Enabling Synergy in IoT
Platform to Service and Beyond

Michael P Andersen
Gabe Fierro
David E. Culler

*Electrical Engineering and Computer Sciences, University of California, Berkeley*
<table>
<thead>
<tr>
<th>“Wide-where”</th>
<th>“Local-where”</th>
<th>“Person-where”</th>
<th>Firmware</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trust and privacy</td>
<td>Secure transport and discovery</td>
<td>Interactive environments</td>
<td>Programming model</td>
<td>Ubiquitous sensing and control</td>
</tr>
<tr>
<td>How to manage this</td>
<td>Personalized comfort</td>
<td>Wearables</td>
<td>How to prototype</td>
<td>Cheap hardware</td>
</tr>
<tr>
<td>Ubiquitous sensing and control</td>
<td>Personal devices</td>
<td>Smart rooms, cars</td>
<td>What’s the right API/OS?</td>
<td></td>
</tr>
</tbody>
</table>
IoT Design Principles

**Extensibility**
- Platforms should support modular changes, updates, upgrades
- Design for a wide range of possible use cases and extensions

**Adaptability**
- Ability to detect and react to new scenarios, devices, applications, usage patterns, interfaces and contexts

**Unattended Operation**
- At IoT scale, humans cannot be involved in every decision/configuration
- Devices/services must discover each other and learn how to interact
Goal of Talk

- These layers cannot be developed in isolation
  - Must *synergize*

- Establish a set of **design principles** for each of the layers in an IoT ecosystem

- Explore construction of full-stack IoT platform that illustrates these principles
IoT Synergy Stack

Local-“where”
- CQBS
- BOSSWAVE

Wide-“where”

Person-“where”
- SVCD: Service Discovery

Firmware
- C++ Userland
- Lua Userland

Firmware
- Synergy Kernel

Hardware
- Storm + Firestorm
Hardware

- CQBS
- BOSSWAVE
- Gateway
- SVCD: Service Discovery
- C++ Userland
- Lua Userland
- Synergy Kernel
- Storm + Firestorm

Extensibility:
- Facilitate reuse of available sensors/actuators

Adaptability:
Unify communication modes:
- Device to device
- Device to Internet
- Device to person

Unattended Assembly:
- Low power
Hardware: Storm module

- WSN mote-class platform:
  - Low-power (2.3μA idle)
  - Powerful processor (Cortex M4)
  - Distinct 802.15.4 radio

- Easily integrated into carrier:
  - Single-sided, solder-on
  - Castellated edges
  - Organized buses

- Runs a TinyOS port
Hardware: Firestorm carrier

- Arduino-header compliant
  - Tap into Maker movement
- Communication is central:
  - Cortex M0 w/ BLE SoC
  - Firmware can use both radios
- Multiple power domains w/ gating:
  - Support variety of power requirements for addons
  - Retain energy efficiency
  - 9.3 uA idle
Hardware: IoT Design Principles

Extensibility
- Arduino form factor: Preserve ability to rapidly innovate
- Substantial RAM/ROM: fewer constraints on code, data

Adaptability
- BLE great for device-phone, device-person
- 802.15.4 great for device-device
- MCUs can cooperate

Unattended Assembly
- Low power: How to achieve low power and extensibility/adaptability?
- Multiple power domains: low idle current, but can also drive up to 800mA
  - Arduino Zero: 1.5mA idle current because of USB debugging
- Low-power hardware is naturally asynchronous
- How to design software architecture + choose programming paradigm to complement this?
- Extension of TinyOS uses MPU to implement kernel/userland separation
Handling Asynchrony

- **Closures**
  - Callback close to invocation
  - `async/await`: pseudo-synchronous code

- **Kernel support**
  - Place event loop in kernel
  - ABI syscalls receive closure that gets scheduled
  - Syscall event handlers executed cooperatively by application

```plaintext
function sleep (t)
  cord.await(storm.os.invokeLater,
             t*storm.os.MILLISECOND)
end

function write_i2c_reg(addr, val)
  cord.await(storm.i2c.write, ...)
  cord.await(storm.i2c.write, ...)
end

write_i2c_reg(PAR_A, VAL_A)
sleep(200)
write_i2c_reg(PAR_B, VAL_B)
sleep(50)
...
```
Lua Userland

- Great for prototyping
- Interactive development on the device
- Replace symbols, variables while program is running
- Code dissemination over network
- High resource usage: all code gets moved to RAM for execution
- Less predictability: garbage collection

- “Active Networks” routing protocol study
- Upload pattern generator, reporting code over network, adjust routing tables
- All in Lua using syscalls
C++ Userland

- Better resource usage (C++11 closures)
- More deterministic, predictable behavior
- Larger production apps
- (Right) smart, networked chair:
  - Multiple sensors
  - Flash filesystem
  - Control logic
  - Heating/Cooling
Firmware: IoT Design Principles

Extensibility
- Develop new syscalls, new userlands

Adaptability
- Choose userland appropriate for application
- Lua: prototyping, C++: production

Unattended Assembly
- Asynchronous programming: easier to capture complex interactions, event-driven logic
- Stable software despite many moving parts
- Works in tandem with asynchronous nature of low-power hardware

Check out IPSN 2016 program for in-depth technical detail on this platform (HW + FW)
Person-where

Adaptability
- Provide for multiple communication usage patterns
- Device/Phone/Person

Unattended Assembly
- Self-describing services
- Discover required functionality automatically
- Do this without human/special app

Extensibility
- Service definitions are updatable
Person-where

CQBS  BOSSWAVE

Gateway

SVCD: Service Discovery

C++ Userland  Lua Userland

Synergy Kernel

Storm + Firestorm

Approach

SVCD: Self-describing device services using brokerless pubsub symmetric over BLE/15.4
SVCD: Synergy Discovery Service

Service Descriptions
- Manifest entry on Github
- Translation to machine-readable or human-readable

- Devices can discover how to use each other
- Single phone app for all services

```
"pm.storm.svc.fsSensors": {
    "id": "0x300f",
    "name": "FireStorm sensing profile",
    "desc": "Sensing profile for FireStorm",
    "attributes": {
        "pm.storm.attr.fsSensors.temperature": {
            "id": "0x401b",
            "name": "Temperature Reading",
            "format": [
                ["s8", "C", "Temperature in Celsius"]
            ]
        },
        "pm.storm.attr.fsSensors.occupancy": {
            ...
        }
    }
}
```
SVCD: Synergy Discovery Service

SVCD Library

- Unified API for defining/exposing/invoking services over both networks
  - advertise(), subscribe(), publish(), etc

- Client interactions appear identical

- Permits range of interactions:
  - E.g. Phone discovers services, creates bindings, then leaves

- About 250 lines of Lua code
Example: Keep Me Comfortable

Desired Temperature: 72°F

- Retrofit a space heater with a Firestorm and Arduino relay shield
- Implement a setpoint service using SVCD
- Setpoint service listens for temperature service advertisements, subscribes
- Turns heater on/off based on average temperature, configured setpoint

while avg_temp() < setpoint do
  storm.io.set(storm.io.GPIO, 1)
End
storm.io.set(storm.io.GPIO, 0)
Person-where: IoT Design Principles

**Extensibility**
- Update manifest of recognized services
- Easy to implement, define new services

**Adaptability**
- Symmetric over BLE/15.4
- Single service description can adapt as users, devices change interaction patterns, applications

**Unattended Assembly**
- Services are self-describing
- Ensembles can form themselves by subscribing/publishing to devices that match expected interface
Adaptability
- Avoid “hardcoded” interactions
- Discovery of relationships based on metadata

Extensibility
- Update, augment metadata as needs or deployment changes

Unattended Operation
- Self-describing devices/services
- Self-assembling applications
- Gracefully react to change
Device Representation

- Each device/service **registers** its capabilities with a **broker**

- Registration contains **metadata** to describe **context**
  - Owner, location
  - Data type, device model
  - Software version
  - Building exposure
  - etc

- Device or driver sends **metadata updates** to broker

---

**Capabilities**

- On/Off
- Brightness
- Hue

<table>
<thead>
<tr>
<th>Point/Command</th>
<th>Light State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location/Desk</td>
<td>Gabe Desk</td>
</tr>
<tr>
<td>Location/Room</td>
<td>410</td>
</tr>
<tr>
<td>Location/Building</td>
<td>Soda Hall</td>
</tr>
<tr>
<td>Device/Model</td>
<td>Philips Hue</td>
</tr>
<tr>
<td>Device/Firmware</td>
<td>v1.2</td>
</tr>
<tr>
<td>Device/Owner</td>
<td>Gabe Fierro</td>
</tr>
</tbody>
</table>
Query-Based Syndication

- Discovery, subscription through **queries** over metadata
- Queries define **relationships**
- Broker forwards updates from devices / services that match

**Application**

Location/Room in [410, 411, 415] AND Location/Building = “Soda” AND Point/Type = “Sensor” AND Point/Sensor = “Temperature”
Query-Based Syndication

Application
Location/Room in [410, 411, 415] AND
Location/Building = “Soda” AND
Point/Type = “Sensor” AND
Point/Sensor = “Temperature”

- Discovery, subscription through queries over metadata
- Queries define relationships
- Broker forwards updates from devices / services that match
Example: Reactive Environment

- Connect smart chairs to the building’s HVAC system
- Human-in-the-loop
- Application discovers chairs in each HVAC zone, adjusts temperature in BMS
- Chairs move, BMS changes: use query-based syndication to maintain consistent view
Local-where: IoT Design Principles

**Extensibility**
- Granular device/service descriptions permits new device types/functions
- Allow metadata schema to evolve

**Adaptability**
- Continuous query-based syndication
- Applications and devices can maintain consistent view of other devices and services

**Unattended Assembly**
- Broker informs subscribers of changes in metadata
- Services, devices can adapt their operation to the set of available resources
- No need for human intervention
Wide-where

Challenge: scalability in IoT involves the administrative load devices place on people

Extensibility
- Need a simple, scalable permissions model

Unattended Operation
- Devices, services must verify identity, data integrity

Adaptability
- Provide these features for many use cases
Identity

- A scalable definition of identity:
  - Easy to verify
  - Easy to create

- SSL certificates?
  - Expensive (though this is getting better)
  - Require cooperation of a hierarchy of certificate authorities

- BOSSWAVE solution:
  - Define identity with ECDSA public key
  - Person/device/service with the private key is equivalent to the entity
  - Identity can be verified using signatures -- no third parties!
Permissions and Trust

- BOSSWAVE permissions mirror how permission work in the real world
  - Peer to peer requesting and granting with delegation to trusted intermediaries
  - Each arrow is a **declaration of trust (DOT)**
  - \((K_{\text{from}}, K_{\text{to}}, URI, Permission)\) -- signed with granter’s key
  - Messages in BOSSWAVE contain **standalone proofs** of permissions
  - Proof contains a **chain of DOTs**
  - Unforgeable, self-verifying
Wide-where: IoT Design Principles

**Extensibility**
- Simple permissions model mirrors how permissions work in real world
- Building blocks to define more complex trust models

**Adaptability**
- Secure over the wide area
- URI + Permissions maps onto many application architectures

**Unattended Assembly**
- Verification of identity, permissions and trust can be done without the cooperation of third parties
Conclusion

- Established design principles for the IoT
  - Extensibility
  - Adaptability
  - Unattended Operation
- Full-stack IoT solution across these principles
- Lots of information!
  - Talk/email me if you want to know more
Thank You!

Questions?

gtfierro@eecs.berkeley.edu
CQBS Walkthrough

Subscriber

Subscribe

SELECT * WHERE
Metadata/Location/Room = 410 AND
Metadata/Point/Type = Sensor AND
Metadata/Point/Measure = Temperature

Initial Results

{UUID: 9b5155d..., <Metadata>, <Readings>}

Updated Results

{New: {UUID: b246949..., <Metadata>, <Readings>}}

Updated Results

{Del: [9b5155d...]}
BOSSWAVE Messages

**Sender**
- App sends msg
- Attach credentials
- Sign message
- TX to URI’s broker

**Broker**
- Verify signature
- Verify credentials
- Queue for TX

**Receiver**
- App receives msg
- Locate handler
- Verify credentials
- Verify signature
Discovery Mechanisms

Current Platforms
- Pairwise master/client binding, or
- Service invocation in local area network
- N devices, N phone apps

Key Features
- Devices should be self-describing
- Descriptions should communicate capabilities
- Apps should determine context of discovered services
- Know when devices change state
SVCD API

- Cross-network service description, utilization
- 250 lines of Lua code
- Translates to BLE and 802.15.4 syscalls
- Service discovery/invocation, publish/subscribe are naturally asynchronous
  - Clean programming model using async userland

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>init()</td>
<td>Initialize sockets, begin advertisements</td>
</tr>
<tr>
<td>add_service(svc_id)</td>
<td>Add new service</td>
</tr>
<tr>
<td>add_attr(svc_id, attr_id, write_fn)</td>
<td>Attach callback function for writes on attribute</td>
</tr>
<tr>
<td>notify(svc_id, attr_id, value)</td>
<td>Notify subscribers of new value</td>
</tr>
<tr>
<td>subscribe(targetip, svc_id, attr_id, on_notify)</td>
<td>Subscribe to changes on attribute on remote service</td>
</tr>
<tr>
<td>unsubscribe(subscription_id)</td>
<td>Unsubscribe</td>
</tr>
<tr>
<td>advert_received(payload, src_ip)</td>
<td>Triggered on a heard advertisement</td>
</tr>
</tbody>
</table>
Gateway

- Transition from small scale to larger area
- Built PC-less border router using Arduino Ethernet shield, 6in4 tunnelling

CQBS | BOSSWAVE

SVCD: Service Discovery

C++ Userland | Lua Userland

Synergy Kernel

Storm + Firestorm
Device Representation

- Representation must support granular discovery of relationships
  - Decompose device into streams
- Not just names of interfaces
  - AllJoyn Lighting Interface
  - Device functionality ≠ interface
- Sensor/actuator channels described with metadata
- Descriptions are extensible
  - How to describe new device?
  - How to describe new functionality?
- Descriptions can change
  - Things are replaced, repaired, upgraded, installed, moved
  - Things change owners, add functionality, etc
Discovery and Communication

- Use queries over metadata for discovery
- Subscribe to streams that match the query
- Query describes desired relationships

**Upshot:** Relationships are explicit

“The dimmable lights in the room I am in”

Devices are implicit, so they are free to change
BOSSWAVE Topology

Client (any lang)  \[\text{Out Of Band comms}\]  BOSSWAVE  \[\text{BW native}\]  Designated router  \[\text{State}\]  BOSSWAVE  \[\text{BW native}\]  BOSSWAVE

Trust domain (e.g. same pc)  \[\text{Out Of Band comms}\]  Client (any lang)  \[\text{Trust domain (e.g. same pc)}\]
# Programming Models

- **Sequential C (C++)**
  - Blocking calls w/ spinning CPU is inefficient
  - Interrupt handlers
  - Arduino, mbed model
  - Code-your-own event loop: listen on multiple sockets?

- **Threads (RiotOS)**
  - Suspend during blocking calls
  - Usually overprovision resources (no virtual memory)
  - Small number of threads
  - Active thread memory unavailable to other threads
Handling Asynchrony

- **Split-Phase** *(nesC -- TinyOS)*
  - Allow processor to sleep, transition during blocking calls
  - “Completion” is in different place than “invocation”
  - Global variables, switch statements
  - Bug prone, hard to read
  - No clean demuxing of signals

```c
task next state() {
    switch(state)
    case par_a_write_addr:
        call I2C.write(...);
        state = par_a_write_val;
        break;
    case par_a_write_val:
        call Timer.startOneShot(200);
        state = par_b_write_addr;
        break;
    case par_b_write_addr:
        call I2C.write(...);
        state = par_b_write_val;
        break;
    case par_b_write_val:
        call Timer.startOneShot(50);
        state = ...;
        break;
    ...
}

event void I2C.writeDone(...) {
    post next_state();
}

event void Timer.fired() {
    post next_state();
}
```