Cambridge Consultants

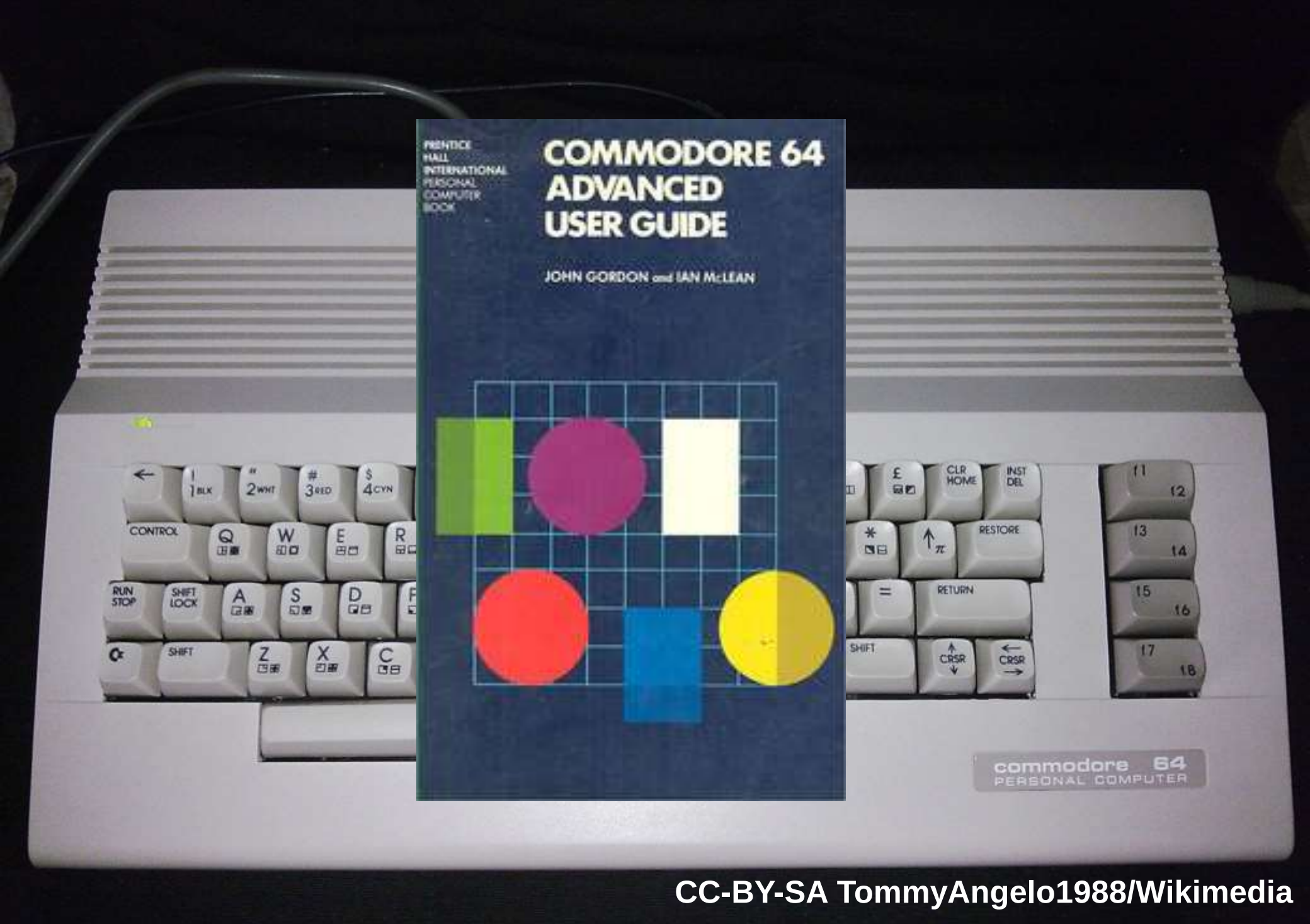
Grease

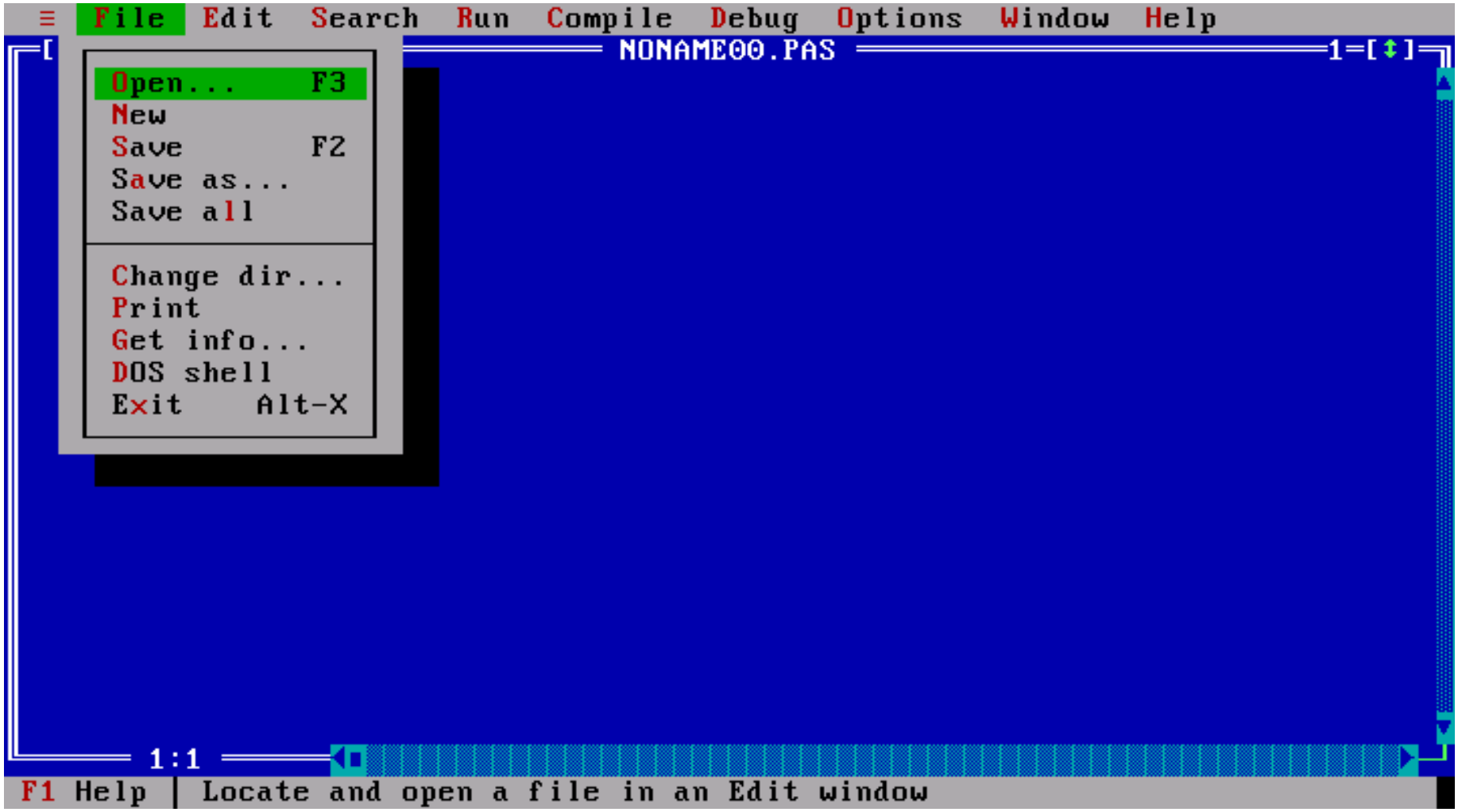
A Message-Passing Approach to Protocol Stacks in Rust

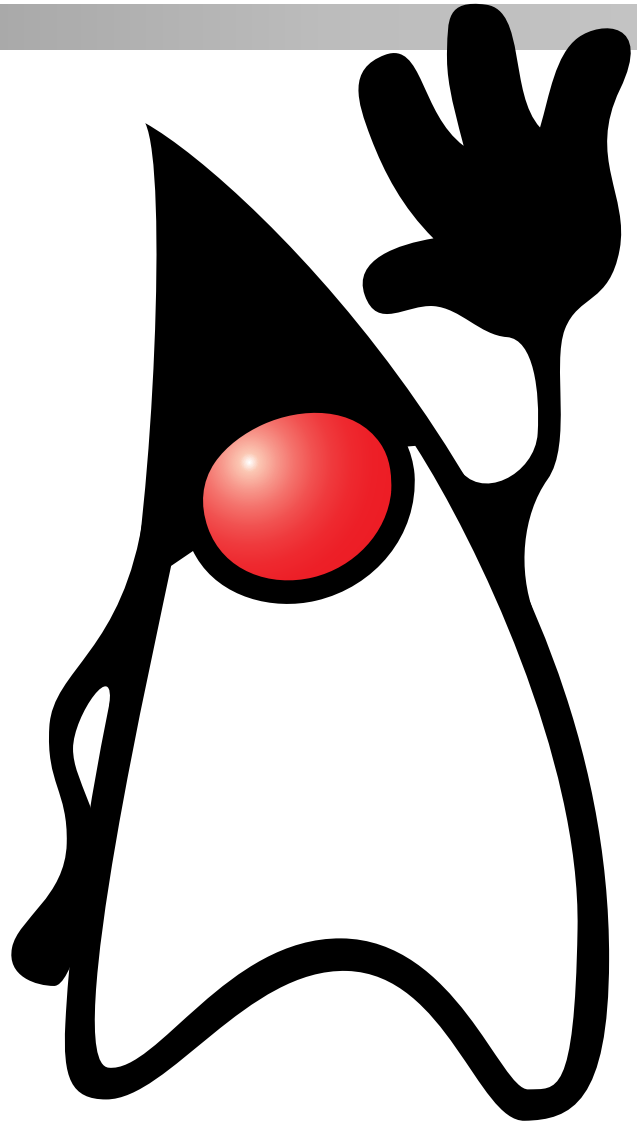
Jonathan Pallant

Cambridge Consultants

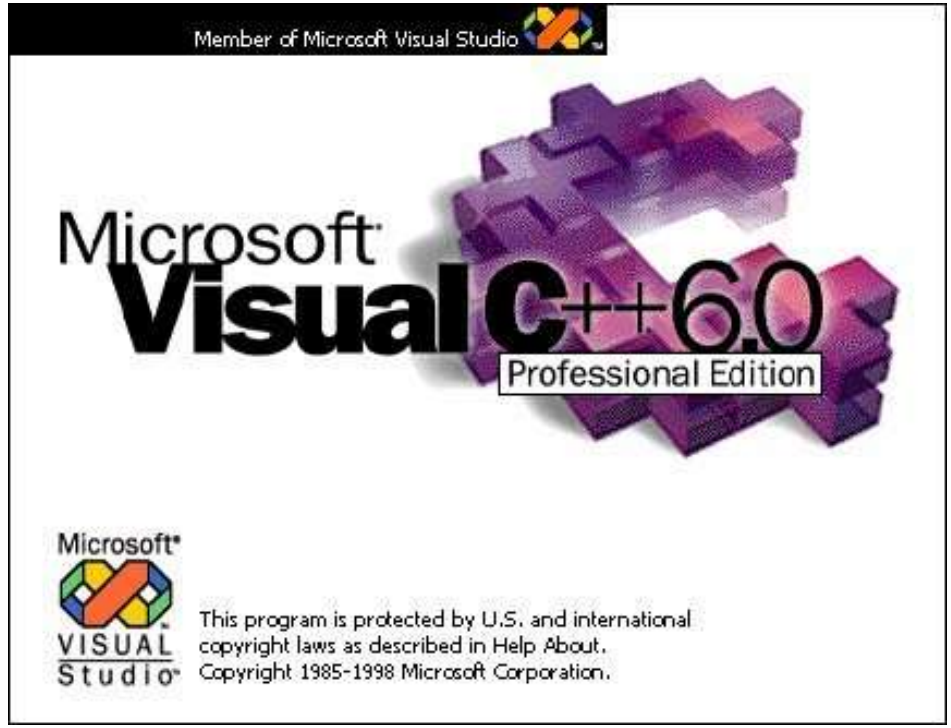
My Journey in Software Engineering




File Edit Search Run Compile Debug Options Window Help
NONAME00.PAS
Open... F3
New
Save F2
Save as...
Save all
Change dir...
Print
Get info...
DOS shell
Exit Alt-X
1:1
F1 Help | Locate and open a file in an Edit window

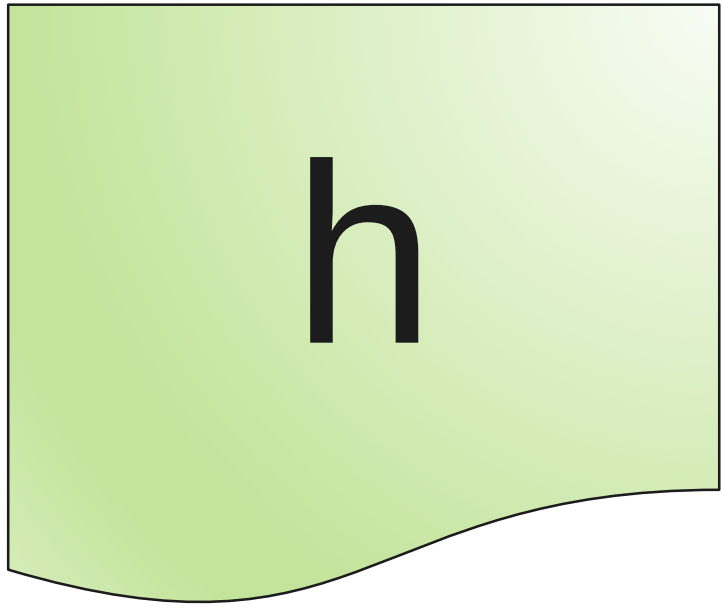


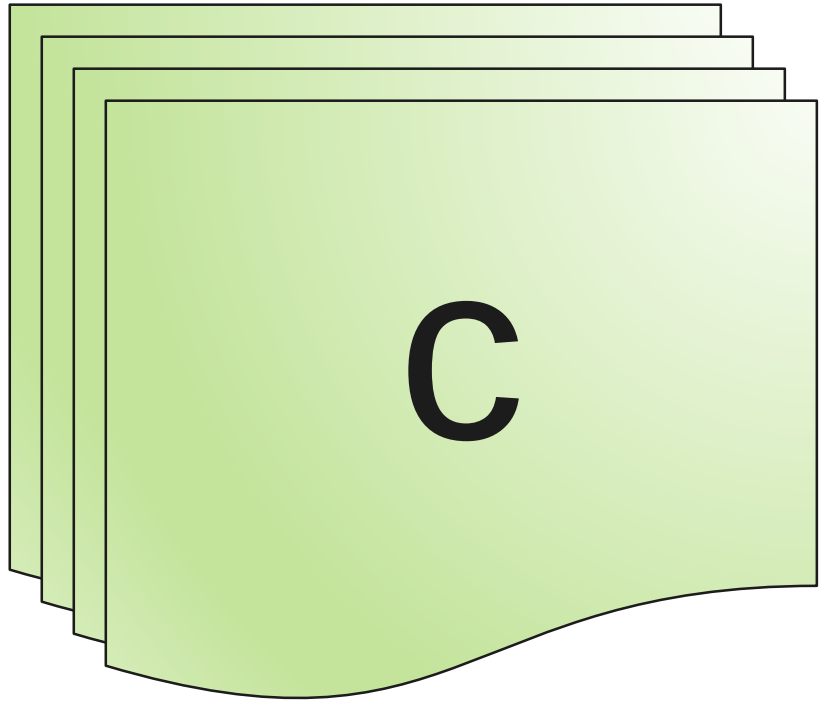

Borland
Delphi™
Personal
7
Unregistered
©2002 Borland Software Corporation


Member of Microsoft Visual Studio
Microsoft
Visual C++6.0
Professional Edition
Microsoft
VISUAL
Studio
This program is protected by U.S. and international
copyright laws as described in Help About.
Copyright 1985-1998 Microsoft Corporation.

At this point…

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


h

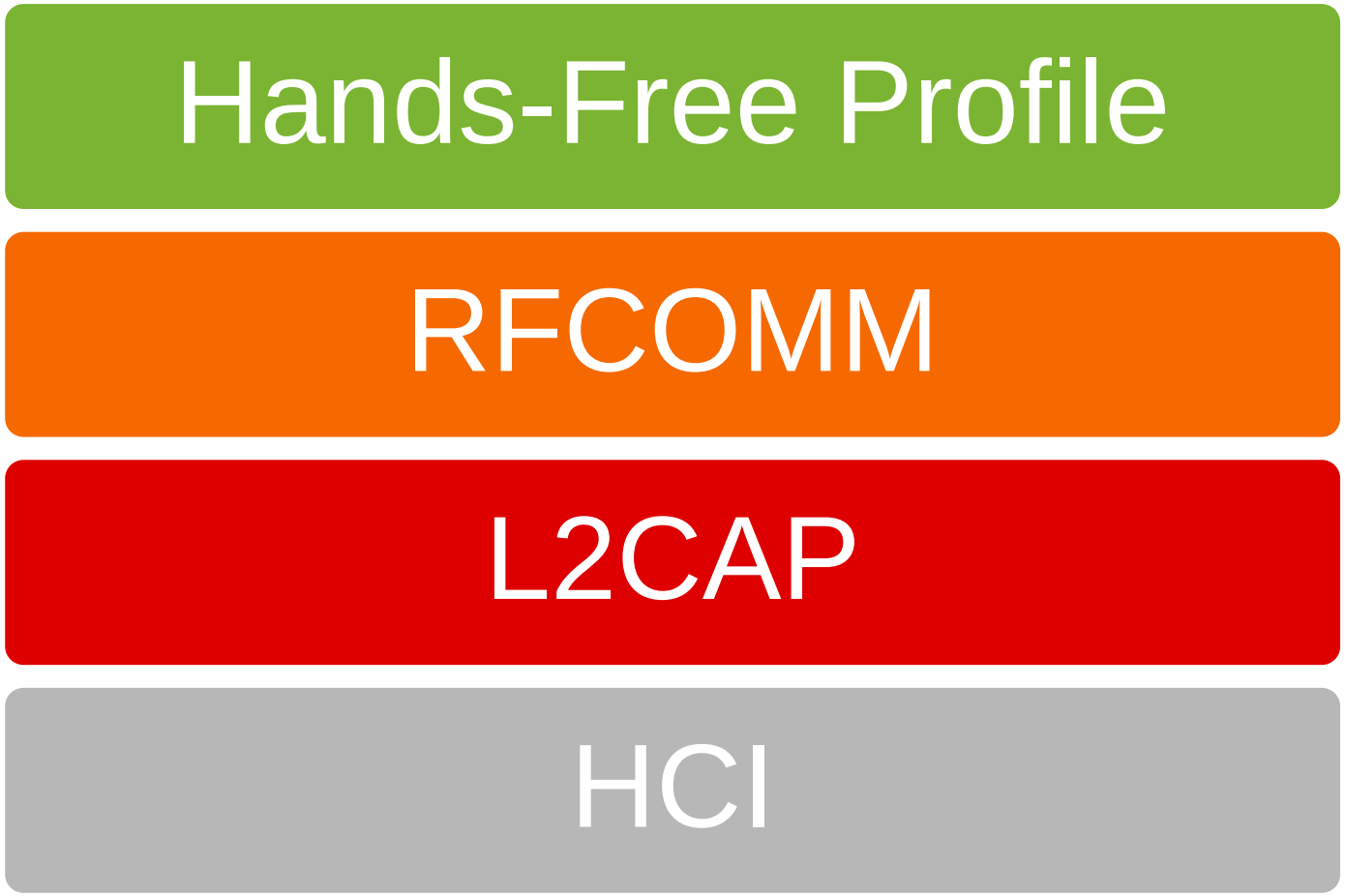

c

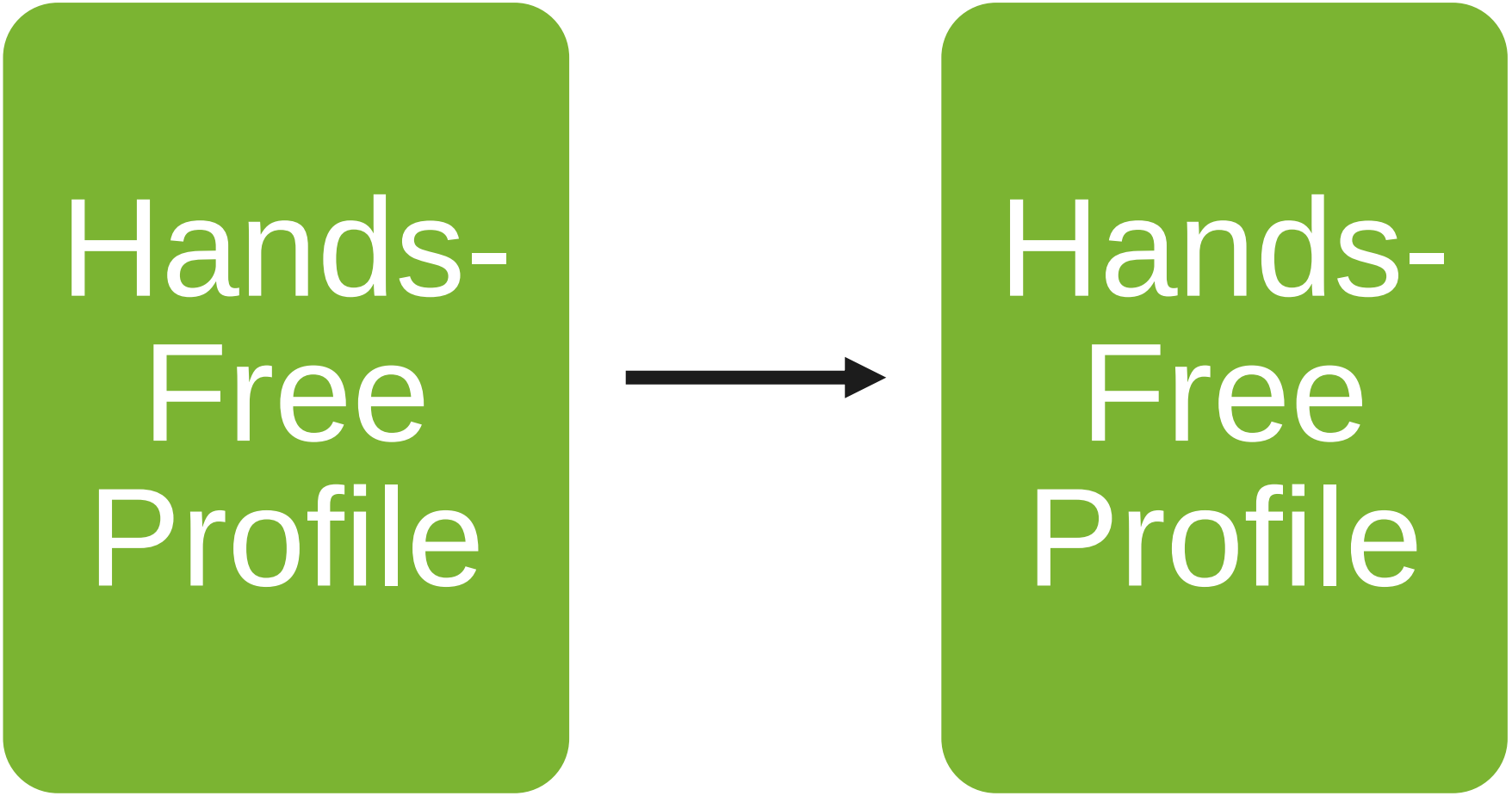

Cambridge
Consultants

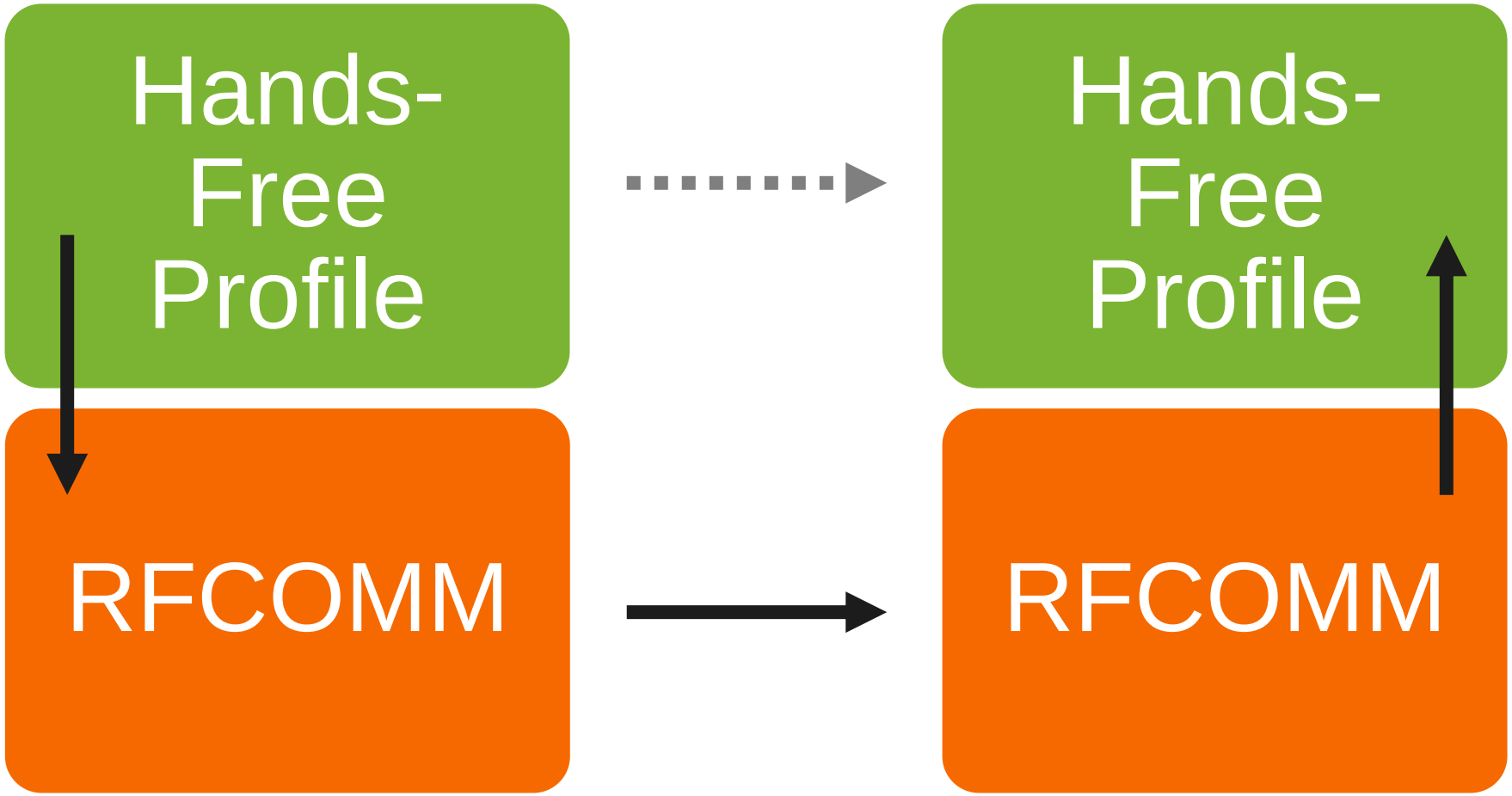

Bluetooth®
iridium®
Everywhere

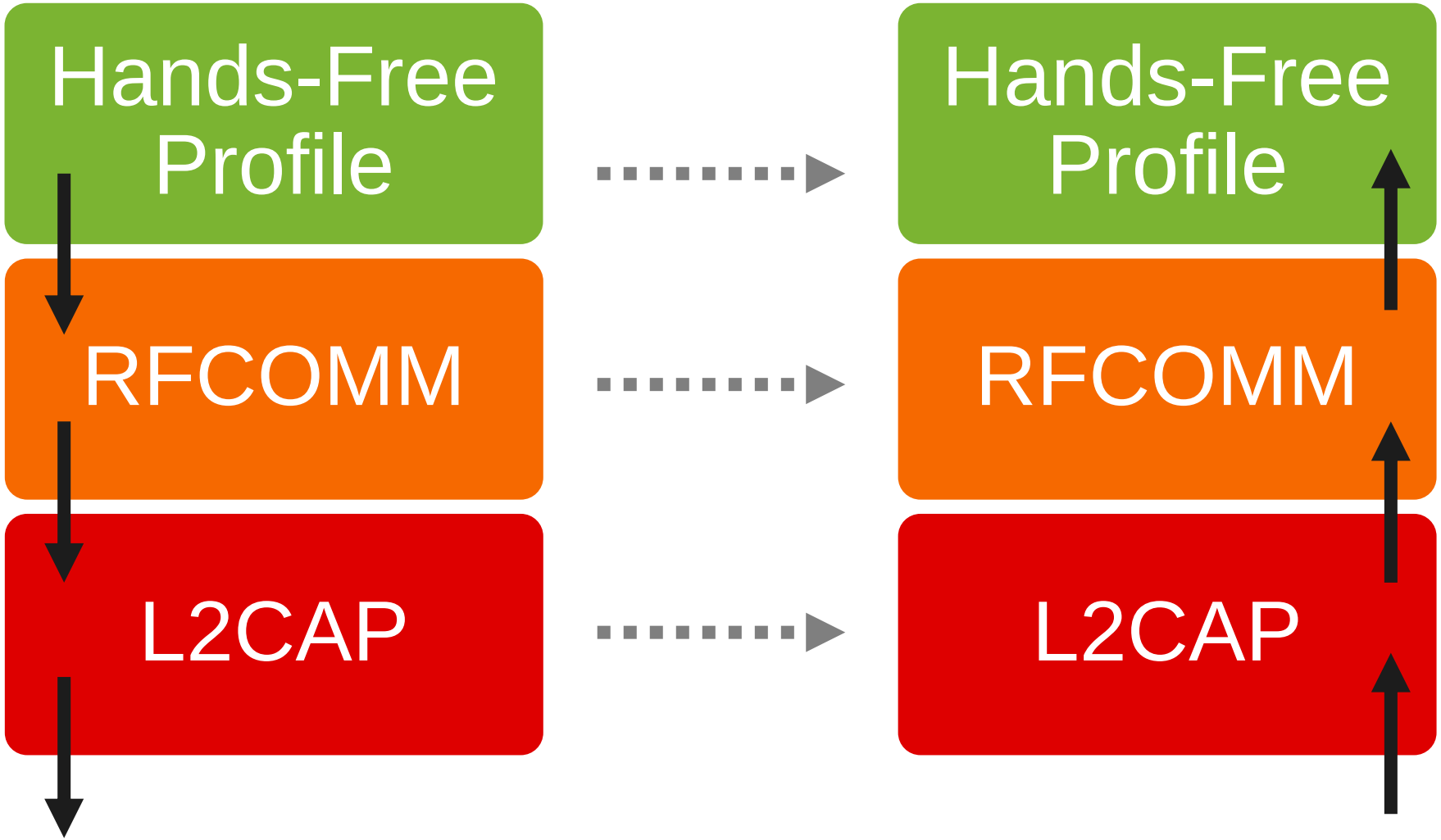
Cambridge Consultants

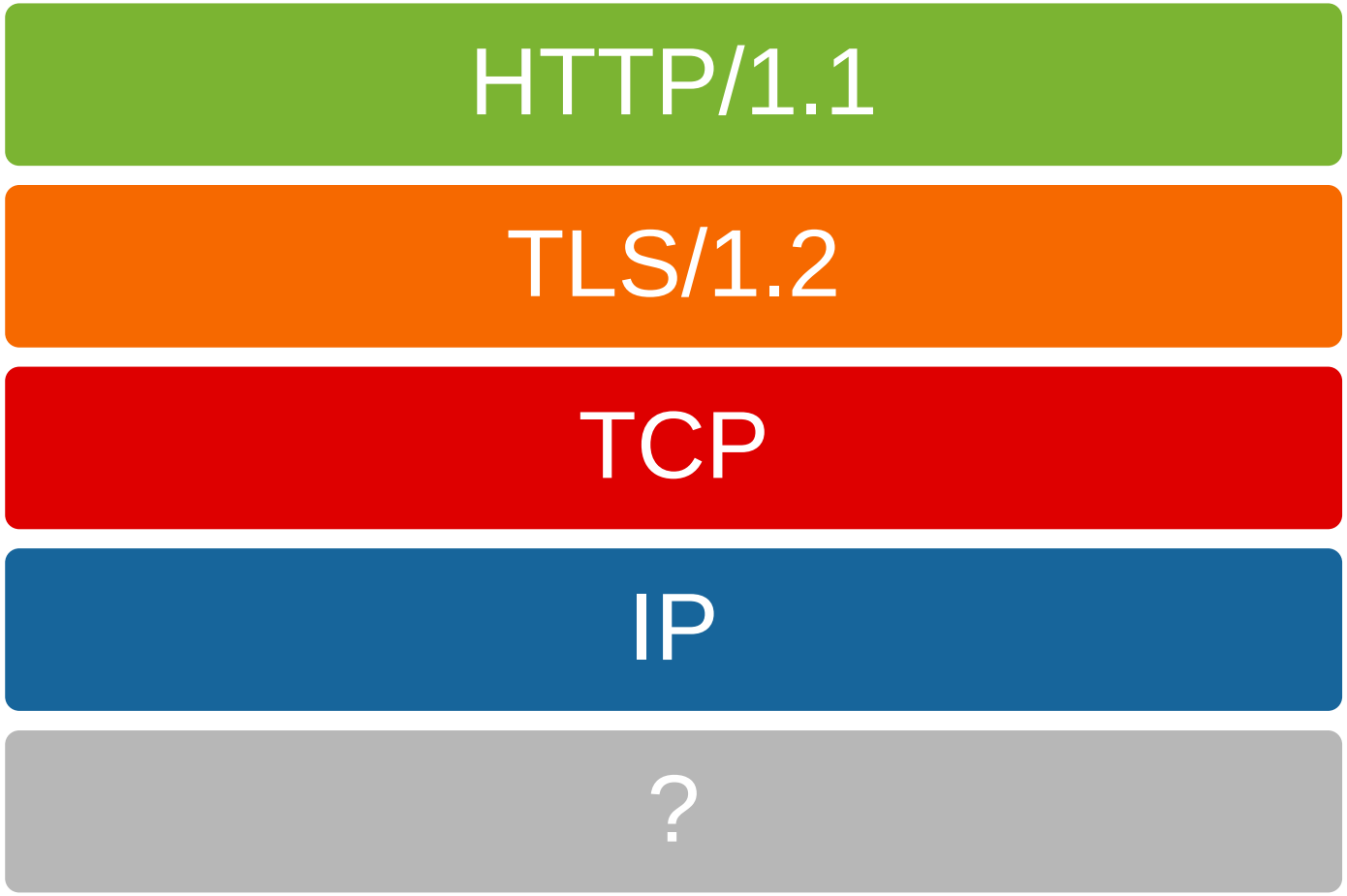
Protocol Stacks


Hands-Free Profile
RFCOMM
L2CAP
HCI


Hands-Free Profile
Hands-Free Profile


Hands-
Free
Profile
Hands-
Free
Profile
RFCOMM
RFCOMM


Hands-Free Profile
RFCOMM
L2CAP
Hands-Free Profile
RFCOMM
L2CAP


HTTP/1.1
TLS/1.2
TCP
IP
?

INTERNATIONAL TELECOMMUNICATION UNION

ITU-T X.200

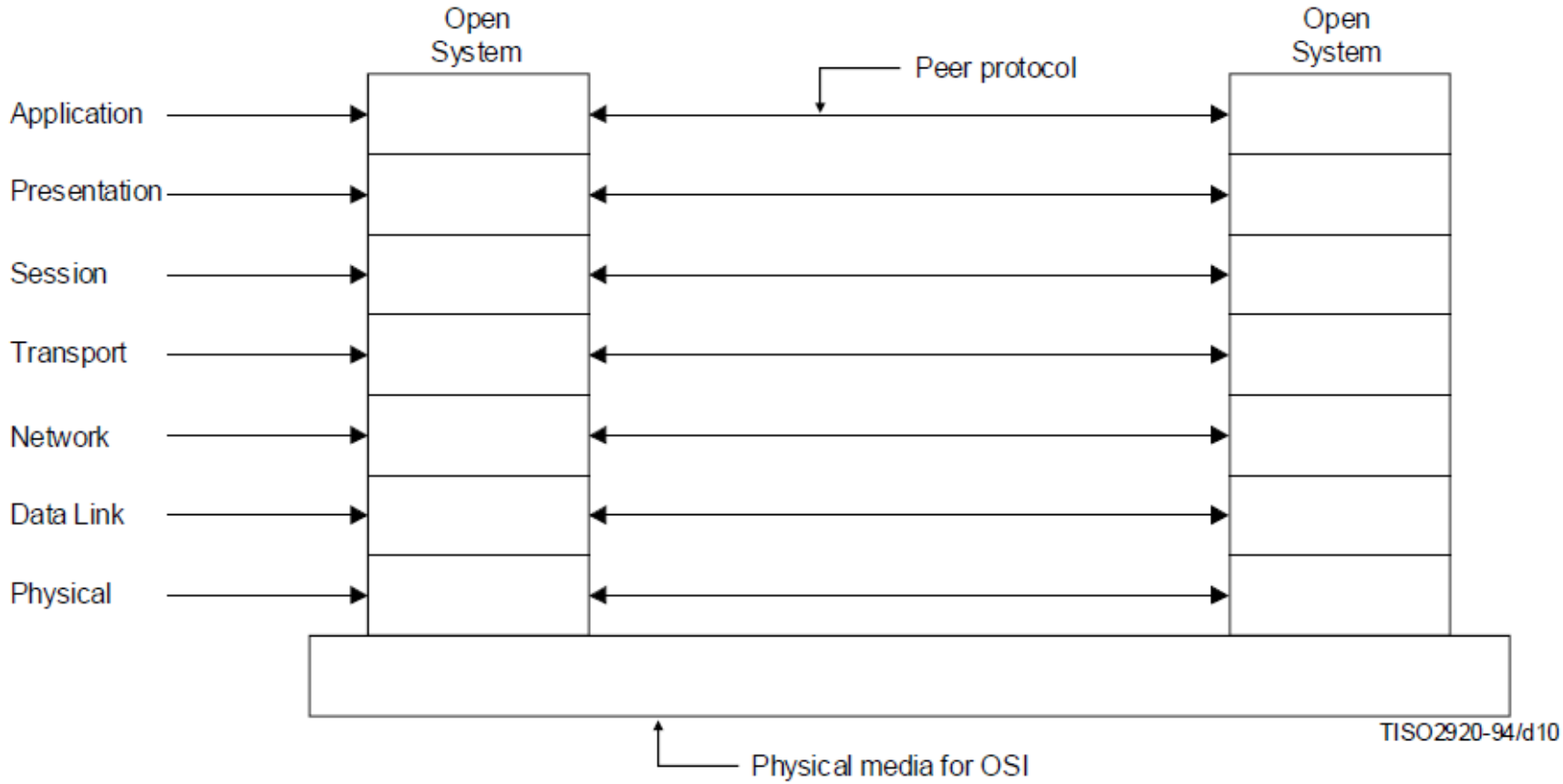
(07/94)

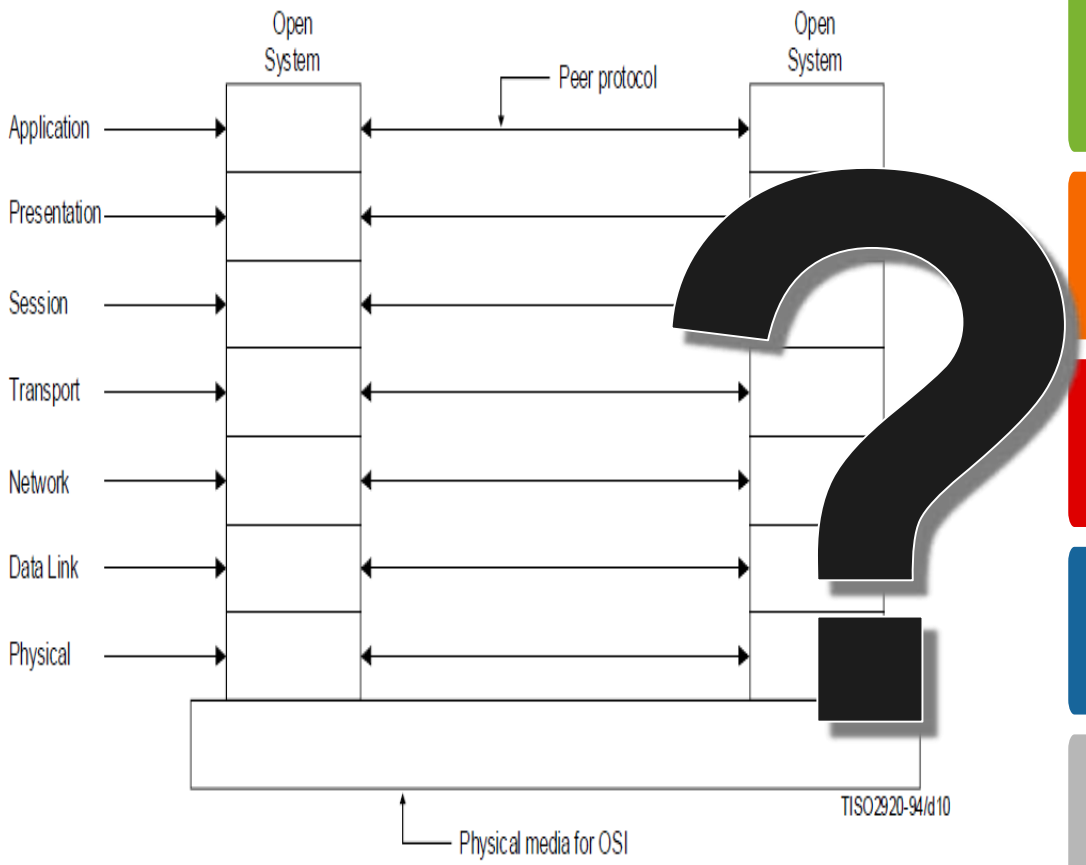
TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

DATA NETWORKS AND OPEN SYSTEM COMMUNICATIONS

OPEN SYSTEMS INTERCONNECTION – MODEL AND NOTATION

INFORMATION TECHNOLOGY – OPEN SYSTEMS INTERCONNECTION – BASIC REFERENCE MODEL: THE BASIC MODEL


Open System
Peer protocol
Open System
Application
Presentation
Session
Transport
Network
Data Link
Physical
TISO2920-94/d10
Physical media for OSI



HTTP

TLS

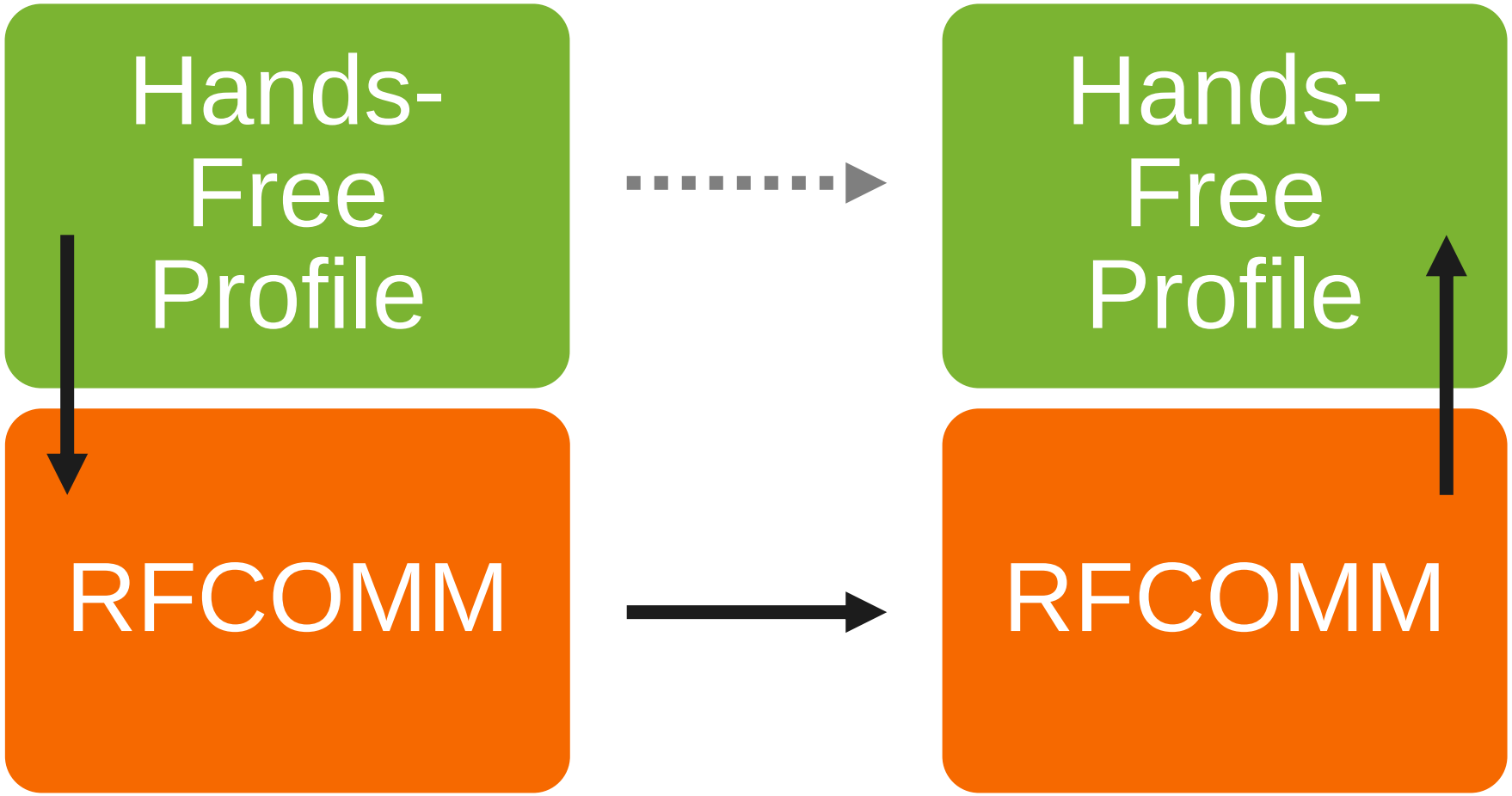
TCP

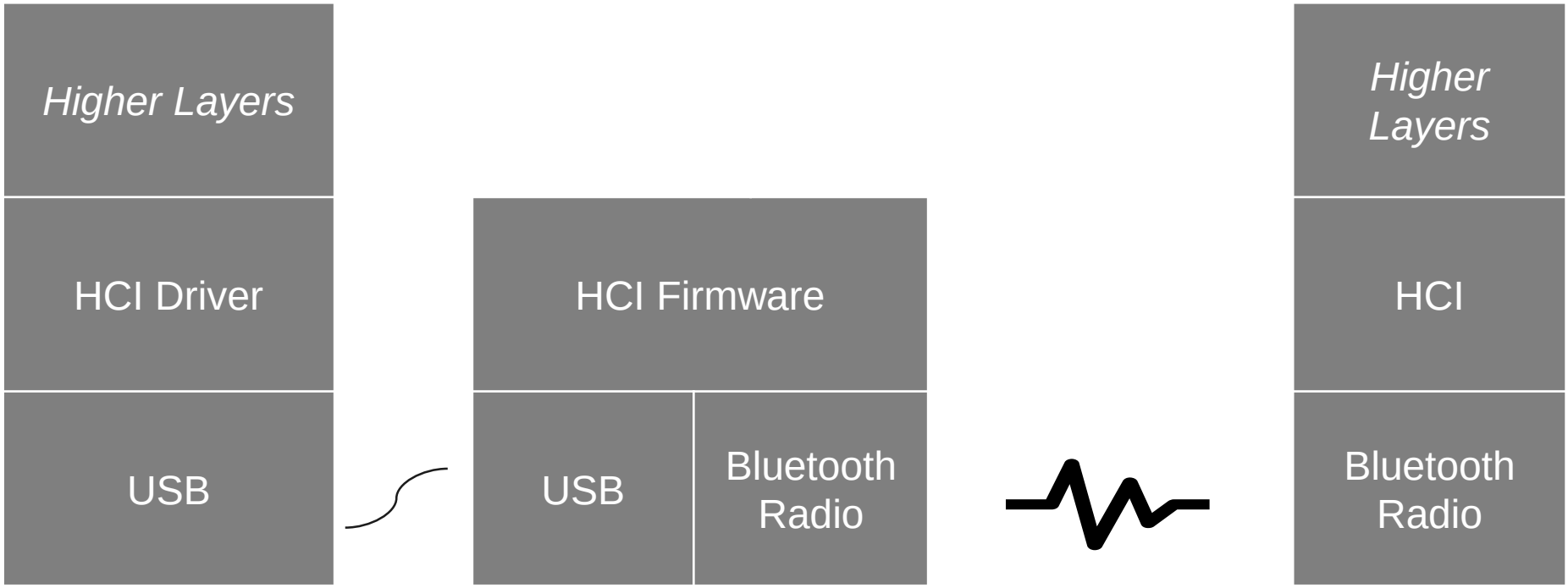
IP

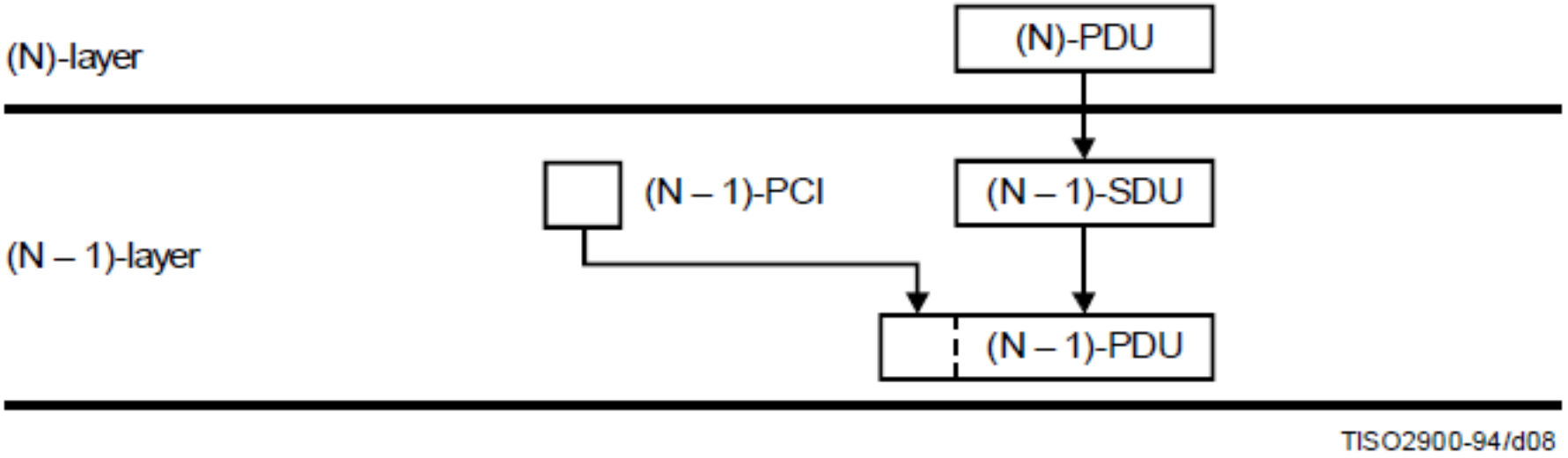
?

Cambridge Consultants

The Layered Model


Hands-
Free
Profile
Hands-
Free
Profile
RFCOMM
RFCOMM

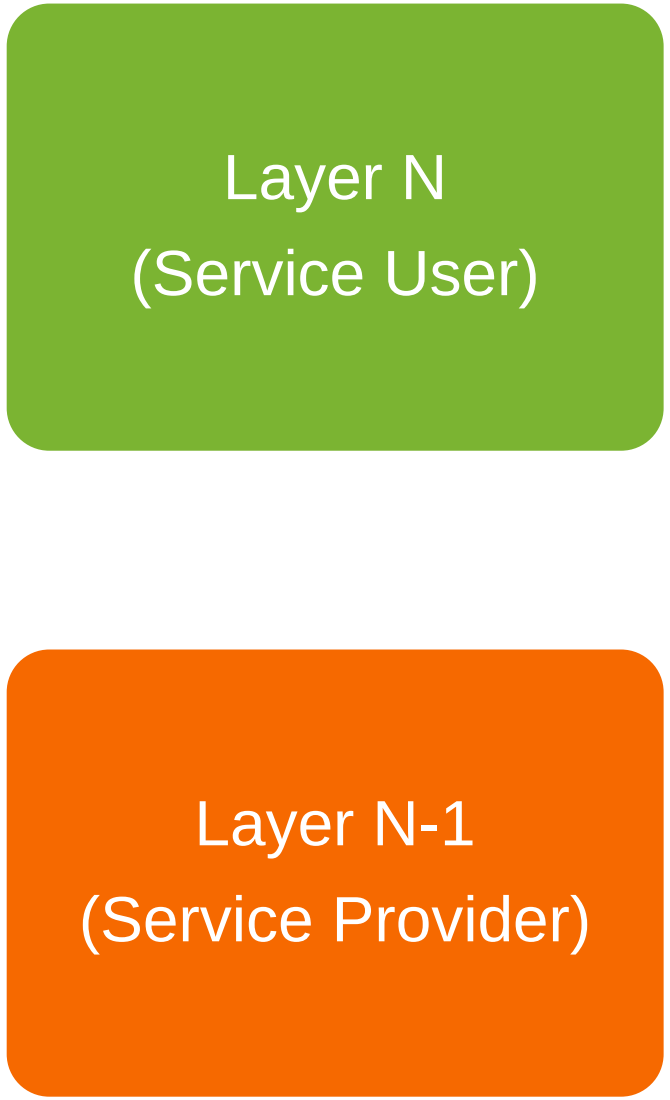

Higher Layers
HCI Driver
USB
HCI Firmware
USB
Bluetooth Radio
Higher Layers
HCI
Bluetooth Radio
PC
USB Bluetooth Module
Single-chip Bluetooth Module

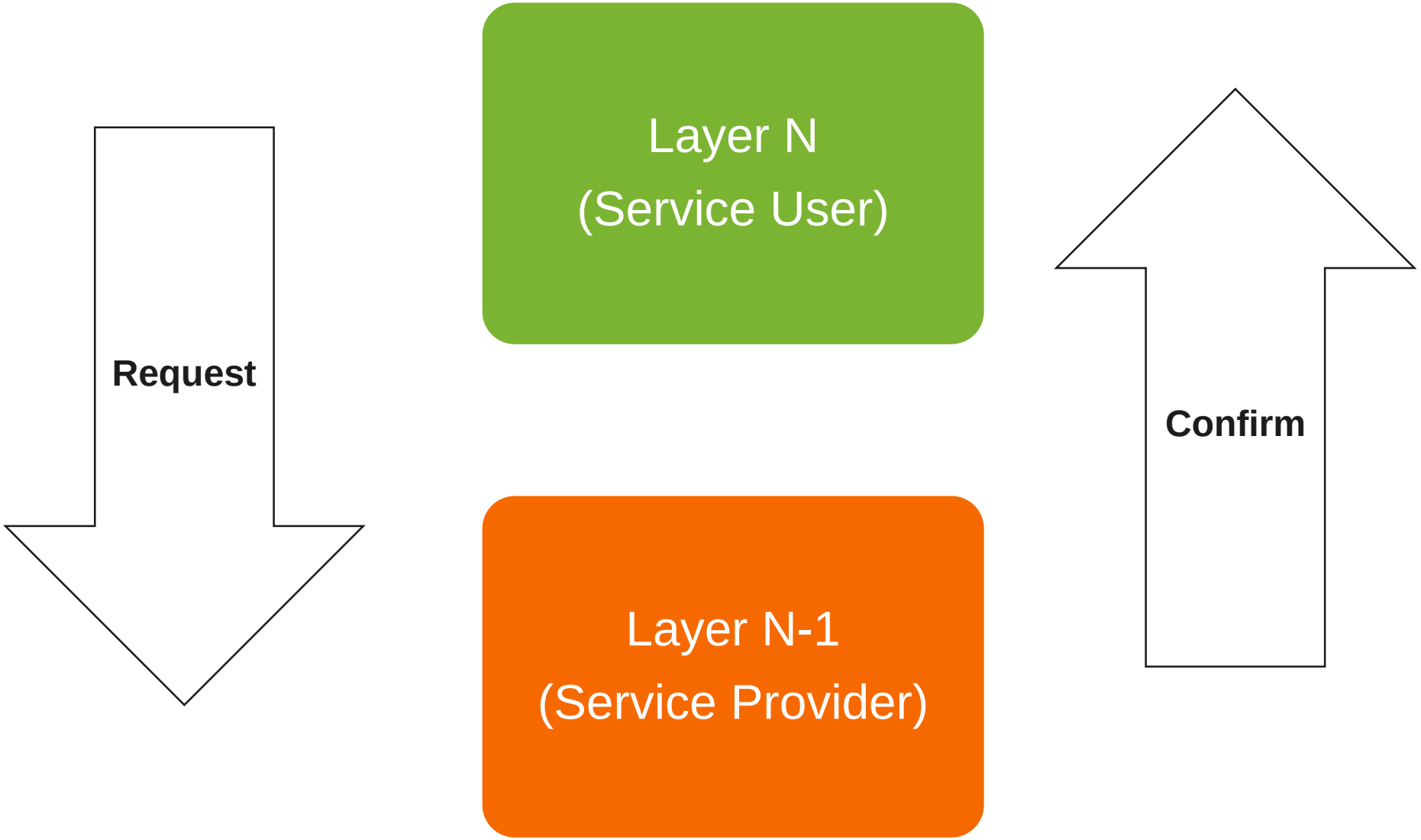


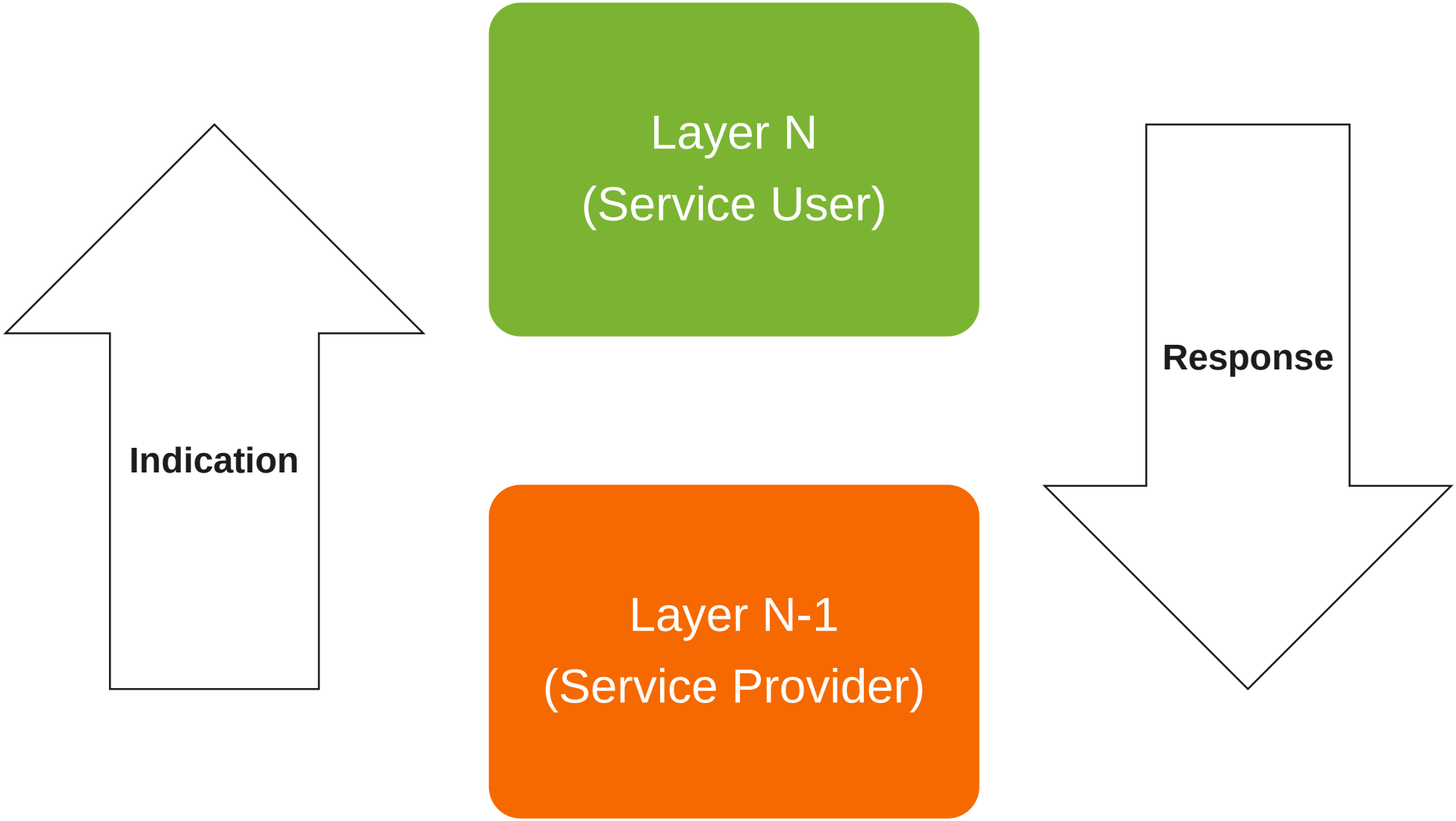
PCI Protocol-control-information
PDU Protocol-data-unit
SDU Service-data-unit

Hands-Free Profile

RFCOMM


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


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


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

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

- Clearly state the intention.
- Matched pairs of REQ/CFM.
- Appropriate case depends on language.

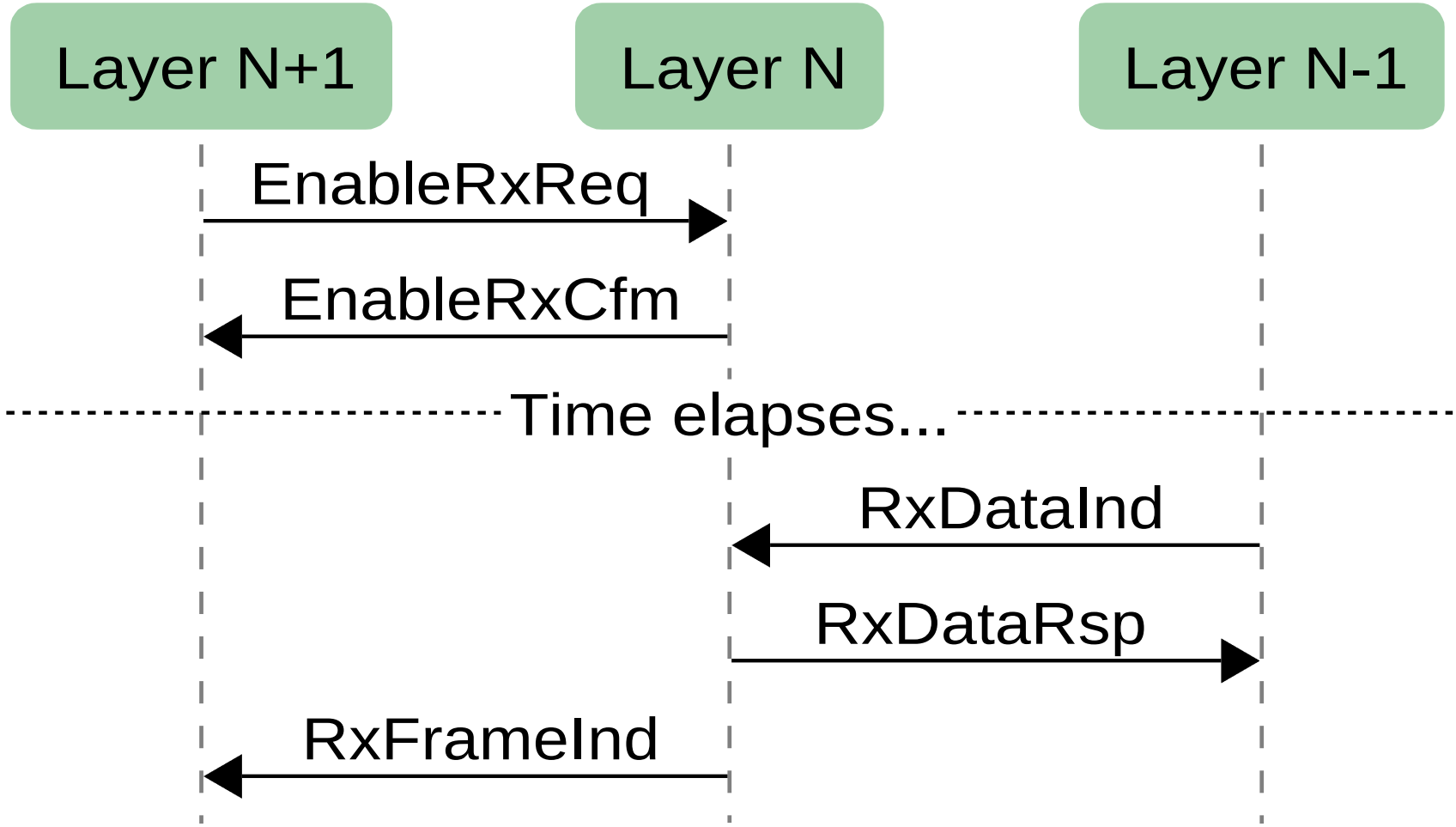
- Bad Names
  - SEND_DATA_REQ / DATA_SEND_CFM
  - DATA_RECEIVED

- Mismatched pair.
- Unclear message type.

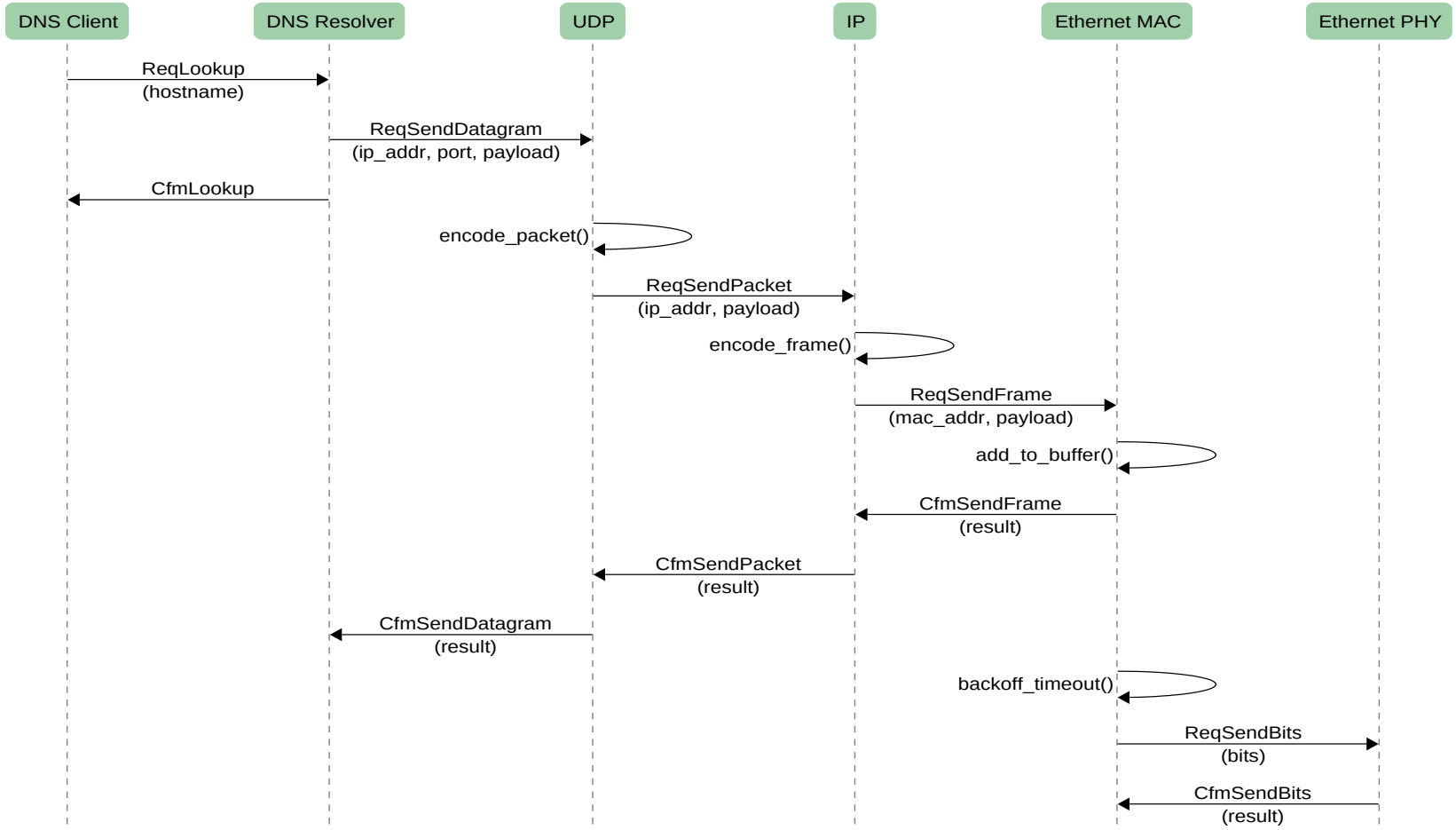
But why?

- Large systems are *large.*
- A consistent set of rules of crucial.
  - Signposting so you don’t get lost.
  - Avoids misunderstanding.
  - This works in practice!

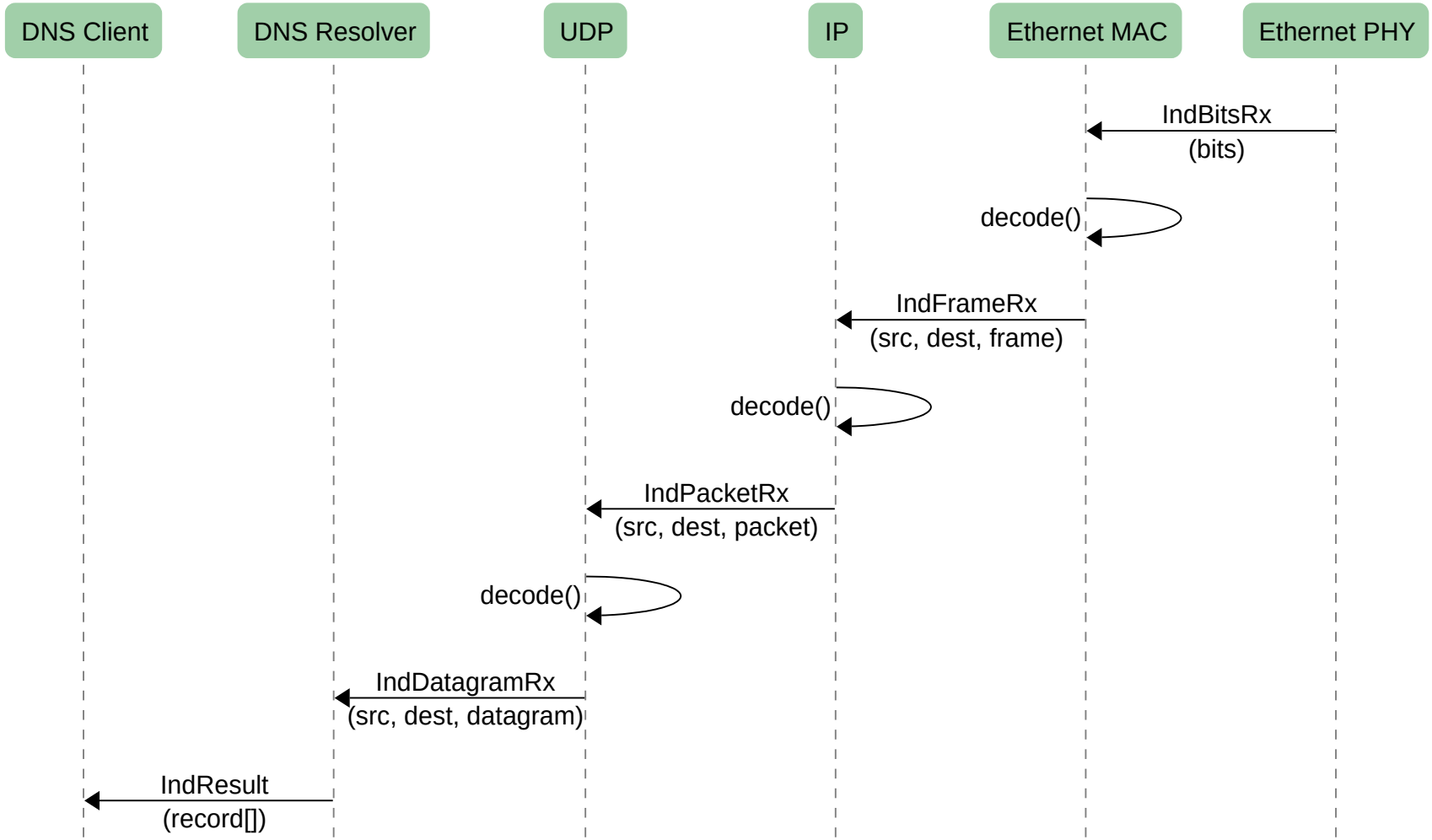
Message Sequence Charts



DNS Example, Part 1



DNS Example, Part 2



You can use this model to
make good software


Bluetooth®
iridium®
Everywhere

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

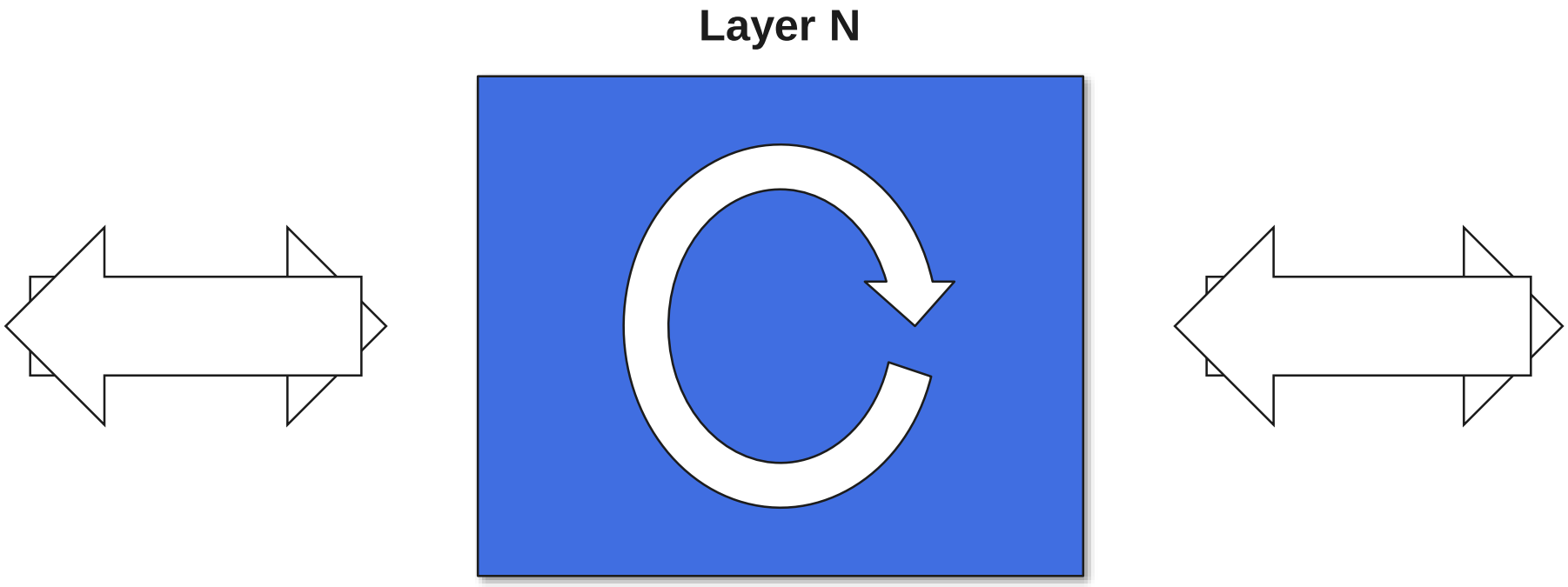
Message Passing

- 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

Cambridge Consultants

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

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

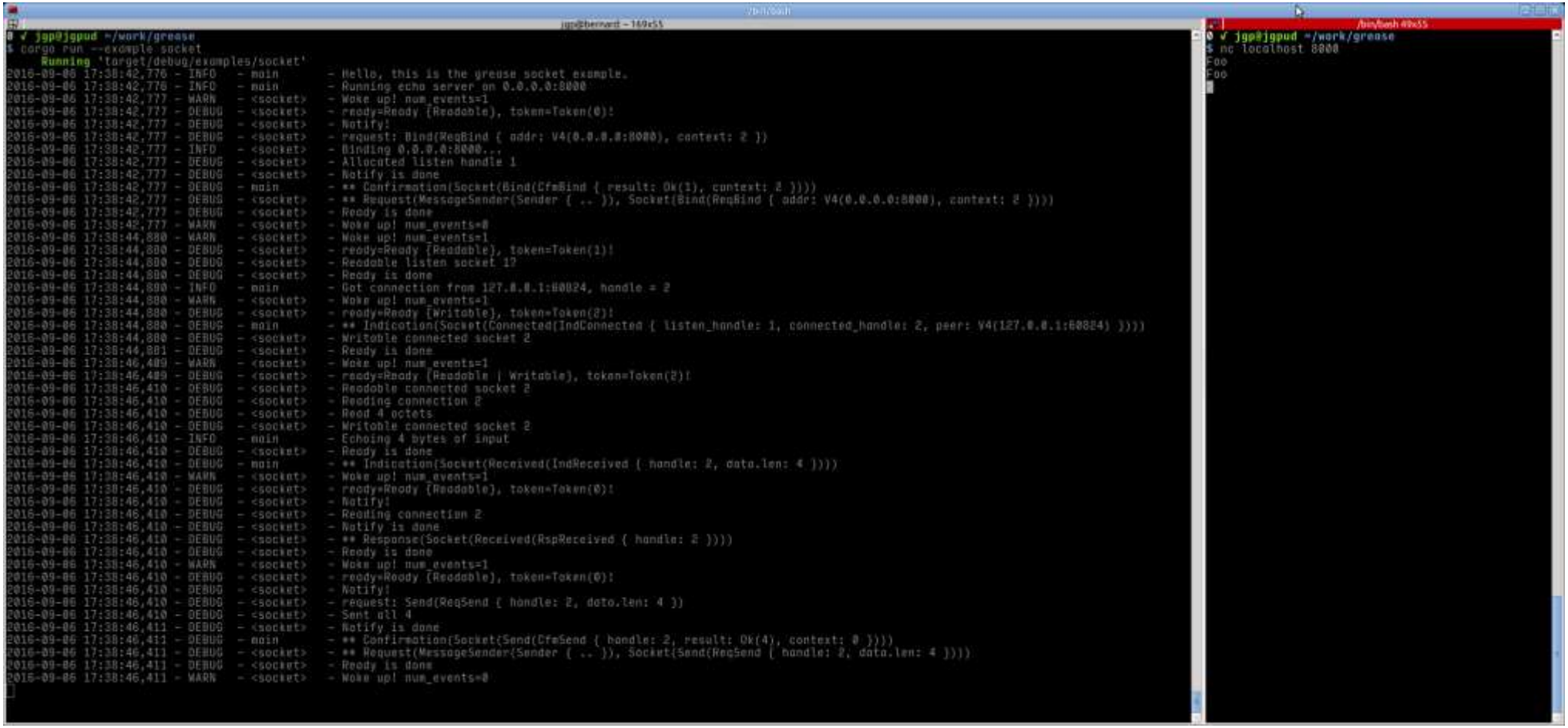
```
pub enum Message {
    Request(MessageSender, Request),
    Confirmation(Confirmation),
    Indication(Indication),
    Response(Response),
}
```

Grease, Mk2

```
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

```
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>;
}
```


jgp@jgpud ~/work/grease
$ cargo run --example socket
Running 'target/debug/examples/socket'
2016-09-06 17:38:42,776 - INFO - main - Hello, this is the grease socket example.
2016-09-06 17:38:42,776 - INFO - main - Running echo server on 0.0.0.0:8000
2016-09-06 17:38:42,777 - WARN - <socket> - Woke up! num_events=1
2016-09-06 17:38:42,777 - DEBUG - <socket> - ready=Ready {Readable}, token=Token(0)!
2016-09-06 17:38:42,777 - DEBUG - <socket> - Notify!
2016-09-06 17:38:42,777 - DEBUG - <socket> - request: Bind(ReqBind { addr: V4(0.0.0.0:8000), context: 2 })
2016-09-06 17:38:42,777 - INFO - <socket> - Binding 0.0.0.0:8000...
2016-09-06 17:38:42,777 - DEBUG - <socket> - Allocated listen handle 1
2016-09-06 17:38:42,777 - DEBUG - <socket> - Notify is done
2016-09-06 17:38:42,777 - DEBUG - main - ** Confirmation(Socket(Bind(CfmBind { result: Ok(1), context: 2 })))
2016-09-06 17:38:42,777 - DEBUG - <socket> - ** Request(MessageSender(Sender { .. }), Socket(Bind(ReqBind { addr: V4(0.0.0.0:8000), context: 2 })))
2016-09-06 17:38:42,777 - DEBUG - <socket> - Ready is done
2016-09-06 17:38:42,777 - WARN - <socket> - Woke up! num_events=0
2016-09-06 17:38:44,880 - WARN - <socket> - Woke up! num_events=1
2016-09-06 17:38:44,880 - DEBUG - <socket> - ready=Ready {Readable}, token=Token(1)!
2016-09-06 17:38:44,880 - DEBUG - <socket> - Readable listen socket 1?
2016-09-06 17:38:44,880 - DEBUG - <socket> - Ready is done
2016-09-06 17:38:44,880 - INFO - main - Got connection from 127.0.0.1:60824, handle = 2
2016-09-06 17:38:44,880 - WARN - <socket> - Woke up! num_events=1
2016-09-06 17:38:44,880 - DEBUG - <socket> - ready=Ready {Writable}, token=Token(2)!
2016-09-06 17:38:44,880 - DEBUG - main - ** Indication(Socket(Connected(IndConnected { listen_handle: 1, connected_handle: 2, peer: V4(127.0.0.1:60824) })))
2016-09-06 17:38:44,880 - DEBUG - <socket> - Writable connected socket 2
2016-09-06 17:38:44,881 - DEBUG - <socket> - Ready is done
2016-09-06 17:38:46,409 - WARN - <socket> - Woke up! num_events=1
2016-09-06 17:38:46,409 - DEBUG - <socket> - ready=Ready {Readable | Writable}, token=Token(2)!
2016-09-06 17:38:46,410 - DEBUG - <socket> - Readable connected socket 2
2016-09-06 17:38:46,410 - DEBUG - <socket> - Reading connection 2
2016-09-06 17:38:46,410 - DEBUG - <socket> - Read 4 octets
2016-09-06 17:38:46,410 - DEBUG - <socket> - Writable connected socket 2
2016-09-06 17:38:46,410 - INFO - main - Echoing 4 bytes of input
2016-09-06 17:38:46,410 - DEBUG - <socket> - Ready is done
2016-09-06 17:38:46,410 - DEBUG - main - ** Indication(Socket(Received(IndReceived { handle: 2, data.len: 4 })))
2016-09-06 17:38:46,410 - WARN - <socket> - Woke up! num_events=1
2016-09-06 17:38:46,410 - DEBUG - <socket> - ready=Ready {Readable}, token=Token(0)!
2016-09-06 17:38:46,410 - DEBUG - <socket> - Notify!
2016-09-06 17:38:46,410 - DEBUG - <socket> - Reading connection 2
2016-09-06 17:38:46,410 - DEBUG - <socket> - Notify is done
2016-09-06 17:38:46,410 - DEBUG - <socket> - ** Response(Socket(Received(RspReceived { handle: 2 })))
2016-09-06 17:38:46,410 - DEBUG - <socket> - Ready is done
2016-09-06 17:38:46,410 - WARN - <socket> - Woke up! num_events=1
2016-09-06 17:38:46,410 - DEBUG - <socket> - ready=Ready {Readable}, token=Token(0)!
2016-09-06 17:38:46,410 - DEBUG - <socket> - Notify!
2016-09-06 17:38:46,410 - DEBUG - <socket> - request: Send(ReqSend { handle: 2, data.len: 4 })
2016-09-06 17:38:46,410 - DEBUG - <socket> - Sent all 4
2016-09-06 17:38:46,411 - DEBUG - <socket> - Notify is done
2016-09-06 17:38:46,411 - DEBUG - main - ** Confirmation(Socket(Send(CfmSend { handle: 2, result: Ok(4), context: 0 })))
2016-09-06 17:38:46,411 - DEBUG - <socket> - ** Request(MessageSender(Sender { .. }), Socket(Send(ReqSend { handle: 2, data.len: 4 })))
2016-09-06 17:38:46,411 - DEBUG - <socket> - Ready is done
2016-09-06 17:38:46,411 - WARN - <socket> - Woke up! num_events=0
/bin/bash 49x55
$ nc localhost 8000
Foo
Foo

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
- Check out https://cambridgeconsultants.com/careers

Cambridge Consultants






UK ▪ USA ▪ SINGAPORE ▪ JAPAN

www.CambridgeConsultants.com

Cambridge Consultants is part of the Altran group, a global leader in Innovation. www.Altran.com

Registered No. 1036296 England