Service-Oriented Architecture (SOA) — From First Principles to Bare-Metal on i.MX RT1050

1) What SOA is (and isn't)

Service-Oriented Architecture (SOA) is an architectural style where capabilities are packaged as **services** with explicit **contracts**. A **service** is a unit of functionality (for instance, "provide the last measured outside air temperature" or "set the lamp test state"). Clients discover and invoke services via standardized interfaces and protocols. The core ideas are **loose coupling**, **contract-first design**, **message orientation**, and **composability**.

- **Loose coupling**: Providers and consumers depend on the contract, not internal implementations.
- **Contract**: A versioned specification defining messages, data types, behaviors (requests, responses, notifications), and error codes.
- **Message orientation**: Interactions happen via well-formed messages (over UART, SPI, CAN, or Ethernet), not ad-hoc function calls.
- **Composability**: Larger features are composed from smaller services without breaking encapsulation.

SOA vs. Microservices: Microservices are a deployment style popular in cloud systems (many independently deployable processes). In embedded avionics we often implement SOA **inside a single Line Replaceable Unit (LRU)** or even a single processor using **partitions** or **tasks**, to satisfy tight real-time constraints and certification objectives. SOA's principles—clear service boundaries, versioned contracts, and message semantics—translate directly to bare-metal and Real-Time Operating System (RTOS) environments.

SOA vs. "just drivers and APIs": A driver function such as `uint32_t ADC_Read();` becomes a **service** only if it is **contracted, versioned, message-based, and discoverable**. A service adds: - a stable **contract** (payload schemas, error codes, versions), - **transport independence** (UART today, Ethernet tomorrow), - **observability** (health/status endpoints), - a **compatibility strategy** (forward/backward).

2) Why SOA matters in avionics

Modern avionics stress modularity: **IMA (Integrated Modular Avionics)**, **ARINC 653 (partitioned operating environments)**, **ARINC 664/AFDX (Avionics Full-Duplex Switched Ethernet)**, **FACE (Future Airborne Capability Environment)**, and **MOSA (Modular Open Systems Approach)** all encourage **well-bounded, replaceable**

capabilities. SOA complements this by defining **how** components interact at the logical level:

- Explicit service contracts support **DO-178C (Software Considerations in Airborne Systems and Equipment Certification)** traceability from requirements to tests.
- Deployments can evolve without re-integration churn: a service implementation may change if the **contract** remains compatible.
- **DDS (Data Distribution Service)**, **ARINC 653 ports**, **ARINC 429 words**, or **ARINC 664 virtual links** can serve as transports for the same service contract.

In safety-critical designs—**DAL (Design Assurance Level) A–E**—SOA doesn't remove determinism; it **forces you to design it** (latency budgets, bounded queues, memory caps, worst-case execution time per service).

---

3) SOA building blocks for a microcontroller

On an **i.MX RT1050** bare-metal target, a tiny yet robust SOA framework looks like this:

1. **Service IDs and Versions**
   Assign a numeric **Service ID** and **major.minor** version to each service. Increment **major** for breaking changes; **minor** for backward-compatible additions.

2. **Message Model**

   - **Request/Response** for commands/queries

   - **Publish/Subscribe** (notifications) for sensor streams

   - **Heartbeat/Health** for liveness & status
     Use a **header** (magic, version, type, service ID, verb, payload length, CRC) and a **payload** (packed C struct with fixed endianness and alignment rules).
3. **Transport Adapters**
   The same service rides over **UART**, **CAN**, **I^{2}C**, or **Ethernet**. Start with **UART** on the EVKB; later swap in another adapter without changing service logic.

4. **Service Registry**
   A table mapping **(service_id, verb)** to handler functions. Keeps dispatch deterministic and auditable.

5. **Determinism**
   Fixed-size buffers; bounded allocations; bounded retries/timeouts; back-pressure on publishers; worst-case execution time analysis per verb handler; minimal work in interrupts.

6. **Safety & Security Hooks**
   Parameter range checks; **defensive decoding**; CRC (Cyclic Redundancy Check) on messages; capability masks; and trace logging for **traceability**.

---

4) Generic example (non-avionics): "LED & Temperature Services" over UART

Two services on one EVKB: - **LED Service (ID 0x0001)** with verbs `SET_STATE(ON/OFF)` and `GET_STATE()`. - **Temperature Service (ID 0x0002)** with verb `GET_DEGC()` returning the latest temperature from an on-board or simulated sensor.

A host or another MCU calls these over UART using the shared message format. Swapping UART for Ethernet later does not alter the **service layer**—only the **transport adapter**.

---

5) Avionics use case: "429 Data Concentrator as a Service"

A small LRU acts as a **Data Concentrator**: it **receives ARINC 429 words** from sensors via an **ADK-8582** ARINC-429 interface module and **exposes them as services** over a system bus (we prototype over UART for the lab; the same contract can later ride over ARINC 664/AFDX or DDS).

Provided services: - **ARINC429 Label Service (ID 0x0429)**
- `GET_LAST(label)` → returns the last received 32-bit ARINC 429 word for the requested label.
- `SUBSCRIBE(label, rate_hz)` → optional notifications for that label.
- `GET_STATS()` → counters (words received, parity errors, overruns).

This decouples the **consumer** (e.g., a display computer) from the low-level ARINC-429 details. The consumer just "asks the service."

---

6) Best-practice design rules (embedded SOA)

- **Contracts first**: Write C structs and message headers/verbs **before** coding. Freeze byte order and packing (big-endian or little-endian but fixed) and **never change** casually.
- **Strict bounds**: Define maximum payload size, ring buffers, and queue depths; test at limits.
- **Time budgets**: Assign worst-case execution time to each verb handler; measure using cycle counters.

- **Fail safely**: On decode errors—bad magic, CRC, or version—return a clear error code and discard payload.
- **Version with care**: Add optional fields at the tail; bump **minor**. Reserve **major** for breaking layout changes.
- **Observability**: Implement `GET_HEALTH()` and `GET_VERSION()` for every service.
- **Security hygiene** (aligned to **DO-326A**, Airworthiness Security): validate inputs, authenticate maintenance commands where practical, and support `LOCKOUT/MAINTENANCE` mode.

---

7) Hands-on Lab 1 — A micro-SOA over UART on EVKB-i.MXRT1050 (Bare-Metal, MCUXpresso SDK)

Lab goal

Build a tiny SOA runtime with: - A **UART transport** using **SDK LPUART** drivers. - A **message header**, **CRC-16**, and a **dispatcher**. - Two services: **LED** and **Board Info** (board name, silicon ID). - A simple **client** on the same board (loopback) or a PC via USB-UART.

> Notes:
> • The EVKB routes the debug console to one LPUART instance (commonly `LPUART1`). Use another LPUART (e.g., `LPUART3`) for the service link, or reuse `LPUART1` only after disabling console prints. Pin routing is in **pin_mux.c**.
> • The code mirrors **SDK driver examples** (e.g., `boards/evkbimxrt1050/driver_examples/lpuart/...`): `BOARD_InitHardware()`, `LPUART_Init()`, `LPUART_WriteBlocking()`, `LPUART_ReadBlocking()`.

Message header and CRC (service_protocol.h/.c)

```c
/* service_protocol.h */
#ifndef SERVICE_PROTOCOL_H
#define SERVICE_PROTOCOL_H

#include <stdint.h>
#include <stdbool.h>

#define SVC_MAGIC             (0xA55Au)
#define SVC_VERSION_MAJOR     (1u)
#define SVC_VERSION_MINOR     (0u)

typedef enum {
    SVC_MSG_REQUEST   = 0,
    SVC_MSG_RESPONSE  = 1,
    SVC_MSG_NOTIFY    = 2,
    SVC_MSG_HEARTBEAT= 3
} svc_msg_type_t;
```

```c
typedef struct __attribute__((packed)) {
    uint16_t magic;         /* 0xA55A */
    uint8_t  ver_major;     /* 1 */
    uint8_t  ver_minor;     /* 0 */
    uint8_t  msg_type;      /* svc_msg_type_t */
    uint8_t  reserved;      /* align to 2-byte for verbs */
    uint16_t service_id;    /* e.g., 0x0001 LED, 0x0002 BoardInfo */
    uint16_t verb;          /* e.g., 1=GET, 2=SET */
    uint16_t payload_len;   /* N bytes following this header */
    uint16_t crc16;         /* CRC-16-IBM over header except crc16 + payload *
/
} svc_frame_t;

#define SVC_ID_LED            (0x0001u)
#define SVC_ID_BOARD_INFO     (0x0002u)
#define SVC_ID_ARINC429       (0x0429u)

#define SVC_VERB_GET          (1u)
#define SVC_VERB_SET          (2u)
#define SVC_VERB_GET_VERSION  (100u)
#define SVC_VERB_GET_HEALTH   (101u)

#define SVC_MAX_PAYLOAD       (128u)

uint16_t svc_crc16_ibm(const uint8_t *data, uint32_t len);

#endif /* SERVICE_PROTOCOL_H */

/* service_protocol.c */
#include "service_protocol.h"

uint16_t svc_crc16_ibm(const uint8_t *data, uint32_t len)
{
    /* Polynomial 0xA001 (reversed 0x8005), init 0xFFFF */
    uint16_t crc = 0xFFFFu;
    for (uint32_t i = 0; i < len; ++i) {
        crc ^= (uint16_t)data[i];
        for (uint8_t b = 0; b < 8; ++b) {
            if (crc & 1u) crc = (crc >> 1) ^ 0xA001u;
            else          crc = (crc >> 1);
        }
    }
    return crc;
}
```

UART adapter and dispatcher (service_uart.c)

The initial implementation uses **blocking** I/O for clarity. Later, port to the **interrupt ring-buffer** style used in SDK examples (`lpuart_interrupt.c`, `lpuart_interrupt_rb_transfer.c`) for non-blocking reads with timeouts.

```c
/* service_uart.c */
#include <string.h>
#include "fsl_lpuart.h"
#include "board.h"
#include "clock_config.h"
#include "pin_mux.h"
#include "service_protocol.h"

/* Select your service UART instance here. Avoid the debug console UART. */
#define SERVICE_LPUART          LPUART3
#define SERVICE_LPUART_IRQn     LPUART3_IRQn
#define SERVICE_LPUART_CLK_FREQ  BOARD_DebugConsoleSrcFreq() /* OK for demo *
/

/* Simple error codes */
#define SVC_OK                  (0)
#define SVC_ERR_BAD_HEADER      (-1)
#define SVC_ERR_BAD_CRC         (-2)
#define SVC_ERR_NO_HANDLER      (-3)
#define SVC_ERR_PAYLOAD         (-4)

typedef int (*svc_handler_t)(uint16_t verb, const uint8_t *req, uint16_t req_
len,
                             uint8_t *resp, uint16_t *resp_len);

/* Registry entry */
typedef struct {
    uint16_t service_id;
    svc_handler_t handler;
    uint8_t  ver_major, ver_minor;
} svc_reg_t;

/* Forward declarations for handlers */
static int svc_led_handler(uint16_t verb, const uint8_t *req, uint16_t req_le
n, uint8_t *resp, uint16_t *resp_len);
static int svc_boardinfo_handler(uint16_t verb, const uint8_t *req, uint16_t
req_len, uint8_t *resp, uint16_t *resp_len);

/* Example registry */
static const svc_reg_t g_registry[] = {
    { SVC_ID_LED,        svc_led_handler,       1, 0 },
    { SVC_ID_BOARD_INFO, svc_boardinfo_handler, 1, 0 },
};
```

```c
static status_t uart_send_bytes(const uint8_t *data, size_t len)
{
    LPUART_WriteBlocking(SERVICE_LPUART, data, len);
    return kStatus_Success;
}

static status_t uart_recv_bytes(uint8_t *data, size_t len)
{
    LPUART_ReadBlocking(SERVICE_LPUART, data, len);
    return kStatus_Success;
}

static int svc_send_response(const svc_frame_t *req_hdr, const uint8_t *paylo
ad, uint16_t payload_len)
{
    svc_frame_t hdr = *req_hdr;
    hdr.msg_type    = SVC_MSG_RESPONSE;
    hdr.payload_len= payload_len;
    hdr.crc16       = 0;

    /* Compute CRC over header (except crc16) + payload */
    uint16_t crc;
    uint8_t  temp[sizeof(svc_frame_t)];
    memcpy(temp, &hdr, sizeof(hdr));
    ((svc_frame_t*)temp)->crc16 = 0;
    crc = svc_crc16_ibm(temp, sizeof(hdr)) ^ svc_crc16_ibm(payload, payload_l
en); /* simple combine */
    hdr.crc16 = crc;

    uart_send_bytes((uint8_t*)&hdr, sizeof(hdr));
    if (payload_len) uart_send_bytes(payload, payload_len);
    return SVC_OK;
}

static const svc_reg_t* find_service(uint16_t sid)
{
    for (size_t i = 0; i < sizeof(g_registry)/sizeof(g_registry[0]); ++i) {
        if (g_registry[i].service_id == sid) return &g_registry[i];
    }
    return NULL;
}

int svc_server_poll_once(void)
{
    svc_frame_t hdr;
    if (uart_recv_bytes((uint8_t*)&hdr, sizeof(hdr)) != kStatus_Success) retu
rn SVC_ERR_BAD_HEADER;
```

```c
    if (hdr.magic != SVC_MAGIC || hdr.ver_major != SVC_VERSION_MAJOR) return
SVC_ERR_BAD_HEADER;
    if (hdr.payload_len > SVC_MAX_PAYLOAD) return SVC_ERR_PAYLOAD;

    uint8_t payload[SVC_MAX_PAYLOAD];
    if (hdr.payload_len) {
        if (uart_recv_bytes(payload, hdr.payload_len) != kStatus_Success) ret
urn SVC_ERR_PAYLOAD;
    }

    /* Verify CRC */
    uint16_t crc_calc;
    uint16_t crc_saved = hdr.crc16;
    hdr.crc16 = 0;
    crc_calc = svc_crc16_ibm((uint8_t*)&hdr, sizeof(hdr)) ^ svc_crc16_ibm(pay
load, hdr.payload_len);
    if (crc_calc != crc_saved) return SVC_ERR_BAD_CRC;

    const svc_reg_t *reg = find_service(hdr.service_id);
    uint8_t resp[SVC_MAX_PAYLOAD];
    uint16_t resp_len = 0;
    int rc = SVC_ERR_NO_HANDLER;

    if (reg && hdr.msg_type == SVC_MSG_REQUEST) {
        rc = reg->handler(hdr.verb, payload, hdr.payload_len, resp, &resp_len
);
    }

    /* Minimal error signaling: if handler failed, return zero-length respons
e with same IDs */
    svc_send_response(&hdr, (rc==SVC_OK) ? resp : NULL, (rc==SVC_OK)?resp_len
:0);
    return rc;
}

/* ========== Example service implementations ========== */

typedef struct __attribute__((packed)) {
    uint8_t state; /* 0=off,1=on */
} led_set_req_t;

static int svc_led_handler(uint16_t verb, const uint8_t *req, uint16_t req_le
n, uint8_t *resp, uint16_t *resp_len)
{
    switch (verb) {
    case SVC_VERB_SET:
        if (req_len != sizeof(led_set_req_t)) return SVC_ERR_PAYLOAD;
        if (((const led_set_req_t*)req)->state) {
            /* BOARD_UserLedOn();  <-- implement per board */
        } else {
            /* BOARD_UserLedOff(); */
        }
        *resp_len = 0;
        return SVC_OK;

    case SVC_VERB_GET:
        /* return current LED state */
        resp[0] = 0; /* TODO: read GPIO; for demo, assume 0 */
        *resp_len = 1;
        return SVC_OK;

    case SVC_VERB_GET_VERSION:
        resp[0] = 1; resp[1] = 0; /* major.minor */
        *resp_len = 2;
        return SVC_OK;

    default:
        return SVC_ERR_NO_HANDLER;
    }
}

typedef struct __attribute__((packed)) {
    uint32_t silicon_id;
    uint8_t  board_name[16];
} boardinfo_get_resp_t;

static int svc_boardinfo_handler(uint16_t verb, const uint8_t *req, uint16_t
req_len, uint8_t *resp, uint16_t *resp_len)
{
    (void)req; (void)req_len;
    if (verb == SVC_VERB_GET) {
        boardinfo_get_resp_t r = {0};
        r.silicon_id = 0x1050C7A1; /* example value */
        const char *name = "EVKB-IMXRT1050";
        for (int i = 0; i < 16; ++i) r.board_name[i] = (uint8_t)(name[i] ? na
me[i] : 0);
        memcpy(resp, &r, sizeof(r));
        *resp_len = sizeof(r);
        return SVC_OK;
    } else if (verb == SVC_VERB_GET_VERSION) {
        resp[0] = 1; resp[1] = 0;
        *resp_len = 2;
        return SVC_OK;
    }
    return SVC_ERR_NO_HANDLER;
}
```

Server `main.c` (bare-metal, SDK style)

This mirrors SDK examples that call `BOARD_InitHardware()` and initialize LPUART with `LPUART_Init()`.

```
/* main.c — SOA server */
#include "fsl_common.h"
#include "fsl_lpuart.h"
#include "board.h"
#include "app.h"            /* optional, if you follow SDK example patterns *
/
#include "pin_mux.h"
#include "clock_config.h"
#include "service_protocol.h"

/* From service_uart.c */
int svc_server_poll_once(void);

int main(void)
{
    BOARD_InitHardware(); /* BOARD_ConfigMPU, BOARD_InitBootPins, BOARD_InitB
ootClocks */

    /* Configure SERVICE_LPUART pins in pin_mux.c beforehand (TX/RX pads rout
ed) */

    lpuart_config_t config;
    LPUART_GetDefaultConfig(&config);
    config.baudRate_Bps = 115200U;
    config.enableRx = true;
    config.enableTx = true;
    LPUART_Init(SERVICE_LPUART, &config, SERVICE_LPUART_CLK_FREQ);

    /* Optional: banner */
    const uint8_t hello[] = "SOA server ready\r\n";
    LPUART_WriteBlocking(SERVICE_LPUART, hello, sizeof(hello));

    while (1) {
        (void)svc_server_poll_once(); /* blocks for a frame, handles it, resp
onds */
    }
}
```

Minimal client note

A PC script can frame the same header/payload to exchange with the board. For on-board loopback, instantiate a “client” task on another UART or use the same UART in half-duplex with controlled sequences. Importantly, the client never calls driver APIs directly; it speaks **service messages** only.

8) Hands-on Lab 2 — ARINC-429 "Label Service" via ADK-8582 over UART

Lab goal

Use the EVKB to **query ARINC-429 labels** exposed by an **ADK-8582** module and serve them over the same SOA runtime. The EVKB acts as a **gateway service**:

- **Service ID 0x0429 (ARINC429)**
  - `GET_LAST(label)` → request `{uint8_t label;}`; response `{uint8_t label; uint32_t word; uint8_t parity_ok;}`
  - `GET_STATS()` → RX/TX counters and error metrics.
  - `SUBSCRIBE(label, rate_hz)` → optional notify stream (advanced).

Wiring & UARTs

- Keep **LPUART1** for the debug console (SDK default).
- Assign **LPUART3** for the **SOA link** and **LPUART4** for the **ADK-8582** interface (or swap per hardware needs). Configure pads in **pin_mux.c**.

ARINC-429 adapter stub

Encapsulate the ADK-8582's UART protocol. The SOA layer must not depend on module specifics.

```c
/* arinc429_adapter.h */
#ifndef ARINC429_ADAPTER_H
#define ARINC429_ADAPTER_H
#include <stdint.h>
#include <stdbool.h>

typedef struct {
    uint32_t rx_words;
    uint32_t parity_errors;
    uint32_t overruns;
} arinc429_stats_t;

bool arinc429_init(void);                           /* init UART to ADK-8582 */
bool arinc429_get_last(uint8_t label, uint32_t *word, uint8_t *parity_ok);
bool arinc429_get_stats(arinc429_stats_t *s);

#endif
```

```
/* arinc429_adapter.c — sketch; fill in with ADK-8582 specifics */
#include "fsl_lpuart.h"
#include "board.h"
#include "arinc429_adapter.h"

/* Choose LPUARTx for ADK link; configure pins in pin_mux.c */
#define ADK_LPUART              LPUART4
#define ADK_LPUART_CLK_FREQ     BOARD_DebugConsoleSrcFreq()

bool arinc429_init(void)
{
    lpuart_config_t c;
    LPUART_GetDefaultConfig(&c);
    c.baudRate_Bps = 115200U; /* or as required by ADK-8582 */
    c.enableTx = true; c.enableRx = true;
    LPUART_Init(ADK_LPUART, &c, ADK_LPUART_CLK_FREQ);
    /* Perform any ADK-8582 reset/handshake here */
    return true;
}

bool arinc429_get_last(uint8_t label, uint32_t *word, uint8_t *parity_ok)
{
    /* Example: send 'L' + label; expect [label][w0..w3][parity] */
    uint8_t cmd[2] = { 'L', label };
    LPUART_WriteBlocking(ADK_LPUART, cmd, sizeof(cmd));

    uint8_t rx[6] = {0};
    status_t st = LPUART_ReadBlocking(ADK_LPUART, rx, sizeof(rx));
    if (st != kStatus_Success) return false;

    if (rx[0] != label) return false;
    *word = ((uint32_t)rx[1]) | ((uint32_t)rx[2]<<8) | ((uint32_t)rx[3]<<16)
| ((uint32_t)rx[4]<<24);
    *parity_ok = rx[5];
    return true;
}

bool arinc429_get_stats(arinc429_stats_t *s)
{
    uint8_t cmd = 'S';
    LPUART_WriteBlocking(ADK_LPUART, &cmd, 1);
    uint8_t rx[12];
    if (LPUART_ReadBlocking(ADK_LPUART, rx, sizeof(rx)) != kStatus_Success) r
eturn false;
    s->rx_words = (uint32_t)rx[0] | ((uint32_t)rx[1]<<8) | ((uint32_t)rx[2]<<
16) | ((uint32_t)rx[3]<<24);
    s->parity_errors = (uint32_t)rx[4] | ((uint32_t)rx[5]<<8) | ((uint32_t)rx
[6]<<16) | ((uint32_t)rx[7]<<24);
    s->overruns = (uint32_t)rx[8] | ((uint32_t)rx[9]<<8) | ((uint32_t)rx[10]<
<16) | ((uint32_t)rx[11]<<24);
    return true;
}
```

ARINC-429 service handler (plug into the registry)

```
/* service_arinc429.c */
#include <string.h>
#include "service_protocol.h"
#include "arinc429_adapter.h"

typedef struct __attribute__((packed)) {
    uint8_t label;
} arinc429_get_req_t;

typedef struct __attribute__((packed)) {
    uint8_t  label;
    uint32_t word;
    uint8_t  parity_ok;
} arinc429_get_resp_t;

typedef struct __attribute__((packed)) {
    uint32_t rx_words;
    uint32_t parity_errors;
    uint32_t overruns;
} arinc429_stats_resp_t;

int svc_arinc429_handler(uint16_t verb, const uint8_t *req, uint16_t req_len,
uint8_t *resp, uint16_t *resp_len)
{
    if (verb == SVC_VERB_GET) {
        if (req_len != sizeof(arinc429_get_req_t)) return -1;
        uint8_t label = ((const arinc429_get_req_t*)req)->label;
        arinc429_get_resp_t r = { .label = label, .word = 0, .parity_ok = 0 }
;
        if (!arinc429_get_last(label, &r.word, &r.parity_ok)) return -2;
        memcpy(resp, &r, sizeof(r));
        *resp_len = sizeof(r);
        return 0;
    } else if (verb == SVC_VERB_GET_STATS) {
        arinc429_stats_t s;
        if (!arinc429_get_stats(&s)) return -2;
        arinc429_stats_resp_t r = { s.rx_words, s.parity_errors, s.overruns }
;
        memcpy(resp, &r, sizeof(r));
        *resp_len = sizeof(r);
        return 0;
    } else if (verb == SVC_VERB_GET_VERSION) {
        resp[0]=1; resp[1]=0; *resp_len=2; return 0;
    }
    return -3;
}
```

Register the handler:

```
/* in service_uart.c registry */
extern int svc_arinc429_handler(uint16_t, const uint8_t*, uint16_t, uint8_t*,
uint16_t*);
static const svc_reg_t g_registry[] = {
    { SVC_ID_LED,        svc_led_handler,       1, 0 },
    { SVC_ID_BOARD_INFO, svc_boardinfo_handler, 1, 0 },
    { SVC_ID_ARINC429,   svc_arinc429_handler,  1, 0 },
};
```

Verification checklist

- **Functional**: `GET_LAST(label)` from a PC client returns a stable 32-bit value; toggling a 429 source changes the returned word.
- **Robustness**: Corrupt a byte; CRC must detect and the server must reject safely.
- **Timing**: Measure request→response latency; document 99.9th percentile and bound any retries.
- **Safety hygiene**: Oversized payloads and unknown service IDs are handled deterministically.

---

9) Strong, realistic avionics scenarios enabled by this SOA

1) **Weight-and-Balance data provisioning**
   A 429-fed mass & balance calculator publishes **gross weight** and **center-of-gravity** as a service. The EICAS (Engine-Indicating and Crew-Alerting System) display queries via `GET_LAST(label)` for weight and CG derived labels. If a new sensor suite is added, the calculator changes internally; the **service contract and IDs remain identical**, limiting integration and certification impact.

2) **Maintenance mode and replay**
   During maintenance, a ground tool connects over a maintenance port (UART or Ethernet) and calls `GET_STATS()` on the ARINC-429 service to download error counters and **replay** last N words for diagnostics. Because the tool speaks the **service contract** (not raw 429), lab and aircraft tooling remain uniform.

3) **Sensor substitution**
   If a primary pitot system fails, a reversionary path activates alternate sensors with different 429 labels. The Data Concentrator maps them to the **same service**; the display doesn't need to know which physical label is feeding it—**the service abstraction keeps the system running**.

---

10) Certification-aligned considerations

- **DO-178C**: Map every service verb to **high-level requirements**, **low-level requirements**, and **tests**. Frame parser and CRC functions may be **DAL A/B** if they affect safety-critical data paths.
- **DO-297 (IMA)**: When moving to a partitioned OS (per **ARINC 653**), deploy each service in a partition with **ARINC ports** as the transport. Preserve the same contracts.
- **DO-326A (Security)**: Even on UART, authenticate maintenance commands (physical presence checks, maintenance key, or locked-out on ground only).

---

11) Extending the lab (advanced)

- Replace blocking UART with the **interrupt ring-buffer** pattern from the SDK example `lpuart_interrupt_rb_transfer.c`. Expose a `svc_transport_recv()` that pops bytes from the ring and assembles frames via a finite-state parser.
- Add **Publish/Subscribe**: implement `SUBSCRIBE(label, rate_hz)` and push **SVC_MSG_NOTIFY** frames at bounded rates.
- Add a **Service Registry Query** verb (`LIST_SERVICES`) that returns all `(service_id, version)` pairs.
- Swap the UART transport for **ENET (Ethernet)** using SDK LwIP examples while keeping the service layer unchanged.

---

12) Quick checklist (best practices recap)

- Contracts frozen and versioned?
- Byte-order, packing, and max sizes documented and enforced?
- CRC validated on every message?
- Bounded buffers and timeouts tested?
- Health/Version verbs implemented for every service?
- Error paths and unknown services handled deterministically?
- Traceability from requirement → code → test captured?

Note on SDK alignment

All code above follows the **MCUXpresso SDK for EVKB-i.MXRT1050** patterns (init via `BOARD_InitHardware()`, drivers like `fsl_lpuart.h`, and board support files such as `pin_mux.c` and `clock_config.h`). Replace placeholder hooks such as `BOARD_UserLedOn()` / `BOARD_UserLedOff()` with the appropriate GPIO control from your SDK board files to ensure the exercises build and run against the provided SDK code base.