Grease

A Message-Passing Approach to Protocol Stacks in Rust
1 My Journey in Software Engineering
2 Protocol Stacks
3 The Layered Model
4 Making Good Software
5 Grease
My Journey in Software Engineering
13 April 2018
Commercially Confidential

1:1
F1 Help | Locate and open a file in an Edit window
At this point…

- I know 11 languages (to some extent)
- I am a programmer, but
- I do not know how to write software.
Hands-Free Profile

RFCOMM

L2CAP

HCI
Hands-Free Profile

Hands-Free Profile
Hands-Free Profile

RFCOMM

L2CAP

Hands-Free Profile

RFCOMM

L2CAP
ITU-T X.200
TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

DATA NETWORKS AND OPEN SYSTEM
COMMUNICATIONS
OPEN SYSTEMS INTERCONNECTION – MODEL
AND NOTATION

INFORMATION TECHNOLOGY –
OPEN SYSTEMS INTERCONNECTION –
BASIC REFERENCE MODEL:
THE BASIC MODEL
The Layered Model
Hands-Free Profile

RFCOMM

Hands-Free Profile

RFCOMM
Higher Layers

HCI Driver

USB

PC

HCI Firmware

USB

Bluetooth Radio

Single-chip Bluetooth Module

Higher Layers

HCI

Bluetooth Radio
(N)-layer

(N - 1)-layer

(N - 1)-PCI

(N - 1)-SDU

(N)-PDU

 PCI  Protocol-control-information
 PDU  Protocol-data-unit
 SDU  Service-data-unit

TISO2900-94/d08
Hands-Free Profile

RFCOMM
Layer N
(Service User)

Layer N-1
(Service Provider)
Layer N
(Service User)

Layer N-1
(Service Provider)

Request

Confirm
Layer N
(Service User)

Layer N-1
(Service Provider)

Indication

Response
Naming Messages

- **Good Names**
  - FRAME_SEND_REQ / FRAME_SEND_CFM
  - FRAME_QUEUE_REQ / FRAME_QUEUE_CFM
  - FRAME_TX_IND
  - DATA_RECEIVED_IND
  - POSITION_REQUIRED_IND / POSITION_REQUIRED_RSP
  - IndDataReceived

- **Bad Names**
  - SEND_DATA_REQ / DATA_SEND_CFM
  - DATA_RECEIVED

- Clearly state the intention.
- Matched pairs of REQ/CFM.
- Appropriate case depends on language.
- Mismatched pair.
- Unclear message type.
But why?

- Large systems are *large*.

- A consistent set of rules is crucial.
  - Signposting so you don’t get lost.
  - Avoids misunderstanding.
  - This works in practice!
Message Sequence Charts

Layer N+1
EnableRxReq
EnableRxCfm

Layer N-1
RxFrameInd
RxDataInd
RxDataRsp

Layer N
Time elapses...
DNS Example, Part 1

DNS Client
- ReqLookup (hostname)
- CfmLookup

DNS Resolver
- ReqSendDatagram (ip_addr, port, payload)
- encode_packet()
- ReqSendPacket (ip_addr, payload)
- encode_frame()
- ReqSendFrame (mac_addr, payload)
- add_to_buffer()
- CfmSendFrame (result)
- CfmSendPacket (result)
- backoff_timeout()
- ReqSendBits (bits)
- CfmSendBits (result)

UDP

IP

Ethernet MAC

Ethernet PHY
DNS Example, Part 2

- DNS Client
- DNS Resolver
- UDP
- IP
- Ethernet MAC
- Ethernet PHY

- IndBitsRx
  - decode()
  - IndFrameRx
    - (src, dest, frame)
    - decode()
    - IndPacketRx
      - (src, dest, packet)
      - decode()
      - IndDatagramRx
        - (src, dest, datagram)
        - IndResult
          - (record[])
You can use this model to make good software
Function-call

- Single threaded

Message-passing

- Multi-threaded
Function-call

• Single threaded

Message-passing

• Multi-threaded
Function Calling stacks

- Many (most?) stacks and OS APIs are based around function calling:
  - Berkeley sockets API, for example
  - Callback functions for asynchronicity
  - Which thread does the callback function execute in?
  - Can take ‘short-cuts’ and poke around in the memory of another module.
// One method per message
result_t foo_tx_data_req(const uint8_t* p);

// One function for all requests
result_t foo_req(const foo_req_t* p);
// One method per message
result_t bar_tx_frame_req(addr_t addr, ...);

// One function for all requests
result_t bar_req(const bar_req_t* p);
Function-call

- Single threaded

Message-passing

- Multi-threaded
Message Passing

- Message passing is more work, but has benefits:
  - API is enforced, and well defined
  - Can’t (easily) poke around with another task’s variables
  - Unit testing is clean
    - No fighting the linker to provide stubs, fakes and mocks
  - Can hook the message passing system to provide debugging
    - He said, she said…
Every Layer is a Task

- Tasks are:
  - 1 Thread (or maybe more…)
  - 1 Queue (or maybe more…)

- The (primary) thread pends on the (primary) queue and performs actions based on the events received, before going to sleep again.
Layer N
Context

- If you allow more than one request to be in-flight at a time then, when a Confirmation is received, the service-user needs to be able to work out which Request it is a reply to.
  - You might need to handle multiple simultaneous connections
  - Requests might take an indeterminate amount of time.

- We use a *Context* field for this.
  - Each layer should use an unambiguous (to that layer) value
    - A pointer/reference
    - A unique integer from an incrementing thread-local value
**Context**

- The *Confirmation* reflects back the *Context* in the *Request*. Perhaps also used in a later *Indication*.
  - Service-user may have some sort of Hash Table to allow fast lookups.

- Context values generated in a given layer do not (generally) go up – they go down.

- Always decided by the Service-User
  - Service Providers must not rely on uniqueness
  - Example/test/production code might always set it to zero!

- For brevity, context values were omitted from the previous diagrams (along with error codes).
Typical C implementation

typedef struct message_t
{
    inter_id_t inter_id;
    prim_id_t prim_id;
    address_t return_address;
    size_t size;
} message_t;

typedef enum http_prim_id_t
{
    HTTP_REGISTER_URL_REQ,
    HTTP_REGISTER_URL_CFM,
    ...
    HTTP_LAST_PRIM // Not a prim
} http_prim_id_t;

typedef struct http_bind_req_t
{
    message_t hdr;
    ip_address_t addr;
    http_bind_context_t ctx;
} http_bind_req_t;

http_register_url_req_t* p_req = message_alloc(
    sizeof(http_register_url_req_t),
    INTER_ID_HTTP,
    HTTP_REGISTER_URL_REQ);
p_req->addr.port = 8000;
p_req->addr.ip[0] = 0x00;
... message_send(OS_QID_HTTP, p_req);
Standard C problems…

- Memory management
- Structure initialisation
- Tagged enumerations
There must be a better way!
Introducing Rust

- Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety.
  - www.rust-lang.org

- Out of Mozilla

- Used in Firefox today on Win/Mac/Linux/Android…

- The Servo HTML5 rendering engine (replacing Gecko) is their use-case
Why should I care?

- Fast like C with excellent C inter-op
- Segmentation faults are impossible*
- Null-pointer dereferences are impossible*
- Buffer overflows are impossible*
- First class build system / documentation generator / code formatting
- Rich, expressive type system
- But unlike C++, the types are sane (e.g. std::string)
Introducing Grease!

- A Message-Passing Approach to Protocol Stacks in Rust
- A proof of concept is available from [https://github.com/cambridgeconsultants/grease](https://github.com/cambridgeconsultants/grease)
Queues in Rust

- Standard library offers ‘mpsc’ channels
  - Multiple Provider Single Consumer

- Could easily substitute in another channel with the same API, e.g. to use an RTOS
  - Would love someone to do this!

- Wrapped into two types:
  - MessageSender – many per channel
  - MessageReceiver – one per channel
Tasks

- Standard library offers a threading API

- Could easily substitute in another threading library with the same API, e.g. to use an RTOS

- Messages can contain smart containers (like Vec) so tasks must be in the same address space
Grease, Mk1

```java
pub enum Message {
    Request(MessageSender, Request),
    Confirmation(Confirmation),
    Indication(Indication),
    Response(Response),
}
```
Grease, Mk2

```rust
pub trait ServiceProvider<REQ, CFM, IND, RSP> {
    /// Call this to send a request to this provider.
    fn send_request(&self, req: REQ, reply_to: &ServiceUser<CFM, IND>);

    /// Call this to send a response to this provider.
    fn send_response(&self, rsp: RSP);

    /// Call this to clone this object so another task can use it.
    fn clone(&self) -> ServiceProviderHandle<REQ, CFM, IND, RSP>;
}
```
Grease, Mk2

```rust
pub trait ServiceUser<CFM, IND> {
    /// Call this to send a confirmation back to the service user.
    fn send_confirm(&self, cfm: CFM);

    /// Call this to send an indication to the service user.
    fn send_indication(&self, ind: IND);

    /// Call this so we can store this user reference in two places.
    fn clone(&self) -> ServiceUserHandle<CFM, IND>;
}
```
**What next?**

- Think about message-passing architectures in your next project
- Think about what Rust can do you for
- Think about what you can do for Rust
- Check out [https://github.com/cambridgeconsultants/grease](https://github.com/cambridgeconsultants/grease)
- Check out [https://cambridgeconsultants.com/careers](https://cambridgeconsultants.com/careers)