by Laminar when using CSPOT as a local runtime. The second implementation (termed *Multinode*) represents the linear regression algorithm as a DAG of 43 Laminar nodes. The CSPOT runtime will implement the computations concurrently (up to the number of cores available to the container). However, in this representation, the nodes represent computations at the finest level of granularity (i.e., with the maximal Laminar and CSPOT overhead). The third implementation is a deployment of the second implementation across machines that results from changing the host identifiers in the *Host* WooF (i.e., one that does not require a recoding, but rather, a redeployment). The nodes responsible for slope and intercept calculation are placed in another machine. This application configuration allows the usage of more computation and efficient pipelining of the dataflow.

As expected, the *Uninode* performance (which incurs the minimal CSPOT overhead) achieves the lowest latency and the highest throughput. The average observed Laminar latency for this benchmark is comparable to the CSPOT `NULL` function dispatch latency reported in [31], leading us to believe that Laminar is performant.

```cpp
void update(double new_x, double new_y) {
    double dt = 1e-2;    // Time step (delta t)
    double T = 5e-2;     // Time constant
    double decay_factor = exp(-dt / T);

    // Decay values
    num *= decay_factor;
    x *= decay_factor;
    y *= decay_factor;
    xx *= decay_factor;
    xy *= decay_factor;

    // Add new datapoint
    num += 1;
    x += new_x;
    y += new_y;
    xx += new_x * new_x;
    xy += new_x * new_y;

    // Calculate determinant and new slope / intercept
    double det = num * xx - pow(x, 2);
    if (det > 1e-10) {
        intercept = (xx * y - xy * x) / det;
        slope = (xy * num - x * y) / det;
    }
}
```

Figure 5: Online linear regression coefficient update routine

In the *Multinode* encoding, *every* operation shown in Figure 5 is executed as a separate Laminar node. This maximal overhead case clearly decomposes the program to a degree in which the overheads for executing each operation dominate the execution latency. The optimal decomposition that balances the cost of sequential overhead with the benefit of concurrency is between these extremes. As part of our future work, we plan to investigate automatic program partitioning and scheduling approaches (such as those discussed in [26]) that seek this balance.

Note that the *Multimachine* deployment of the *Multinode* encoding shows better throughput than for the same encoding on a single machine. This difference is due to the default CSPOT configuration for the 2.0 release and the number of available cores. CSPOT, by

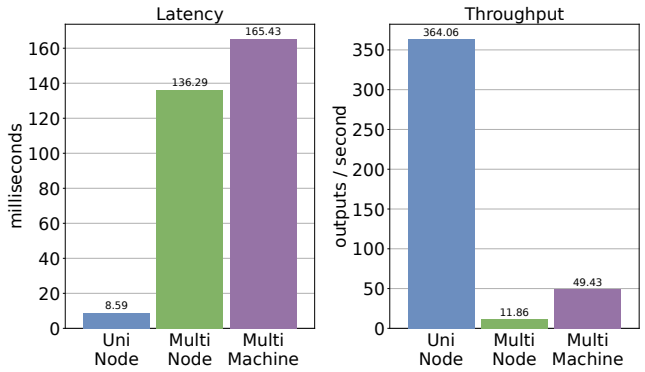

Figure 6: Benchmark result of linear regression example

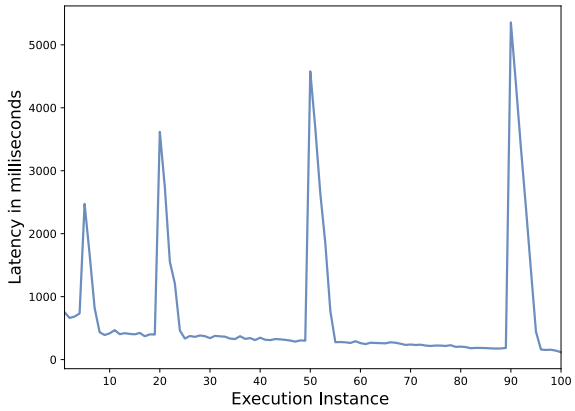

Figure 7: Benchmark result for Latency of Execution with Network Partition and Duplicates

default, sets a limit of 15 on the number of concurrently active handlers, and does not direct Docker to use all of the available cores. Thus, the *Multinode* execution on a single machine is over-sequentialized and experiences resource contention, but when two machines are used, the application achieves a modicum of pipeline parallelism. We plan a thorough performance analysis of the interaction between CSPOT and Laminar, and we anticipate potential performance improvements will result compared to those observed in this early prototype.

Since Laminar stores all program state in persistent CSPOT WooFs, it should be possible in principle to halt the Laminar program at any point in its execution and to resume it from where it halted, even when distributed across an arbitrary number of machines. That is, Laminar is continually checkpointing the program state through its use of persistent logs, so resumption should be possible from the last logged state.

We have yet to explore the full crash consistency and recovery properties of Laminar, however, we have begun to study its resistance to network partitioning. CSPOT appends an element to a WooF and assigns it a sequence number as a transaction.

Thus, two possible failure conditions exist when appending a datum to a WooF using RPC across a network interface. Either the append fails because the network could not fully transmit the RPC request, or it succeeds, but the response carrying the sequence number back to the requester as a return value is lost. The former case is addressed by a timeout and retry mechanism, while the latter requires duplicate suppression.

To demonstrate the robustness of Laminar in the face of network partitions, we set up the previously described *Multinode* Linear Regression application across two machines, then induce network partitions for varying time periods, measure the response time, and verify the correctness of the application execution. We run the *Multinode* Linear Regression with 100 inputs and feed the inputs to the nodes at 1 input per second. We introduce a network partition at inputs 5, 20, 50 and 90 for 2, 3, 4 and 5 seconds, respectively. Figure 7 shows the latency of execution instance for each input. In this experiment, we also simulate the loss of a response by having Laminar issue each CSPOT `Put` function twice, back-to-back, to create duplicates (as if the response were lost and a retry succeeded).

The results show that the system is fault-tolerant and resumes execution without severely impacting the latency of subsequent iterations. Laminar ensures the execution is sequential even though events can arrive at the destination node after the network is back up in random order due to randomness in the backoff times between retries. The result shows that the system takes 0.5s on average after the network recovery to continue processing inputs. The recovery time after a network partition is a function of the input data rate, proportional to the number of handlers trying to send the events and the network downtime, leading to various back-off times for retry. In addition, recovery time is affected by the overall application design. Unsurprisingly, we found that the resulting latency graph looks identical in the scenario when we do not simulate the duplicate suppression mechanism, which is implemented by Laminar checking the internally generated sequence number and discarding data with the identical execution number, i.e., a very simple operation.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we described Laminar – a dataflow program representation for distributed IoT application programming, and evaluated its initial implementation using CSPOT, a distributed serverless infrastructure for IoT. We showed that the high-level dataflow semantics of Laminar are performant, increase programmer convenience, and improve application reliability thereby simplifying the development of distributed heterogeneous IoT systems. Our evaluation of Laminar demonstrates the flexibility of its programming model, the ease with which it can be deployed across devices, and resilience in the face of network failures.

Currently, Laminar exploits append-only logs to checkpoint every operation and preserve causal dependencies. In future work, Laminar will be augmented to utilize this information for pause, rollback, and replay functionality to debug distributed applications and increase fault tolerance. It will also serve as the compile target for a high-level language, offering users even greater programming ease. In addition, we plan on performing more detailed studies to understand better the possible performance optimizations and exact reliability properties the underlying serverless log-based infrastructure affords.

ACKNOWLEDGMENTS

This work has been supported in part by NSF awards CNS-2107101, CNS-1703560, and ACI-1541215.

REFERENCES

[1] 2023. Keysight VEE. https://www.keysight.com/us/en/home.html.
[2] ambience 2023. The Ambience Operating System. https://github.com/MAYHEM-Lab/ambience.
[3] Ambiencepersonal 2022. Personal Communication with Ambience Developers. https://github.com/FatihBAKIR, December, 2022.
[4] AWS Greengrass 2019. AWS Greengrass. https://aws.amazon.com/greengrass/ [Online; accessed 12-Sep-2019].
[5] AWS Lambda 2023. AWS Lambda – Serverless Compute - Amazon Web Services. https://aws.amazon.com/lambda/.
[6] Fatih Bakir, Chandra Krintz, and Rich Wolski. 2021. CAPLets: Resource aware, capability-based access control for iot. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 106–120.
[7] Michael R Berthold, Nicolas Cebron, Fabian Dill, Thomas R Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Kilian Thiel, and Bernd Wiswedel. 2009. KNIME-the Konstanz information miner: version 2.0 and beyond. *AcM SIGKDD explorations Newsletter* 11, 1 (2009), 26–31.
[8] Michael Blackstock and Rodger Lea. 2014. Toward a distributed data flow platform for the web of things (distributed node-red). In *Proceedings of the 5th International Workshop on Web of Things*. 34–39.
[9] craigshoemaker. 2023. Azure Functions Overview. https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview.
[10] John T Feo, David C Cann, and Rodney R Oldehoeft. 1990. A report on the Sisal language project. *J. Parallel and Distrib. Comput.* 10, 4 (1990), 349–366.
[11] Emden R. Gansner, Eleftherios Koutsofios, and Stephen North. 2006. Drawing graphs with dot.
[12] googlefuncs 2021. Cloud Functions. https://cloud.google.com/functions.
[13] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The essence of JavaScript. In *ECOOP 2010–Object-Oriented Programming: 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings 24*. Springer, 126–150.
[14] John Guttag, James Horning, and John Williams. 1981. FP with data abstraction and strong typing. In *Proceedings of the 1981 conference on Functional programming languages and computer architecture*. 11–24.
[15] Paul Hudak and Joseph H Fasel. 1992. A gentle introduction to Haskell. *ACM Sigplan Notices* 27, 5 (1992), 1–52.
[16] Wei-Tsung Lin, Fatih Bakir, Chandra Krintz, Rich Wolski, and Markus Mock. 2019. Data repair for distributed, event-based IoT applications. In *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*. 139–150.
[17] Seyed Hossein Mortazavi, Mohammad Salehe, Carolina Simoes Gomes, Caleb Phillips, and Eyal De Lara. 2017. Cloudpath: A multi-tier cloud computing framework. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. 1–13.
[18] MSIoT 2023. IoT Edge | Microsoft Azure. https://azure.microsoft.com/en-us/services/iot-edge/.
[19] Rishiyur S Nikhil. 1993. The parallel programming language Id and its compilation for parallel machines. *International Journal of High Speed Computing* 5, 02 (1993), 171–223.
[20] Rishiyur S Nikhil and Keshav K Pingali. 1989. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 11, 4 (1989), 598–632.
[21] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. An overview of the Scala programming language. (2004).
[22] OpenFaaS 2020. OpenFaaS. https://www.openfaas.com [Online; accessed 1-Sep-2020].
[23] Tobias Pfandzelter and David Bermbach. 2020. tinyfaas: A lightweight faas platform for edge environments. In *2020 IEEE International Conference on Fog Computing (ICFC)*. IEEE, 17–24.
[24] purescript 2023. The Purescript Language. https://www.purescript.org.
[25] Marek Sadowski, Lennart Frantzell, and Sadowski. 2020. Apache OpenWhisk–Open Source Project. *Serverless Swift: Apache OpenWhisk for iOS developers* (2020), 37–57.
[26] Vivek Sarkar. 1987. *Partitioning and scheduling parallel programs for execution on multiprocessors*. Stanford University.
[27] Stephen Skedzielewski and John Glauert. 1985. IF1 An Intermediate Form for Applicative Languages. *Lawrence Livermore National Laboratory Manual M-170, Livermore, CA* (1985).
[28] David A Turner. 1985. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture: Nancy, France, September 16–19, 1985*. Springer, 1–16.
[29] Jeffrey D Ullman. 1998. *Elements of ML programming (ML97 ed.)*. Prentice-Hall, Inc.
[30] William W Wadge, Edward A Ashcroft, et al. 1985. *Lucid, the dataflow programming language*. Vol. 303. Academic Press London.
[31] Rich Wolski, Chandra Krintz, Fatih Bakir, Gareth George, and Wei-Tsung Lin. 2019. CSPOT: portable, multi-scale functions-as-a-service for IoT. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*. ACM, 236–249. https://doi.org/10.1145/3318216.3363314

A AUTOGENERATED DATAFLOW REPRESENTATION

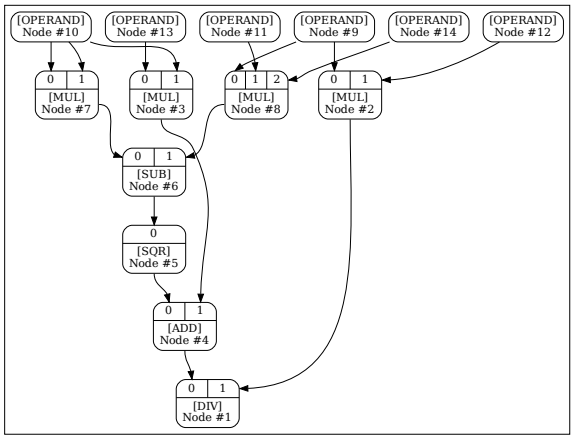

Figure 8: Laminar Autogenerated Graphical Dataflow Representation of the Quadratic Formula. The inputs to the formula a, b, c are represented by node#9, node#10, and node#11, respectively. Node#12 transmits the constant 4, node#13 transmits the constant -1, and node#14 transmits the constant 2. Each operation is marked, and numbers at the top of each node enumerate input ports. Output ports can connect to multiple input ports, implementing fan-out.