CSPOT: A Serverless Platform of Things

University of California, Santa Barbara Computer Science Technical report Number 2018-01 *

Rich Wolski
Computer Science Department
University of California, Santa Barbara

Chandra Krintz
Computer Science Department
University of California, Santa Barbara

Wei-tsung Lin
Computer Science Department
University of California, Santa Barbara

Functions-as-a-Service (FaaS) has emerged as a new, scalable technology for implementing cloud-based web services. As an event-driven programming paradigm, FaaS systems are also gaining in popularity as a technology for implementing the "back end" of Internet of Things (IoT) applications.

In this paper, we describe CSPOT – a portable, multi-scale FaaS system for implementing IoT applications. CSPOT is designed to run at all scales employed by cloud-based IoT applications. The current implementation is available for micro-Linux devices, edge computing devices, moderately-sized private clouds, and public clouds. CSPOT features include a lightweight runtime system that automatically logs causal relationships between FaaS function invocations, and a portable, append-only storage abstraction which also serves as a communication substrate for functions.

1 Introduction

Functions-as-a-Service (FaaS) [13, 11, 15] has become a popular cloud-native service for developers to use to build and deploy scalable web services. FaaS addresses two challenges facing developers and system administrators wishing to build and maintain scalable web services.

First, it automatically controls the execution of computations in response to changes in web service request load (e.g. autoscaling). Programmers write "functions" – short-running stateless computations – that are either invoked by other functions or automatically triggered by "events" that the cloud presents to the web service. For example, it is possible to register a function to be triggered each time a new web service request arrives, when ever a record is accesses in a scalable data base, when ever a queue entry is added to a scalable queuing service, etc. That is, than writing the web service as a procedural, iterative program, a FaaS developer programs using an *event-based* programming style. All services that the web services needs to use from the hosting cloud are accessed via triggered events as are all computations that the service must perform.

For this reason, FaaS functionality is often termed "serverless" computing since deployment (which requires the provisioning and maintenance of "servers") is handled automatically by the FaaS service. FaaS programmers need only upload and register their functions. These functions can only compute locally (and for short periods that are capped by the cloud provider) and access cloud services through APIs and language-specific interfaces (i.e. SDKs).

Secondly, it reduces the operational dollar cost associated with the hosting of a scalable web service. The typical non-FaaS approach requires the system administrator to first provision a set of virtual resources in a cloud, and then to deploy the web service software to those resources. Each resource (they are typically rented separately) incurs an occupancy or usage charge, the sum total of which constitutes the recurring cost of maintaining the web service.

For FaaS implementation of a web service, only function execution incurs a charge which is often only a fraction of a cent. Because functions are only executed in response to events, the service operator pays

*This work is supported in part by NSF CNS-1703560, OAC-1541215, CCF-1539586, CNS-1218808, ACI-1541215

only for what functions it executes. In so doing, she avoids the typical charges associated with renting the necessary virtual servers to host computations, network load balancers, databases, etc.

While originally developed for scalable web services, FaaS has recently garnered interest from those who are developing applications for "The Internet of Things" (IoT). IoT is a technological possibility in which all (or almost all) common objects become able to communicate using The Internet. In the near term, its less ambitious goals are to integrate sensing and actuation capabilities ubiquitously within various environments (e.g. Smart Cities [6], home automation, etc.)

Because many IoT applications incorporate simple sensing devices they can be programmed as event-driven systems. This affinity for the same programming model supported by public-cloud FaaS systems coupled with the relatively low operational dollar cost that FaaS offers has led many to believe that FaaS will be the technology of choice for implementing IoT applications using cloud computing.

FaaS, IoT and "The Cloud"

Curiously, from a technological point of view, there are currently serious impediments to the use of public-cloud FaaS for IoT applications. Firstly, FaaS is not a distributed computing paradigm. It works well within the confines of a single cloud, but the current FaaS offerings do not include support for wide-area distribution. Secondly, the current IoT offerings from the various public cloud vendors [14, 16] are based on the MQTT [28] protocol with a publication-subscription protocol for Internet-based applications. MQTT programs (and those written for MQTT-SN that target sensor networks) abstract the event-driven programming details associated with data acquisition behind a "pub-sub" interface. Thus an application that incorporates both the vendor-supplied IoT and FaaS technologies must include two separate event-driven sets of software components. One set acquires the data from sensors and operates the publication interface to an implementation of MQTT and the other subscribes to MQTT telemetry and triggers FaaS functions with the subscription data. That is, applications that couple AWS IoT and AWS Lambda [14] or Azure IoT Suite and Azure Functions [16] require two different event-driven program components (one for MQTT publication and one for the FaaS processing) as well as an MQTT-based configuration and deployment.

Additionally, the public cloud offerings do not allow for hierarchical application design in the wide area. "Edge [21, 22]" or "Fog [17]" computing systems recognize that communication delay or energy expenditure (in remote IoT applications) can be reduced when the system can move the computation to the data as easily as it can move the data to the computation. Thus one model of IoT application deployment uses compute and storage devices located at the "edge" of the network (i.e. "near" the sensing devices in terms of network latency and bandwidth) to host computations rather than communicating all data to a public-cloud data center for processing.

Public cloud vendors are beginning to offer edge computing capabilities but these capabilities require a tight integration with the public cloud data centers. For example, AWS Greengrass [9] allows for some AWS Lambda (i.e. FaaS) functionality to execute in an edge computing device, but all persistent storage must reside in the public cloud and not at the edge. Thus while the computations might be structured hierarchically (e.g. as a streaming pipeline from edge to cloud) the application itself can not exploit locality for performance or robustness reasons.

A Serverless Platform of Things

The goal of CSPOT is to allow IoT applications to use a single FaaS programming model "end-to-end" – from devices through edge computing infrastructure to large-scale public clouds – as a single "platform" that supports a common set of abstractions. It is "serverless" in that CSPOT functions do not interact with their deployment technologies. CSPOT programs define event handlers that are triggered by updates to a common storage abstraction that can be hosted an any level of the device-edge-cloud hierarchy. Only state persisted in these abstractions survives function execution – all other locally-generated function state is discarded by the CSPOT runtime.

CSPOT is also a first attempt to build such a platform. It is intentionally designed with "thinner" abstractions than what is available in commercial FaaS systems. There are two reasons for the decision to make CSPOT less abstract than its commercial counterparts. The first is that the successful concepts developed by CSPOT should become the base upon which higher-level abstractions can be constructed. That

is, it represents the first step in a "bottom-up" approach to building a full-fledged "Platform of Things." Secondly, IoT applications must be able to integrate a widely heterogeneous set of devices, infrastructure components, and software services. By defining an "assembly language" for a Platform of Things, the goal is to allow maximal integration capability at the assured expense of programmability and programmer productivity.

2 CSPOT Abstractions

CSPOT defines three abstractions:

- **Namespaces** which root separate hierarchical name spaces for CSPOT storage abstractions,
- **Wide Area Objects of Functions** (WooFs) which are persistent, append-only memory objects and
- **Handlers** which can be triggered when a data item is appended to a WooF.

The programming model is event-driven where events are triggered by WooF append operations. All data that persists beyond the execution of a CSPOT handler must do so as data that has been appended to one or more WooFs. Thus a CSPOT application consists of event-triggered computations that may generates volatile local state but that result in updates to application "variables" that are global to the application, have append-only semantics, and which are persistent.

Each WooF is named uniquely within a CSPOT Namespace and Namespaces cannot overlap. To allow for the exploitation of locality, each Namespace has a location (e.g. a host machine) and communication between Namespaces corresponds to a communication message being sent from the Namespace where the Handler is performing an update to the Namespace in which the target of the update is located.

Each Handler is also logically part of a Namespace and Handlers cannot be invoked across Namespaces. That is, a Handler can only operate directly on WooFs within the Namespace that contains it.

Each WooF is logically a log of append operations where each element that is appended is an untyped memory region of fixed size. The element size for a WooF is set when the WooF is created and cannot be changed over its lifetime.

Each append to a specific WooF returns a unique sequence number associated with the element that has been appended to the WooF.

The length of append history maintained in each WooF is fixed and set when the WooF is created. All elements between the element most recently appended and the "earliest" element in the history can be accessed (i.e. there are no missing elements between elements that are present in the WooF history).

WooF Names

A WooF name is a URI [7] beginning with the string "woof://" which specifies a hierarchical name (i.e. a "path") to the WooF object itself. If a host specifier and or port are included in the name, then they designate the Internet address of a host where the WooF is stored. If they are missing, then the host is assumed to be local.

Examples of valid WooF names include

- woof://hostname.cs.ucsb.edu/home/smartfarm/woofObject
- woof:///var/CSPOT /woofObject

The first example names a WooF called **woofObject** located in the Namespace **/home/smartfarm** on the host **hostname.cs.ucsb.edu**. The second example names a WooF called **woofObject** on the local host in the Namespace **/var/CSPOT /woofObject**.

The CSPOT API

The CSPOT API consists of a create operation that defines the name, element size, and history length for each WooF, a put operation that appends an element to a WooF (returning its sequence number) and a get operation that returns the element corresponding to a sequence number passed as an argument. It also includes operations that allow a programmer to get the attributes associated with a WooF.

The native API for CSPOT is written for the C programming language. The API calls are as follows.

- **int WooFCreate(char *woof_name, unsigned long element_size, unsigned long history_size)**
  - **woof_name** is the local or fully qualified name of the WooF to create.
  - **element_size** refers to the number of bytes in a memory region.
  - **history** size refers to the number of elements (not bytes) in the WOOF history.
  - returns ¡ 0 on failure.

The **WooFCreate()** creates a WooF the local Namespace. Currently, the messaging support for create is disabled so that only a create call from a handler or CSPOT client program running on the same machine as where the Namespace is hosted is possible. Once message authentication is implemented, this restriction will be removed.

- **unsigned long WooFPut(char *woof_name, char *handler_name, void *element)**
  - **woof_name** is the local or fully qualified name of an existing WooF
  - **handler_name** is the name of a file in the WooF's Namespace that contains the handler code or it is NULL. When **handler_name** is NULL, no handler will be triggered after the append of the element.
  - **element** is an "in" parameter that points to an memory region to be appended to the WooF. The call returns the sequence number of the element or an unsigned representation of -1 on failure.

The **WooFPut()** function causes the untyped memory pointed to by the **element** parameter to be appended to the specified WooF. The size of the memory is determined by the element size that was specified when the WooF was created. When the call succeeds, the sequence number assigned to the appended element is returned to the caller. When it fails, the unsigned representation of -1 is returned. The **handler** parameter specifies the name of a handler binary located in the same Namespace as the WooF that CSPOT will invoke once the append has been successfully completed. The call to **WooFPut()** completes when the append is successful and not when the Handler completes. If the **handler** parameter is NULL, CSPOT will not invoke a handler.

- **int WooFGet(char *woof_name, void *element, unsigned long seq_no)**
  - **woof_name** is the local or fully qualified name of an existing WooF.
  - **element** is an "out" parameter that will be filled in with data retrieved from the WooF.
  - **seq_no** is the sequence number of the element to be returned.

A call to **WooFGet()** returns the element corresponding to the sequence number that is passed as an argument. If the sequence number is valid (i.e. it is the sequence number returned to a previously occurring call to **WooFPut()** for the same WooF) then the memory pointed to by the **element** parameter is filled in with the data for that element and the return value from the call is greater than or equal to zero. If the sequence number is not valid or if the WooF does not exist in the Namespace, the call returns a negative value. Note that the size of the memory region that **WooFGet()** writes is determined by the element size specified when the WooF was created.

- **unsigned long WooFGetLatestSeqno(char *woof_name)**
  - **woof_name** is the local or fully qualified name of an existing WooF.

A call to **WooFGetLatestSeqno()** returns the sequence number of the most recent successful append operation for the specified WooF. If the call fails, then an unsigned representation of -1 is returned.

Note that **WooFGet()** and **WooFGetLatestSeqno()** are unsynchronized. That is, a call sequence such as the one show below

```
seq_no = WooFGetLatestSeqno(a_woof);
if(seq_no != (unsigned long)-1) {
  err = WooFGet(a_woof,element,seq_no);
}
```

returns the element that was at the "tail" of the WooF at the time of the call to **WooFGetLatestSeqno()** was serviced in the Namespace.

Handlers and Clients

The CSPOT API can be invoked either within Handlers or from "client" programs that are interacting with one or more CSPOT applications. Each Handler must have the following C-language prototype.

```
int handler_function_name(WOOF *wf,
          unsigned long seq_no,
          void *ptr)
```

where **handler_function_name** is a legal C-language function name (i.e. the handler will be compiled as a C-language function having these three arguments). The first parameter is a C-language pointer to a structure defined by CSPOT for manipulating WooFs. At some future point in the development of CSPOT , the parameter will become opaque, but both for debugging and possible performance optimizations, the structure that the CSPOT runtime uses internally is available to a Handler. The second parameter is the sequence number that CSPOT assigned to the append to the WooF. Note that the append occurs before the Handler is invoked by the runtime. The third parameter is an "in" pointer that points to a copy of untyped memory that has been appended. This memory is logically immutable so a change to it by the Handler to the memory pointed to by the **ptr** function does not persist beyond the Handler's execution lifetime. By convention, Handlers are expected to return zero on success and a negative value on failure.

At present, Handlers should only persist state via a call to **WooFPut()**. However they are not restricted from making network connections to access outside services, some of which could also persist application state.

Clients are programs written in any language that can operates the CSPOT API. Logically, a call to a CSPOT API function from a program that is not a Handler results in a request to the WooF's Namespace to perform the operation on behalf of the caller and to return the results.

3 Building a CSPOT Application

This section describes how to build and execute a simple CSPOT application. It also includes several examples of slightly more complex applications and a description of how to use CSPOT as part of an IoT installation.

Building CSPOT

The current CSPOT code base is available from `https://github.com/MAYHEM-Lab/cspot.git`. It is currently only tested for Centos 7, although it is functional for Ubuntu and Rasperian in client-form only. Note that WooF names must include the CSPOT Namespace port number explicitly for Rasperian clients.

To build CSPOT for Centos 7, run the script

```
install-centos7.sh
```

as the root user. This script will attempt to install the dependencies required by CSPOT and several example CSPOT applications that are included in the CSPOT repository. Once the installation script has completed, running the `make` command in the code directory will build CSPOT .

Hello World

Listing 1 shows a simple "hello world" CSPOT handler.

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

#include "woofc.h"
#include "hw.h"

int hw(WOOF *wf,
       unsigned long seq_no,
       void *ptr)
{

  HW_EL *el = (HW_EL *)ptr;
  fprintf(stdout,
    "hello world\n");
  fprintf(stdout,
    "from woof %s at %lu with string: %s\n",
      wf->shared->filename,
      seq_no,
      el->string);
  fflush(stdout);
  return(0);

}
```

Listing 1: Hello World Handler

The Handler both prints "Hello World" and also a string it reads from the WooF. Like `pthreads`, the handler must unmarshal its arguments from the WooF memory. In this example, the application defines a C-language data structure in `hw.h` that formats the memory in each element of the WooF.

```
struct obj_stc
{
       char string[255];
};
typedef struct obj_stc HW_EL;
```

Listing 2: Hello World data type

Thus a call to **WooFPut()** that invokes the Handler `hw` Handler can append a string of up to 255 bytes. When `hw` is invoked by the CSPOT runtime, it will be passed the sequence number and a pointer to a memory region that contains the data appended to the WooF with this sequence number.

To run the application requires a client which must create the WooF and then call **WooFPut()** specifying the `hw` Handler. In this example, the following client program does both.

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

#include "woofc.h"
#include "woofc-host.h"
```

```
#include "hw.h"

#define ARGS "W:"
char *Usage = "hw-client -W woof_name\n";

char Wname[4096];
int main(int argc, char **argv)
{
        int c;
        int err;
        HW_EL el;
        unsigned long ndx;

        while((c = getopt(argc,argv,ARGS)) != EOF) {
                switch(c) {
                        case 'W':
                                strncpy(Wname,optarg,sizeof(Wname));
                                break;
                        default:
                                fprintf(stderr,
                                "unrecognized command %c\n",(char)c);
                                fprintf(stderr,"%s",Usage);
                                exit(1);
                }
        }

        if(Wname[0] == 0) {
                fprintf(stderr,"must specify woof name\n");
                fprintf(stderr,"%s",Usage);
                fflush(stderr);
                exit(1);
        }

        WooFInit();

        err = WooFCreate(Wname,sizeof(HW_EL),5);
        if(err < 0) {
                fprintf(stderr,"couldn't create woof from %s\n",Wname);
                fflush(stderr);
                exit(1);
        }

        memset(el.string,0,sizeof(el.string));
        strncpy(el.string,"my first bark",sizeof(el.string));

        ndx = WooFPut(Wname,"hw",(void *)&el);
        if(WooFInvalid(ndx)) {
                fprintf(stderr,"first WooFPut failed for %s\n",Wname);
                fflush(stderr);
                exit(1);
        }
        return(0);
}
```

Listing 3: Contents of hw-client.c

A call to **WooFCreate()** specifies that each element will be **sizeof(EL_HW)** bytes so that each call to put or get will manipulate a full structure. The history size is set to 5 although since the program does not iterate it could be 1.

The Build Model

The CSPOT runtime system (cf Section **??**) launches each handler as a process within a Docker container [18] passing it the canonical set of handler arguments. Thus, each handler must be compiled as separate Linux program that will be forked and execed within the container. To do so, CSPOT uses a C-language wrapper to supply the *main()* function. This wrapper (called a *shepherd*) performs all of the initialization necessary to prove the handler with access to the appropriate element within the target WooF. It then calls the handler, passing the necessary arguments and, when the handler has completed, it "disconnects" the process from the WooF.

Because the C language does not support reflection, integrating the handler's entry point with the shepherd requires a textual substitution within the code for the shepherd. Specifically, the file *woofc-shepherd.c* contains the string *WOOF_HANDLER_NAME* which must be replaced by the name of the handler entry point when the handler is compiled. Typically, this substitution takes place in a makefile using the Linux utility `sed` (as shown below)

```
sed 's/WOOF_HANDLER_NAME/${HAND1}/g' ${SHEP_SRC} > ${HAND1}_shepherd.c
${CC} ${CFLAGS} -c ${HAND1}_shepherd.c -o ${HAND1}_shepherd.o
${CC} ${CFLAGS} -o ${HAND1} ${HAND1}.c ${HAND1}_shepherd.o\\
        ${WOBJ} ${SLIB} ${LOBJ} ${MLIB} ${ULIB} ${LIBS}
```

Listing 4: Building a CSPOT Handler Binary

In this example, *SHEP_SRC* is defined to be "woofc-shepherd.c" and *HAND1* is defined to be "hw". Thus, this sequence of makefile directives substitutes the string "hw" for the string "WOOF_HANDLER_NAME" in the file *woof-shepherd.c* and creates a new file with the name *hw_shepherd.c*. This file is then compiled and linked with the object file created from *hw.c* to create a binary with the file name `hw`. The CSPOT runtime looks for the binary having this name based on the handler name specified in a call to **WooFPut()** (cf Listing 3).

Running Hello World

To execute the application requires that the client binary, handler binary, a container binary, and a binary for the CSPOT platform all be copied to some directory on the host that will be hosting the Namespace. In this example, the Namespace is */hw-namespace* and we assume that the user executing the commands has write permissions on that directory and also that CSPOT is installed in `/usr/local/src/cspot`. Thus the following commands must be executed to initialize the Namespace for "Hello World."

```
mkdir -p /hw-namespace
cp /usr/local/src/cspot/apps/hello-world/hw /hw-namespace
cp /usr/local/src/cspot/apps/hello-world/hw-client /hw-namespace
cp /usr/local/src/cspot/woofc-namespace-platform /hw-namespace
cp /usr/local/src/cspot/woofc-container /hw-namespace
```

To initiate the platform in the Namespace *hw-namespace* run the commands

```
cd /hw-namespace
./woofc-namespace-platform >& namespace.log.txt &
```

The platform will start the CSPOT runtime and log all output to the file *namespace.log.txt* in that directory.

Once the platform is executing, to run "Hello World" execute the commands

```
cd /hw-namespace
./hw-client -W hw-woof
```

In this example, because the client calls **WooFCreate()** it must be executing on the same machine as the platform. Also, the WooF name given to the client will be interpreted relative to the current working directory if it is supplied as a simple name (and not as a path name).

Docker routes standard out to the process that calls `docker run` so the output from the `hw` hander will be printed as the output from `woofc-namespace-platform` and will appear in the file */hw-namespace/namespace.log.txt*.

CSPOT Application Structure

The "Hello World" example shows the general structure of CSPOT applications. Each application requires

- a Namespace that correspond to a directory in the file system of the host machine,
- a copy of the `woofc-namespace-platform` binary compiled for the host machine to be located in the Namespace directory,
- a copy of the `woofc-container` binary compiled for the host machine to be located in the Namespace directory,
- all handler binaries (compiled using the *woofc-shepherd.c* wrapper for the host machine) to be copied to the Namespace directory, and
- at least one client program (responsible for initiating execution).

For "Hello World" the client program creates a WooF so it must also run on the host machine. The program `hw-cleint` shown in Listing 3 interprets its only argument as the name of a WooF that should be created in the Namespace directory which is inferred to be the current working directory. That is, `hw-cleint` must run from the Namespace directory because it uses the current working directory to create the WooF name that `woofc-namespace-platform` can parse to determine the WooF to create and access.

CSPOT does not require that all client programs reside in the Namespace directory, however. The CSPOT repository contains several examples of different client programs, each of which can be run from anywhere in the file system. In addition, client programs (like those described below) can be run from remote machines as long as they do not attempt to create WooFs. That is, a remote client can access a WooF, but it must be created by a user with sufficient permissions to the Namespace directory to create the WooF on the host machine.

The Senspot API

Often, it is desirable to acquire data from a host or device that isn't running the CSPOT runtime system but which should be processed by a CSPOT application. It is possible to write specific remote clients for each application but in the case where each datum is a single, small value, CSPOT includes *Senspot* remote clients, and a *Senspot* local client for creating WooFs. Each WooF must be created manually by a user with write permissions to the Namespace directory on the machine hosting the WooF. Once created, however, the data in the WooF can be accessed from any host using the *Senspot* clients.

The *Senspot* API consists of the following interface programs.

- **senspot-init -W woof-name -s size** defines a WooF in the namespace corresponding to the prefix of the WooF name (or the current working directory if there is no prefix) having space for `size` elements.
- **senspot-put -W woof-name -T type** puts a typed value to the WooF named by the -W argument. The argument `type` is a single character that indicates the C data type conveyed in the put. Valid data types are
  - **'d'** for double precision floating point
  - **'i'** for C-language integer
  - **'l'** for C-language long integer
  - **'s'** for C-language string up to 1024 bytes in length

  senspot-put reads the value from the standard input.
- **senspot-get -W woof-name -S seq_no** reads the value and formats it from the element having sequence number **seq_no**. If the **-S** option is elided, the value most recently appended to the WooF is returned.

Senspot is intended for use with scripting or other languages to provide simple, one-dimensional data acquisition functionality. The example shown below illustrates how to use the Senspot API to construct a simple "load average" sensor for Centos 7 Linux. In the example, the machine hosting the Namespace for the WooF has IP address 128.111.47.51.

On the Namespace machine (128.111.47.51), the user must initialize a Senspot WooF for the load average sensor in a Namespace. In this example, she has write permissions to the directory `/load-avg-namespace` on the Namespace machine. To initialize the WooF, she executes the command

```
cd /load-avg-namespace
./senspot-init -W la-sensor -s 10000
```

The **-s** argument specifies that the WooF will store a history of load-average measurements that has a length of $10,000$. As described previously, the **senspot-init** command contacts the Namespace platform that must be running to service requests for the Namespace.

On the source machine where load-average measurements are to be gathered, the following BASH script uses the Linux `uptime` command to report the current load average and to post it to the WooF.

```
#!/bin/bash

WOOFNAME=woof://128.111.47.51/load-avg-namespace/la-sensor

/usr/bin/uptime | awk '{print $10}' | sed 's/,//g' | senspot-put -W $WOOFNAME -T 'd'
```

The script assumes that the **senspot-put** binary is in the execution path for the user.

Each time the script is executed, it will append the current load average measurement to the WooF located in the Namespace `/load-avg-namespace` on the host 128.111.47.51.

To read the last load-average value appended to the WooF, execute the command

```
senspot-get -W woof://128.111.47.51/load-avg-namespace/la-sensor
```

senspot-get returns the value, the Linux epoch time for the time when the measurement was posted via **senspot-put**, the IP address of the host generating the measurement, and the sequence number.

Live Temperature Prediction: a Complex Application Example

The UCSB SmartFarm [25, 29] project is using CSPOT as part of an IoT application that is designed to aid growers and farm managers in preventing frost damage to crops. One method of frost prevents uses large wind-generating fans to mix warm air aloft with cold air near the ground. The fans are typically propane or diesel powered, causing considerable expense (in terms of fuel cost) and carbon emissions when they are in use. Thus farmers would like to know, with a considerable degree of accuracy the temperature differential between the air at approximately 10 meters altitude and at 1 meter altitude and many location in their growing blocks.

Current solutions to this problem rely on manual labor to drive or ride though the property reading fixed thermometers. This solution is error prone and expensive since the labor force must work through the night when frost is likely (at least in the California Central Valley).

The UCSB SmartFarm project is developing a system for deploying inexpensive, low power temperature sensors at multiple altitudes to be able to gather the temperature information in real time, to analyze it for the temperature gradients that indicate fan activiation is necessary, and that monitor the temperature change caused by fan activiation to ensure that it is effective but also not over used. Currently, at each measurement location, the project deploys a Raspberry Pi with an attached temperature and humidity sensor. This installtion uses a battery and a small set of solar panels the battery during the day so that it can run at night.

However, during the course of the developing the team noticed that the internal CPU temperature, as reported by the on-board health-and-status interfaces implemented by Raspian [5] (a Linux variant for the Raspberry Pi platform) was highly correlated with outdoor temperature.

Figure 1 shows time series traces for the outdoor temperature (as measured by a commercial-grade meterological station) and the internal CPU temperature for a Raspberry Pi "Zero" [4] located at the

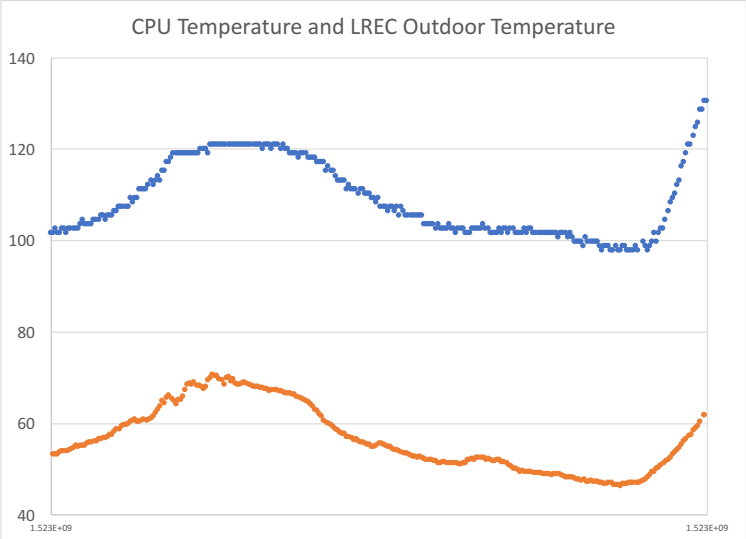

Figure 1: Time series trace of outdoor temperature and CPU temperature taken from LREC. The units of the y-axis are degrees Farenheit. The sensor generates a measurement every 5 minutes.

Lindcove Research and Extension Center (LREC) in Exeter, CA. The meterologocal station measures outdoor temperature at 10 meters and the Raspberry (located in a weatherproof container) is at a 1 meter altitude.

From the Figure (and a number of other experiments including those that use commercially available meterological data) it is clear that outdoor temperature can be predicted from CPU temperature. However, note that there are some discrepencies in shape between the two curves. To generate an accurate prediction of outdoor temperature, the application smooths the CPU series using Singular Spectrum Analsys (SSA [23]) and the computes a linear regression between the smoothed CPU series and the observed outdoor temperature. SSA requires a number of lags of autocorrelation to use and a finite history. In Figure 1 the system chooses up to 12 lags (30 minutes) over a history of 24 measurements (2 hours). It recomputes both the smoothed series and the regression coefficients in CSPOT handlers every time a new outdoor measurement is posted to a WooF. Similarly, every time a new CPU temperature measurement is posted, it uses the most recently computed regression coefficients to predict the outdoor temperature.

Figure 2 shows all three series: the CPU temperature series, the outdoor measurement series, and the predicted outdoor temperature series. In this figure, the measurements are shows as individual markers and the solid line shows the predictions. Despite some obvious deviation, over this time period (24 hours), the mean absolute error between the measured outdoor temperature and the predicted outdoor temnperature is 0.73 degrees Farenheit and the standard deviation for this error is 0.61.

By obviating the need to fit the Raspberry Pi with an external temperature sensor, it is possible to reduce the cost of measuring outdoor temperature at a number of locations by as much as 50%. For example, it is possible to optimize the use of frost-prevention measures in an agricultural setting by taking accurate air temperature readings from a number of different distributed locations. By using the internal CPU tempreature senor as an outdoor thermometer, the cost of implementing such a system at scale can be reduced dramatically.

Figure 3 shows the structure of the application. A client application component running on the Raspbery Pi executes a call to **WooFPut()** every time a CPU temperture measurement is to be taken (every 5 minutes for the data in Figure 2). Similarly, an agent that is polling a web site where the weather station posts its data executes a **WooFPut()** to store a raw outdoor temperature measurement (again, every 5 minutes in the previous figure).

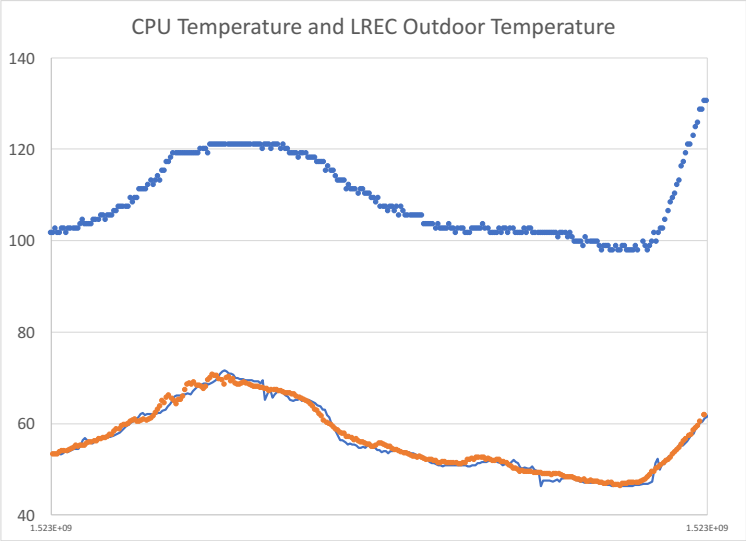

Figure 2: Time series trace of outdoor temperature, CPU temperature, and predicted outdoor temperature taken from LREC. The units of the y-axis are degrees Farenheit. The sensor generates a measurement every 5 minutes.

To improve data durability as well as overall robustness of the system this initil data is then relayed by two intermediary client applications that fetch the data using **WooFGet()** and then forward it via **WooFPut()**. Note that these intermediaries can query both source and target WooFs to ensure that all sequence numbers are effectively forwarded.

The puts to the seconday data replica WooFs each trigger a separate handler function. When data is put to the CPU temperature replica WooF, the handler fetches the latest model that has been fit using the most recent outdoor temperature data and makes a prediction of outdoor temperature that it puts to the prediction WooF. When new outdoor temperature data is put to the temperature replication WooF, a handler is triggerd to compute a new prediction model using the latest outdoor temperature and lates CPU temperature data. The model parameters are put to a model WooF where they can be fetched by the CPU measuurement handler that is triggered when a new value arrives at the CPU replica WooF.

This application architecture is one of several different possible architectures that would produce equivalent results. In particular, it would have been possible to have puts to the initial data WooFs trigger model generation and temperature prediction directly (i.e. without the use intermediary clients and replica WooFs).

However, from a deployment perspective, this architecture offers several advantages. First, the sensors are located in a remote location where network connectivity is both low quality and power intensive over long distances. Thus, the first level WooFs are hosted in an out building near the sensors on an "edge cloud" [19] that communicates with the sensors via a local, isolated network.

The replica WooFs are hosted in a data center cloud that also runs the intermediary client application components. The advantage of this approach is that the edge cloud need not take responsibility for delivering data to the data center cloud, thereby freeing cloud resources to maximize the chance of correct data acquisition in the face of a lossy network. Also, by replicating the data in the data center cloud, it is possible for the edge cloud and the data center cloud to operate independently, the latter using "stale data." In this way, loss of network connectivity to the edge cloud allows an outdoor temperature prediction using out-of-date temperature information. Because outdoor temperature does not fluctuate significantly on a 5 minute time scale, network outages of a relatively short duration do not cause the application to have to

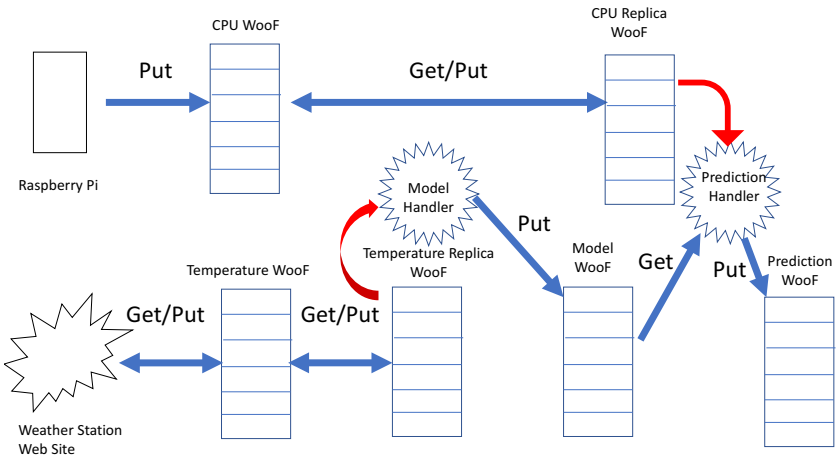

Figure 3: CSPOT Frost Prevention Application Structure

"fail stop."

Note that this deployment architecture is not built into the application itself. That is, the entire application can be hosted on the edge cloud or (if the sensors can communicate with the wide-area internet) in a data center cloud. Note also that the intermediary clients can be removed transparently. That is, if the WooFs at the edge do puts in handler that are triggered, the "back end" of the application will not need to change. Thus application structure can be altered to fit different deployment, reliability, and power-usage requirements while the application, itself, remains unchanged.

4 CSPOT 1.0 Implementation

The current implementation (version 1.0) of CSPOT uses Linux memory-mapped files [1] as the operating system storage abstraction for WooFs. It also isolates function handlers as Linux processes executing within a docker container associated with each Namespace. Handler execution (but not WooF appends) constitute "events" within the system. CSPOT uses an append-event log within each Namespace to trigger Handler execution. As a result, the causal order of all events withing a Namespace is available as a debugging aid. Cross Namespace invocation uses ZeroMQ [8] as a messaging substrate and threads within the container proxy Namespace-external operations. This section discusses the implementation tradeoffs and possible future enhancements to the current CSPOT implementation.

WooFs

Each WooF is implemented as a separate memory-mapped Linux file containing a typed header structure and space to contain some number of fixed-sized elements. The header includes the local file name of the WooF, element size, the number of elements that are retained in the append history, and the current sequence number.

Each WooF keeps a circular buffer of appended elements. The space for the buffer is located immediately after the header in the memory-mapped file and head and tail incices are kept in the WooF header.

In this way, WooFs are self-describing in that all of the information necessary to manipulate a WooF are contained in the WooF itself. When a WooF is "opened" its contents are mapped into the memory space of

the process opening the WooF as shared memory. Thus multiple threads and, indeed, multiple processes can access a WooF concurrently using the information contained in the WooF header. To implement synchronization for internal operations, the WooF header includes two Linux semaphores [2]. The first implements mutual exclusion for operations like buffer head and tail index update, sequence number assignments, etc. The second allows threads to synchronize on the "tail" of the WooF so that when a new append occurs, they can be activated.

Handlers, Containers, and the Event Log

When the Namespace platform begins executing for a Namespace, it launches a Docker container for the Namespace that shares the Namespace directory in which all WooFs and Handlers for the Namespace must be located. Docker includes an option to specify where, in the container's directory structure, a directory shared with the host must be located. By using the same location within the container (e.g. "/CSPOT ") the API can locate the WooF and Handlers within the container.

Each handler is compiled as a separate Linux executable program. When an invocation of **WooFPut()** includes a handler name, the API code appends the element specified in the call to **WooFPut()** to the WooF and then appends an event record specifying the WooF, the sequence number of the element that has been appended, and the handler name to an event log for the Namespace.

The main process within the container spawns several threads that synchronize on the tail of the event log using a semaphore in the event log header. These threads "claim" events from the log by atomically appending a claim record for an unclaimed event and then call Linux `fork()` and `exec()` on the handler binary. When **WooFPut()** is called from within a Handler, the sequence number of the caller is including in the event record indicating that it is the "cause" of the Handler firing. Thus the event log that serves as the dispatch structure for handlers includes the dependency information necessary to determine causal order.

The event log also includes a host specifer so that cross Namespace dependencies can be tracked. Currently, this feature is not implemented and, instead, the host identifier for the local Namespace is used. That is, the container spawns several message request threads to handle extra-Namespace operations. Each thread acts as a proxy for a remote requester and call **WooFPut()** on the local Namespace. Because **WooFPut()** does not include a host specifier in its argument list, the host identifier for the local Namespace is used as the "cause" of the event.

5 CSPOT Performance

The goal of the CSPOT performance profile is to facilitate lightweight execution on infrastructure across a spectrum of scales. To investigate the nature of this profile, we benchmark the CSPOT runtime using the hosts shown in Table 1

Host	Make and Model	CPU	Memory	OS
RPi 3	Raspberry Pi 3	1.2 GHz Quad Core ARMv8	1 GB	Raspian 9.3
NUC	Intel NUC 6i7KYK	2.6 GHz Intel Core i7-6770HQ	32 GB	CentOS 7.2
Aristotle	cg1.4xlarge	2.1 GHz Xeon, 4 vCPU	8 GB	CentOS 7.2
AWS EC2	m5.xlarge	2.5 GHz Xeon, 4 vCPU	16 GB	CentOS 7.5

Table 1: Edge cloud testbed for CSPOT performance benchmarking

We deploy CSPOT on a Raspberry Pi 3 [3] located in a powered, outdoor setting in Goleta, California. We have instantiated an edge cloud using Intel NUC devices [24] running Eucalyptus version 4.3 [27, 20] at

the University of California, Santa Barbara (UCSB). Aristotle [12] is a private cloud running Eucalyptus 4.3 and located at UCSB that participates in a multi-campus cloud federation. Finally, we include an *m5.xlarge* instances in Amazon's EC2 [26] in the *us-east-1* region.

Table 2 shows the mean, standard deviation, and 95^{th} percentile function invocation performance over a number of different exectuion platforms.

Host	Mean (ms)	std-dev (ms)	95% (ms)
RPi 3	37	6.8	48
NUC	4.0	0.63	4.9
NUC-VM	6.5	3.3	15
Aristotle	5.0	1.6	7.0
AWS EC2	5.0	0.96	6.6
AWS Lambda	253	90	584

Table 2: Comparison of CSPOT function dispatch times across device, edge, and cloud hosts. The statistics are each computed from 100 function invocations and the units are milleseconds

In each cell of the table containing a numeric value, the units are milleseconds. The benchmark application runs a client on the same host where the WooF is located causing the CSPOT to bypass the ZeroMQ messaging layer. The client records the current time using the `gettimeofday()` system call and then executes **WooFPut()** to put this time stamp into a pre-installed WooF. The call to **WooFPut()** invokes a handler that reads the first time stamp, also calls `gettimeofday()`, and writes both timestamps to a second WooF. After the experiment is complete, the benchmark reads the sequence of timestamp pairs from the second WooF. The difference between each pair of time stamps records the time needed by the CSPOT runtime to support a handler invocation. Both the client and the handler are written in C.

In addition, for comparison, we have also implemented the benchmark in Python for execution using AWS Lambda [13]. The handler is triggers by a store to DynamoDB [10] using a Lambda trigger.

Currently, the CSPOT implementation throttles handler invocation requests so that no more tha 5 are active at a single moment in time (which is a tunable parameter) per namespace. In Table 2, each statistic is computed over 100 consecutive invocations which are batched four-at-a-time. The table also includes two edge deployments: one in an edge cloud which uses KVM as a hypervisor to host the namespace and the client on an Intel NUC (marked *NUC-VM* in the table) and another which runs the CSPOT runtime on the NUC itself (and not in a virtual machine).

The table shows that CSPOT is able to achieve high rates of invocation performance regardless of the scale of the system on which it is deployed. The NUC, Aristotle, and EC2 invocation times are all similar which is somewhat surprising given their very different internal engineering and scale. Curiously, the mean of the virtualized deployment on the NUC is higher than the others due to the presence of more values near the tail of the distribution. That is, in the virtualized case, the run time occasionally generates a much longer execution time (as evidenced by the $95-th$ percentile which is much larger) than in the non-virtualized case. It is not possible to repeat the virtualized versus non-virtualized experiment on Aristole or EC2 however the 95^{th} percentile in each case is much closer to the mean than in the *NUC-VM* case indicating that the effect is not as pronouced if it is present.

Moreover, the latency is considerably lower for CSPOT across all platforms that for the Pyhton implementation in Lambda. For the Lambda benchmark, we report run that takes place after a "warm start." That is, we run the benchmark once to allow AWS to employ what ever state caching it might implement internally (e.g. container persistence) and then run the benchmark again immediately. The timing data is

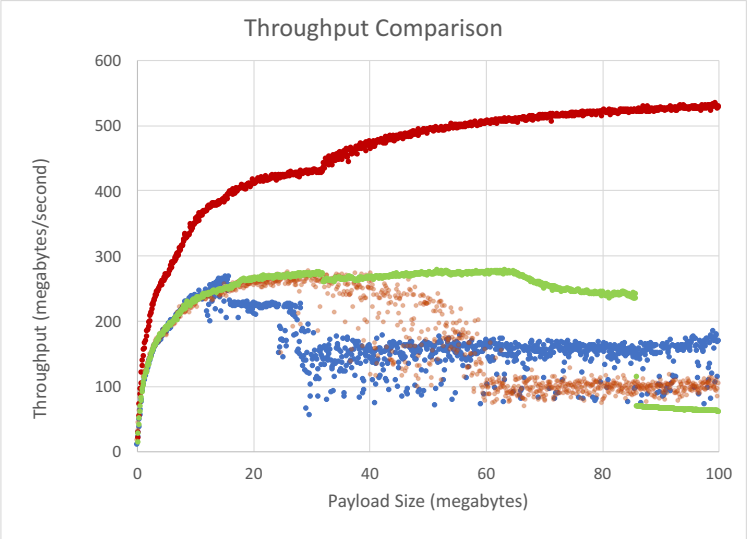

Figure 4: Comparison of throughput rates as a function of payload size across edge and cloud platforms

taken from the second run.

From the perspective of invocation latency, CSPOT is clearly high performance a lightweight across a spectrum of platform scales.

Figure 4 compares the throughput that CSPOT is able to achieve on various platforms. The benchmark repeats the latency probe detailed in Table 2 with payloads of increasing size. That is, for each size payload (shown on the x-axis in the figure), the benchmark records the total time necessary to deliver 100 of those payloads (via 100 sequential calls to **WooFPut()**) and then computes the overall observed bandwidth (shown on the y-axis). The experiment starts with a payload of 100K bytes and increases by 100K until a maximum payload of 100M bytes per transfer. It aggegrates the time necessary to make 100 transfers of each payload size to compute throughput.

The figure compares a three different cloud deployments of CSPOT (edge cloud, Aristotle private cloud, AWS) and an unvirtualized deployment on the Intel NUC. The edge cloud consists of Intel NUC processors (*cf* Table 1 running Eucalyptus which uses KVM as does Aristotle. However, AWS uses an internal and propietary version of Xen [30]. In the figure, the three cloud platforms achieve more or less the same maximal throughput as a function of payload size. The curves are not smooth, with the highest variation in the edge cloud. However up to a payload size of 10M bytes, the edge cloud, Aristotle, and AWS achieve approximately the same performance (as evidenced by the blue, green, and amber features in the graph respectively). Aristotle's performance is slightly less, but not largely so.

At approximately 10M bytes per transfer, the edge cloud (blue graph features) performance reaches a maximum which is approximately constant at 220 megabytes per second until the payload size reaches 65M bytes per transfer. At that point in the experiment, the performance become more variable through the end of the trace.

In contrast, Aristotle (amber graph features) experiences a variable but gradually decreasing throughput between approximately 40M bytes per transfer and 60M bytes per transfer. Above 60M bytes per transfer, the Aristotle performance stabilizes at a lower total throughput of approximately 100 megabytes per second.

The performance on AWS is more stable through the range of transfer sizes. It reaches a maximum of approximately 260 megabytes per second at a transfer size of 20M bytes per transfer, but then begins to experience a decrease in througput beyond 63M bytes per transfer follwed by a sudden "cliff" at 84M bytes per transfer.

Note that virtualization appears to impose a significant throughput penalty, regardless of the scale of the systems on which the experiments are run. To investigate the effect of virtualization on CSPOT hosting, we ran the same experiment on the the same edge cloud machine that we used to host the edge-cloud virtualized experiment. That is, the red graph features in Figure 4 and the blue graph features are from the same machine – an Intel NUC that is part of the edge cloud. The difference is that the blue graph features come from an instance of CSPOT running in a virtual machine using the KVM hypervisor whereas the red graph features come from CSPOT running on the unvirtualized operating system. While there is some variation throughout the unvirtualized experiment, it is less and the throughput is significantly higher.

Performance Variability

All three clouds – the edge cloud, Aristotle, and AWS – were operating in a production computing mode at the time of this experiment. That is, each cloud was hosting applications running by multiple users and we made no attempt to avoid interference due to multi-tennancy. Moreover, the experiment is not randomized with respect to transfer sizes. It starts with a 100K byte transfer size and conducts 1024 consecutive experiments (increasing the payload size by 100K bytes each time) where each experiments is 100 separate consecutive transfers. As a result, each experiment spans several minutes rasing the possibility that the degree fo interferences due to multi-tennancy could vary over the course of an experiment.

This performance variability (particularly for AWS) seemed to indicate that CSPOT running in a multi-tenant, virtualized enviromnment (e.g. a cloud) might experience some interference either from other competing tenants or due to "noise" in the virtualization or operating systems software. Indeed, the difference between the means and 95^{th} percentiles in Table 2 seems to indicate such variability.

Note that we did not make the edge cloud node quiescient during the unvirtualized test. That is, while the CSPOT test was running, that node continued to host the same unrealted virtual machines used by other edge-cloud applications. Thus while there may be some tennancy effects, they appear exacerbated by virtualization.

To understand these effects, for each payload size shown in Figure 4, we extract the transfer (among 100 such transfers) that experienced the *minimum* delay. This fastest transfer corresponds to the highest achieved throughput (computed as payload size divded by transfer time) for that given payload size among 100 back-to-back trials of that size. Figure 5 graphs this "best" (highest) throughput for each payload size. Comparing the maximum throughputs shown in Figure 5 to the "overall" throughputs shown in Figure 4 indicates that there is quite a bit of variation in the virtualized cases, most likely due to multi-tenant interference or operating system noise. The maximum throughput graphs shown in Figure 5 show smoothly increasing throughput as a function of payload size up to some host-specific maximum. Curiously, *all* of the graphs (including that for the unvirtualized edge cloud graph) show a sharp discontinuity when the payload size exceeds 32M bytes. For all of the virtualized deployments (edge cloud, Aristotle, and AWS) the maximum throughput drops sharply when the payload is 32M bytes or larger. Somewhat inexplicably, the maximum throughput *increases* for payloads 32M bytes or larger when CSPOT is run unvirtualized.

Taken together, these results indicate that CSPOT achieves very low latencies compared to AWS Lambda (cf Table 2) but the combination of hypervisor virtualiztion and Linux containers (used by Docker) has a dramatic effect on throughput. Specifically, virualized deployment of CSPOT appears to experience significantly greater variability in throughput compared to an unvirtualized deployment (with similar tennancy competition). Moreover, the maximum throughput appears almost halved when CSPOT is deployed in *any* cloud setting compared to an unvirtualized deployment on a small, portable system (i.e. the Intel NUC used in this study).

These variability effects are consistent with our previous study of AWS Lambda [**?**] but still bear greater scrutiny. In particular, CSPOT's novel use of memory-mapped files and Docker volumes to implement its log-based event runtime system may be exposing Linux performance characteristics that are atypical. However, if and when very low latency and/or high throughput are needed, CSPOT executing unvirtualized (while still providing isolation at the namespace level through Docker containers) appears a viable alternative.

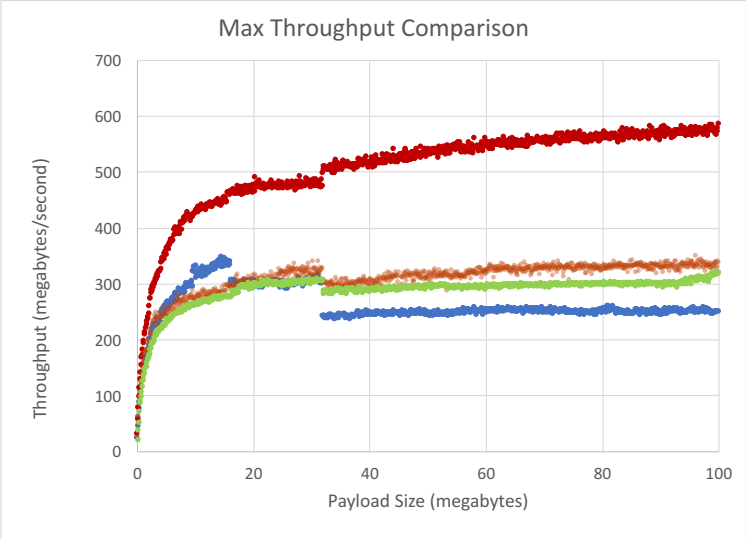

Figure 5: Comparison of maximum throughput rates as a function of payload size across edge and cloud platforms

Data Replication

Because CSPOT intertwines computation, communication, synchronization, and persistence, it is also possible to combine replication for data durability with replication for latency reduction. To illustrate that possibility, we have written an application that takes a local time stamp and calls **WooFPut()** on a remote WooF to store it. The handler for this WooF takes a local time stamp and calls **WooFPut()** on a remote handler, and so on, until the end of the "chain" of puts. In this way, the data is replicated on each WooF before it is forwarded to the next.

To generate the data shown in Figure 6, we implement this "chain" replication strategy using the hosts shown in Table 1. The source of the data is the Raspberry Pi 3 located in an outdoor setting in Goleta, CA. It calls **WooFPut()** on a WooF hosted in an edge cloud sited at the University of California, Santa Barbara (using Intel NUCs and Eucalyptus). The handler running on the edge cloud, calls **WooFPut()** on a WooF hosted in Aristotle (also located at U.C. Santa Barbara) when then calls **WooFPut()** to a WooF hosted in AWS's *us-west-2* availability zone. The figure "stacks" the respective latencies with the lowest in each stack being Pi-to-edge (blue), the next lowest being edge-to-Aristotle (light-green), and the last being Aristotle to AWS (red). U.C. Santa Barbara (UCSB) is located approximately 3 miles from the outdoor site where the Raspberry Pi is sited. the Pi uses 802.11 "WiFi" to communinicate to a wireless tower connected to a cable-modem service. The edge cloud connected to UCSB's campus network via 1 gigabit ethernet. Aristotle has a 10 gigabit ethernet campus network connection and a connection to CENIC [**?**] where it routes to the rest of the Internet (including AWS and the *us-west-2* region).

Table 3 compares the end-to-end latency (the sum of the intermediate latencies) to the AWS Lambda experiment shown in Table 2. From the table, the mean latency for AWS Lambda to trigger a single function as a result of a store ot DynamoDB without cross-region replication or global tables enabled. The data durability of DynamoDB is high because it claims to replicate the data stored in it among multiple availability zones in the region where it is invoked (*us-west-2* in this experiment). However, the AWS Lambda application is local to AWS. That is, the 253 millisecond average latency is the time to store a timestamp pair from within AWS in the same region and availability zone. By contrast, CSPOT has a mean end-to-end latency of 320 milliseconds, but that time includes the *all* of the network latencies between the device and AWS. Moreover, the standard deviation and 95^{th} percentile for CSPOT are lower and each trial creates

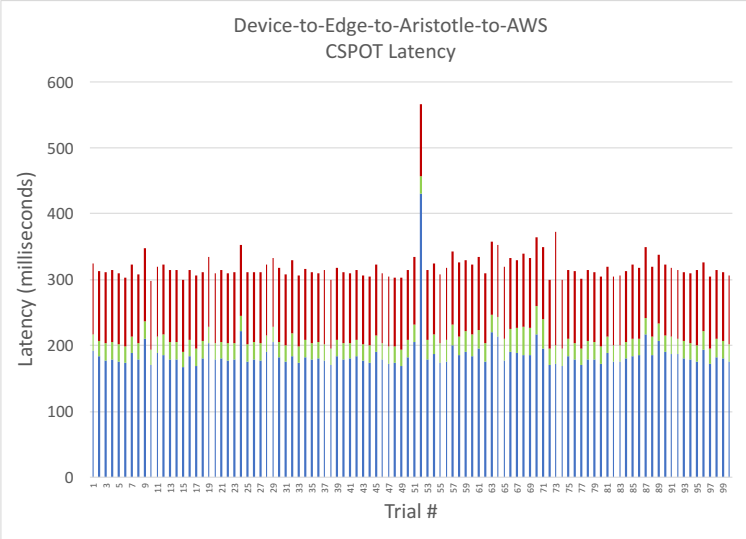

Figure 6: "Chain" replication from Raspberry Pi to edge cloud to Aristotle to AWS. 100 sepparate, back-to-back trials (x-axis). The lowest section of each bar (blue) depicts Pi-to-edge latency, the second from the lowest (green) shows the edge-to-Aristotle latency, and the top part of each bar (red) shows Aristotle-to-AWS latency. The units are milliseconds.

Host	Mean (ms)	std-dev (ms)	95% (ms)
AWS Lambda	253	90	584
device-edge-Aristotle-AWS	320	28	352

Table 3: Comparison of AWS Lambda to CSPOT chain replication across the wide area. The statistics are each computed from 100 function invocations and the units on the y-axis are milleseconds and the x-axis shows each trail.

three replicas (one in the edge cloud, one in Aristotle, and one in AWS) creating 3-replica data durability. Note, also, that the replicas are distributed at the edge (where they can be consumed rapidly in a closed-loop setting), in a large cloud for analysis (Aristotle), and in a public cloud (where even greater scale is available and other features like content distribution services).

These results indicate that CSPOT provides a new, high-performance, and distributed platform for building and deploying cloud-based IoT applications. The performance of individual CSPOT abstractions is significantly faster than the state-of-the-art "Functions-as-a-Service" offering from AWS and the distribution capabilities make it possible to develop replication strategies that are lower latency with similar data durability characteristics.

References

[1] The linux mmap system call. http://man7.org/linux/man-pages/man2/mmap.2.html. [Accessed electronically, March 2018].

[2] Linux semaphores. http://man7.org/linux/man-pages/man7/sem_overview.7.html. [Accessed electronically, March 2018].

[3] Raspberry pi three. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. [Accessed electronically, March 2018].

[4] Raspberry pi zero. https://www.raspberrypi.org/products/raspberry-pi-zero/. [Accessed electronically, March 2018].

[5] Raspian os. https://www.raspberrypi.org/downloads/raspbian/. [Accessed electronically, March 2018].

[6] Smarter cities for smarter growth. https://www-935.ibm.com/services/us/gbs/bus/html/smarter-cities.html. [Accessed electronically, March 2018].

[7] Uniform resource identifier. http://en.wikipedia.org/wiki/Uniform_Resource_Identifier. [Accessed electronically, March 2018].

[8] Zeromq desitributed messaging. http://zeromq.org. [Accessed electronically, March 2018].

[9] Amazon. Amazon GreenGrass, 2017. https://aws.amazon.com/greengrass/ Accessed 15-Sep-2017.

[10] Amazon DynamoDB. https://aws.amazon.com/dynamodb/. [Online; accessed 15-Nov-2016].

[11] Amazon Serverless computing or Google Function as a Service (FaaS) vs Microservices and Container Technologies. https://www.yenlo.com/blog/amazon-serverless-computing-or-google-function-as-a-service--faas-vs-microservices-and-container-technologies [Online; accessed 1-Nov-2016].

[12] Aristotle Cloud Federation. https://federatedcloud.org [Online; accessed 22-Aug-2016].

[13] AWS Lambda. https://aws.amazon.com/lambda/. [Online; accessed 15-Nov-2016].

[14] AWS Lambda IoT Reference Architecture. http://docs.aws.amazon.com/lambda/latest/dg/lambda-introduction.html [Online; accessed 1-Nov-2016].

[15] Azure Functions. https://azure.microsoft.com/en-us/services/functions/. [Online; accessed 15-Nov-2016].

[16] Berdy, N. How to use Azure Functions with IoT Hub message routing, 2017. https://azure.microsoft.com/en-us/blog/how-to-use-azure-functions-with-iot-hub-message-routing/.

[17] Bonomi, F., Milito, R., Zhu, J., and Addepalli, S. Fog computing and its role in the internet of things. In *MCC workshop on Mobile cloud computing* (2012), pp. 13–16.

[18] Docker. https://www.docker.com [Online; accessed 1-Nov-2016].

[19] Elias, A. R., Golubovic, N., Krintz, C., and Wolski, R. Wheres the bear?–automating wildlife image processing using iot and edge cloud systems. In *ACM Conference on IoT Design and Implementation* (2017).

[20] Eucalyptus Documentation. https://docs.eucalyptus.com/eucalyptus/4.3.0/ [Online; accessed 22-Aug-2016].

[21] Floyer, D. The Vital Role of Edge Computing inthe Internet of Things. http://wikibon.com/the-vital-role-of-edge-computing-in-the-internet-of-things/ [Online; accessed 22-Aug-2016].

[22] Garcia Lopez, P., Montresor, A., Epema, D., Datta, A., Higashino, T., Iamnitchi, A., Barcellos, M., Felber, P., and Riviere, E. Edge-centric computing: Vision and challenges. *ACM SIGCOMM Computer Communication Review 45*, 5 (2015), 37–42.

[23] Golyandina, N., and Zhigljavsky, A. *Singular Spectrum Analysis for time series*. Springer Science & Business Media, 2013.

[24] Intel NUC 6i7KYK. https://www.intel.com/content/www/us/en/nuc/nuc-kit-nuc6i7kyk-features-configurations.html [Online; accessed April 2018].

[25] Krintz, C., Wolski, R., Golubovic, N., Lampel, B., Kulkarni, V., Sethuramasamyraja, B., Roberts, B., and Liu, B. SmartFarm: Improving Agriculture Sustainability Using Modern Information Technology. In *KDD Workshop on Data Science for Food, Energy, and Water* (Aug. 2016).

[26] Murty, J. *Programming Amazon Web Services: S3, EC2, SQS, FPS, and SimpleDB*. O'Reilly Media, Inc., 2009.

[27] Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., and Zagorodnov, D. The eucalyptus open-source cloud-computing system. In *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on* (2009), IEEE, pp. 124–131.

[28] Stanford-Clark, A., and Truong, H. L. Mqtt for sensor networks (mqtt-s) protocol specification. *International Business Machines Corporation version 1* (2008).

[29] UCSB SmartFarm. http://www.cs.ucsb.edu/~ckrintz/projects/index.html. [Online; accessed March 2018].

[30] Xen Virtual Machine Monitor Performance. http://www.cl.cam.ac.uk/Research/SRG/netos/xen/performance.html.