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) \cite{functions-as-a-service, serverless-computing, cloud-native-development} 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 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

---

*This work is supported in part by NSF CNS-1703560, OAC-1541215, CCF-1539586, CNS-1218808, ACI-1541215*
(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 [5], 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 [11, 13] are based on the MQTT [21] 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 [11] or Azure IoT Suite and Azure Functions [13] 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 [17, 18]” or “Fog [14]” 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 [8] 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 at 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 generate 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 [6] 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

```c
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 \texttt{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.

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

where \texttt{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 \texttt{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 \texttt{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 operate 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 \texttt{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

\texttt{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 \texttt{make} command in the code directory will build CSPOT.

**Hello World**

Listing 1 shows a simple “hello world” CSPOT handler.

```c
#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\n", Usage);
        exit(1);
    }
}
if (Wname[0] == 0) {
    fprintf(stderr, "must specify woof name\n");
    fprintf(stderr, "%s\n", 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 [15] 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 \texttt{woofc-shepherd.c} contains the string \texttt{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 \texttt{sed} (as shown below)

\begin{verbatim}
sed \textquotesingle{}s/\texttt{WOOF\_HANDLER\_NAME}/\{HAND1\}/g\textquotesingle{} \$(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\}
\end{verbatim}

Listing 4: Building a CSPOT Handler Binary

In this example, \texttt{SHEP\_SRC} is defined to be \texttt{woofc-shepherd.c} and \texttt{HAND1} is defined to be \texttt{hw}. Thus, this sequence of makefile directives substitutes the string \texttt{hw} for the string \texttt{WOOF\_HANDLER\_NAME} in the file \texttt{woof-shepherd.c} and creates a new file with the name \texttt{hw\_shepherd.c}. This file is then compiled and linked with the object file created from \texttt{hw\_c} to create a binary with the file name \texttt{hw}. The CSPOT runtime looks for the binary having this name based on the handler name specified in a call to \texttt{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 \texttt{/hw-namespace} and we assume that the user executing the commands has write permissions on that directory and also that CSPOT is installed in \texttt{/usr/local/src/cspot}. Thus the following commands must be executed to initialize the Namespace for “Hello World.”

\begin{verbatim}
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
\end{verbatim}

To initiate the platform in the Namespace \texttt{hw-namespace} run the commands

\begin{verbatim}
cd /hw-namespace
./woofc-namespace-platform >& namespace.log.txt &
\end{verbatim}

The platform will start the CSPOT runtime and log all output to the file \texttt{namespace.log.txt} in that directory. Once the platform is executing, to run “Hello World” execute the commands

\begin{verbatim}
cd /hw-namespace
./hw-client -W hw-woof
\end{verbatim}

In this example, because the client calls \texttt{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 \texttt{docker run} so the output from the \texttt{hw} handler will be printed as the output from \texttt{woofc-namespace-platform} and will appear in the file \texttt{/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 \texttt{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-client` 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-client` 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` reads the value and formats it from the element having sequence number `seq`. 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 \texttt{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 \texttt{uptime} command to report the current load average and to post it to the WooF.

\texttt{#!/bin/bash}

\texttt{WOOFNAME=woof://128.111.47.51/load-avg-namespace/la-sensor}

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

The script assumes that the \texttt{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 \texttt{/load-avg-namespace} on the host 128.111.47.51.

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

\texttt{senspot-get -W woof://128.111.47.51/load-avg-namespace/la-sensor}

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

\textbf{Live Temperature Prediction: a Complex Application Example}

The UCSB SmartFarm [20, 22] 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 prevention 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 locations 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 activation is necessary, and that monitor the temperature change caused by fan activation 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 installation 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 [4] (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 meteorological station) and the internal CPU temperature for a Raspberry Pi “Zero” [3] located at the Lindcove Research and Extension Center (LREC) in Exeter, CA. The meteorological 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 meteorological data) it is clear that outdoor temperature can be predicted from CPU temperature. However, note that there are some discrepancies in shape between the two curves. To generate an accurate prediction of outdoor temperature, the application smooths the CPU series using Singular Spectrum Analys (SSA [19]).
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 shown as individual markers and the solid line shows the predictions. Despite some obvious deviation, over this time period (3 days), the mean absolute error between the measured outdoor temperature and the predicted outdoor temperature is 1.2 degrees Fahrenheit and the standard deviation for this error is 1.2.

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 temperature sensor 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 Raspberry Pi executes a call to \texttt{WooFPut()} every time a CPU temperature 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 \texttt{WooFPut()} to store a raw outdoor temperature measurement (again, every 5 minutes in the previous figure).

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 \texttt{WooFGet()} and then forward it via \texttt{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 secondary 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 triggered to compute a new prediction model using the latest outdoor temperature and latest CPU temperature data. The model parameters are put to a model WooF where they can be fetched by the CPU measurement 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” [16] 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 “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 within a Namespace is available as a debugging aid. Cross Namespace invocation uses ZeroMQ [7] 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 \texttt{WooFPut()} includes a handler name, the API code appends the element specified in the call to \texttt{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 \texttt{fork()} and \texttt{exec()} on the handler binary. When \texttt{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 specifier 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 \texttt{WooFPut()} on the local Namespace. Because \texttt{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.
References


