NanoLambda:
FaaS at All Resource Scales for IoT

Speaker: Gareth George

Co-authors: Fatih Bakir, Rich Wolski, and Chandra Krintz
RACELab: https://sites.cs.ucsb.edu/~ckrintz/racelab.html
Background

- IoT devices are increasingly prevalent producers of data
- Programming & processing data remains a challenge
  - On device processing:
    - Often difficult to implement
    - Portability, security, maintainability all are challenges
  - Cloud / Edge Cloud processing
    - Easy to program in high level languages
    - Tools such as FaaS provide a homogenous and scalable execution environment
    - Cloud introduces expensive network cost and latency
      - IoT devices will produce 80 zettabytes a year (*IDC) by 2025

*IDC market analysis firm on IoT data [https://www.idc.com/getdoc.jsp?containerId=prUS45213219](https://www.idc.com/getdoc.jsp?containerId=prUS45213219)
Background

What if we could bring FaaS to the device?

The challenge is that FaaS is currently limited to the Cloud or Edge*

- Data must be moved from device to data center
- *High power cost for low power devices to transmit over WiFi*
- Poor network infrastructure in rural areas
- Existing FaaS runtimes are limited to Linux-based edge systems

*AWS GreenGrass and similar services*
Introducing NanoLambda

- NanoLambda: platform for running FaaS handlers across all tiers
  - On Device
  - Cloud / Edge
  - Compatible with AWS Lambda
- Goals
  - Ease of development
  - Portability
  - Small code and memory footprint
  - Security
  - Uniform programming methodology
- At the smallest scales
  - ESP8266 with 96KB of ram and 512KB of program flash storage
  - CC3220SF with 256KB of ram and 1MB of program flash storage
Deploying FaaS Functions

```
# create zip file with python application
zip edgelambda.zip edgelambda.py
# upload it to open source lambda service
# for compilation and deployment
aws lambda create-function
--function-name scheduled_pred_main
--zip-file fileb://edgelambda.zip
--handler edgelambda.new_accel_sample
--runtime python3.6
--endpoint-url http://<lambda service address>:1180
--no-sign-request
--no-verify-ssl
```

NanoLambda Cloud/Edge
1. stores code in object store service
2. compiles and caches compact bytecode representation on-demand

When IoT devices find a handler is not cached locally, they request bytecode representation from NanoLambda service.
NanoLambda Architecture

NanoLambda Cloud/Edge

CSPOT Backend Object Store
Python 3.6 CSPOT Handler
CSPOT WooF Invocation Log

IoTPy Code/API translation and packaging service

NanoLambda On Device

IoTPY on device and remote execution capabilities
CSPOT WooF: measurement logs

Sensors & drivers: custom OS with monitoring C++ threads
NanoLambda Architecture

Comprised of two core systems

- **NanoLambda Cloud/Edge**
  - FaaS handler registry & edge execution environment
  - Remote code compilation and bytecode delivery to IoT devices

- **NanoLambda On Device**
  - Provides on-device handler execution capabilities with *IoTPy*
  - Leverages Python VM isolation to provide isolation
IoTPy Design

- Why python?
- Why not an existing interpreter like micropython?
  - Lacks key embedding features
  - Binary size - micropython 620KB binary vs IoTPy 290KB binary
- IoTPy features
  - Lean memory footprint by leveraging NanoLambda Cloud/Edge for bytecode generation
  - Object-oriented VM implementation & first class embedding support
- IoTPy provides a C/C++ interface for native extensions / functions
  - Built in libraries include: math, json, device, and interaction with NanoLambda Cloud/Edge’s Lambda service
- Security
  - Python VM provides memory protection and container-like isolation
NanoLambda Cloud/Edge

- Service provides two REST API servers offering
  - Persistent object storage compatible with S3
  - A FaaS service that deploys functions written for AWS Lambda
- Built with CSPOT -- a low level framework providing FaaS primitives
  - S3 is implemented as a layer on top CSPOT’s append only object storage
  - Lambda is implemented with event handlers triggered by invocation log updates
  - Handlers are run in Linux containers allowing for concurrent but isolated execution
- Provides a registry of function definitions stored in S3 service
- Binary API for fetching compiled function bytecode
NanoLambda On Device

- Runs python handlers on non-Linux IoT devices
- Much like NanoLambda Cloud/Edge, invocations triggered by log events
  - Events can originate from sensors on device
  - Data can also be delivered remotely over CSPOT’s network
- Each append runs a C-language handler function invoking IoTPy
  - On cold start function bytecode is requested from NanoLambda Cloud/Edge
  - IoTPy caches bytecode & interpreter state to accelerate future runs
Execution Offloading

NanoLambda On Device is code compatible with NanoLambda Cloud/Edge

- On Device supports devices as small as ESP8266 and the CC3220SF
- NanoLambda Cloud/Edge runs on Linux at the edge and in the cloud

Portability: The choice to use NanoLambda allows for On Device, at the Edge, in the private Cloud, or directly on AWS Lambda
Predictive Maintenance Application

- Predictive Maintenance is a technique using sensors to detect part failure
- We examine failure detection in motors using accelerometers
- Setup:
  - Accelerometer attached to a motor reads vibration magnitude 5 times a second
  - Data is appended to a WooF for persistence, a history of 32 records is kept.
  - Each append triggers failure detection handler to run
- Handler is benchmarked running on NanoLambda On Device and NanoLambda Cloud/Edge for various problem sizes and configurations
Predictive Maintenance Application

def new_accel_sample(payload, ctx):
    global fan_mdl
    transformed = []
    for record in payload:
        transformed.append(magnitude(record))
    payload = None
    prob = ktest(transformed, reference)
    return str(prob)

def ktest(datalist1, datalist2):
    n1 = len(datalist1)
    n2 = len(datalist2)
    datalist1.sort()
    datalist2.sort()
    j1 = 0
    j2 = 0
    d = 0.0
    fn1 = 0.0
    fn2 = 0.0
    while j1 < n1 and j2 < n2:
        d1 = datalist1[j1]
        d2 = datalist2[j2]
        if d1 <= d2:
            fn1 = (float(j1)+1.0)/float(n1)
            j1 += 1
        if d2 <= d1:
            fn2 = (float(j2)+1.0)/float(n2)
            j2 += 1
        dtemp = math.fabs(fn2-fn1)
        if dtemp > d:
            d = dtemp
        ne = float(n1*n2)/float(n1+n2)
        nesq = math.sqrt(ne)
        prob = ksprob(nesq*0.12+0.11/nesq)*d
        return d, prob, ne

def ksprob(alam):
    fac = 2.0
    sum = 0.0
    termbf = 0.0
    a2 = -2.0*alam*alam
    for j in range(1, 11):
        term = fac*math.exp(a2*j*j)
        sum += term
        if math.fabs(term) <= 0.001*termbf or math.fabs(term) <= 1.0e-8*sum:
            return sum
        fac = -fac
        termbf = math.fabs(term)
    return 1.0
Plotting Power & Invocation Latency

**Fig. 5.** Comparison of power draw over request lifecycle for various KS problem sizes using both remote and local strategies.
Naive Offloading Scheduler

Naive algorithm:
- Pick the lowest latency (time to result) execution strategy based on history
  - Local (On Device) or Remote (Cloud)
- Every 16 invocations reset the history to allow model to recover from network spikes

![Graph of Average Time (MS) Per Invocation vs Problem Size](chart.png)

Fig. 6. Comparison of average invocation latency for local invocation strategy, remote invocation strategy, and offloading scheduler invocation strategy.
Naive Offloading Scheduler Power

Average Power (uJ) Per Invocation vs Problem Size

- Local Only Scheduler
- Remote Only Scheduler
- Naive Scheduler
Concluding Remarks

NanoLambda Contributions:

- Power & Latency Savings
- Ease of development
- Reprogrammability
- Portability
- Security
Thank you!

Authors: Gareth George (Presentor), Fatih Bakir, Rich Wolski, Chandra Krintz

Contact: gareth@ucsb.edu

'Graphics: Flaticon.com'. This presentation has been designed using resources from Flaticon.com