IoTivity-Constrained: IoT for tiny devices

Kishen Maloor
Open IoT Summit North America 2017
Overview

- Introduction
- Background in OCF
- Constrained environment characteristics
- IoTivity-Constrained architecture
- Porting IoTivity-Constrained
- Building Applications
Introduction

- Open Connectivity Foundation (OCF)
  - Publish open IoT standards
- IoTivity-Constrained
  - Small foot-print implementation of OCF standards
  - Runs on resource-constrained devices and small OSes
  - Quickly customize to any platform
OCF standards

A brief background
OCF resource model

- RESTful design: Things modeled as resources with properties and methods
- CRUDN operations on resources (GET / OBSERVE, POST, PUT, DELETE)
- OCF roles
  - Server role: Exposes hosted resources
  - Client role: Accesses resources on a server

Resource URI
- rt: Resource Type
- if: Resource Interface
- p: Policy
- n: Resource Name

Properties
OCF “well-known” resources

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Fixed URI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discovery</td>
<td>/oic/res</td>
</tr>
<tr>
<td>Device</td>
<td>/oic/d</td>
</tr>
<tr>
<td>Platform</td>
<td>/oic/p</td>
</tr>
<tr>
<td>Security</td>
<td>/oic/sec/*</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Refer to the OCF Core spec at [https://openconnectivity.org/resources/specifications](https://openconnectivity.org/resources/specifications)
OCF protocols

- Messaging protocol: CoAP (RFC 7252)
- Data model: CBOR (RFC 7049) encoding of OCF payloads
- Security model: DTLS-based authentication, encryption and access control*
- Transport: UDP/IP; being adapted to Bluetooth

*Refer to the OCF Security spec at https://openconnectivity.org/resources/specifications
Resource discovery

Multicast GET coap://224.0.1.187:5683/oic/res

Unicast response

[URI: /a/light; rt = [“oic.r.light”], if = [“oic.if.rw”],
p= discoverable, observable]
GET and PUT requests

Unicast GET coap://192.168.1.1:9000/a/light

Unicast response

[URI: /a/light; state = 0, dim = 0]

Unicast PUT coap://192.168.1.1:9000/a/light
PayLoad: [state=1;dim=50]

Unicast response

Status = Success
OBSERVE and Notify

Unicast GET coap://192.168.1.1:9000/a/light; Observe_option= 0

Unicast response

[URI: /a/light; state = 1, dim = 50]

Notify Observers

[URI: /a/light; state = 0, dim = 0, sequence #: 1]
Constrained environment characteristics
Constrained device classes

- RFC 7228

<table>
<thead>
<tr>
<th>Name</th>
<th>Data size (e.g., RAM)</th>
<th>Code size (e.g., Flash)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class 0, C0</td>
<td>&lt;&lt; 10 KiB</td>
<td>&lt;&lt; 100 KiB</td>
</tr>
<tr>
<td>Class 1, C1</td>
<td>~ 10 KiB</td>
<td>~ 100 KiB</td>
</tr>
<tr>
<td>Class 2, C2</td>
<td>~ 50 KiB</td>
<td>~ 250 KiB</td>
</tr>
</tbody>
</table>

Must accommodate (at a minimum) OS + Network stack + drivers + IoTivity-Constrained application
Hardware

- Low RAM and flash capacity
- Low power CPU with low clock cycle
- Battery powered devices
Software

- Lightweight OS
- No dynamic memory allocation
- Many options (OS, network stack, …) -> API fragmentation
- Execution context design and scheduling strategy
IoTivity-Constrained features

- OCF roles, resource model, methods, data model, protocol and flows
- CoAP Block-wise transfers (RFC 7959)
  - Application pre-configures MTU size for specific device / deployment
  - Reduce buffer allocations in the network and link layers
- OCF security model
  - Onboarding, provisioning and access control*

*Refer to the OCF Security spec at [https://openconnectivity.org/resources/specifications](https://openconnectivity.org/resources/specifications)
IoTivity-Constrained architecture
Architectural goals

- OS-agnostic core
- Abstract interfaces to platform functionality
- Rapid porting to new environments
- Static memory allocation
- Modular and configurable
Architecture

On {Zephyr, Contiki, RIOT, Mynewt, Linux, …}

Embedded IoT Application

APIs

IoTivity-Constrained Core

Abstract Interfaces

- Clock
- PRNG
- Connectivity
- Persistent Storage

IoTivity-Constrained Framework

Ports
- Zephyr
- RIOT
- Contiki
- Linux
- Mynewt

Concrete Implementations of Interfaces
Core block

Interact uniformly with Interface implementations for all ports
Event loop execution

```c
while(...) {
    oc_main_poll()
}
```

Embedd IoT Application

Resource Layer

Messaging

Security

Connectivity

Interrupt events

Callbacks

oc_main_poll(): process all outstanding events
Idle mode and signaling

... // Initialize a semaphore
while (1) {
    oc_clock_time_t next_event = oc_main_poll();
    // next_event is the absolute time of the next scheduled
    // event in clock ticks. Meanwhile, do other tasks
    // or sleep (e.g., wait on semaphore)
... 
// Framework invokes a callback when there is new work
static void signal_event_loop(void) {
    // Wake up the event loop (e.g., signal the semaphore)
}
Porting IoTivity-Constrained
Platform Abstraction

IoTivity-Constrained Core

Abstract Interfaces
- Clock
- PRNG
- Connectivity
- Persistent Storage

IoTivity-Constrained Framework

APIs

Embedded IoT Application

On {Zephyr, Contiki, RIOT, Mynewt, Linux, …}
Zephyr adaptation

IoTivity-Constrained

IoTivity-Constrained Core

Zephyr Port: Adaptation Layer

APIs: Clock, Semaphore, PRNG, Networking, Flash

UDP/IP

802.15.4 Bluetooth Ethernet

Drivers, BSP

Kernel

Hardware Platform

Zephyr RTOS

IoT application
/ Set clock resolution in IoTivity-Constrained’s configuration
/ file: config.h
#define OC_CLOCK_CONF_TICKS_PER_SECOND (...)  
typedef uint64_t oc_clock_time_t; // timestamp field width

// Declared in port/oc_clock.h
// Implement the following functions using the platform/OS’s
// APIs, For eg. on Linux
// using clock_gettime()
void oc_clock_init(void);
oc_clock_time_t oc_clock_time(void);
Connectivity

// Declared in port/oc_connectivity.h
// Implement the following functions using the platform’s
// network stack.
int oc_connectivity_init(void);
void oc_connectivity_shutdown(void);
void oc_send_buffer(oc_message_t *message);
void oc_send_discovery_request(oc_message_t *message);
uint16_t oc_connectivity_get_dtls_port(void);

• oc_message_t contains remote endpoint information (IP/Bluetooth
  address), and a data buffer
Connectivity events

- Capture incoming messages by polling or with blocking wait in a separate context and construct an oc_message_t object
- Message injected into framework for processing via oc_network_event()
- Based on nature of OS or implementation, might require synchronization

```c
void oc_network_event_handler_mutex_init(void);
void oc_network_event_handler_mutex_lock(void);
void oc_network_event_handler_mutex_unlock(void);
```
// Declared in port/prng.h
// Implement the following functions to interact with the
// platform’s PRNG
void oc_random_init(void);
unsigned int oc_random_value(void);
void oc_random_destroy(void);
Persistent storage

// Declared in port/oc_storage.h
// Implement the following functions to interact with the
// platform’s persistent storage
// oc_storage_read/write must implement access to a key-value store

int oc_storage_config(const char *store_ref);
long oc_storage_read(const char *key, uint8_t *buf, size_t size);
long oc_storage_write(const char *key, uint8_t *buf, size_t size);
Building Applications
Application structure

- Implemented in a set of callbacks
  - Initialization (Client / Server)
  - Defining and registering resources (Server)
  - Resource handlers for all supported methods (Server)
  - Response handlers for all requests (Client)
  - Entry point for issuing requests (Client)
- Run event loop in background task
- Framework configuration at build-time (config.h)
main() {
    static const oc_handler_t handler = {
        .init = app_init,
        .signal_event_loop = signal_event_loop,
        .register_resources = register_resources
    };

    oc_main_init(&handler);

    while (1) {
        oc_clock_time_t next_event = oc_main_poll();
    }
}
void app_init(void) {
    oc_init_platform("Intel", NULL, NULL);
    oc_add_device("/oic/d", "oic.d.light", "Lamp", "1.0", "1.0", NULL, NULL);
}

- Populate standard OCF resources (platform / device)
Defining a resource

```c
void register_resources(void) {
    oc_resource_t *res = oc_new_resource("/a/light", 1, 0);
    oc_resource_bind_resource_type(res, "core.light");
    oc_resource_bind_resource_interface(res, OC_IF_R);
    oc_resource_set_default_interface(res, OC_IF_R);
    oc_resource_set_discoverable(res, true);
    oc_resource_set.observable(res, true);
    oc_resource_set_request_handler(res, OC_GET, get_light, NULL);
    oc_add_resource(res);
}
```
bool light_state;
int brightness;
....

static void get_light(oc_request_t *request,
    oc_interface_mask_t interface, ...) {
    // Call oc_get_query_value() to access any uri-query
    oc_rep_start_root_object();
    oc_rep_set_boolean(root, state, light_state);
    oc_rep_set_int(root, brightness_level, brightness);
    oc_rep_end_root_object();
    oc_send_response(request, OC_STATUS_OK);
}
Resource discovery

```c
oc_do_ip_discovery("oic.r.light", &discovery, NULL);
....
oc_server_handle_t light_server;
char light_uri[64];
...
oc_discovery_flags_t discovery(..., const char *uri, ..., oc_server_handle_t *server,
,...) {
    strncpy(light_uri, uri, strlen(uri));
    memcpy(&light_server, server, sizeof(oc_server_handle_t));
    return OC_STOP_DISCOVERY;
    // return OC_CONTINUE_DISCOVERY to review other resources
}```
// Populated in the discovery callback
oc_server_handle_t light_server;
char light_uri[64];
...
oc_do_get(light_uri, &light_server, "unit=cd", &get_light,
LOW_QOS, NULL);
...
void get_light (oc_client_response_t *data) {
  oc_rep_t *rep = data->payload;
  while (rep != NULL) {
    // rep->name contains the key of the key-value pair
    switch (rep->type) {
    case BOOL:
      light_state = rep->value_boolean; break;
    case INT:
      brightness = rep->value_int; break;
    }
    rep = rep->next;
  }
}
Framework configuration

• Set at build-time in a file `config.h`
  • Number of resources
  • Number of payload buffers and maximum payload size
  • Memory pool sizes
  • MTU size (for block-wise transfers)
  • Number of DTLS peers
  • DTLS connection timeout
  • ...
Project configuration

• make [BOARD=<type>] menuconfig
  • Stack size for main thread
  • Network stack options
    • IPv6, UDP, 6LoWPAN, 6LoWPAN_IPHC
    • Number of network contexts (sockets)
    • Number of network RX/TX buffers, data size
    • Bluetooth host options: L2CAP CoC, GATT, master/slave modes
    • Layer 2 options: IEEE 802.15.4, Bluetooth LE, Ethernet
  • PRNG implementation
IPv6 over BLE (IPSP) support

- Transport UDP/IPv6 over BLE L2CAP (RFC 7668)
- Build 6LN with IPv6, 6Lo_IPHC, BT peripheral mode and L2CAP CoC
- Use sample IPSS for connection setup
- Supported on Arduino 101*
- Test communication with Linux (>= 3.16) as master/central

Conclusion
Summary and plans

• Growing interest from community and prospective OCF vendors
• OCF standards compliance; participate in OCF Plugfest
• Motivate definition of constrained device profile in OCF spec
• Investigate special requirements of industrial and healthcare verticals
• Addition of independent, higher-level components

• We welcome your contributions!
Questions?

Source code: https://gerrit.iotivity.org/gerrit/gitweb?p=iotivity-constrained.git

IoTivity mailing list: iotivity-dev@lists.iotivity.org

Kishen Maloor

kishen.maloor@intel.com