



# ElasticFrame

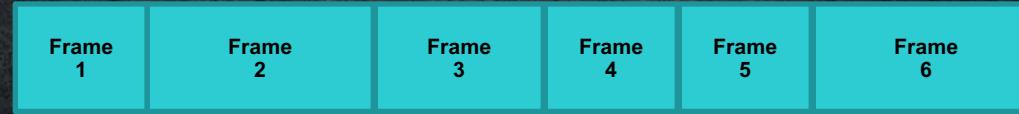
Protocol



# THE PROBLEM

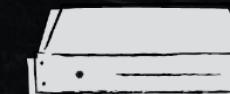


Source framing / ADTS / NAL / NV12 / Private Data



Some unknown underlying framing  
(UDP/SRT/RIST/RTMP)

The receiver does not understand  
The data content



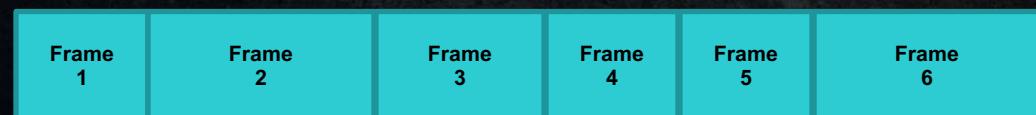
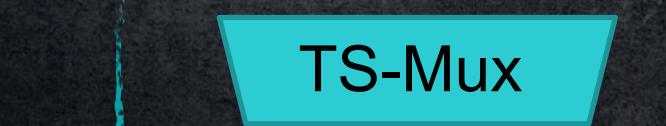
# THE OLD MEDIA WAY (MPEG-TS)



Source framing / ADTS / NAL / WHATEVER



Some underlying framing



The receiver do understand



# MPEG-TS MULTIPLEX SOLUTION

- Designed to solve media transport over legacy infrastructure (ASI / ATM aso.)
- Do not deal with out of order delivery
- Adds delay in CBR mode
- Has no built-in support for native IP multiplexing (multiple ports)
- High overhead 2% for the top layer (TS packet) then depending on underlying PES frame size ~6% overhead.
- Do not (IP networks) match underlying MTU (TS was designed to match ATM MTU but later in the standardization process ATM changed MTU to 48 bytes)
- Has no working sense of loss for IP networks (only 4 bits CC in fragment level)

# BUT THERE ARE SOLUTIONS OUT THERE

- Most solutions binds to an underlying transport
- Some suffers from HOL (RTMP aso.)
- Some do not take a media format (JPEG-XS/ H264) to protocol (Any) responsibility
- ALL of them adds unnecessary headers and lowers the payload of what you want to transport
- Some are under licenses that you do not want in your code and some are closed source.



# ELASTICFRAMEPROTOCOL (SEND)

Super simple API ->

Creation of the source

```
ElasticFrameProtocol myEFPSender(MTU, ElasticFrameMode::sender);  
myEFPSender.sendCallback = std::bind(&sendData, std::placeholders::_1);
```

Generating packets of MTU-size to be sent by the underlying transport layer ->

```
void sendData(const std::vector<uint8_t> &subPacket) {
```

When the user instructs the protocol to send data of some sort

```
myEFPSender.packAndSend(myData, ElasticFrameContent::h264, 0, 'ANXB', 2, NO_FLAGS);
```

# ELASTICFRAMEPROTOCOL (RECEIVE)

Super simple API ->

Creation of the receiver (callback + unfinished frames timeout + head of line blocking flush timeout )

```
ElasticFrameProtocol myEFPReciever(0,ElasticFrameMode::receiver);
myEFPReciever.receiveCallback = std::bind(&gotData, std::placeholders::_1);
myEFPReciever.startReceiver(5, 2);
```

Unpack the incoming data->

```
myEFPReciever.receiveFragment(myData,0);
```

Get the incoming frames

```
void getData(ElasticFrameProtocol::pFramePtr &frame) {
}
```

# ELASTICFRAMEPROTOCOL

- Lowest latency possible
- Supports out of order delivery or managed head-of-line blocking
  - Dynamic out of order delivery to strict Head of line delivery of data.
  - Details later in this presentation
- Low overhead (MTU – 8 bytes payload !!)
  - Example MTU 1500 == overhead of 0.5%
- $(MTU - 8) * 65536$  bytes super-frame size
  - Example SRT == 90 Megabytes maximum frame size.
- Support ‘first come first serve’ with multiple inputs from the same source.
  - One 1+1 mode (other modes to be implemented)
- Can multiplex 256 streams (details later in this presentation)

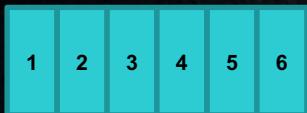
# HOW IT WORKS HIGH LEVEL

# ELASTICFRAMEPROTOCOL

Source frames (example H264 NAL)



Source frames (example ADTS)



Source frames (example Private data)



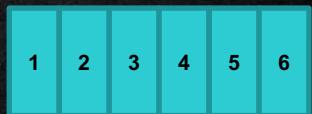
EFP

Some underlying framing  
(Any with MIN\_MTU of 256)

Destination frames (example H264 NAL)



Destination frames (example ADTS)

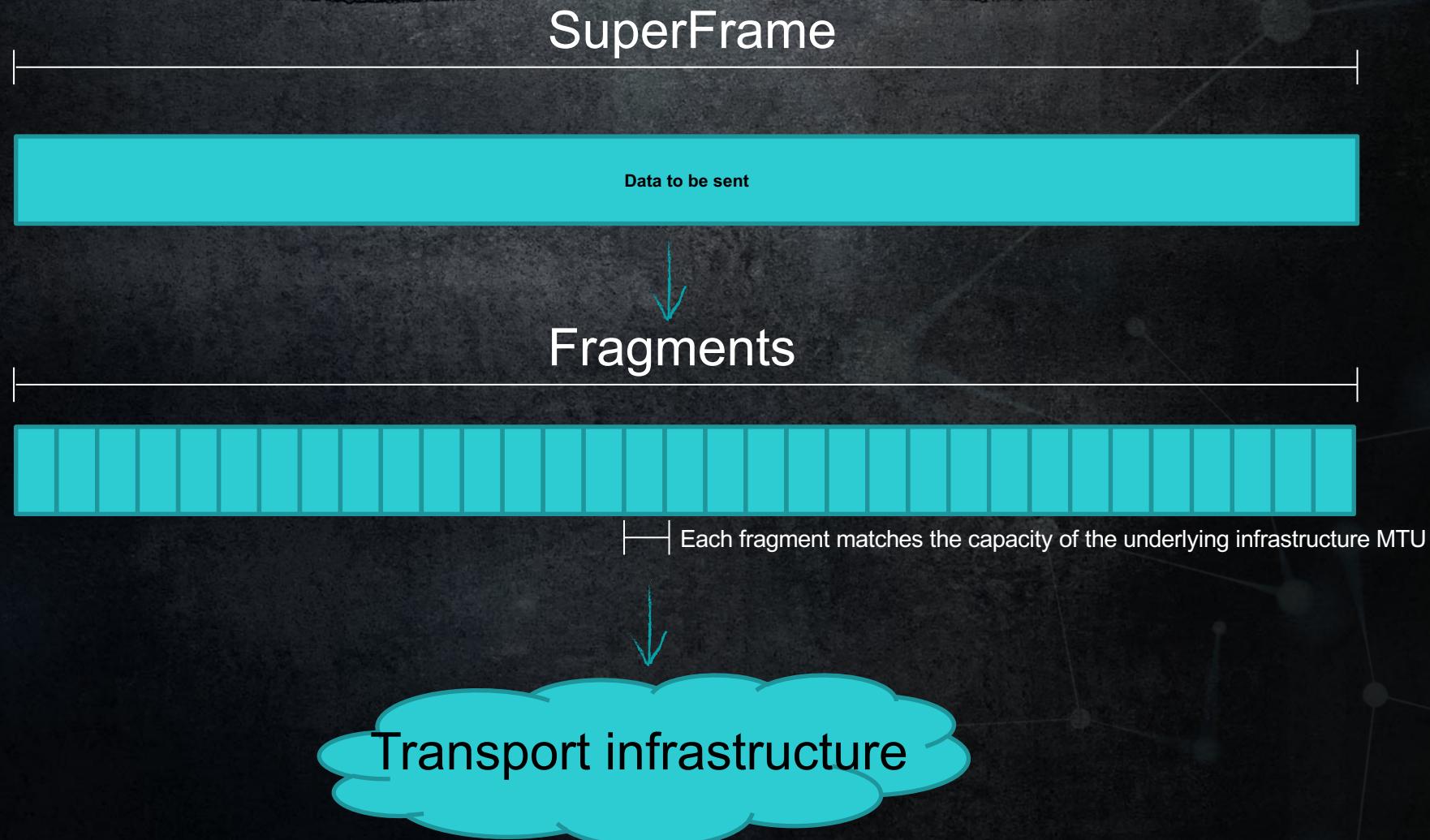


Destination frames (example Private data)



- Maintains delivery order
  - Also supports out of order
- PTS support
- HOL aging support
- DataType
- Data integrity indication

# ELASTICFRAMEPROTOCOL FRAGMENTS



# ELASTICFRAMEPROTOCOL FRAGMENTS

- A superFrame consists of
  - $X * \text{type1}$  fragments if the superFrame is larger than MTU - sizeof(type2Frame)
  - Always a type2 frame (single or at the end following  $x * \text{type1}$  frames)

SuperFrame



```
struct ElasticFrameType1 {  
    uint8_t hFrameType = Frametype::type1;  
    uint8_t hStream = 0;  
    uint16_t hSuperFrameNo = 0;  
    uint16_t hFragmentNo = 0;  
    uint16_t h0fFragmentNo = 0;  
};
```

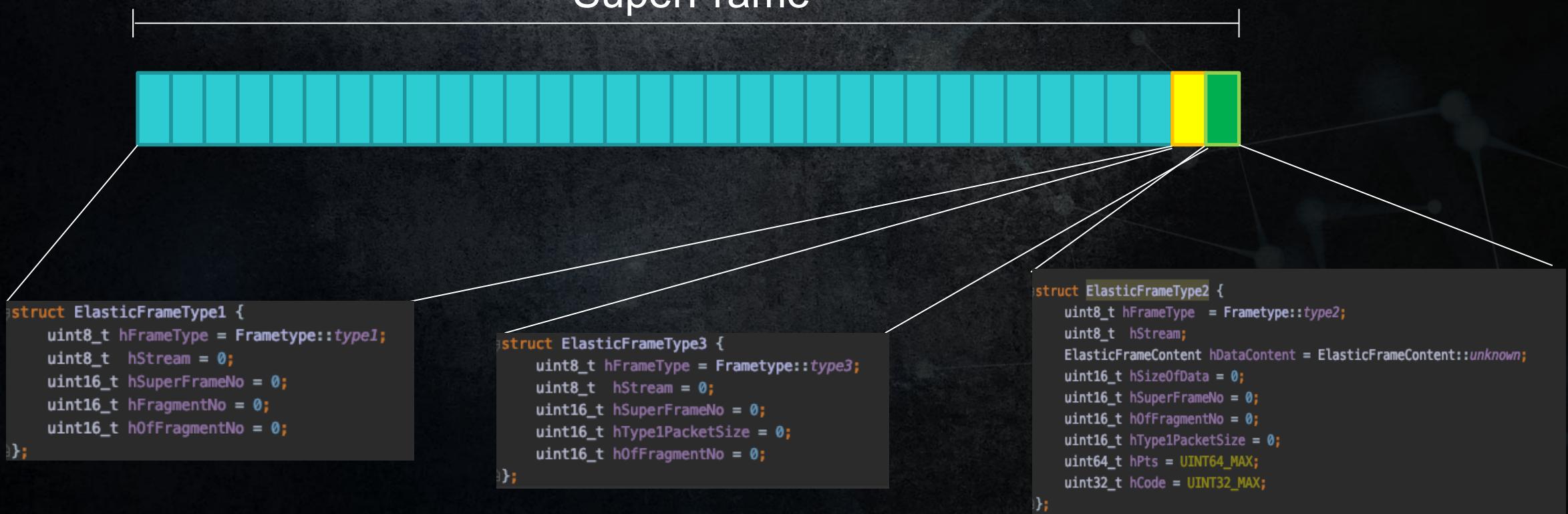
```
struct ElasticFrameType2 {  
    uint8_t hFrameType = Frametype::type2;  
    uint8_t hStream;  
    ElasticFrameContent hDataContent = ElasticFrameContent::unknown;  
    uint16_t hSizeOfData = 0;  
    uint16_t hSuperFrameNo = 0;  
    uint16_t h0fFragmentNo = 0;  
    uint16_t hType1PacketSize = 0;  
    uint64_t hPts = UINT64_MAX;  
    uint32_t hCode = UINT32_MAX;  
};
```

# ELASTICFRAMEPROTOCOL FRAGMENTS

There is a corner case where the excess data does not fit the type2 frame

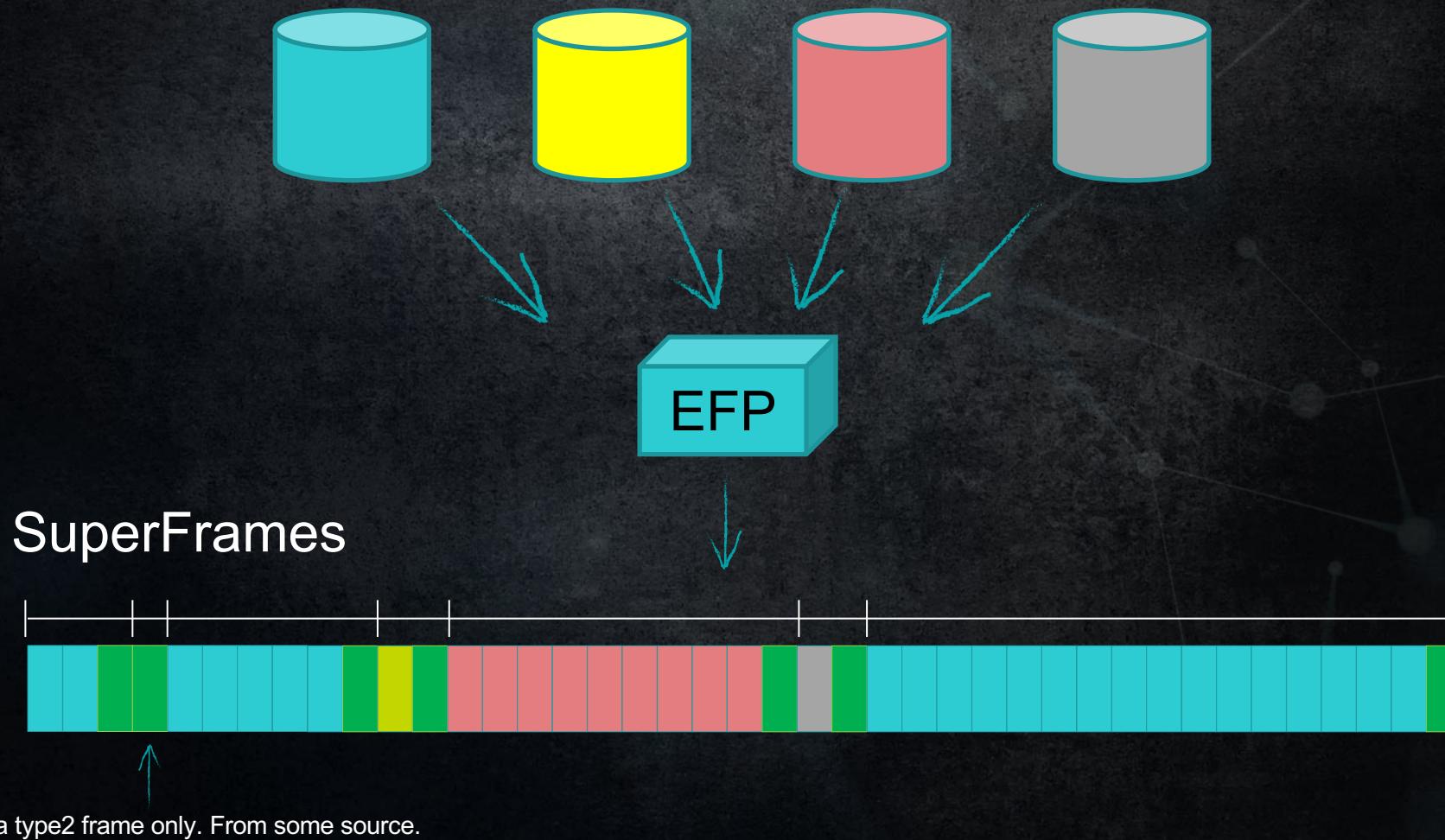
The protocol then ‘tail’ the data in a type3 frame and then terminate the super frame with a type2 declaring the stream content. (The stream concept is detailed later).

SuperFrame

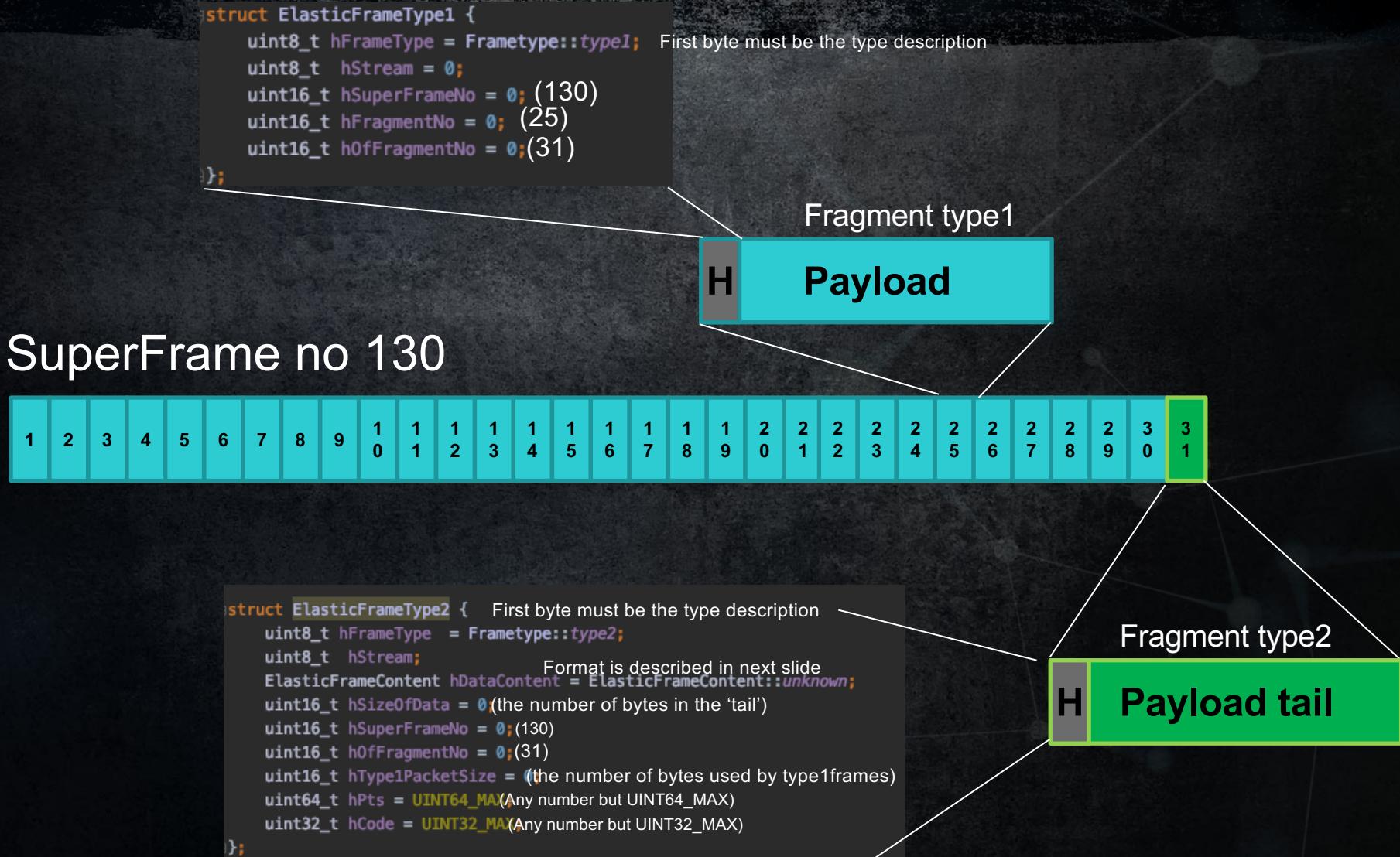


# ELASTICFRAMEPROTOCOL SUPERFRAMES

Different data sources

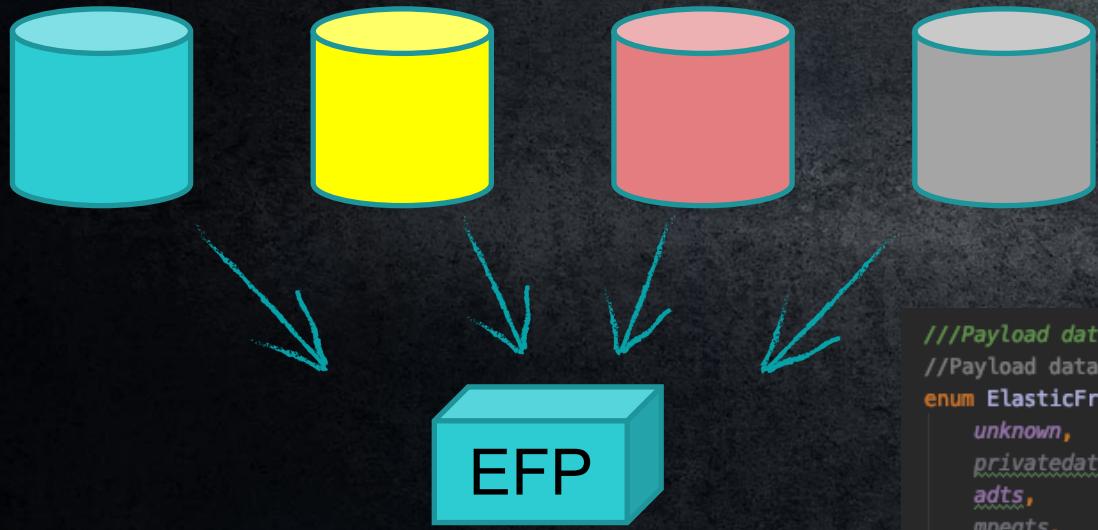


# ELASTICFRAMEPROTOCOL HIERARCHY



# ELASTICFRAMEPROTOCOL CONTENT DEFINITION

## Different data sources



- Datatypes are defined in the fragments
- If the MSB is set (uint8\_t) then
  - 'Code' contain a format description
  - Usually a variant. Example, Annex B or AVCC framing of H.264

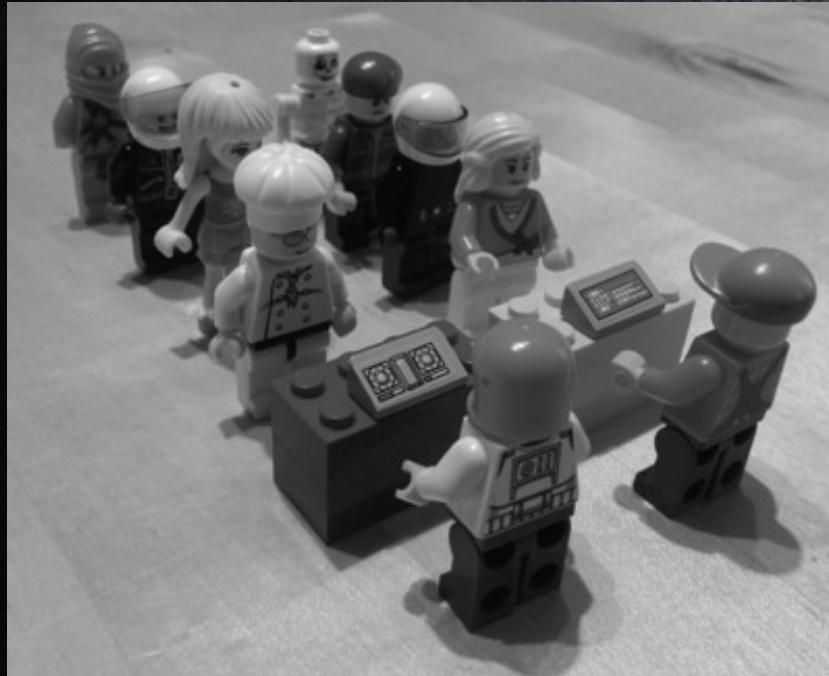
```
//Payload data types
//Payload data defines ----- START -----
enum ElasticFrameContentDefines : uint8_t {
    unknown,                      //Standard
    privatedata,                  //Any user defined format
    ads,                          //Mpeg-4 AAC ADTS framing
    mpeats,                       //ITU-T H.222 188byte TS
    mpeges,                       //ITU-T H.222 PES packets
    jpeg2000,                     //ITU-T T.800 Annex M
    jpeg,                         //ITU-T.81
    jpeaxs,                        //ISO/IEC 21122-3
    pcmaudio,                     //AES-3 framing
    ndi,                           /*TBD*/
    //Formats defined below (MSB='1') must also use 'code' to define the data format in the superframe
    didssdid = 0x80,               //FOURCC format
    sdi,                           //FOURCC format
    h264,                          //ITU-T H.264
    h265,                          //ITU-T H.265
};

//(FOURCC) (Must be the fourcc code for the format used)
//(FOURCC) (Must be the fourcc code for the format used)
//ANXB = Annex B framing / AVCC = AVCC framing
//ANXB = Annex B framing / AVCC = AVCC framing
```

# DELIVERY MECHANISM

# ELASTICFRAMEPROTOCOL DELIVERY

Head of line blocking



First come first serve

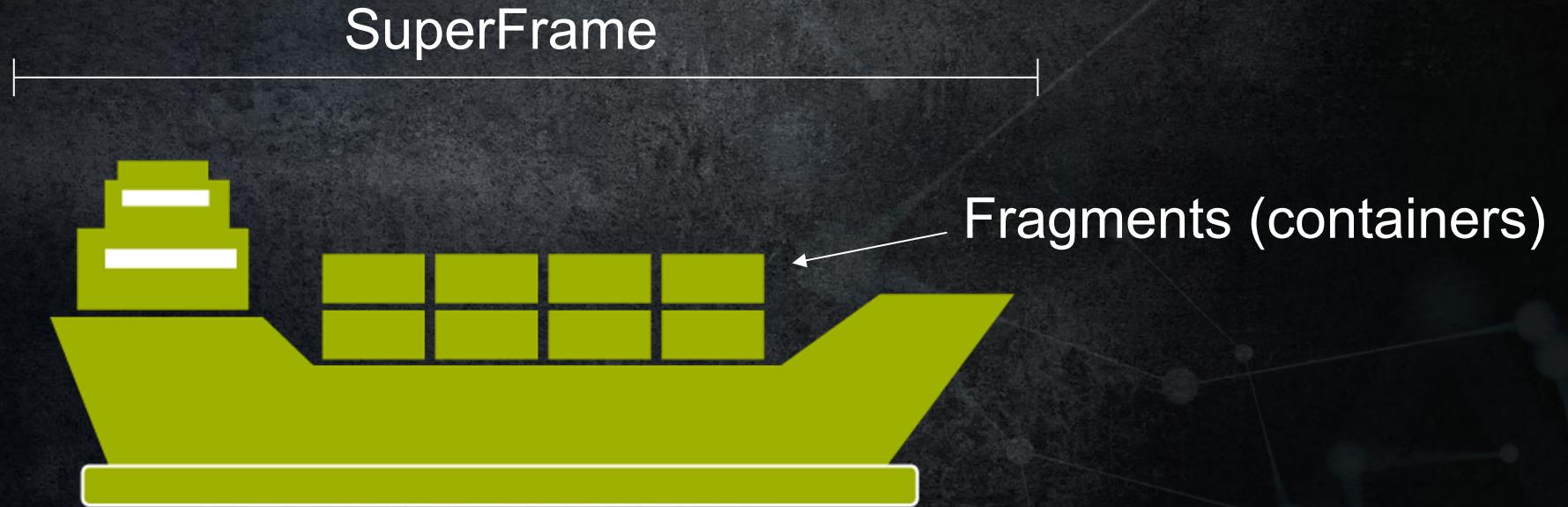


Both modes are supported by the protocol

# ELASTICFRAMEPROTOCOL HOL EXAMPLE

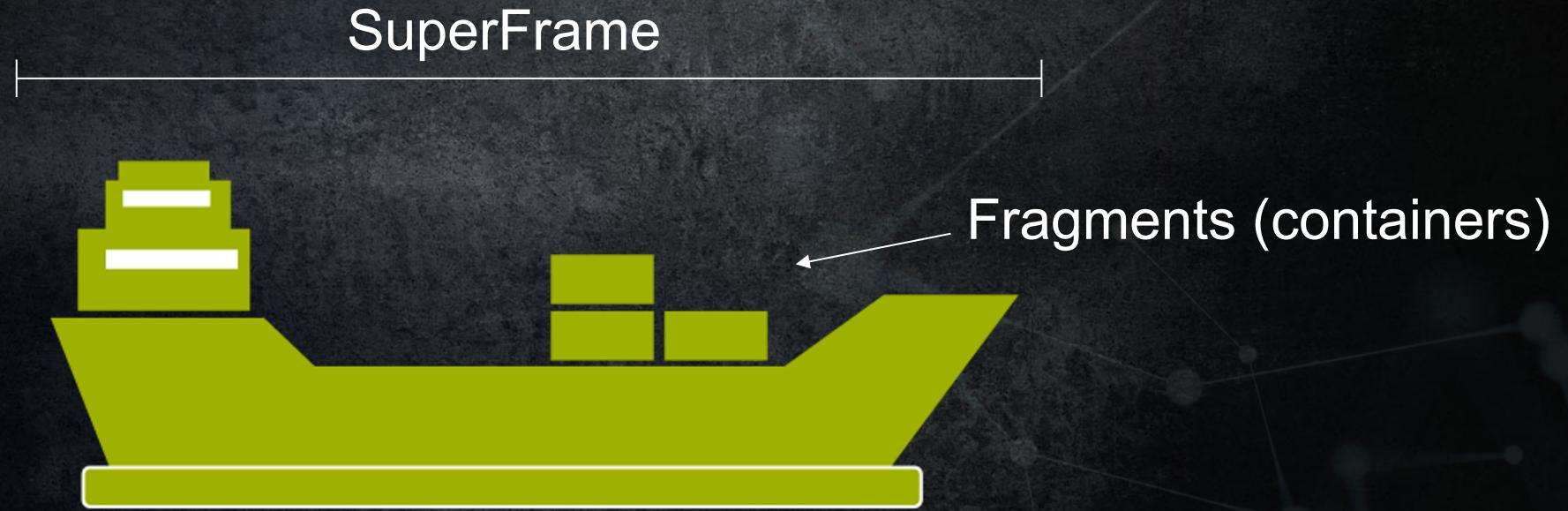
OBSERVE!!

Simplified examples to better explain the inner logic!



This example superFrame is from the datatype green and it contains all fragments meaning it can anchor and deliver it's gods

# ELASTICFRAMEPROTOCOL HOL EXAMPLE



This example superFrame is from the datatype green and it is missing fragments meaning if it anchors and deliver it's goods the receiver will not get all of It's expected payload

# ELASTICFRAMEPROTOCOL HOL EXAMPLE

A perfect day



35



34



33



32



31

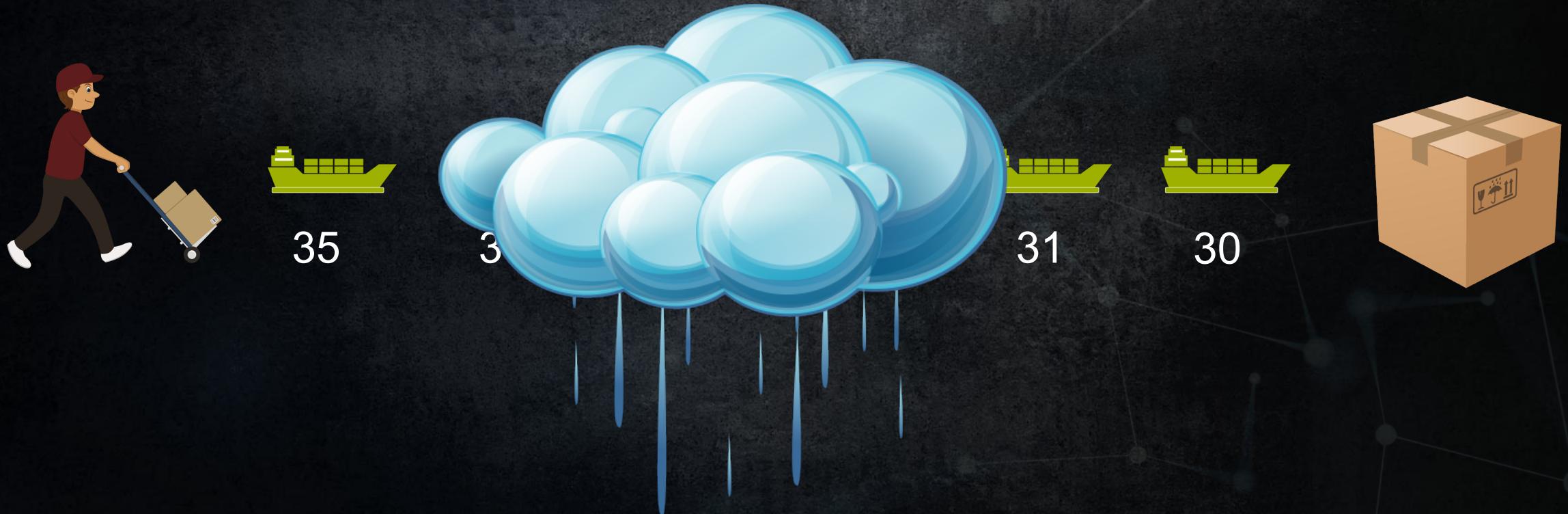


30



# ELASTICFRAMEPROTOCOL HOL EXAMPLE

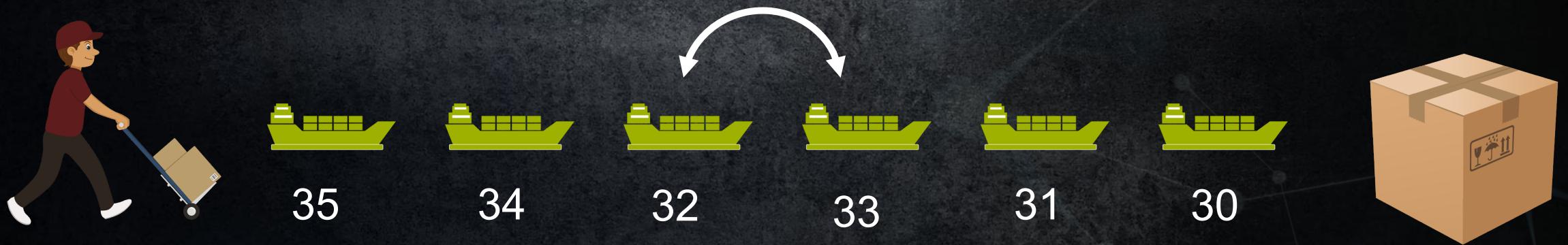
Some worse conditions



# ELASTICFRAMEPROTOCOL HOL EXAMPLE

Some worse conditions

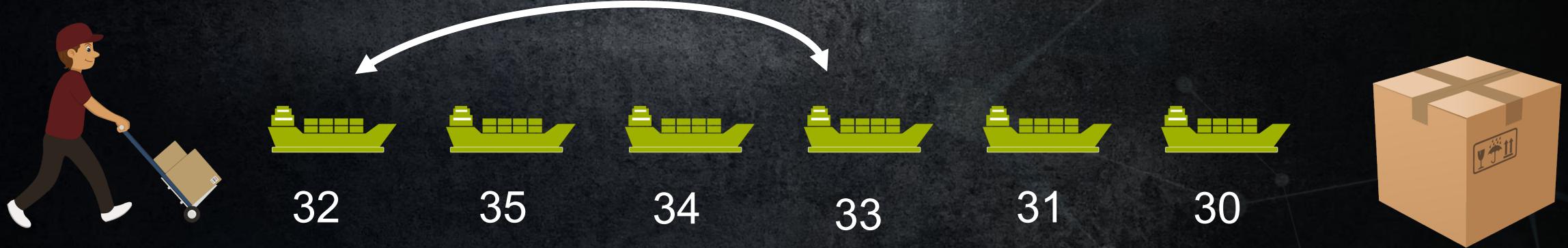
Delayed boat number 32 not much but a slight delay



At the receiving end  
They will wait for boat 32, when it  
arrives they will unload 32 before  
unloading boat 33

# ELASTICFRAMEPROTOCOL HOL EXAMPLE

Some worse conditions  
Delayed boat number 32 even more



```
myEFPReciever.startUnpacker(3, 1);
```

Number of 10ms cycles to wait for missing boats

At the receiving end they will wait for boat 32 before unloading boat 33 and boats with higher numbers. But if boat 32 takes too long time they will just unload the boats that arrived (in order) and ignore boat 32 when or if it arrives..

# ELASTICFRAMEPROTOCOL HOL EXAMPLE

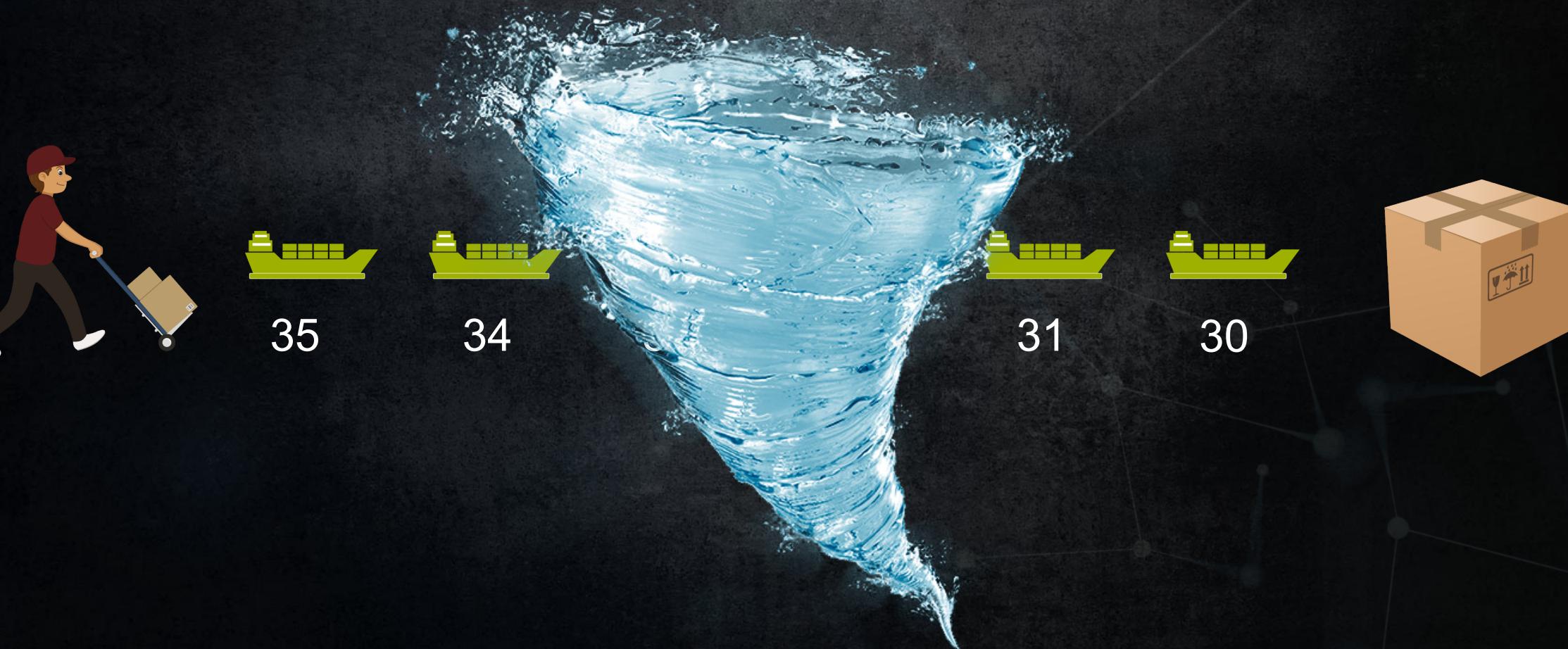
A terrible storm



35



34



31

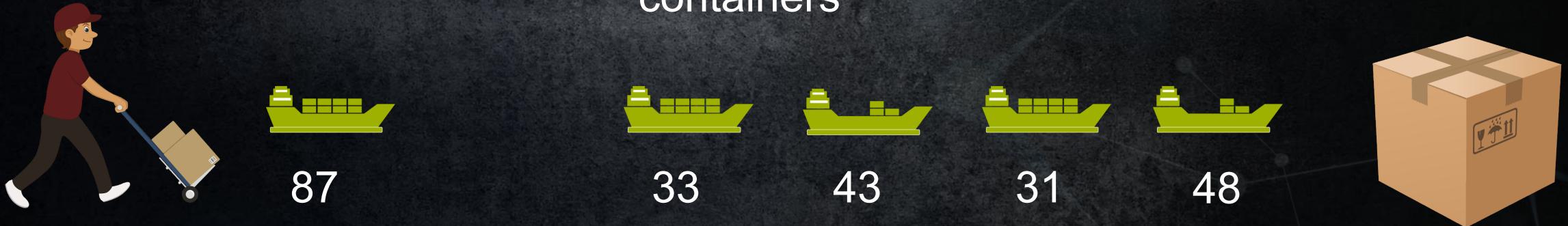


30

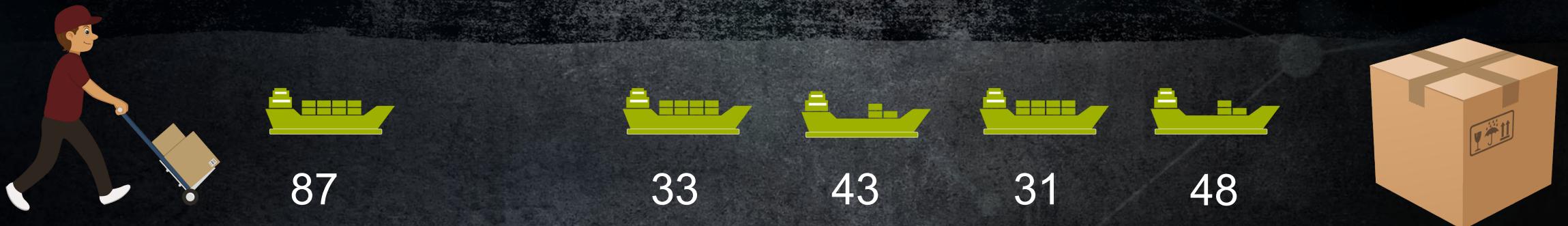


# ELASTICFRAMEPROTOCOL HOL EXAMPLE

This storm was so bad some boats went missing. Some boats came out of order and many boats lost containers



# ELASTICFRAMEPROTOCOL HOL EXAMPLE



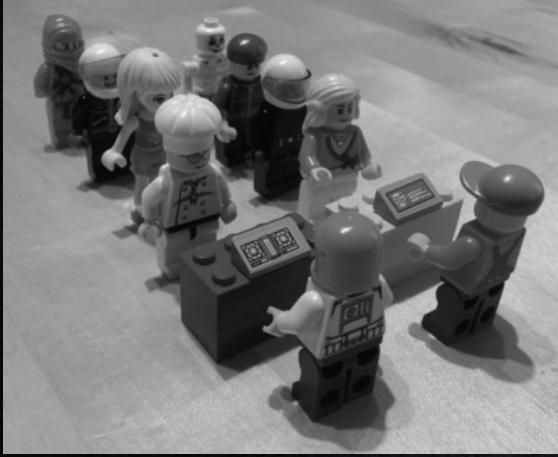
- In the above example 31 will be unloaded.
- Then we wait for boat 32 to arrive but if the timeout triggers we will unload 33
- Then we will wait again for any higher number boats to arrive with all it's containers but if they don't we unload 43 and discard 48 and 87.
  - 43 and 48 will be marked (not complete shipments)
- If boat 43 would be numbered 32 we would deliver 31. Then wait to see if we could find any missing containers that fell over board. Then after that we would proceed. The missing container search timeout is

`myEFPReciever.startUnpacker(3, 1);`

Number of 10ms cycles to wait for missing containers

# ELASTICFRAMEPROTOCOL HOL EXAMPLE

Head of line blocking



```
myEFPReciever.startUnpacker(3, 1);
```

Number of 10ms cycles to wait for missing containers

Number of 10ms cycles to wait for missing boats

(You can wait for containers for a longer time than waiting for missing boats in EFP)

# ELASTICFRAMEPROTOCOL FIRST COME. EXAMPLE

First come first serve



```
myEFPReciever.startUnpacker(3, 0);
```

↑  
Set to 0

# ELASTICFRAMEPROTOCOL FIRST COME. EXAMPLE

A perfect day



35



34



33



32



31



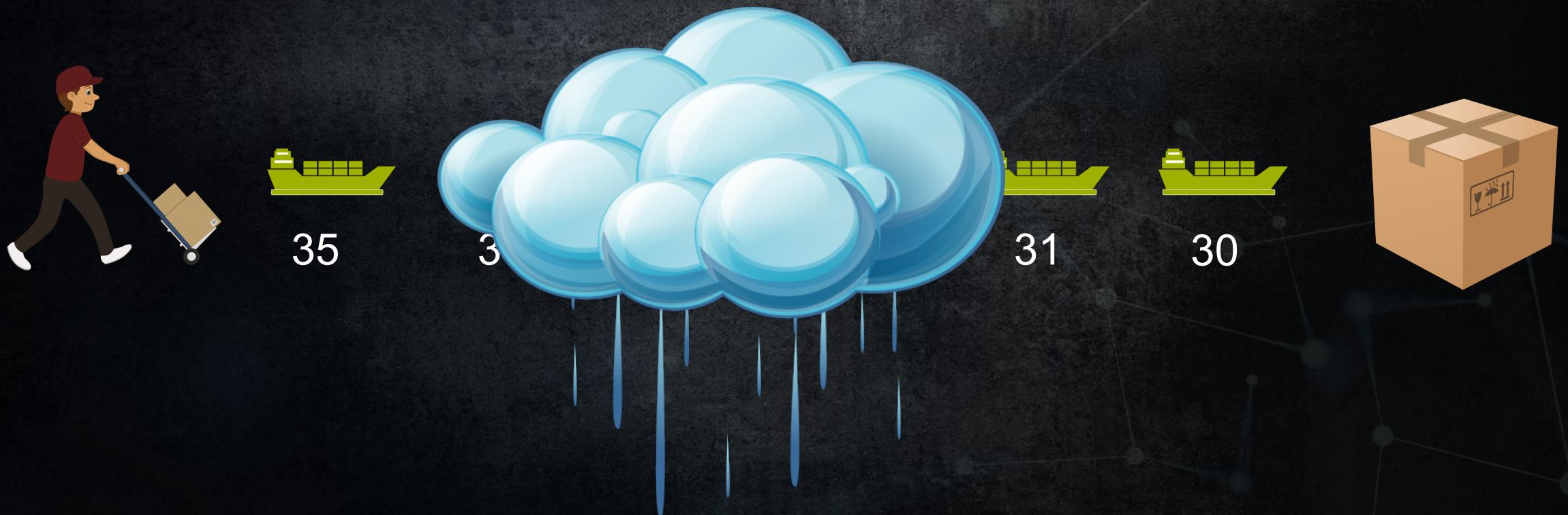
30



Exactly as HOL works

# ELASTICFRAMEPROTOCOL FIRST COME. EXAMPLE

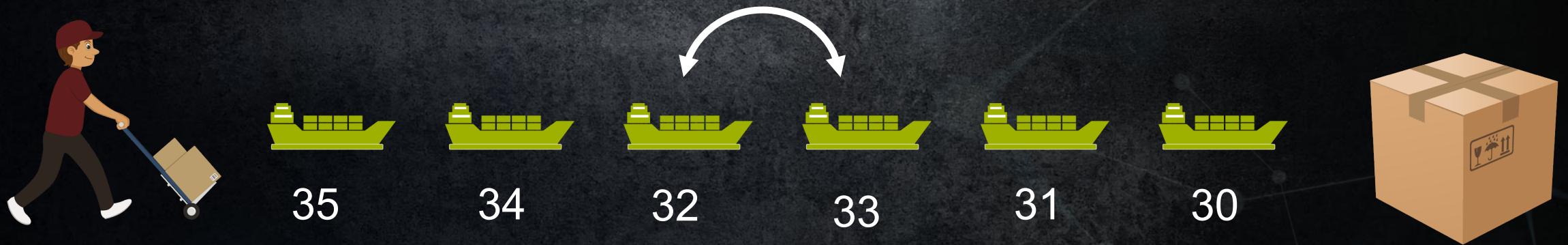
Some worse conditions



# ELASTICFRAMEPROTOCOL FIRST COME. EXAMPLE

Some worse conditions

Delayed boat number 32 not much but a slight delay



At the receiving end  
They will unload boats as they arrive  
No matter the number.

# ELASTICFRAMEPROTOCOL FIRST COME. EXAMPLE

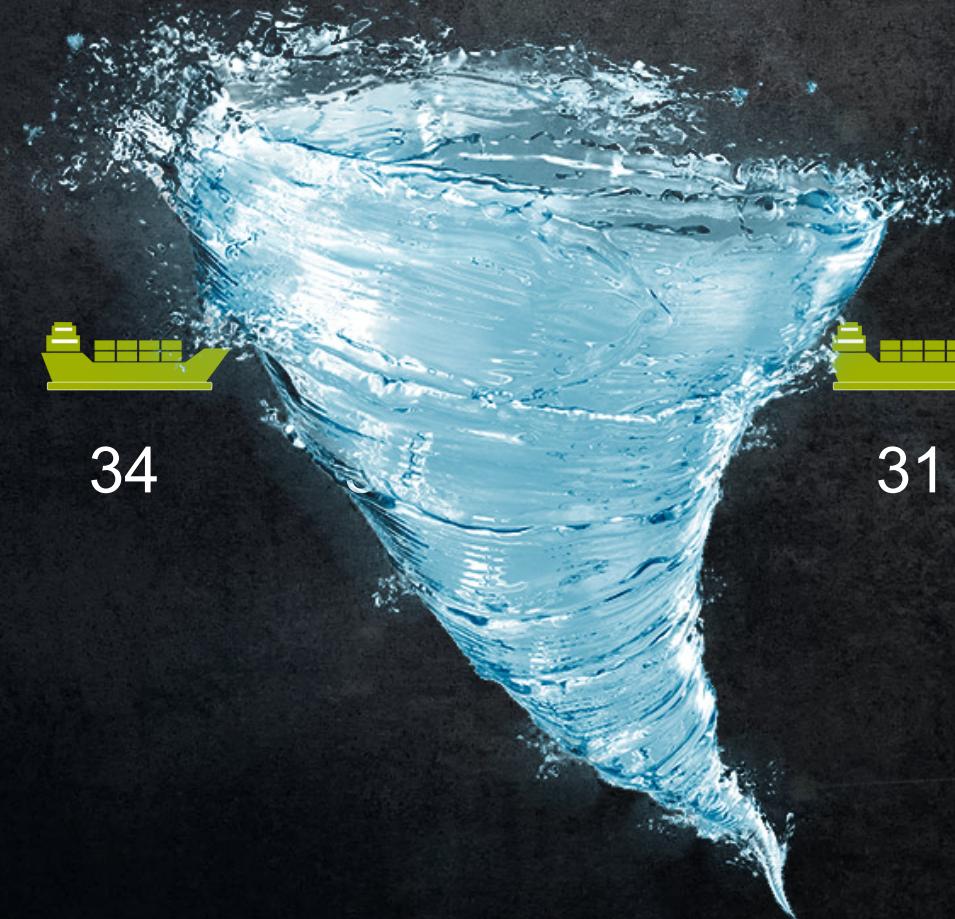
A terrible storm



35



34



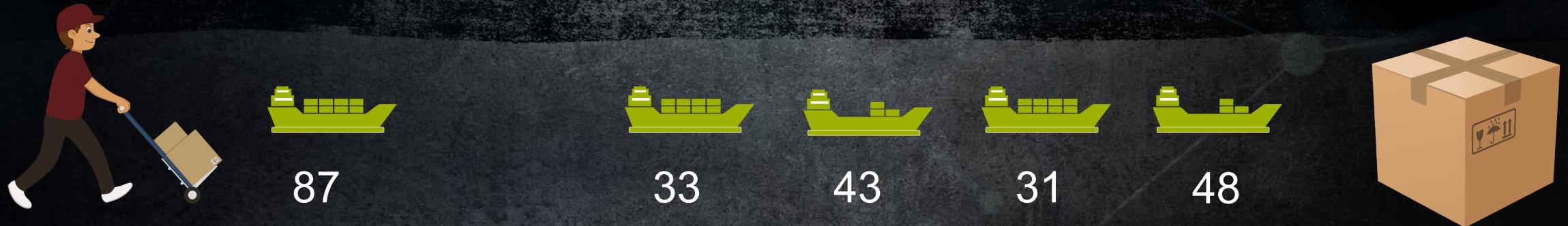
31



30



# ELASTICFRAMEPROTOCOL FIRST COME. EXAMPLE



- In the above example. Boat 31, 33 and 87 will be unloaded immediately.
- They will search for the containers for boat 43 and 48. If they find them they will deliver the content from those boats.
- Let's play with the scenario that we found the containers for boat 43 then we deliver ASAP.. But we did not find 100% of boat 48's containers . We then unload what we got. But tell the receiver that it's not everything.
- If containers from boat 48 appear later (after we delivered), we just ignore them.

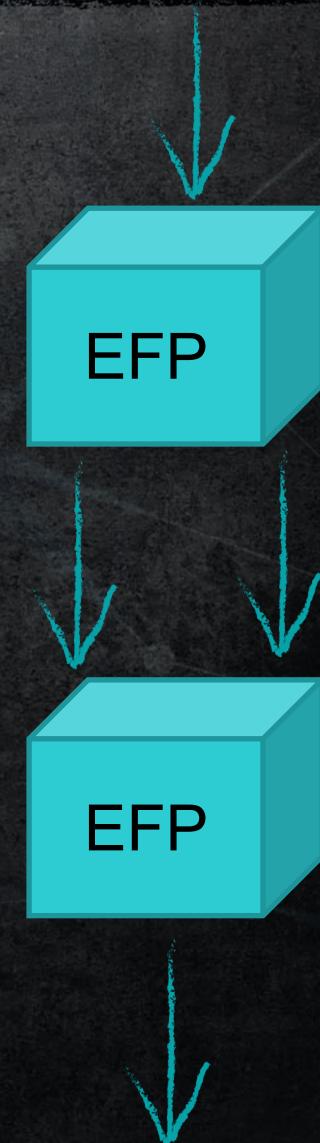
```
myEFPReciever.startUnpacker(3, 0);
```

Number of 10ms cycles to wait for missing containers

**1 + 1**

# ELASTICFRAMEPROTOCOL DELIVERY

- Stateless 1 + 1
- Must originate from same source
- Protects the network layer
- Can be used to lower delay
- Or lower protection protocols requirements
- Will consume 2 x BW



# ELASTICFRAMEPROTOCOL PROTECTION

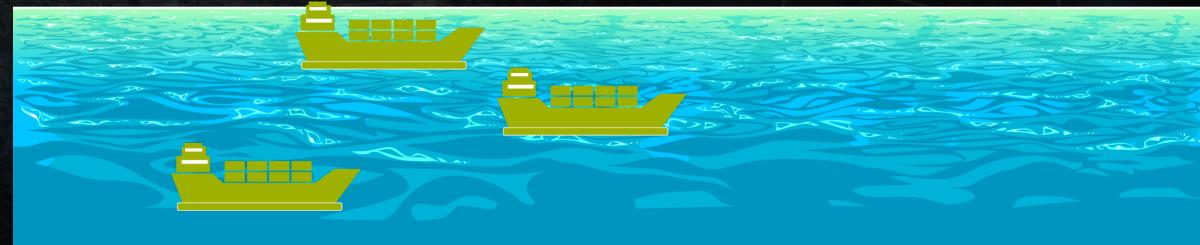
- Our shipper is encountering a situation where he needs to send ships from A to B.
- There are two paths to reach B from A, Route 1 and 2
- There is always a storm on either route1 or route 2.. Sometimes on booth routes.



Route 1



Route 2

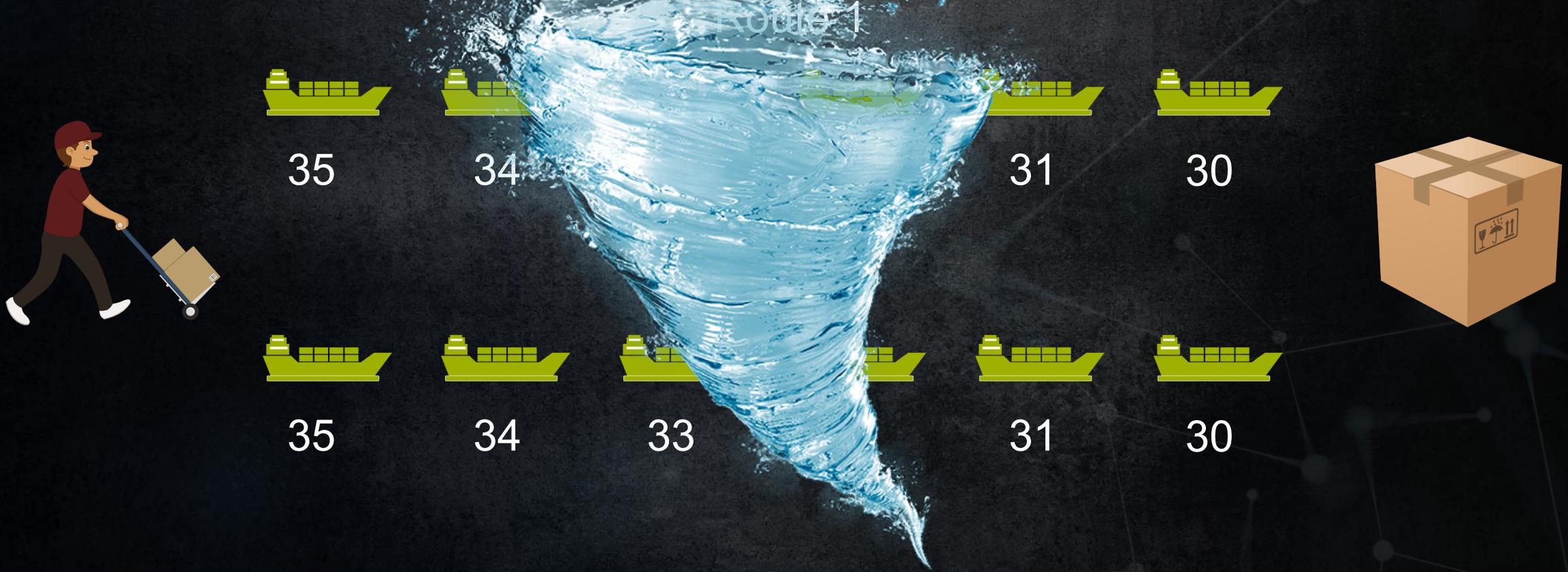


# ELASTICFRAMEPROTOCOL 1 + 1



- Our shipper decides to ship the same content twice. One taking route 1 and the other route 2 so that when the storms hit the shipments ->

# ELASTICFRAMEPROTOCOL 1 + 1

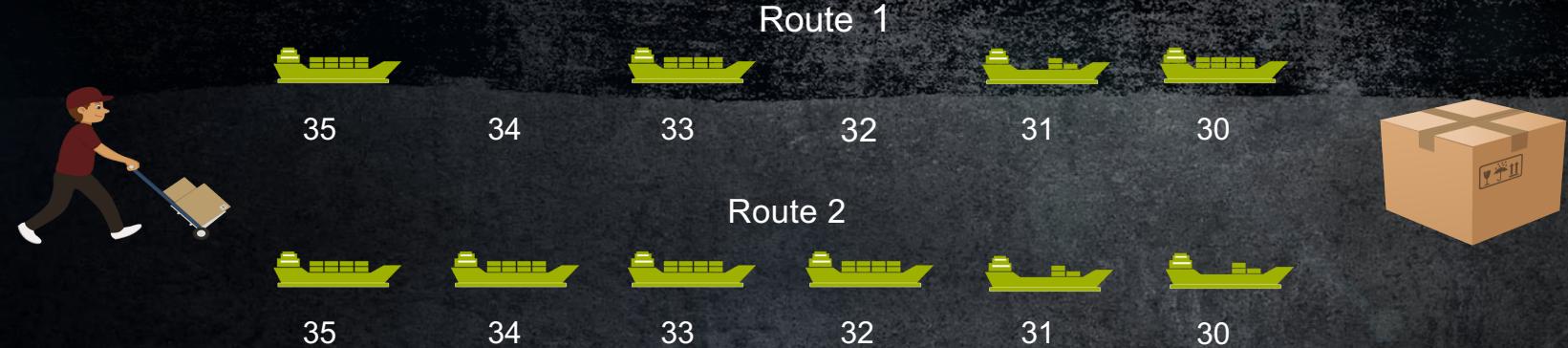


# ELASTICFRAMEPROTOCOL 1 + 1



During this event route1 and route2 where both hit by storms.  
After the storm the fleet of boats looks as above.

# ELASTICFRAMEPROTOCOL 1 + 1

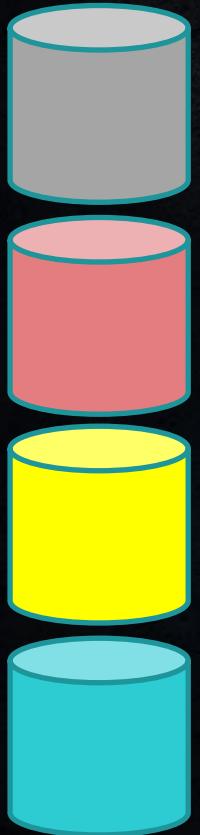


- 30 would be offloaded (both boats) and the containers matching would be first come first serve. The duplet containers are discarded.
- The containers from 31 same thing hoping for that there would be at least one of each container on either of the boats. If not that shipment will be non complete. And they would wait to see if the missing containers float ashore during the time out period.
- 33 is same as 30 but in this case we will get 200% hit-rate since both ships came complete.
- Boat 34 was lost on route1 but since we got a complete shipment on route 2 we can deliver this payload.
- Same thing for 35 with a 200% payload success.

# PROTOCOL INTERNAL

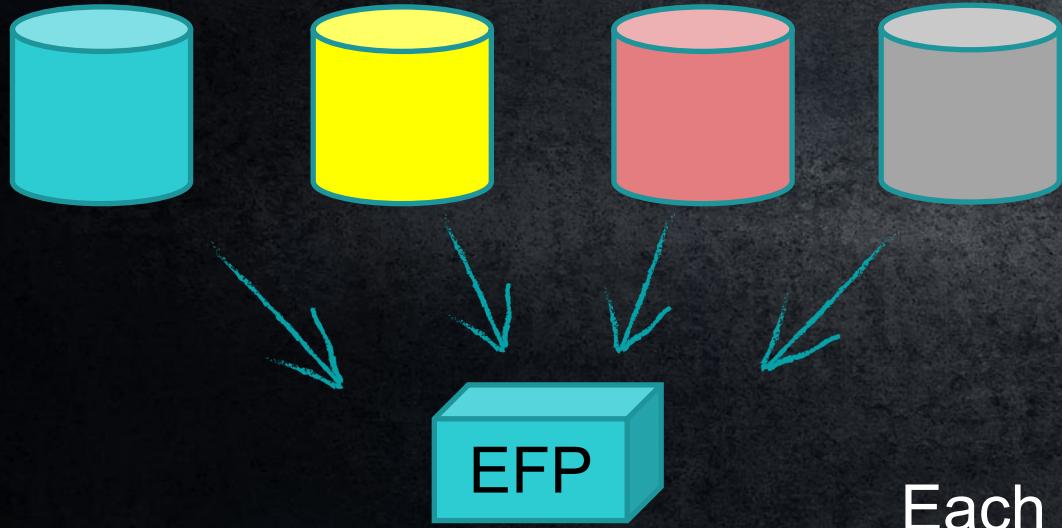
# ELASTICFRAMEPROTOCOL STREAMS

## Introduction to EFP-Streams



# ELASTICFRAMEPROTOCOL STREAMS

Different data sources



The data source can be of the same type  
Example.

H264 Annex B for data source 1  
H264 Annex B for data source 2  
MP4 ADTS for source 3  
MP4 ADTS for source 4  
MP4 ADTS for source 5

Each data source gets mapped to a stream

EFP has a stream-pool  
of 255 streams

(can be extended if  
needed)



# ELASTICFRAMEPROTOCOL STREAMS



Streams are stateless and can be replaced by another data sources. Of same type or different data type.

Can be used by external 1+1 mechanisms switching data from different sources

# ELASTICFRAMEPROTOCOL STREAMS

Streams makes it possible to send multiple services of the same type using the same underlying socket



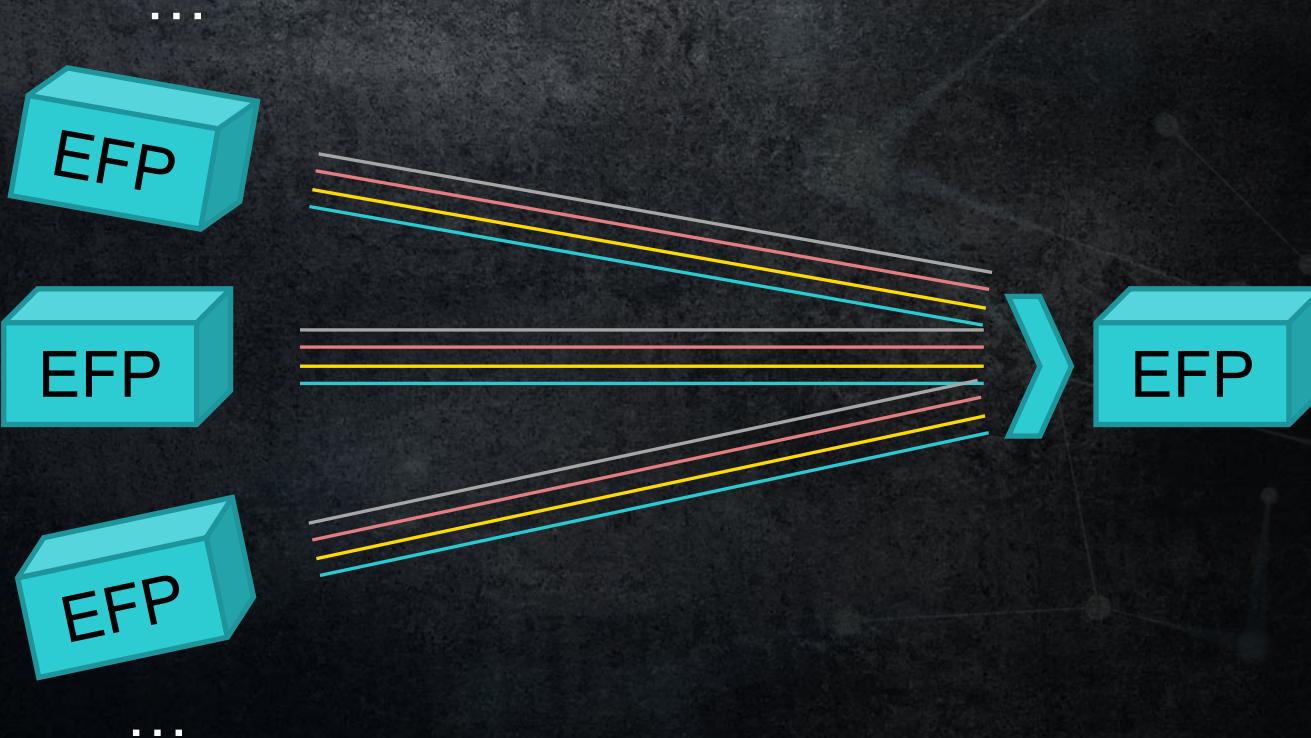
EFP provides:

- Stream
- StreamType
- PTS

Use private data for any additional information

# ELASTICFRAMEPROTOCOL STREAMS

An EFP receiver can receive traffic from 255 EFP senders each sending up to 255 streams



# WORKING WITH PRIVATE DATA

# ELASTICFRAMEPROTOCOL PRIVATE DATA

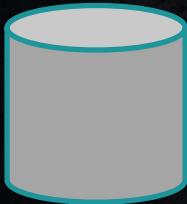
## 4 Different ways of sending private data

- *A stream*
  - Is a contiguous flow of data just as for example a video stream or audio stream
  - May be used for streaming content not defined by the protocol
- *Embedded data*
  - Embedded inside the actual payload of a stream
  - May be used to embed smaller payloads associated with that particular frame of data.
- *Type 0*
  - A packet inserted at any timepoint in the stream of network packets
  - May be used for inserting time critical data anywhere in the stream of all other multiplexed data.
  - A use case could be insertion of markers or time transfer protocols.
- *Opportunistic data transport*
  - Data filling packets up to MTU where the streams do not.
  - May be used for sending non time critical background data where there is no need to meet a timed delivery of the content.

All methods above may be combined.

# ELASTICFRAMEPROTOCOL STREAM

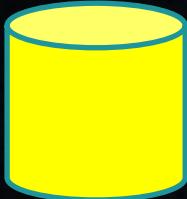
A stream can be of a known defined standard  
Or user defined



JPEG2000 (ITU-T T.800 Annex M)



ITU-T H.264 (Annex B)

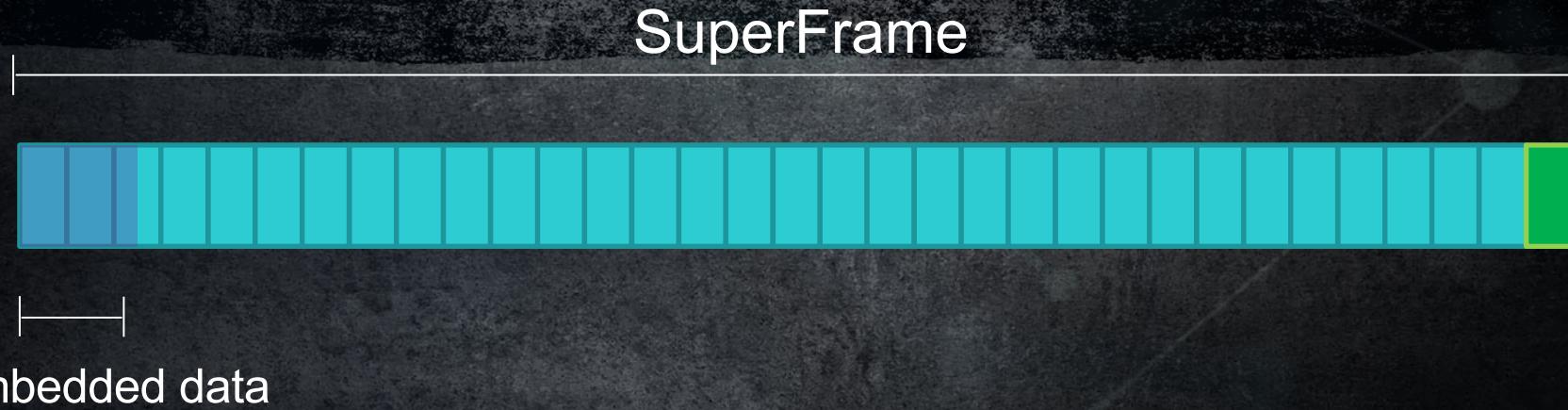


User defined



ITU-T H.264 (Annex B)

# ELASTICFRAMEPROTOCOL EMBEDDED DATA



- Private data can also be inserted at the beginning of a superFrame
  - In the embedded data segment (explained later)
- That private data can more easily be associated with that particular payload data since the callback containing the data also contains the private data.
- Using the EMBEDDED\_PAYLOAD flag indicates there is embedded data

# ELASTICFRAMEPROTOCOL EMBEDDED DATA

## SuperFrame



### Embedded data

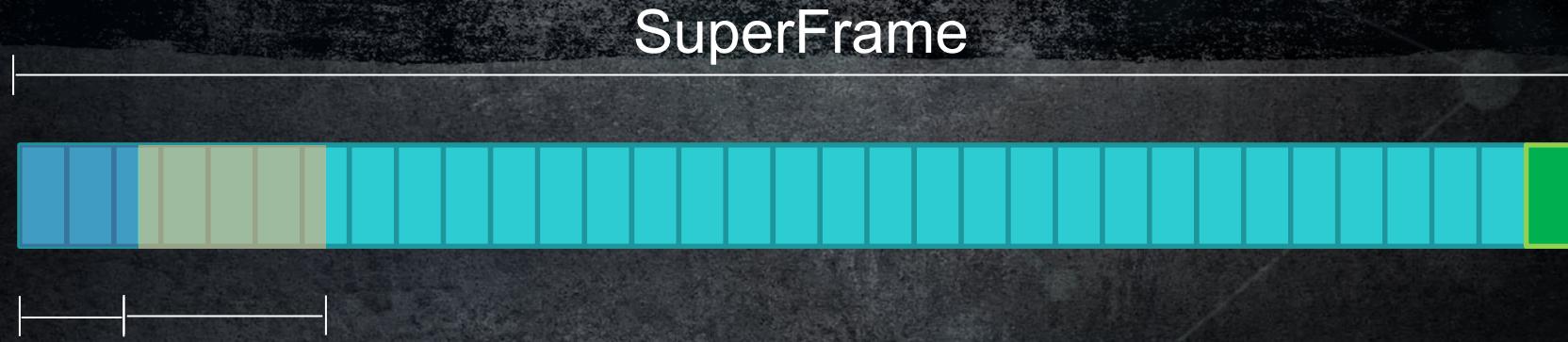
- Embedded data can be maximum 65535 bytes



```
struct ElasticEmbeddedHeader {  
    uint8_t embeddedFrameType = ElasticFrameEmbeddedContentDefines::illegal;  
    uint16_t size = 0;  
};
```

```
enum ElasticFrameEmbeddedContentDefines : uint8_t {  
    illegal,           //may not be used  
    embeddedPrivateData, //private data  
    h222PMT,          //pmt from h222 pids should be truncated to uint8_t leaving the LSB bits only then map to streams  
    mp4FragBox,        //All boxes from a mp4 fragment excluding the payload  
    lastEmbeddedContent = 0x80  
    //defines below here do not allow following embedded data.  
};
```

# ELASTICFRAMEPROTOCOL EMBEDDED DATA

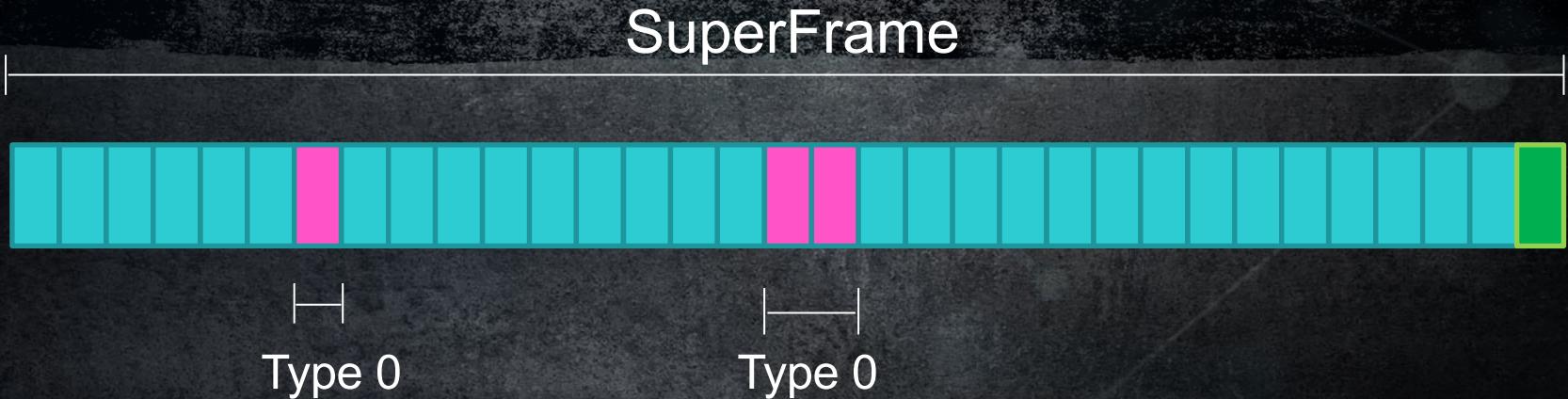


Embedded data

- Multiple embedded data segments can be concatenated. The last embedded content bit defines no further embedded content.

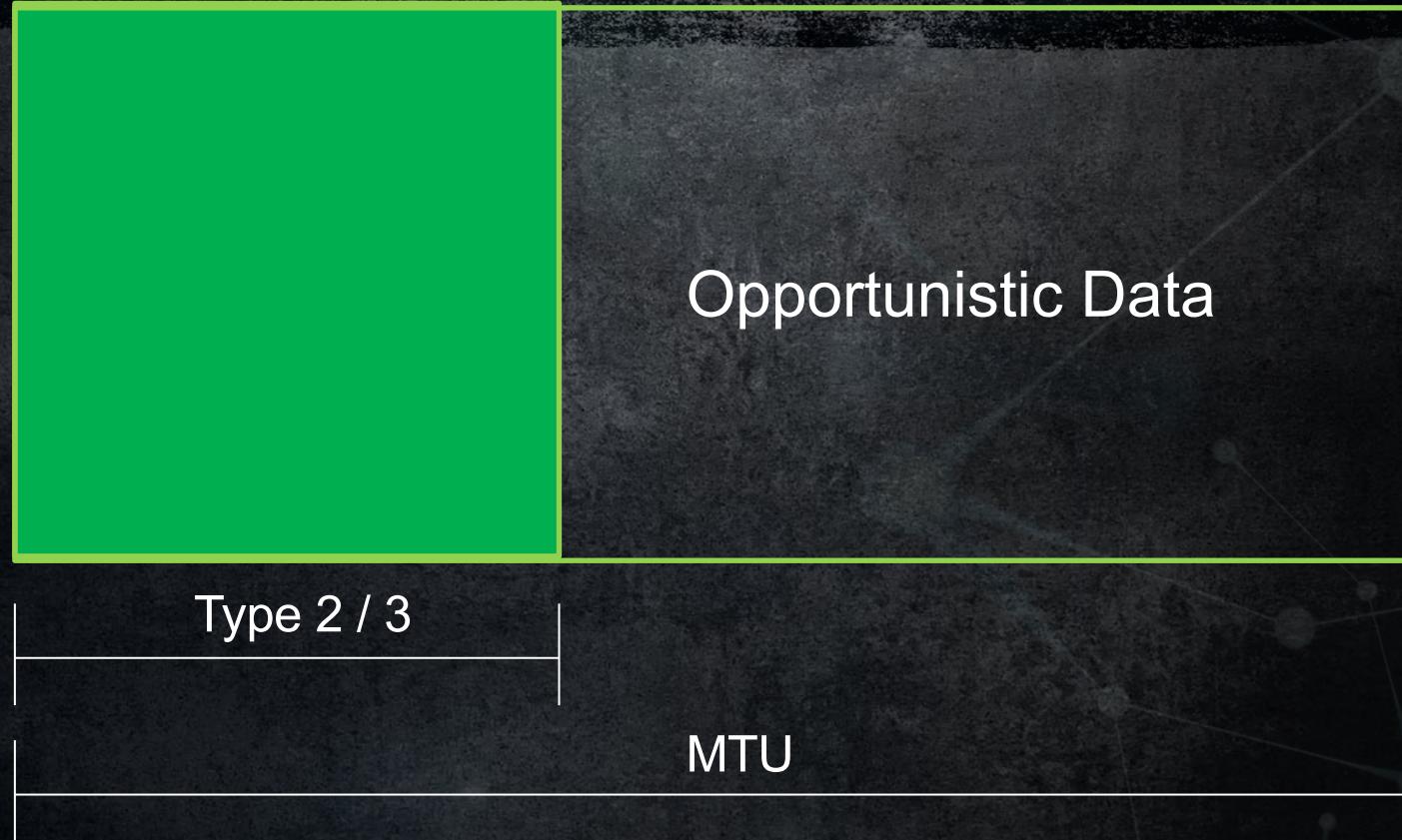
*lastEmbeddedContent = 0x80*

# ELASTICFRAMEPROTOCOL TYPE 0 FRAMES



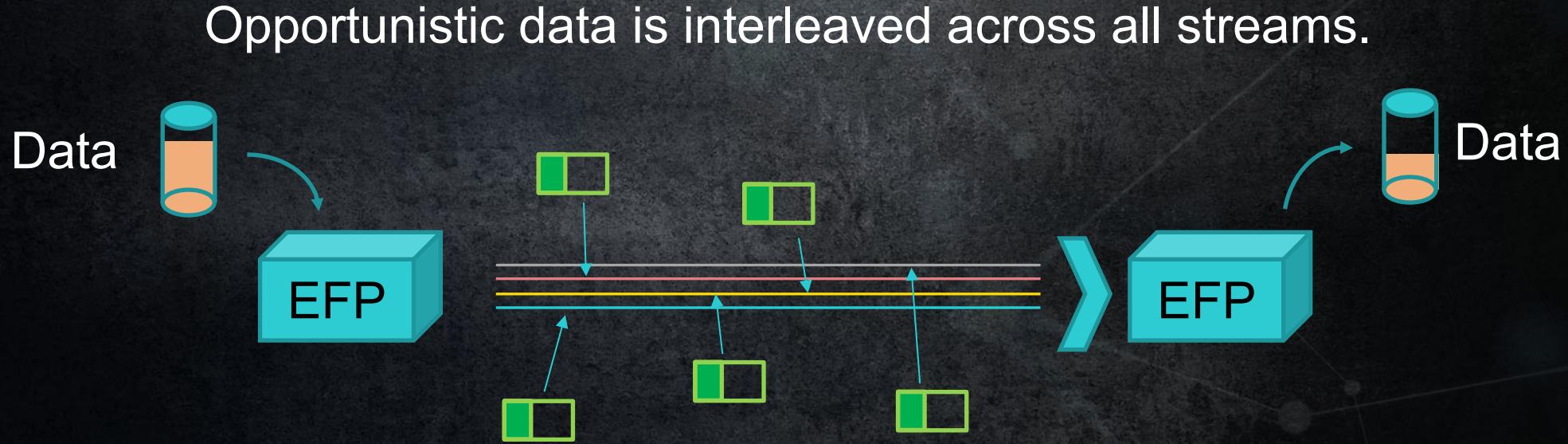
- Any fragment in the stream can also contain a Type 0 fragment. Type 0 may contain private data, the size of MTU-1.
- A type0 can be inserted wherever. Before the first fragment or after the last or anywhere in between. Any number of type0 in a row can also be inserted.
- There is no sense of data integrity or counters for type0 frames. That has to be provided by a higher level protocol.

# ELASTICFRAMEPROTOCOL OPPORTUNISTIC DATA TRANSPORT



- Opportunistic data is using the possible extra space (to MTU) in type2 and Type 3 frames
- Opportunistic data provides no guarantees at all when it comes to bps

# ELASTICFRAMEPROTOCOL OPPORTUNISTIC DATA TRANSPORT



Opportunistic data is interleaved across stream to better utilize the tail MTU mismatch of all possible fragments.

The benefit of doing that is better utilization and higher transfer rates.

CHECK IT OUT AT :

GIT CLONE GIT@BITBUCKET.ORG:UNITXTRA/EFP.GIT

# WHATS NEXT

- Tests. Tests. Tests. And more Tests.
  - Need to reach higher coverage both use cases and code
- API documentation
- Optimize
  - The scan trough bucket code can be optimized by looking at buckets +- last delivered bucket for example. Invalidating all buckets outside the scope using a lower frequency full scan.
- The callback is blocking the worker thread. Change to a non-blocking method.
  - Signal data ready. Use a queue with it's own lock and lock only around the queue access.

THANKS FOR ATTENDING

