NanoLambda : FaaS at All Resource scales for IoT

Gareth George, Fatih Bakir, Rich Wolski, Chandra Krintz

UC Santa Barbara

Background

- IoT devices are increasingly prevalent producers of data
- Writing code for processing data on IoT devices remains a challenge.
    - Difficult to implement
    - Portability, security and maintainability all are challenges

- Cloud / Edge processing:
    - Easy to program
    - Easy to scale – FaaS
    - Drawback – Cost of sending data, latency etc.

Background

- Function as a service –
  - Easy scalability
  - Event driven, Resource conserving
  - Faster development
- FaaS is limited to linux based resource rich systems.
- This paper tries to provide uniform FaaS environment for highly resource constrained IoT devices.

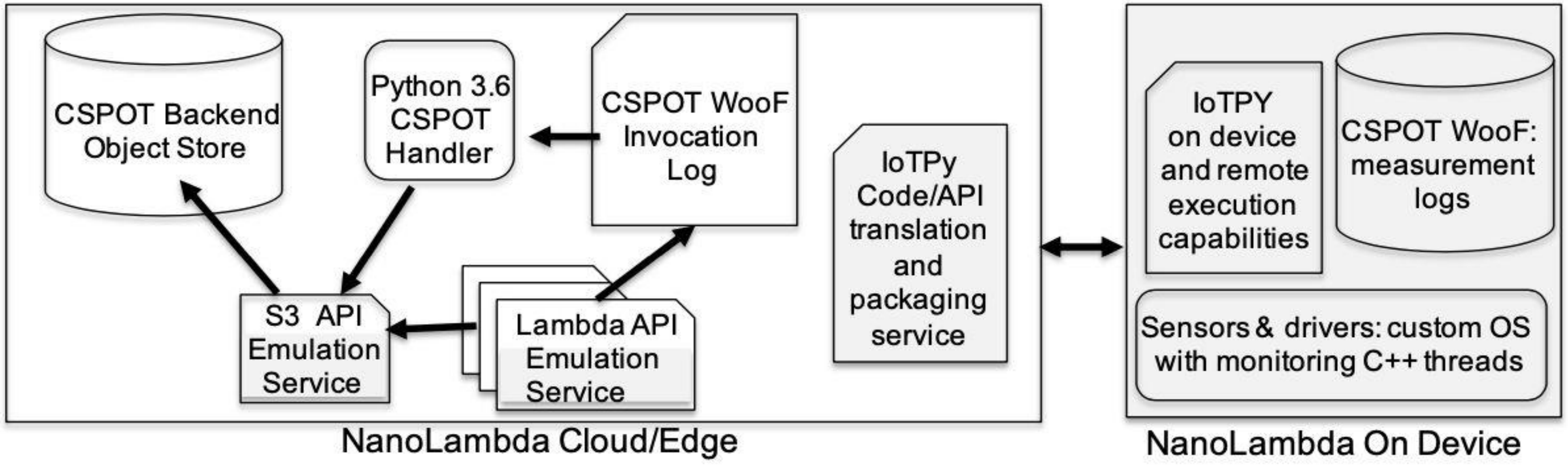
NanoLambda

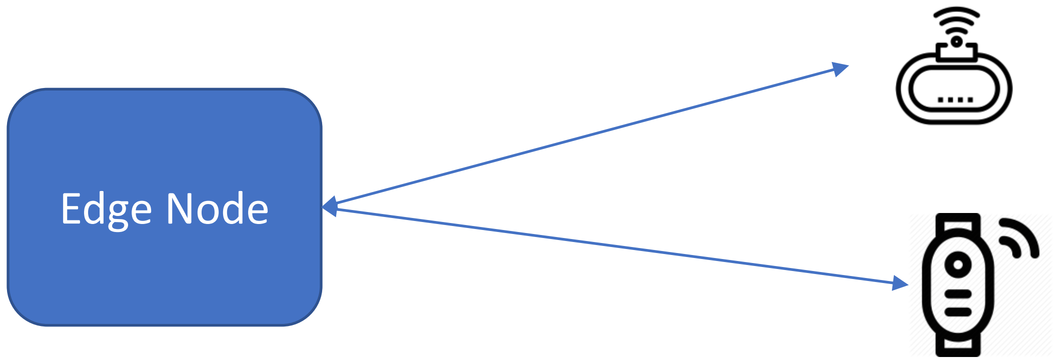
- NanoLambda: platform for running FaaS handlers across all tiers
  - On Device
  - Cloud / Edge
  - Compatible with AWS Lambda
- Goals :
  - Ease of development
  - Portability
  - Small code and memory footprint
  - Security (Isolation)
  - Uniform programming methodology
- At the smallest scales
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

NanoLambda Architecture

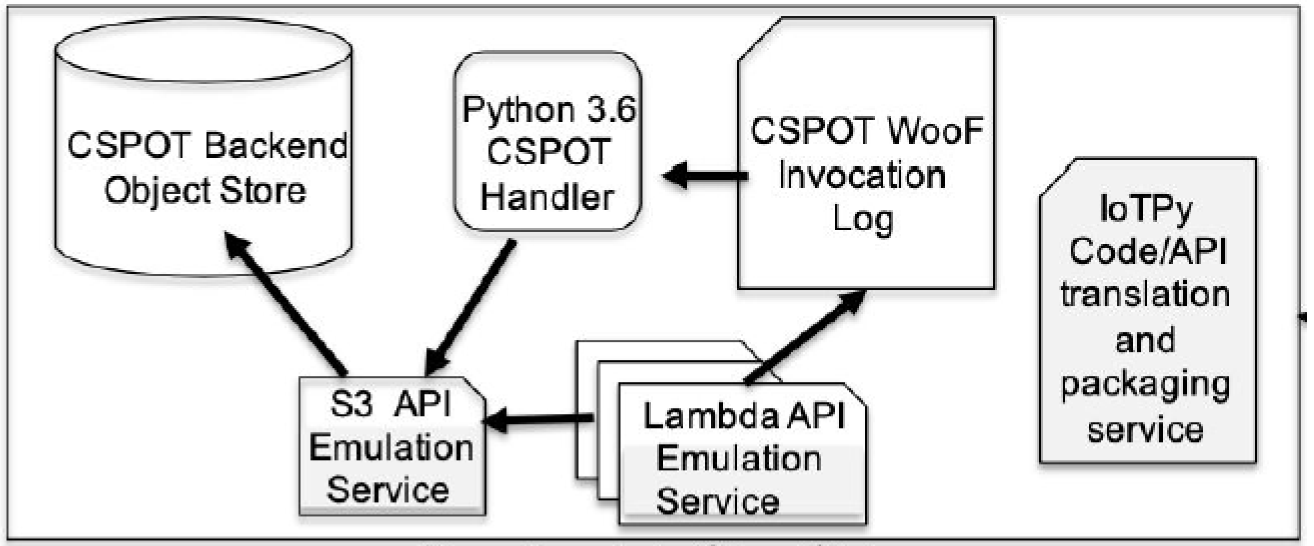
- Based on CSPOT.
  - FaaS deployment model
  - Fault resilient distributed storage
  - Cloud and Edge portability (Only for Linux based systems)
- Comprises of 2 cores systems:
  - NanoLambda Cloud/Edge
  - NanoLambda On Device

NanoLambda Architecture


CSPOT Backend Object Store
Python 3.6 CSPOT Handler
CSPOT WooF Invocation Log
IoTPy Code/API translation and packaging service
S3 API Emulation Service
Lambda API Emulation Service
NanoLambda Cloud/Edge
IoTPY on device and remote execution capabilities
CSPOT WooF: measurement logs
Sensors & drivers: custom OS with monitoring C++ threads
NanoLambda On Device



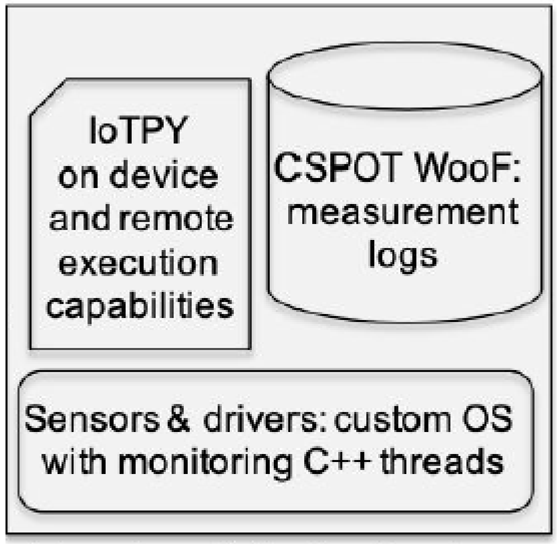
NanoLambda Cloud/Edge


CSPOT Backend Object Store
Python 3.6 CSPOT Handler
CSPOT WooF Invocation Log
IoTPy Code/API translation and packaging service
S3 API Emulation Service
Lambda API Emulation Service

NanoLambda Cloud/Edge

- Service provides two REST API servers offering
  - S3 emulation
  - A FaaS service that deploys functions written for AWS Lambda
- Built with CSPOT
  - 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



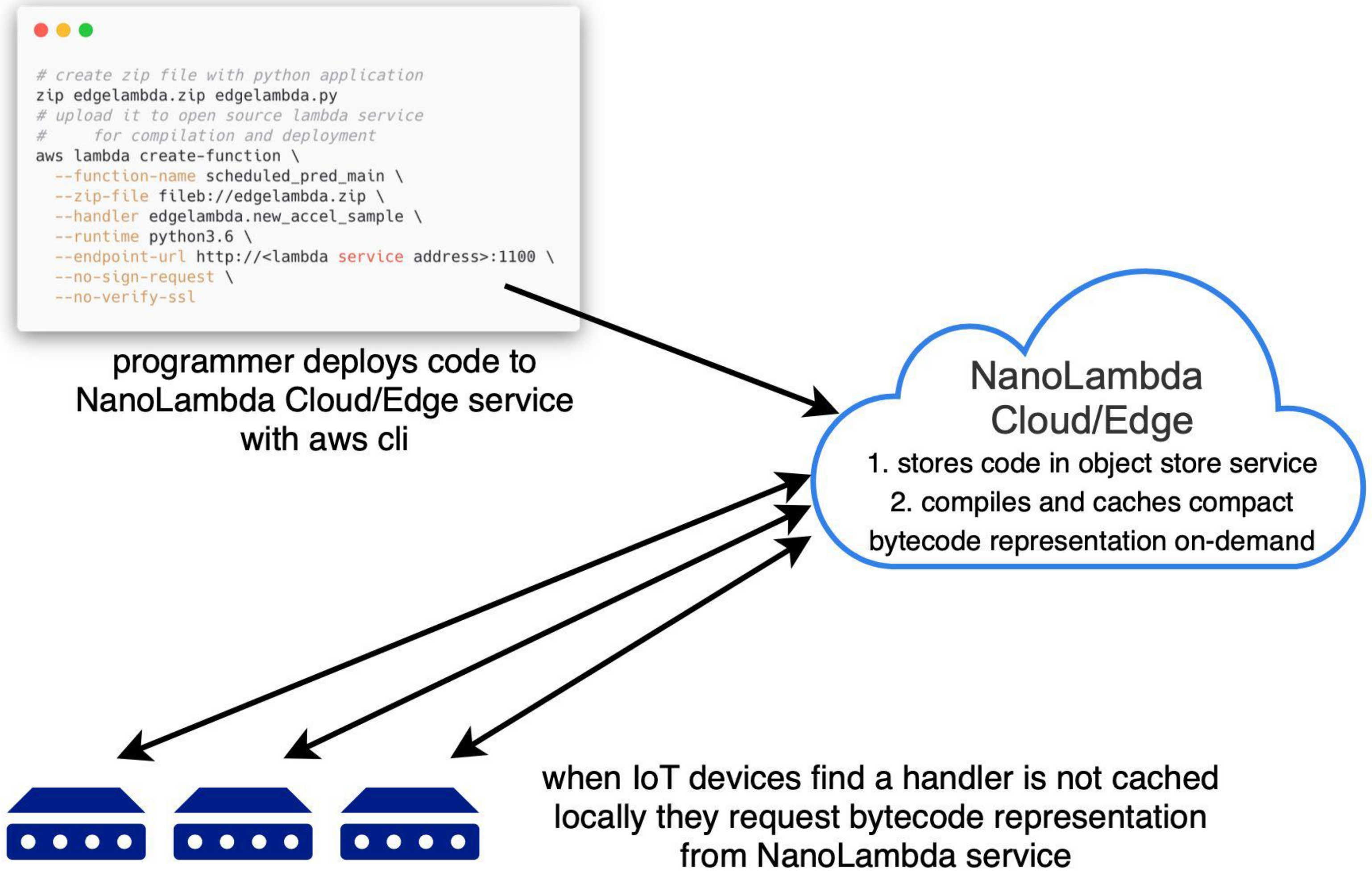
NanoLambda On Device

- 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

IoTPy

- Extremely lightweight Python interpreter built from the ground up with embedding in mind.
- 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
- Security
  - Python VM provides memory protection and container-like isolation

Deploying 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>:1100 \
  --no-sign-request \
  --no-verify-ssl
```

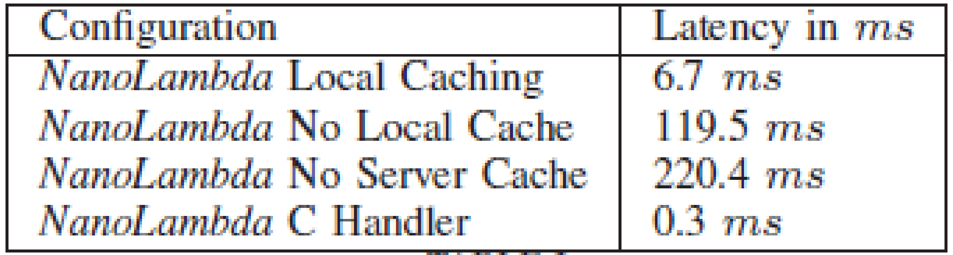
programmer deploys code to NanoLambda Cloud/Edge service with aws cli

NanoLambda Cloud/Edge

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

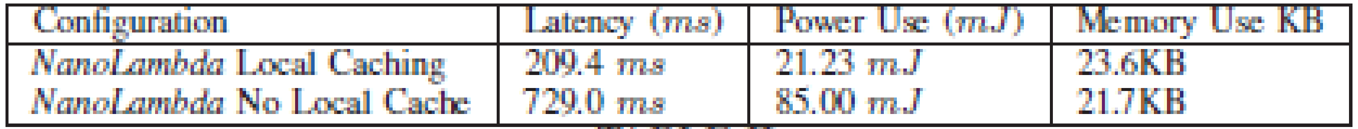
when IoT devices find a handler is not cached locally they request bytecode representation from NanoLambda service

Evaluation



Configuration	Latency in ms
NanoLambda Local Caching	6.7 ms
NanoLambda No Local Cache	119.5 ms
NanoLambda No Server Cache	220.4 ms
NanoLambda C Handler	0.3 ms

TABLE I
AVERAGE LATENCY FOR 100 ITERATIONS OF THE HANDLER FUNCTION



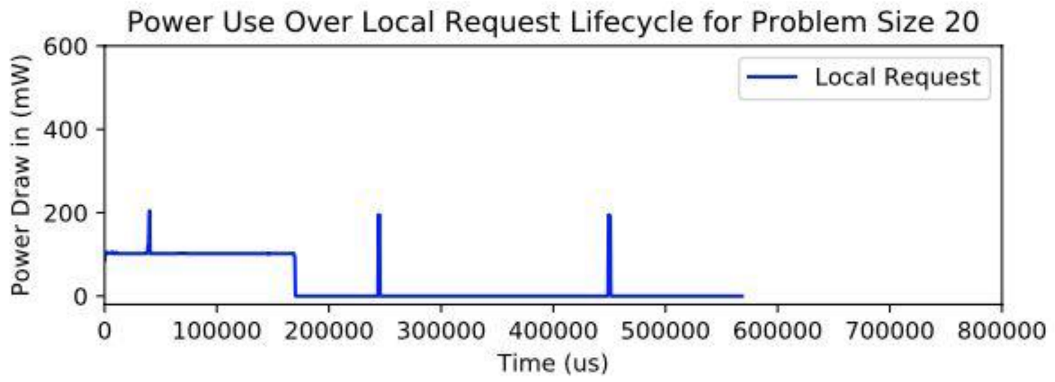
Configuration	Latency (ms)	Power Use (mJ)	Memory Use KB
NanoLambda Local Caching	209.4 ms	21.23 mJ	23.6KB
NanoLambda No Local Cache	729.0 ms	85.00 mJ	21.7KB

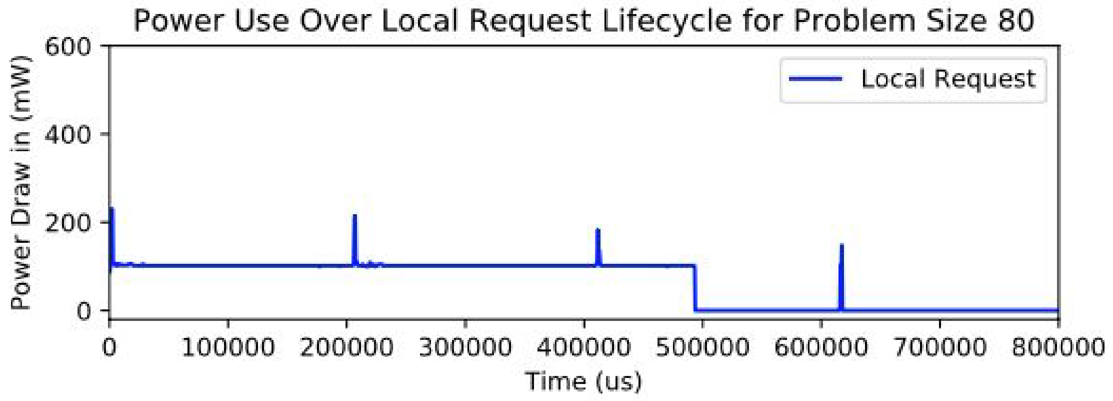
TABLE II
AVERAGE TIMES/POWER CONSUMPTION OVER 100 ITERATIONS OF THE HANDLER FUNCTION ON THE CC3220SF MICROCONTROLLER WITH A KS PROBLEM SIZE OF 32 IN EACH CONFIGURATION.

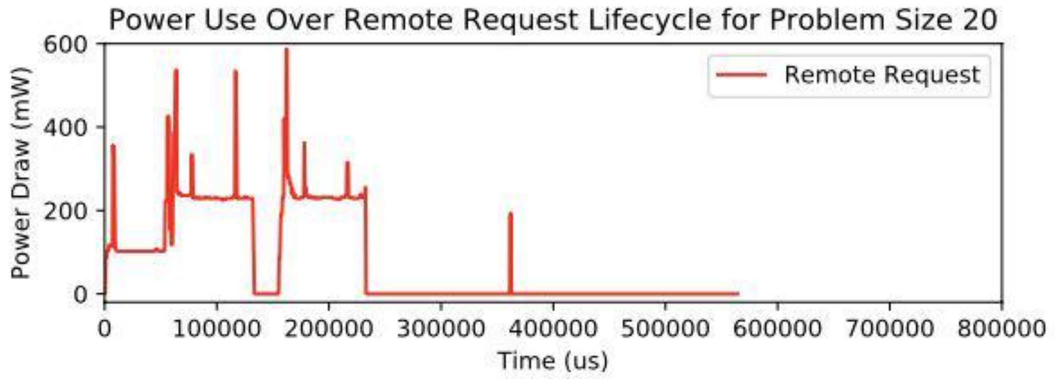
Predictive Maintenance Application

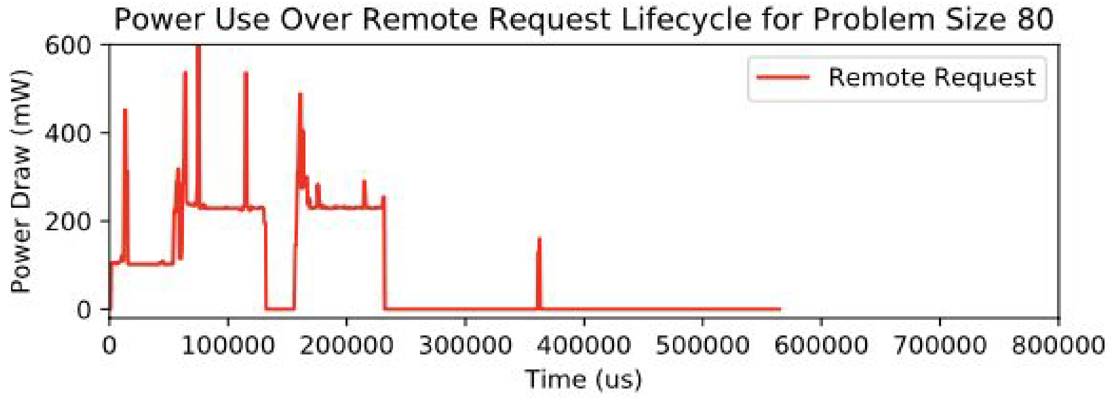
- Predictive Maintenance is a technique using sensors to detect part failure
- We examine failure detection in motors using accelerometers
- Setup:
  - Accelerometer attached to a motor reads vibration magnitude 5 times a second
  - Data is appended to a WooF for persistence, a history of 32 records 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

Evaluation


Power Use Over Local Request Lifecycle for Problem Size 20
Local Request
Power Draw in (mW)
Time (us)
0
200
400
600
0
100000
200000
300000
400000
500000
600000
700000
800000


Power Use Over Local Request Lifecycle for Problem Size 80
Local Request
Power Draw in (mW)
Time (us)
0
200
400
600
0
100000
200000
300000
400000
500000
600000
700000
800000


Power Use Over Remote Request Lifecycle for Problem Size 20
Remote Request
Power Draw (mW)
Time (us)
0
200
400
600
0
100000
200000
300000
400000
500000
600000
700000
800000


Power Use Over Remote Request Lifecycle for Problem Size 80
Remote Request
Power Draw (mW)
Time (us)
0
200
400
600
0
100000
200000
300000
400000
500000
600000
700000
800000

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

Naïve scheduler

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

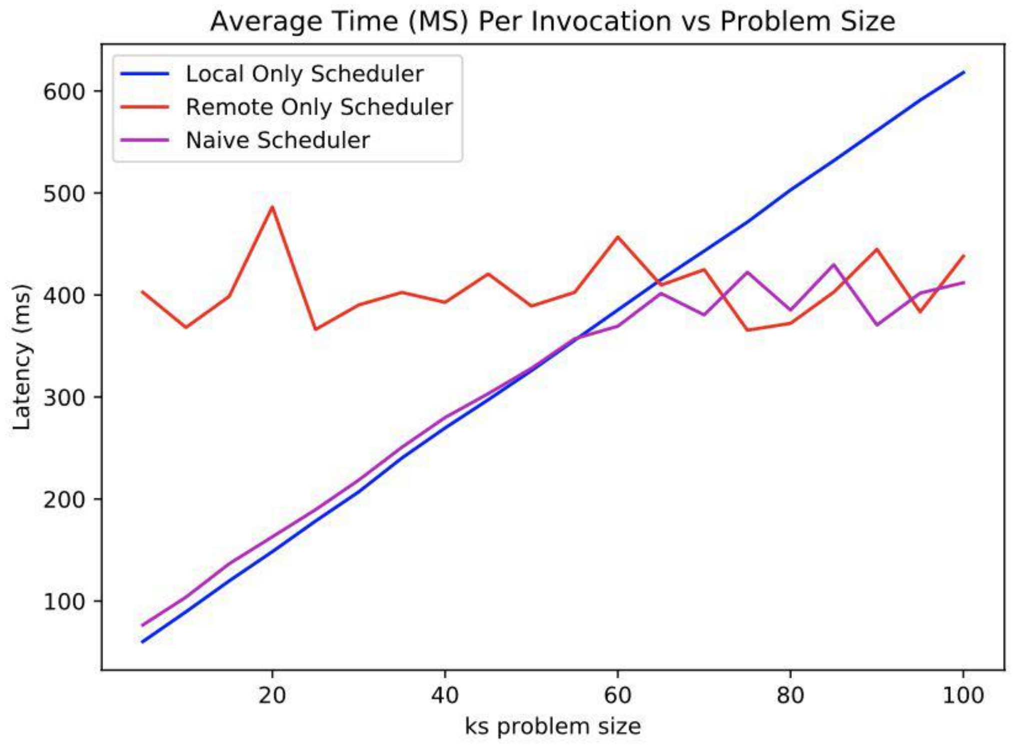


Fig. 6. Comparison of average invocation latency for local invocation strategy, remote invocation strategy, and offloading scheduler invocation strategy.

Discussion

- Each handler is run in a separate Python VM, common packages may be dynamically linked? – Lightweight Isolation
- Support for Python packages.
- Authentication and authorization