`Go`ing the extra mile with Rust

Hari Bhaskaran

18 Dec 2020

A pragmatic comparison of Go vs Rust

Please keep an open mind. No Gophers or Crabs will be harmed

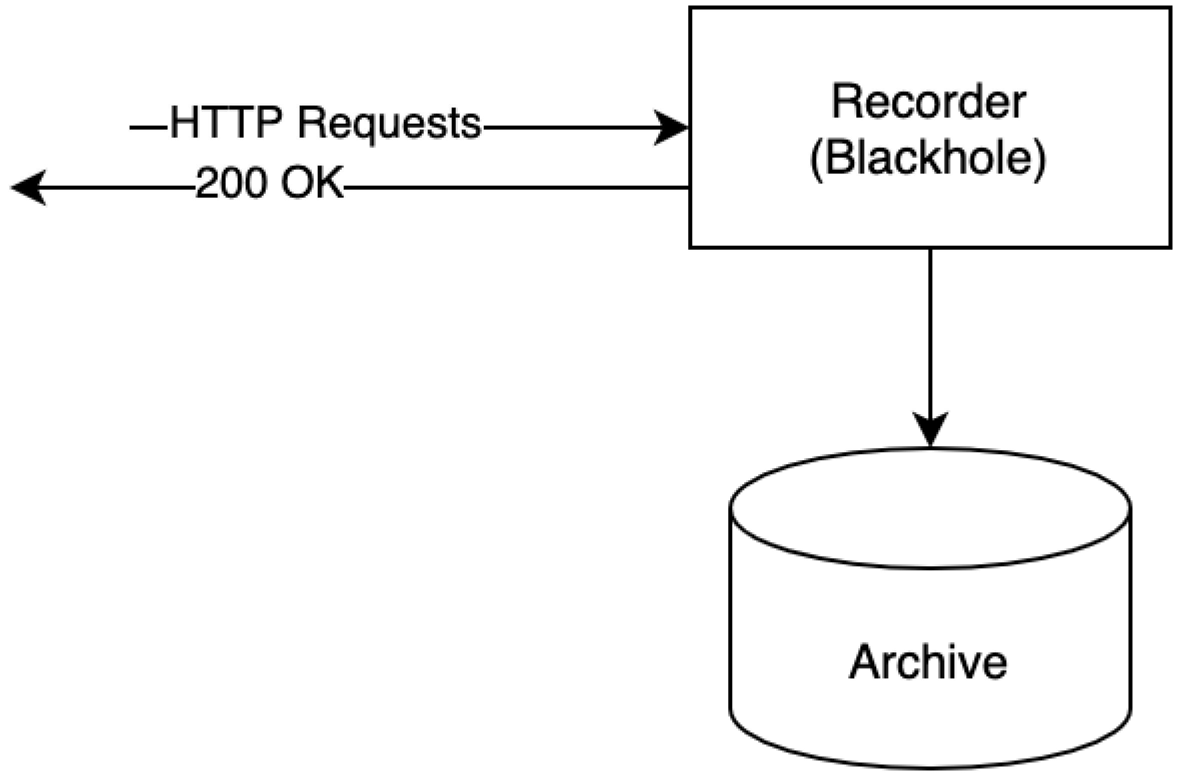
- A utility written in both languages
- Go version is functional, Rust version is WIP
- Both are opensource

Introducing `blackhole`

- Capture incoming HTTP(s) traffic and act like a `/dev/null`
- Record traffic for replay / debugging later.
- A utility knife with many uses
- Available at http://github.com/adobe/blackhole

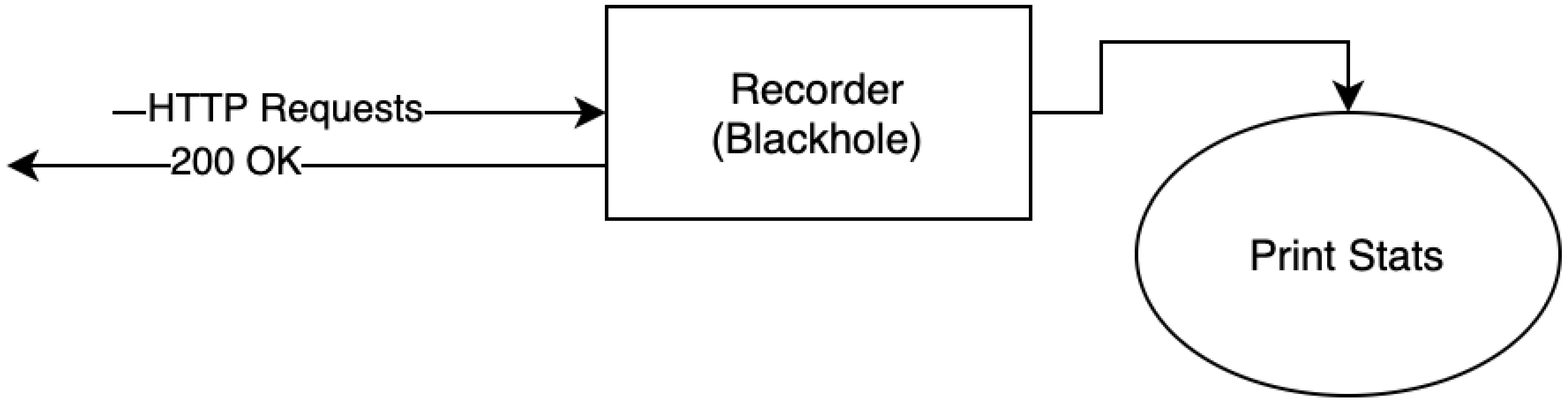
Basic Usage

- Capture traffic by running this *instead* of your real service and record requests
- Requires client to not expect anything more fancier than a `200 OK`



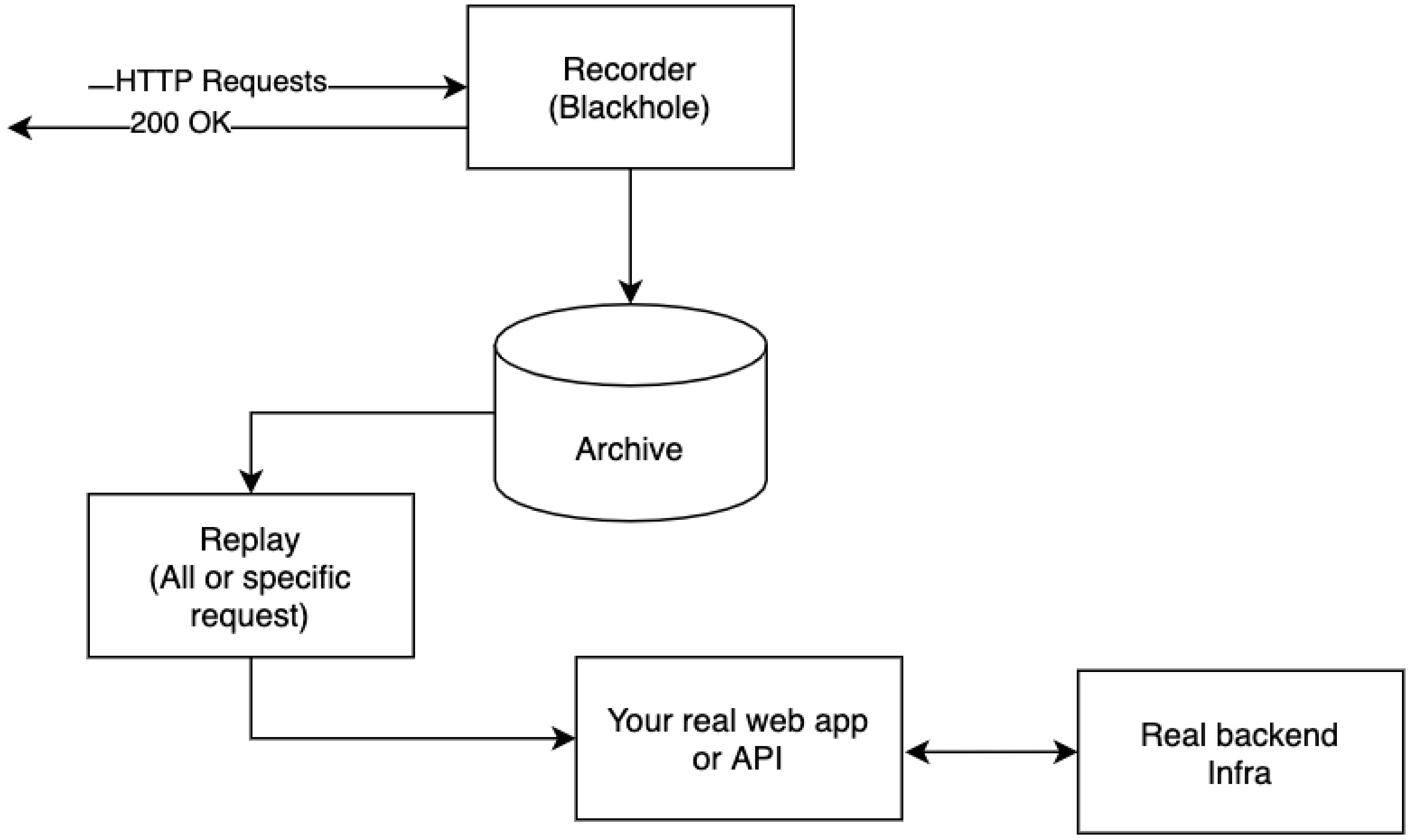
Usecase #1

- Capture traffic by running this *instead* of your real service and drop requests.
- To check
  - theoretical maximums for your network / node
  - Stress-test your web client
  - Stress-test your Server-to-Server API client.



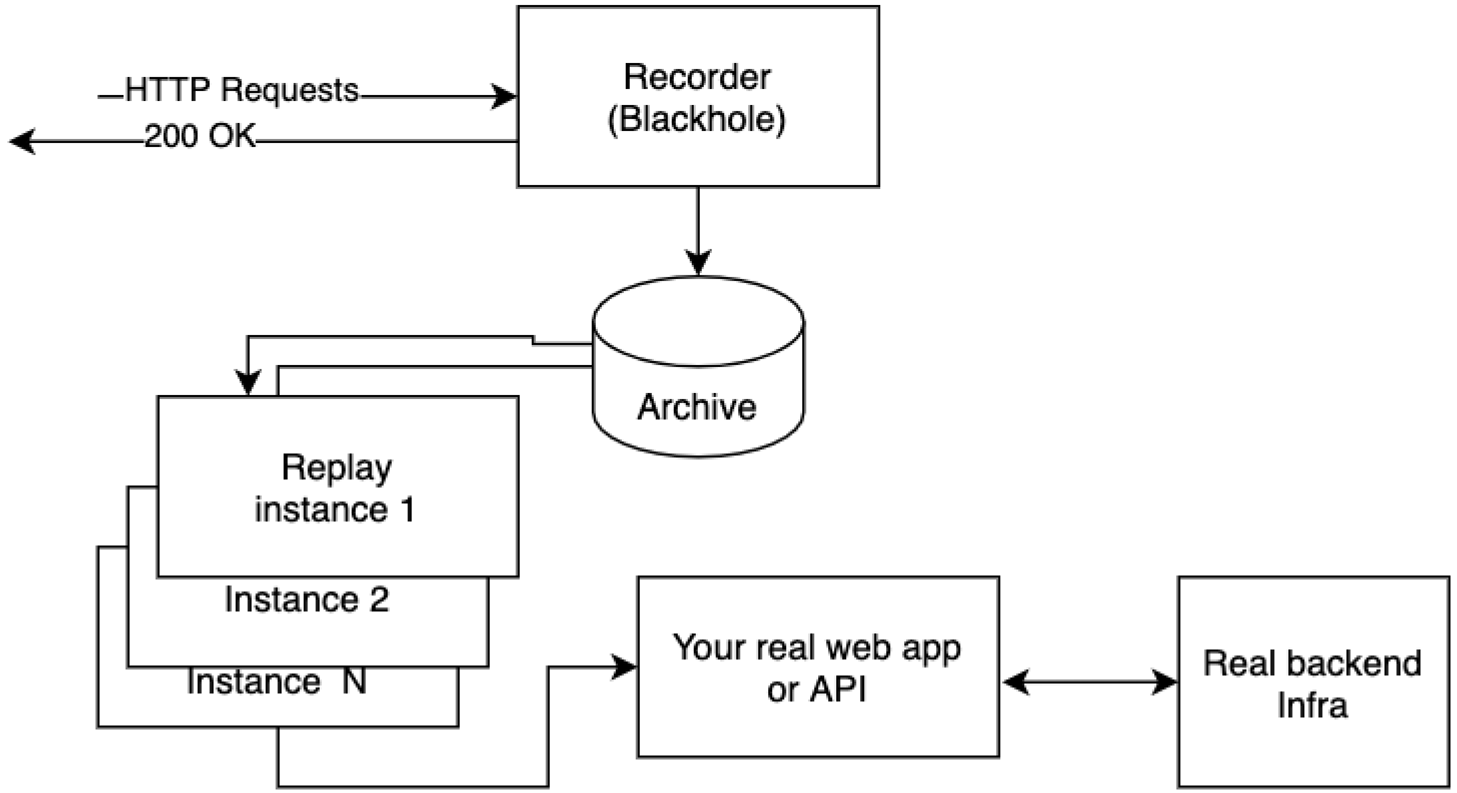
Usecase #2

- Capture traffic and replay specific requests.



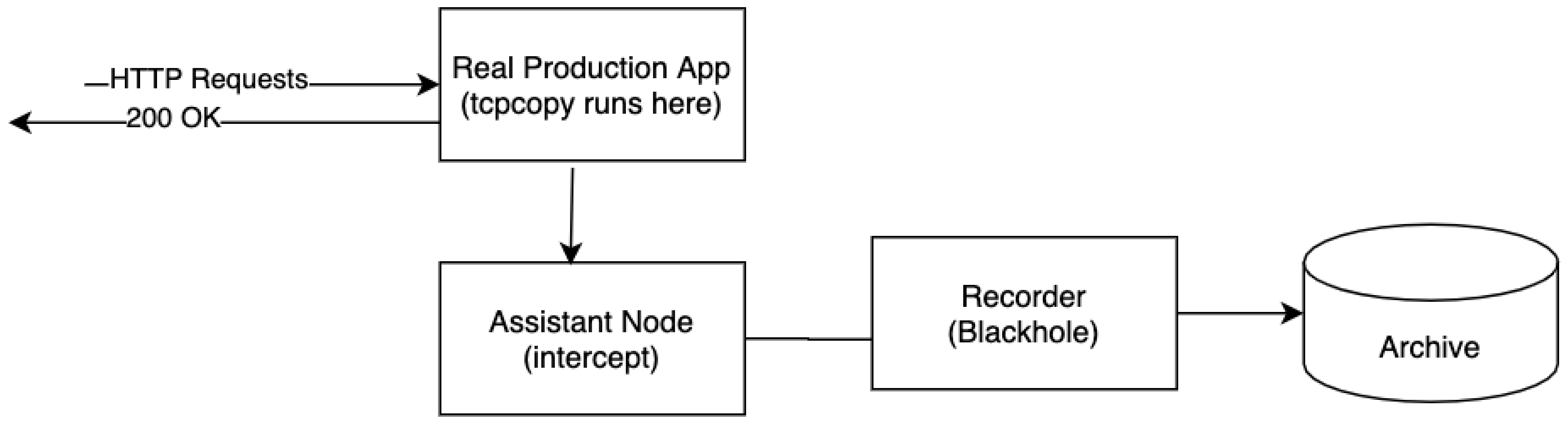
Usecase #3

- Create "more traffic" with real test data.



Usecase #4

- For clients where a simple http 200 wouldn't do
- You have a need to record / sample real production traffic
- NOTE: SSL termination must be external


HTTP Requests
200 OK
Real Production App (tcpcopy runs here)
Assistant Node (intercept)
Recorder (Blackhole)
Archive

Wait? where is the Rust connection?

- Go version can record ~150k req/sec of 2k payload on a Macbook Pro
- Was curious where the limit was and I had a Rust-ing itch to scratch.
- WIP http://github.com/adobe/void (experimental/prototype-level, not functional)

Optimization factors: Important ones (common to both Go or Rust)

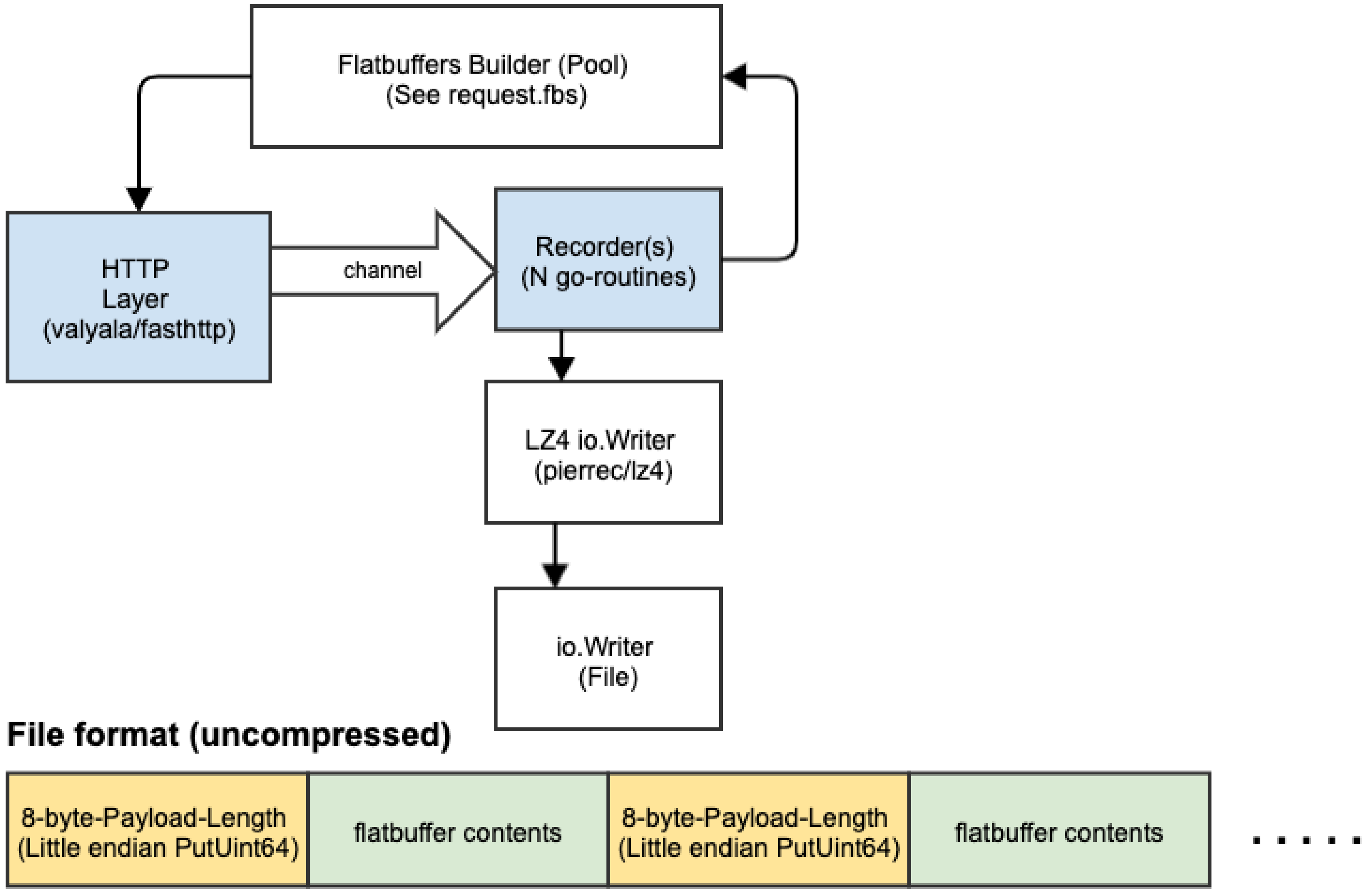
- Client connection reuse
- Creation of a stack (real or green thread) per request
- Memory allocations and copies per request
- Stack allocation vs heap allocation
- General quality and efficiency of code
- 3rd Party module/crate support

Optimization factors: Less important

Contrary to what you might think...

- Go's garbage collector (will explain why)

Key architecture


Flatbuffers Builder (Pool)
(See request.fbs)
HTTP Layer (valyala/fasthttp)
channel
Recorder(s) (N go-routines)
LZ4 io.Writer (pierrec/lz4)
io.Writer (File)
File format (uncompressed)
8-byte-Payload-Length (Little endian PutUint64)
flatbuffer contents
8-byte-Payload-Length (Little endian PutUint64)
flatbuffer contents
. . . . .

Also see https://google.github.io/flatbuffers/flatbuffers_benchmarks.html

Key architecture - HTTP layer fasthttp

Handler code in Go

- ctx object is reused and backed up by reusable buffers
- promises `0 allocs/op` - that is no new heap alloc per request
- No dangling references allowed when method terminates

```
// fastHTTPHandler is the request handler in fasthttp style, i.e. just plain function.
func fastHTTPHandler(ctx *fasthttp.RequestCtx) {

    if recordReqChan != nil {
        ar := request.CreateRequestFromFastHTTPCtx(ctx)
        recordReqChan <- ar
    }
}
```

Key architecture - HTTP layer fasthttp

Well slightly more code

```
srv := &fasthttp.Server{
    Handler: fastHTTPHandler,
}
err := srv.Serve(_ln)
if err != nil {
    log.Fatalf("http server failed with error: %+v", err)
}
```

Key architecture - HTTP layer in Rust

Rust (Decision time.. can I `async` or not?)

- hyper.rs for HTTP layer
- tokio::sync::mpsc::channel for channel
- Ideally want MPMC (Multiple Producer, Multiple Consumer)
- Settling for MPSC (Single Consumer)
- Each crate needs to be careful vetted for `blocking` calls.

Key architecture - HTTP layer Rust handler

Handler code in Rust (lots of freedom, closure nirvana)

```
let make_svc = make_service_fn(|_conn| {
    let (tx, rx) = tokio::sync::mpsc::channel(1000);

    let output_directory = output_directory.to_string();

    task::spawn(async move { attempt4::recorder(output_directory, rx).await });

    // tx is now a separate clone for each instance of http-connection
    async move {
        Ok::<_, Infallible>(service_fn(move |req: Request<Body>| {
            attempt4::handle(req, record, tx.clone())
        }))
    }
});

let server = Server::bind(&addr).serve(make_svc);
```

Key architecture - HTTP layer Rust handler

Handler code in Rust (lots of freedom, closure nirvana)

```
pub fn service_fn<F, R, S>(f: F) -> ServiceFn<F, R>
where
    F: FnMut(Request<R>) -> S,
    S: Future,
{
    ServiceFn {
        f,
        _req: PhantomData,
    }
}
```

Key architecture - HTTP layer Rust handler

Handler code in Rust (lots of freedom, closure nirvana)

```
pub async fn handle(
    req: Request<Body>,
    record: bool,
    mut tx: Sender<Box<flatbuffers::FlatBufferBuilder<'static>>>,
) -> Result<Response<Body>, Infallible> {

....
....
}
```

Key architecture - Serialization via Flatbuffers#

Back to Go. Remember this peice of code?

```
func fastHTTPHandler(ctx *fasthttp.RequestCtx) {

    if recordReqChan != nil {
        ar := request.CreateRequestFromFastHTTPCtx(ctx)
        recordReqChan <- ar
    }
}
```

Key architecture - Serialization via Flatbuffers#

It is all about (close to) "zero allocations" in the hot path

```
func CreateRequest(
    id, method, uri, headers, body []byte) (mr *MarshalledRequest) {
    mr = arPool.Get().(*MarshalledRequest)

    mr.fb.Reset()
    idFB := mr.fb.CreateByteString(id)
    methodFB := mr.fb.CreateByteString(method)
    uriFB := mr.fb.CreateByteString(uri)
    headersFB := mr.fb.CreateByteString(headers)
    bodyFB := mr.fb.CreateByteVector(body)
    fbr.RequestStart(mr.fb)
    fbr.RequestAddId(mr.fb, idFB)
    fbr.RequestAddMethod(mr.fb, methodFB)
    fbr.RequestAddUri(mr.fb, uriFB)
    fbr.RequestAddHeaders(mr.fb, headersFB)
    fbr.RequestAddBody(mr.fb, bodyFB)
    req := fbr.RequestEnd(mr.fb)
    mr.fb.Finish(req)

    return mr
}
```

Key architecture - Serialization via Flatbuffers#

Reading from channel and recording to disk

```
Loop:
for {
    select {
    case req, more := <-recordReqChan:
        if !more {
            break Loop
        }
        numRequests++
        err = rf.SaveRequest(req, false)
        if err != nil {
            ...
        }
        ...
        ...
```

Key architecture - Serialization via Flatbuffers#

Reading from channel and recording to disk

```
func (rf *Archive) SaveRequest(req *request.MarshalledRequest, flushNow bool) (err error) {
    ...
    fbBytes := req.Bytes()
    fbLen := len(fbBytes)

    defer req.Release()

    binary.LittleEndian.PutUint64(lbuf, uint64(fbLen))

    n, err := rf.Write(lbuf)
    if err != nil {
        ...
    }
    n, err = rf.Write(fbBytes)
    if err != nil {
        ...
    }
    ...
    ...
}
```

Key architecture - Serialization via Flatbuffers#

Real advantage of FlatBuffers comes in when you *read or deserialize*

```
// Request returns a pointer to the fbr.Request represented by the UnmarshalledRequest
func (umr *UnmarshalledRequest) Request() *fbr.Request {
    return fbr.GetRootAsRequest(umr.data, 0)
}
```

Key architecture - Writing to disk

For Go, there isn't really much to do compress the output stream

https://godoc.org/github.com/pierrec/lz4#NewWriter (https://godoc.org/github.com/pierrec/lz4#NewWriter)

```
func NewWriter(dst io.Writer) *Writer
```

NewWriter returns a new LZ4 frame encoder. No access to the underlying io.Writer is performed. The supplied Header is checked at the first Write. It is ok to change it before the first Write but then not until a Reset() is performed.

I/O Subsystem is automatically asynchronous and Go does the best possible sequencing already

Key architecture - Writing to disk

For Rust, there is little more work involved

- No `async/tokio` compatible crates available for LZ4
- Currently using `flate2` crate
- Still haven't figured out how to enable "async" variant :o

```
flate2 = { version = "0.2", features = ["tokio"] }
```

Key architecture - Writing to disk

For Rust, there is little more work involved

```
let mut raw_fp1 = std::fs::File::create(filename.clone()).expect("Create failed");

//let mut raw_fp2 = tokio::fs::File::create(filename.clone())
//    .await
//    .expect("Create failed");

let mut encoder = GzEncoder::new(raw_fp1, Compression::Default);

while let Some(builder) = rx.recv().await {
    let finished_data = builder.finished_data();
    encoder.write(finished_data).expect("write failed");
    super::POOL.attach(builder) // return builder to the pool
}
```

Current state of Rust prototype

- Many features are missing, but those are less of a problem
- Performance of Rust version is "90%" of the Go version. *Yes, I understand I have limited experience with Rust compared to Go*
- Ideally want to find the limit of what is possible *if I go the extra mile*
- Code legibility and maintanability is still important.
- Contributions and help welcome! http://github.com/adobe/void (http://github.com/adobe/void)

Thank you

Hari Bhaskaran
18 Dec 2020
hari@adobe.com (mailto:hari@adobe.com)
https://twitter.com/yetanotherfella (https://twitter.com/yetanotherfella)