User-driven in-network computing at the (IoT) edge

Jörg Ott
Teemu Kärkkäinen

22 November 2019

https://www.cm.in.tum.de/
Context

- Internet of Things
  - Specific device capacities rather than just generic compute power
  - Resources not always easy to scale

- Mobile users
  - Location dependencies rather than “arbitrary” function placement as a function of RTT
  - Local orchestration
  - Responsibility in on mobile devices

- Decomposition
  - Reusable – possibly stateless – functions
  - Fine granularity
  - Dynamic instantiation of processing graphs (DAGs)
Two models to provisioning

• Cloud-driven operation
  • CDN-style: Functions being pushed from the cloud towards the user
  • Doesn’t change the fundamental nature of centralized operation

• User-/Device-driven operation
  • Functions are received from and invoked by the user on demand
User-driven model

- Searching for devices in the vicinity
  - Access points, cell towers, embedded systems with computing power
  - Sensors and actuators
- Discovery and service / function identification
- Service composition by combining functions from devices
- Mobile code execution by pushing functions to devices
Two Examples

1. Lua-based mobile code execution
2. Trigger-action framework leveraging Bluetooth Low Energy Beacons for networking

Commonalities
• Client-driven
• Microcontrollers
• Broadcast networks with strictly local discovery
• (Extension via Internet feasible but not yet integrated)
1. Lua-based Mobile Code Execution

• Instance of a mobile, pervasive computing environment

Node architecture
Basics and operation

- Lua Process VMs: Generic execution platform
- Sensing + actuation hardware: Node-specific capabilities

- Function properties: node capabilities
  - Nodes beacon their capabilities + rendezvous information (= SSID)
  - Functions contain metadata expressing dependencies
  - 2-stage matching

- Mobile node as orchestrator
  - Picks devices
  - Transfers mobile code – instantiation governed by the executing node
  - Collects results

- Different operation modes for code
  - Pull for one-time operations
  - Push for repeating readings
• Some prototype observations (ESP32)
  • Discovery and code transfer dominate execution time
    • Can be amortized across multiple scripts
    • BLE + Wi-Fi efficiency have an impact, so does device density
  • > 100 concurrent clients with reasonable tasks feasible
2. Distributed Trigger-Action Framework

- Flexibly programmable smart environments
- Distributed variant of IF-THIS-THEN-THAT (IFTTT)


© 2019 Jörg Ott
Basics and operation

- Model comprising trigger and actions
- Flexibly combined by program logic as minimal mobile code

- Function properties: (Type ID, instance ID) | (Definition ID)
  - Drivers have custom APIs, need to ensure matching signals
  - Metadata messages to announce capabilities

- BLE beacons as a bus system
  - To discover nearby devices
  - To learn about system capabilities
  - To spread rules
  - To distribute signals and thus cause actions

- Extreme case: Moving only computation, no data
  - Minimal data conveyed implicitly in the data
  - Larger data volumes could use auxiliary communication channels
Protocol messages

- Triples: (type, length, type-specific part)
- Trigger / action definitions
  - Definition ID | Type ID | Instance | Type-specific
  - e.g. (red, green, blue)
- Rule definitions
  - Rule ID | Trigger ID | Action ID
- Trigger signals
  - Type | Seq # | Type-specific
  - list of trigger IDs | Bloom filter
- Descriptor
  - Device metadata
Basics and operation (2)
In-network compute operation

1. Function properties
2. Discovery
3. Choice / Placement
4. Orchestration
5. Execution

Code snippet:

```c
int f(int a, int b);
int g(char *c1, char *c2);
char *h(float e);
```

Diagram:

- **f**
  - **g**
  - **h**

**Diagram Diagram:**

- **f**
  - **g**
  - **h**
In-network compute operation

1. Function properties

2. Discovery

3. Choice / Placement

4. Orchestration

5. Execution

Identification
- Name-based (type [,instance])
- Attribute-based
- Implicit (w/ mobile code)

Parameterization: input, output
- Implicit

Requirements / dependencies
- Implicit
- Dedicated interpreter
In-network compute operation

1. Function properties
2. Discovery
3. Choice / Placement
4. Orchestration
5. Execution

Many flavors of service discovery
- Broadcasting / multicasting
- Anycasting
- Directories
- Function / service routing
  - After mapping
  - Named-based

Broadcast network
- Beaconing
- Probing
In-network compute operation

1. Function properties
2. Discovery
3. Choice / Placement
4. Orchestration
5. Execution

Different scopes
- Network-wide
- Regional
- Local

Orchestrator vs. client

Resource consent

Client-driven
- Discovery-based choice
- Function invocation
- Code instantiation
In-network compute operation

1. Function properties
2. Discovery
3. Choice / Placement
4. Orchestration
5. Execution

Different scopes
- Network-wide
- Regional
- Local

Orchestrator vs. client
Degree of self-orchestration

Client-driven
- Construction of a process pipe
- Explicit by arranging functions
- Implicit via a bus
In-network compute operation

1. Function properties
2. Discovery
3. Choice / Placement
4. Orchestration
5. Execution

Execution of functions
- “Server” instances waiting for calls
  - Continuously running
  - Dynamically instantiated

Data flow
- Point-to-point transport
- Encapsulated in beacons

Program flow
- Via orchestrator call sequence
- Via addresses in beacons
Conclusion

• In-network computing for broadcast networks
  • Compute, storage, and networking in each node
  • Beyond a distributed system as network complexity grows
  • Different levels of abstraction and expressiveness
  • Even small code snippets may suffice

Two meta aspects = challenges

• Pushing control into the network
  • Moving away from a central coordinator constantly in charge
  • Autonomous in-network operation of program logic

• Abstracting composability via API signatures
  • Which outputs can connect to which inputs
  • Need more than a Unix or packet pipe model
  • Data + metadata

© 2019 Jörg Ott