Introduction to AMQP 1.0

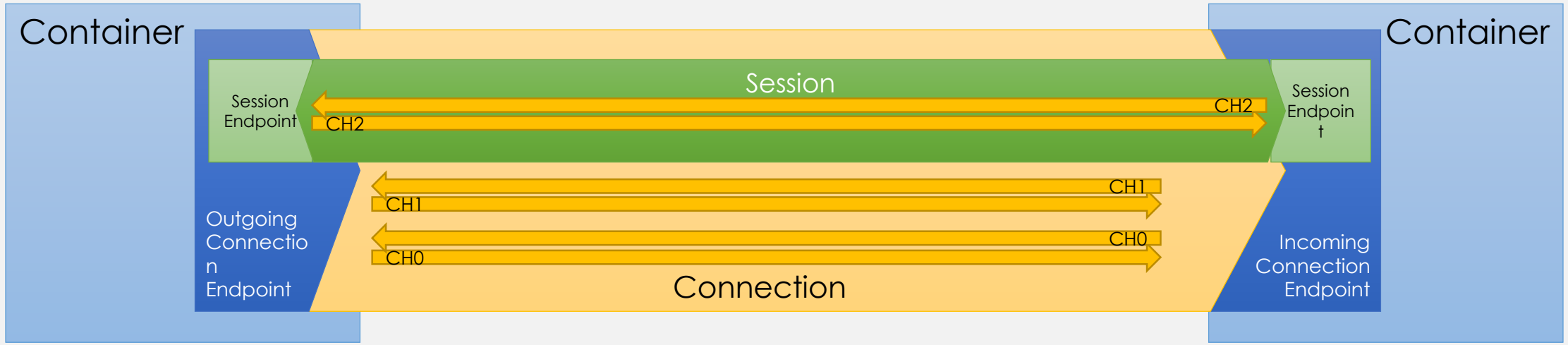
AMQP Foundations

Clemens Vasters
Architect, Messaging, Microsoft Corporation
clemensv@microsoft.com

What is AMQP 1.0?

- Secure, compact, symmetric, multiplexed, reliable, binary transfer protocol to move messages between applications
- International ISO/IEC 19464:2014 standard
- Layered model
  - Transport and connection security protocol
  - Frame transfer protocol
  - Message transfer protocol
- Not tied to any particular message source or target model or topology. Supports classic message brokers, modern hyper-scale messaging infrastructure, and peer-to-peer exchange

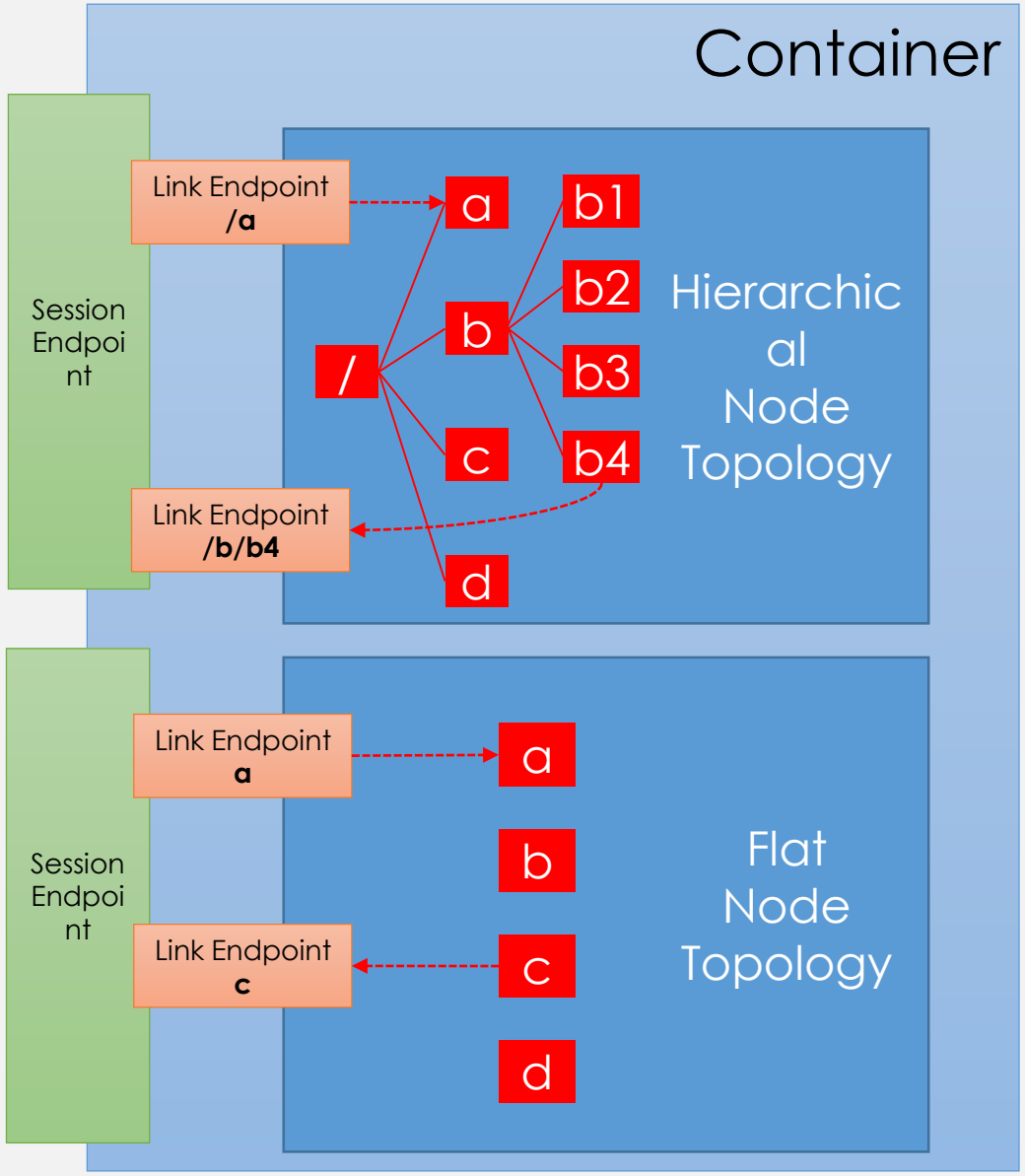
Frame Transfer Protocol Overview



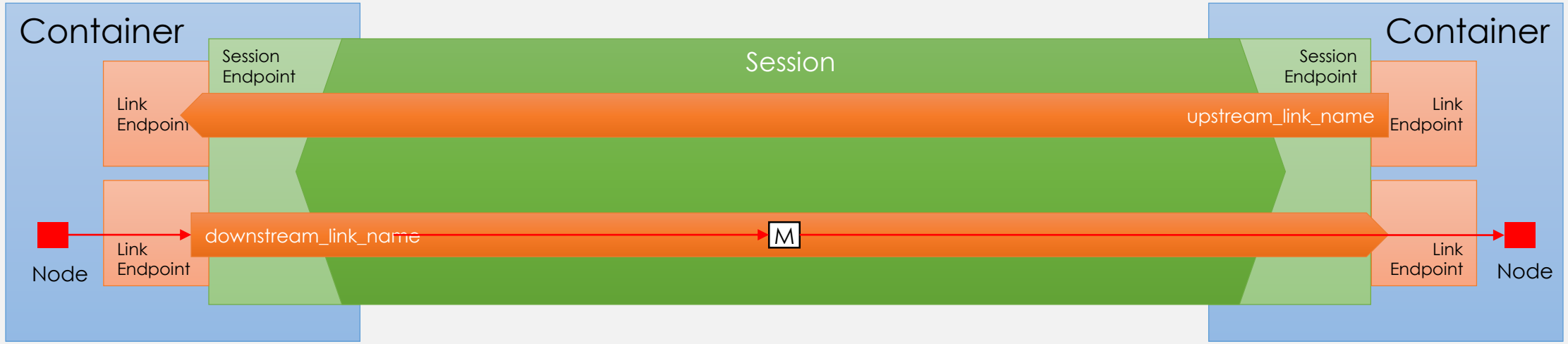
- Containers (Apps) connect to other containers
- Connections manage transfer capacity (frame size, channel count)
- Bidirectional sessions formed over pairs of unidirectional channels
- Sessions allow for multiplexed conversations between

Containers and Nodes

- Container
  - Communicating app
- Node
  - Addressable entity within the app
  - Nodes can be organized in any way the application sees fit; flat, hierarchy, or graph
  - Node can be a queue or topic or relay or event store or …
- Link
  - Links connect via paths to nodes



Message Transfer Protocol Overview



- Unidirectional, named links are formed over sessions
- Links manage flow individually through link credits
- Links can outlive collapsed connection/sessions and can be recovered

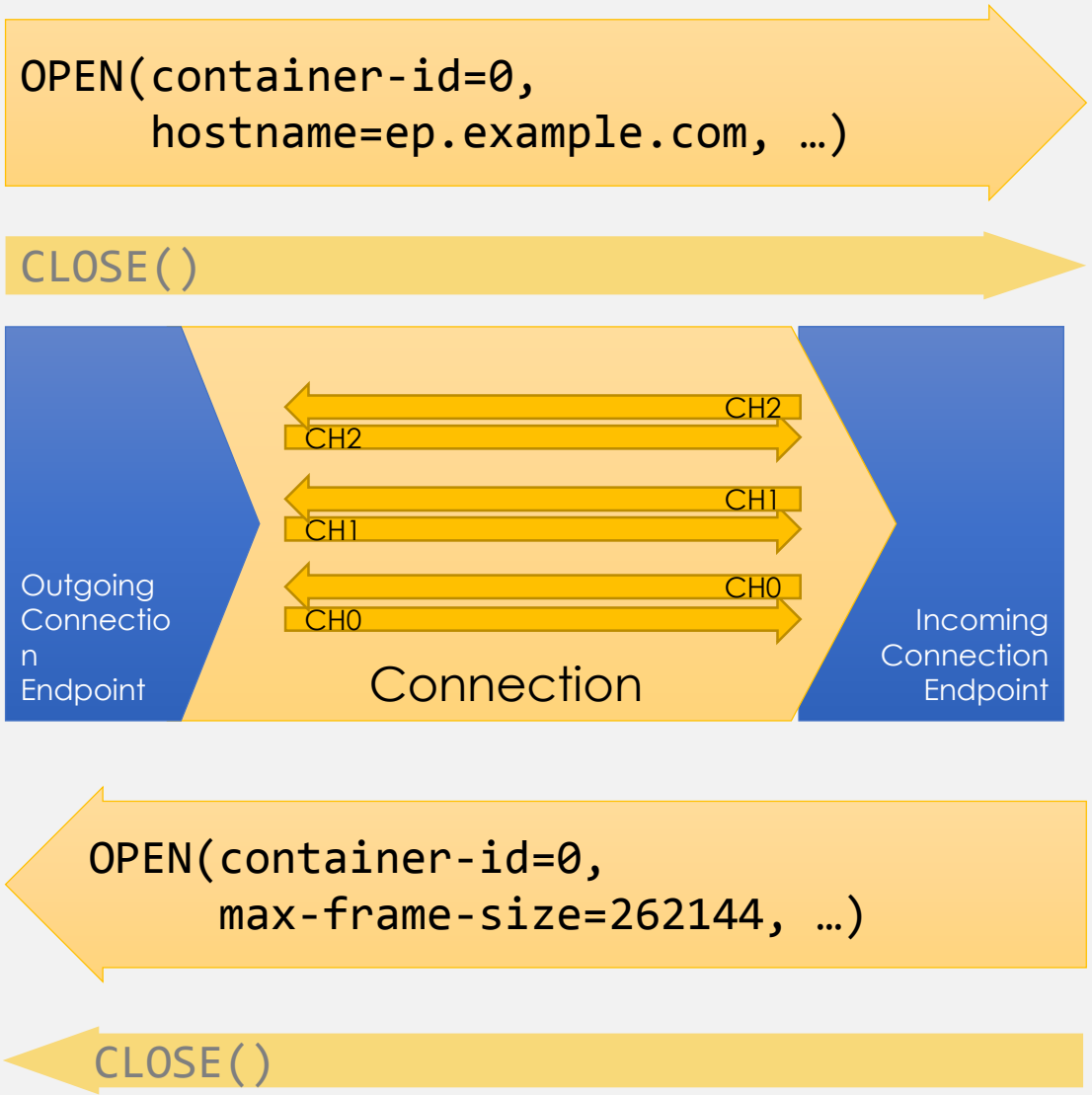
AMQP
Advanced Message Queuing Protocol

AMQP 1.0 Core Protocol Elements

Connections, Security, Sessions, and Links

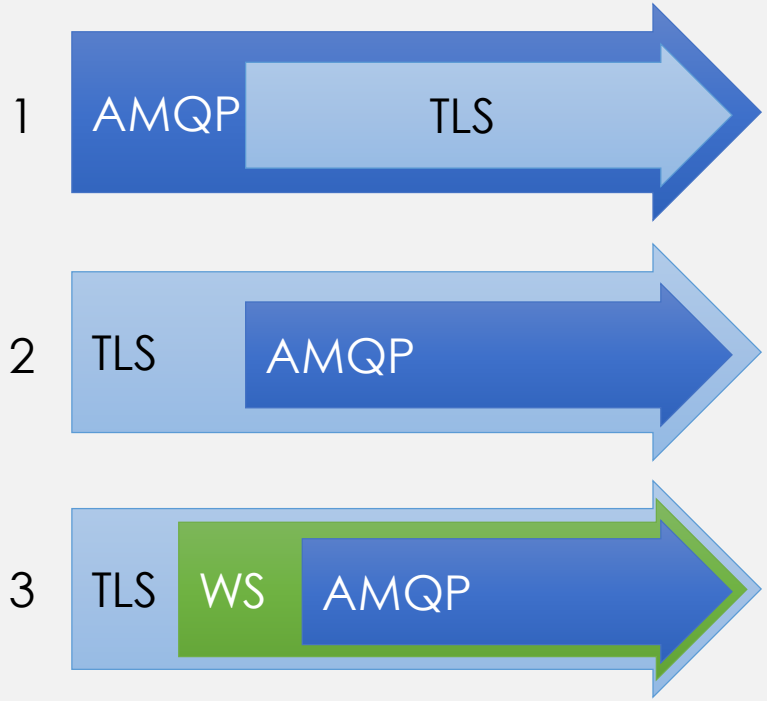
Connection

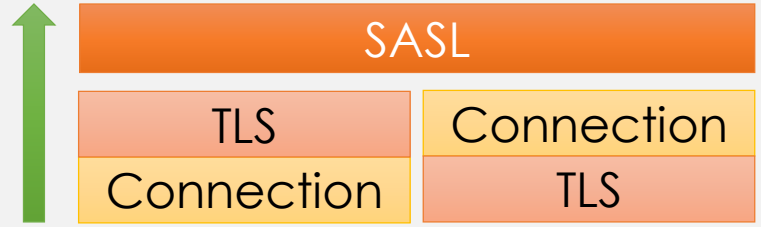
- Layered over network stream-transport (usually TCP; SCTP, Pipes are suitable)
- Connection provides a reliably ordered sequence of frames w/ negotiated maximum frame size
- Frames flow over a negotiated number of multiplexed, unidirectional channels
- Explicit idle-timeout management
- Connection preamble indicates protocol version. Integration with transport-level security; TLS and SASL security can be negotiated inline.



Connection-Level Security Layers

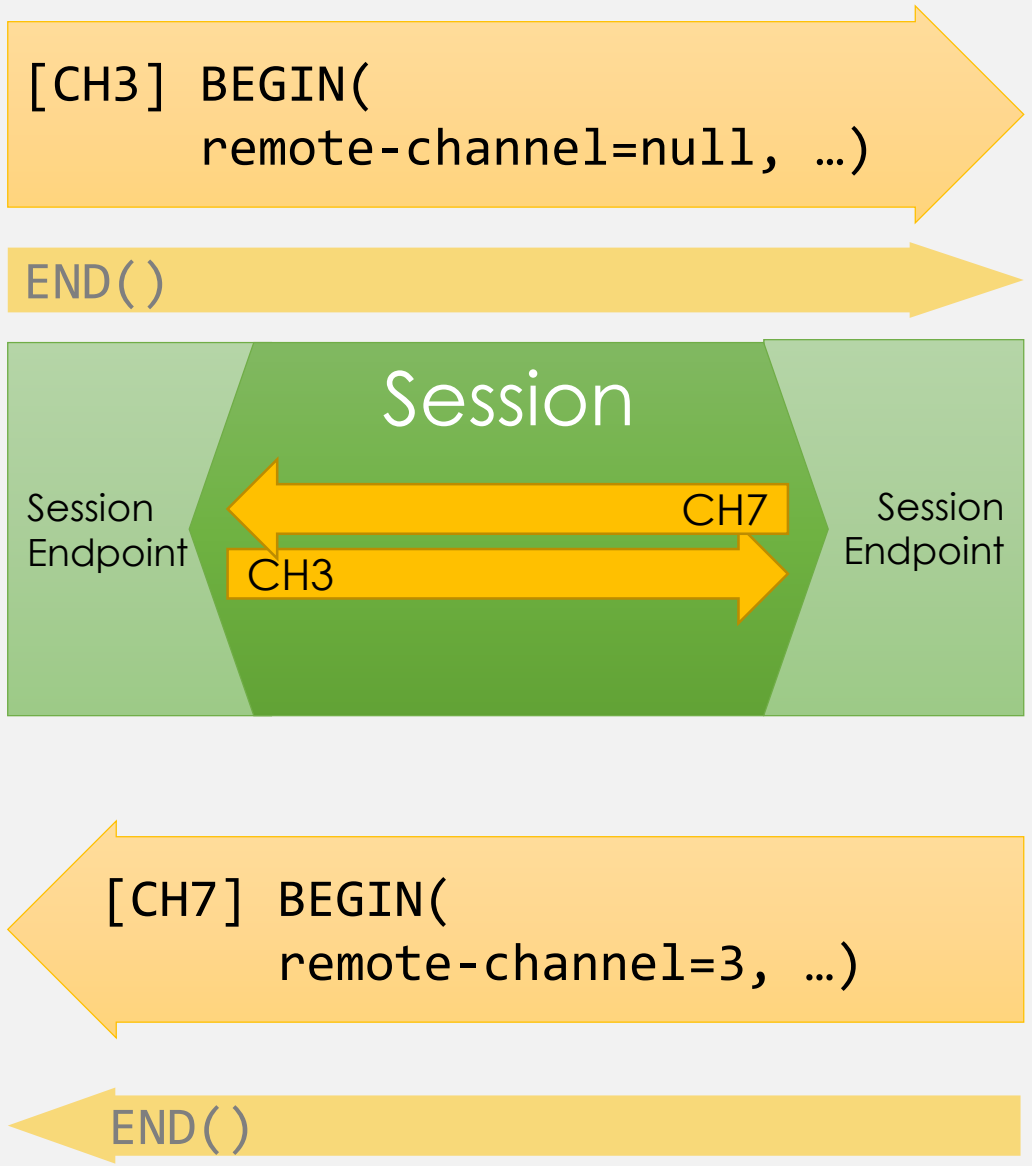
- Transport-Level Security (TLS/"SSL")
  1. Single-port model for AMQP and secure AMQP
     - Sender indicates desire for TLS session with a preamble (protocol id 2) first, then AMQP version negotiation occurs. TLS negotiation occurs inline
  2. "Pure TLS" model for AMQPS on dedicated port 5671
     - Sender opens TLS session to server on port 5671 without upgrade
  3. WebSockets tunnel on port 443
     - Sender creates WebSocket with TLS and overlays AMQP [AMQP 1.0 over WS spec draft]
- SASL Authentication
  - Optional. Negotiated with a preamble (protocol id 3) after initial TCP version negotiation and before AMQP version negotiation
  - Supports any SASL authentication model
  - SASL External permits binding TLS client authentication context to the AMQP connection





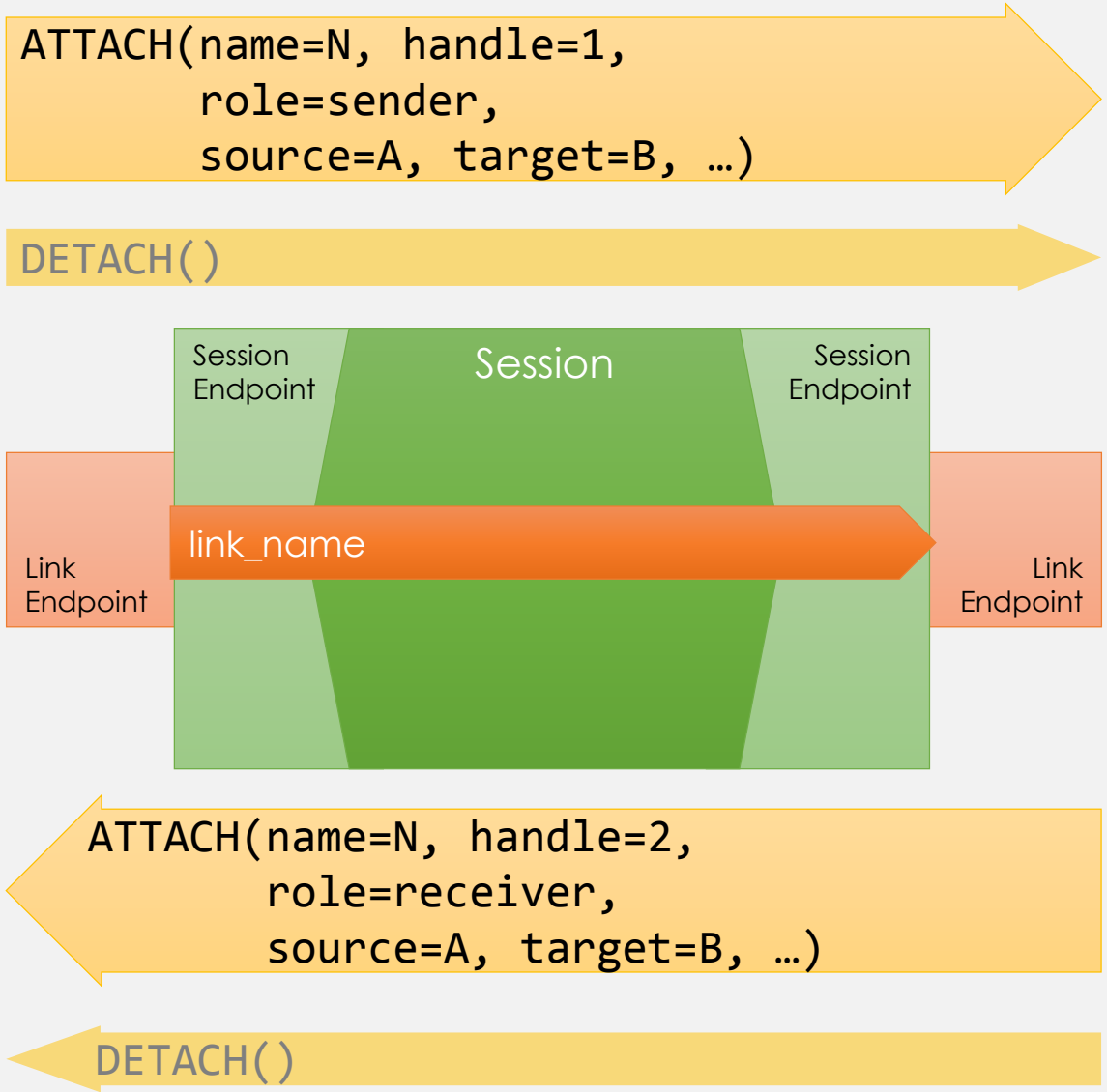
Session

- Binds two unidirectional channels to form a bidirectional, sequential conversation
- Provides a window-based flow control model
  - Number of total transfer frames that sender and receiver can hold in buffers at any given time
  - Decouples credit-based flow control driven by API and capacity management requirements of the underlying platform
- A connection can support



Link

- Named, unidirectional transfer route for messages from “source” to “target” node
- A session can accommodate any number of concurrent links
- Links in either direction can be initiated by either peer.
- Links can be recovered on a different connection/session when the previous broke
- Credit-based flow control to model flow management for various API shapes

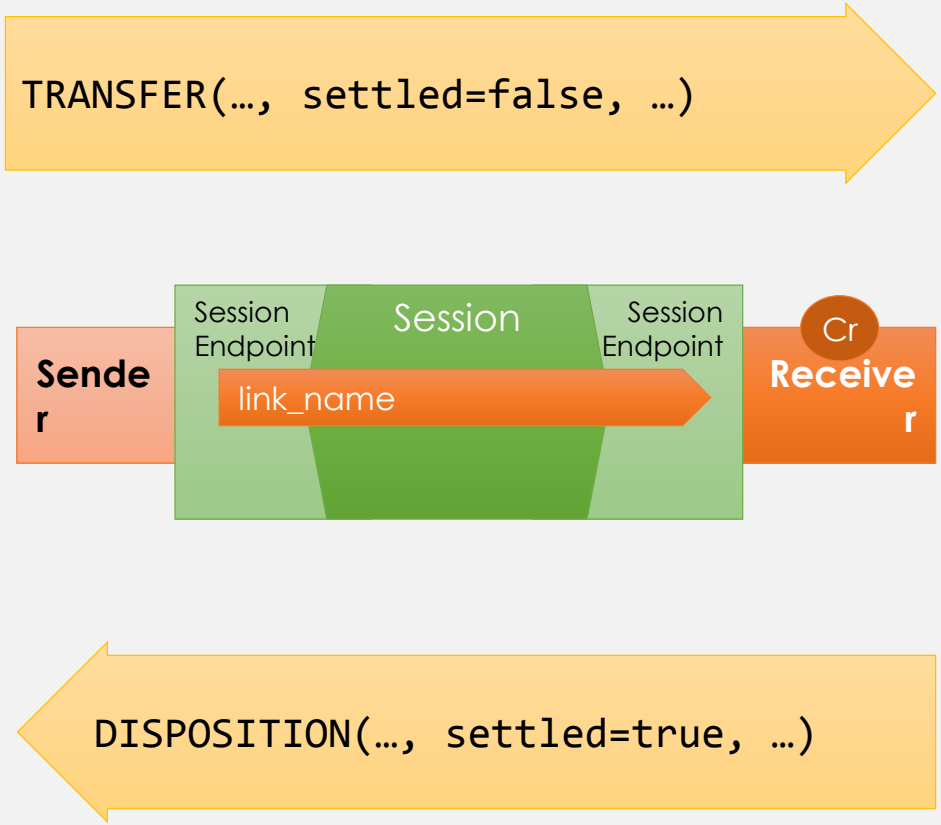


AMQP
Advanced Message Queuing Protocol

Message Transfers

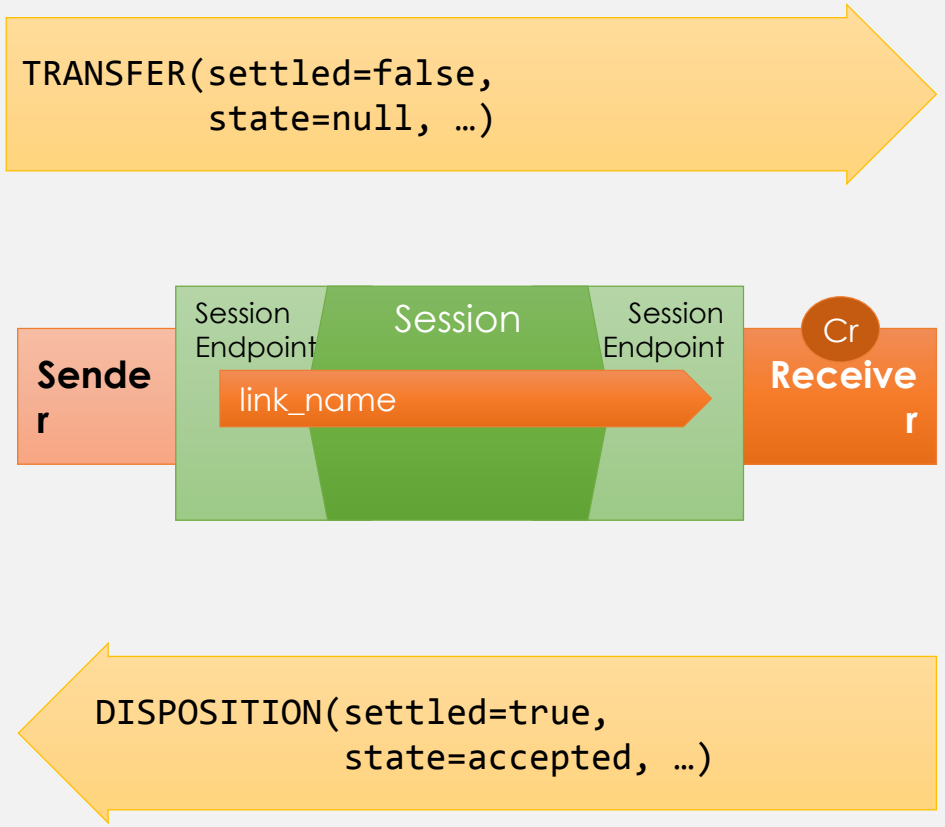
Message Transfers – Settlement

- Messages are transferred fire-and-forget ("settled") or requiring explicit settlement, allowing for various delivery state handling and delivery assurance models
- At-Most-Once / Fire-And-Forget
  - `TRANSFER(settled=true)`
- At-Least-Once
  - → `TRANSFER(delivery_tag=DT, **settled=false,** state=S_0)`
  - ← `DISPOSITION(delivery_tag=DT, **settled=true,** state=T_0)`



Message Transfers – Delivery State

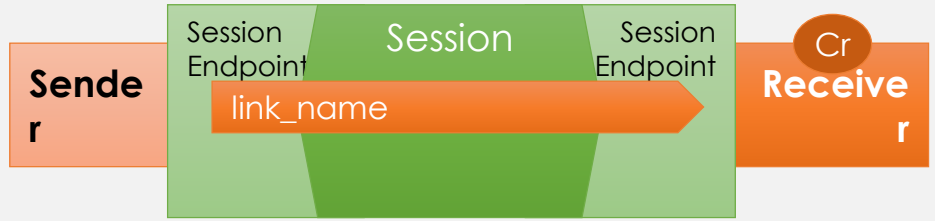
- Intermediary States (transfer ongoing)
  - `received` – used for link recovery negotiation, indicating that an unsettled message had been received
- Terminal States (transfer is done)
  - `accepted` – the message has been accepted by the receiver (OK)
  - `rejected` – the message has been rejected by the receiver *(with error/diagnostics details)*
  - `released` – the message has been abandoned by the receiver and should be redelivered
  - `modified` – the message has been released (see above) and should be modified at the source as indicated in the details



Message Transfer – Recovering Links

- Links can be recovered from previous, broken or abandoned connections/sessions
- Message delivery states are exchanged as the link is re-attached by the peers
- Transfers are subsequently recovered to resend, reconcile state, or continue settlement
  - `TRANSFER(…,resume=true, …)`

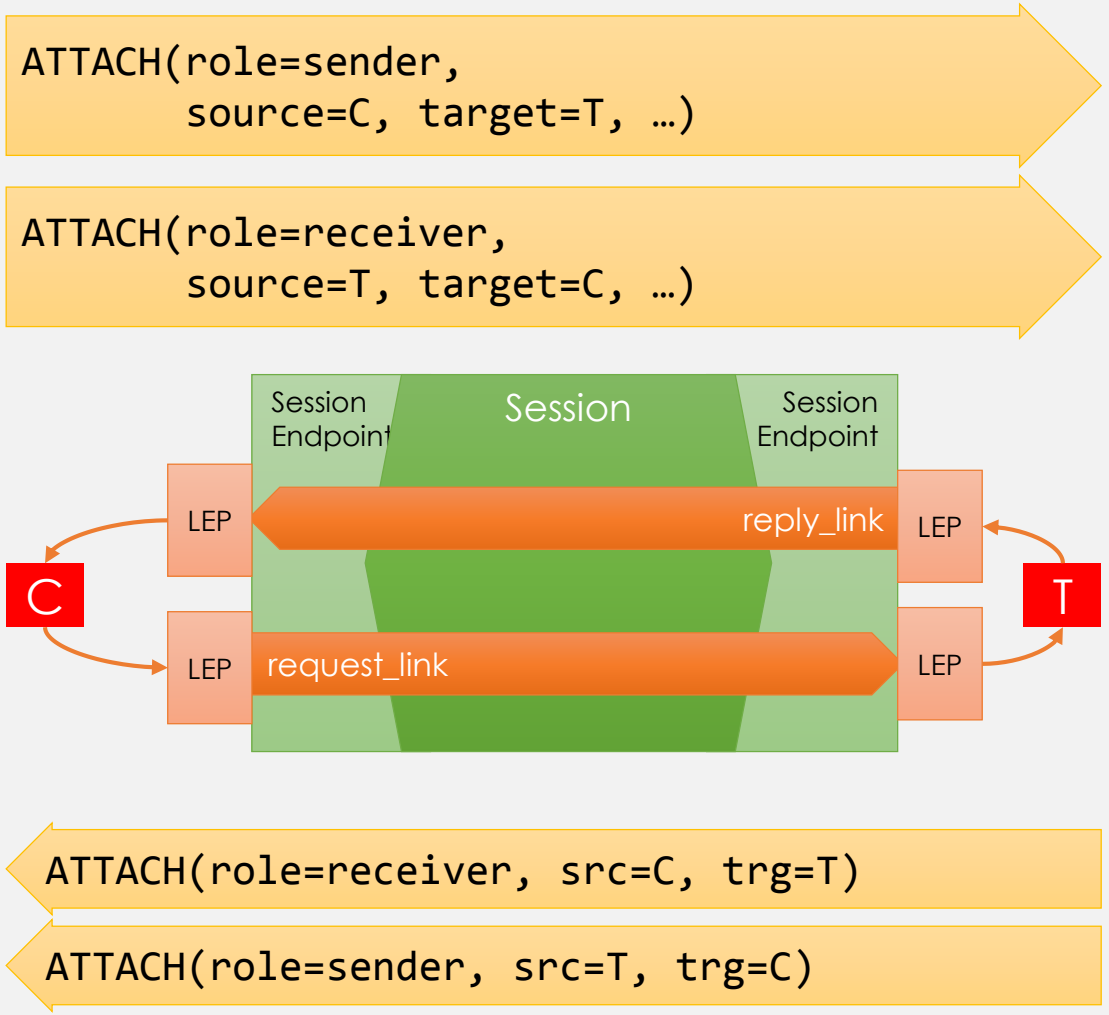
```
ATTACH(role=sender, src=A, trg=B,
       unsettled =
       {  1-> null,
          2-> null,
          3-> accepted,
          4-> null }, …)
```




```
ATTACH(role=sender, src=A, trg=B,
       unsettled =
       {  1-> released,
          2-> null,
          3-> accepted }, …)
```

Modeling Bi-Di and Request/Response

- For a request/response relationship, the client C initiates two links with the target node T
  - [C→T] role=sender
  - [T→C] role=receiver
- The client's response node can be scoped to the conversation or global. Referenced in *request.reply-to*
- Correlation via *reply.correlation-id* set to *request.correlation-id* (or *request.message-id* if absent)



AMQP
Advanced Message Queuing Protocol

Transfers – Flow Control

AMQP Body Frame Types

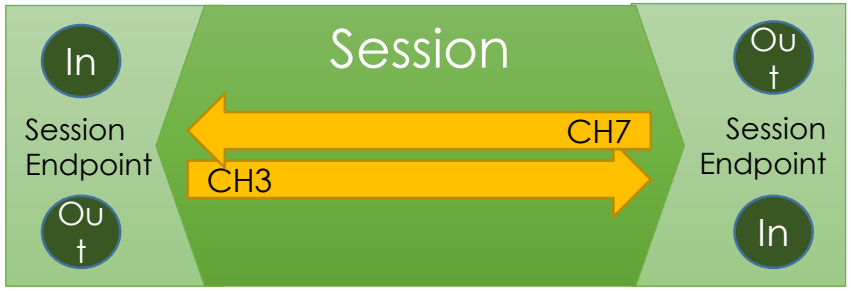
Type		Connection	Session	Link	Description
OPEN	✓	**Handle**			Connection open
BEGIN	✓	Inspect	**Handle**		Session begin
ATTACH	✓		Inspect	**Handle**	Link attach
FLOW			Inspect	**Handle**	Update flow control state
TRANSFER			Inspect	**Handle**	Transfer message
DISPOSITION			Inspect	**Handle**	Update transfer state
DETACH	✓		Inspect	**Handle**	Link detach
END	✓	Inspect	**Handle**		Session end
CLOSE	✓	**Handle**			Connection close

Already discussed

Session Flow Control

Platform Resource Management

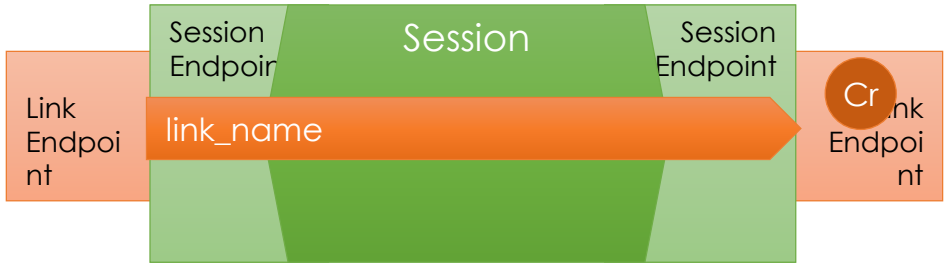
- Each session endpoint has an incoming and outgoing transfer window defining an upper bound for messages in transit
- Window size is expressed in frame count; maximum frame size and window size gate the memory capacity required for a session
- Transfers decrement the remaining window sizes. An outgoing window size of <=0 suspends transfers. Window sizes are updated/reset with FLOW



Link Flow Control

Application-Level Message Flow Management

- Each link has an associated link-credit, which is the number of messages the receiver is currently willing to accept
- Messages can flow when the remaining link-credit is greater than zero. Only the receiver can manipulate the link credit value.
- Transfers decrement the remaining link credit. A link credit of <=0 suspends transfers. Link credit is updated with FLOW

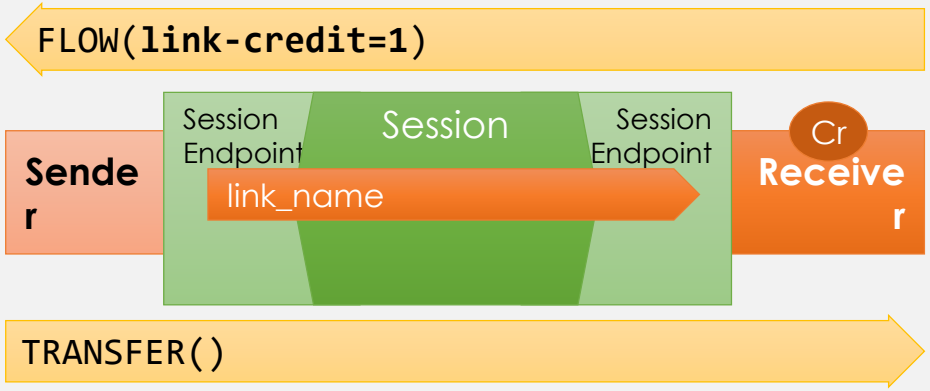


Wait! Why two flow control models?

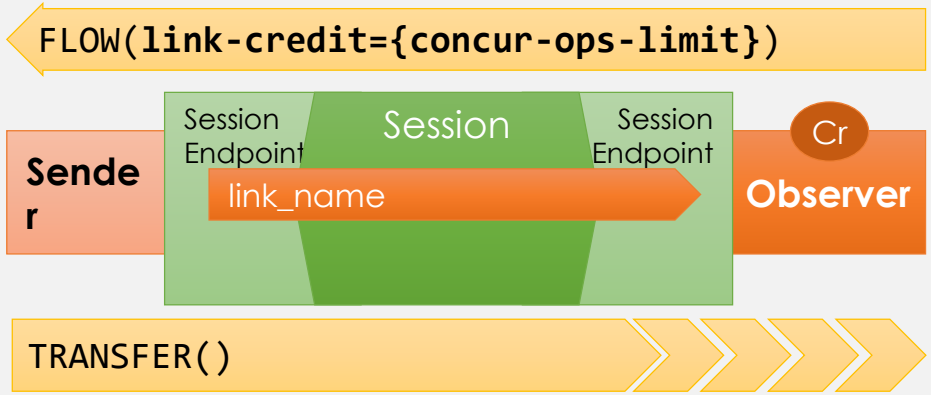
- Session flow control protects the platform
  - Guards high-scale and hyper-scale platform implementations from overcommitting transfer resources to sessions; shields from unexpected transfer bursts
  - Allows for session-level capacity management and throughput throttling
- Link flow control protects the application and supports API gestures
  - Guards an application node from having to accept more messages than it can currently handle.
  - Prevents concurrent, multiplexed session/link traffic from being backed up behind messages that cannot currently be handled
  - Enables on-demand receive, pre-fetch receive, and just-in-time receive ("push") operations

Some API Patterns

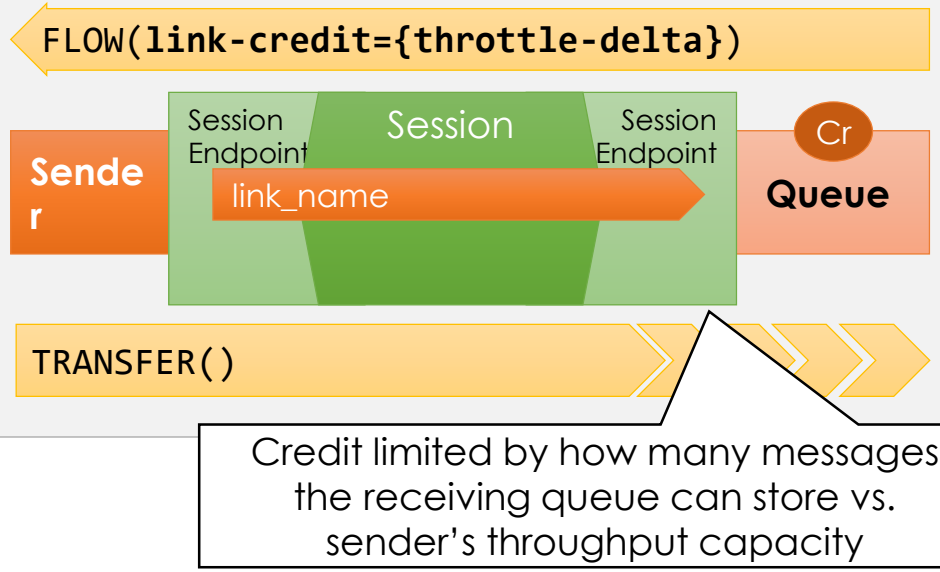
Imperative: `receiver.Receive()`



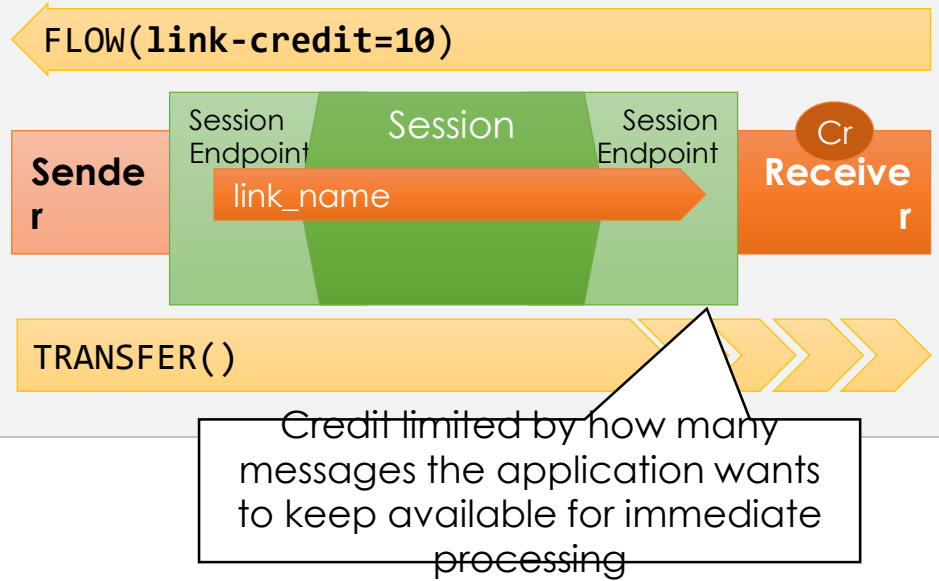
Reactive: `observer.Subscribe(callback)`



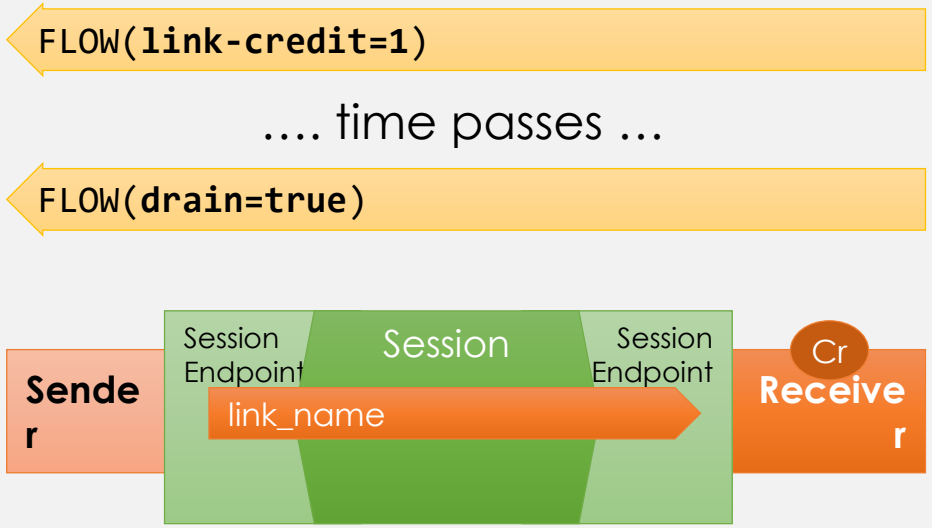
Throttled: `queueClient.Send()`



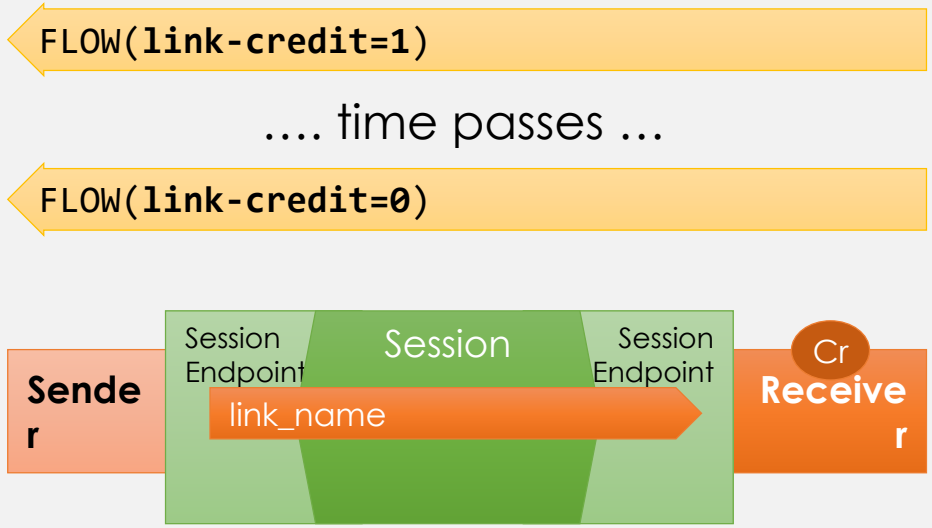
Prefetch: `receiver.SetPrefetch(10)`



Application Timeout/Cancellation/Pause



"drain" allows any immediately pending transfers to complete and then promptly reduces the link credit to zero.



Setting the link-credit to zero immediately stops the link. Pending messages are no longer transferred. Resume with increasing credit.

AMQP
Advanced Message Queuing Protocol

AMQP on the Wire

Primitive Types

The AMQP Type System

- AMQP has its own, portable, language-independent type system
  - Used for all core AMQP protocol elements
  - Optimized for fast and flexible processing and compact representation
  - Defines a broad range of primitive and composite types
  - Permits descriptive references to external type systems
  - Can be used to express and encode message payloads
- Supports schema-bound and schema-free encoding
  - Schema-bound (like Avro/ProtoBuf/Thrift)
    - All fixed protocol elements
    - Only transmit data and rely on out-of-band-shared metadata for interpretation.
  - Schema-free (like JSON/MsgPack)

Schema Example – Composite Type

- AMQP Schema notation is XML
  - DTD included in the core spec
- Type classes
  - **primitive** – built-in types
    - Includes array, list, and map
  - **composite** – multi-value type
    - Inherits from "list" primitive
  - **restricted** – restriction of existing type
    - Restricts source type value space
- Archetypes
  - **provides** – categorization of types
- Descriptor
  - Unique string and numerical descriptors
- Composite: Fields
  - Zero or more fields

properties – composite type schema

```
<type name="properties" class="composite"
      source="list" provides="section">
   <descriptor name="amqp:properties:list"
               code="0x00000000:0x00000073"/>
    <field name="message-id" type="*" requires="message-id"/>
    <field name="user-id" type="binary"/>
    <field name="to" type="*" requires="address"/>
    <field name="subject" type="string"/>
    <field name="reply-to" type="*" requires="address"/>
    <field name="correlation-id" type="*" requires="message-id"/>
    <field name="content-type" type="symbol"/>
    <field name="content-encoding" type="symbol"/>
    <field name="absolute-expiry-time" type="timestamp"/>
    <field name="creation-time" type="timestamp"/>
    <field name="group-id" type="string"/>
    <field name="group-sequence" type="sequence-no"/>
    <field name="reply-to-group-id" type="string"/>
</type>
```

Schema Example – Restricted Type

- Restricted types inherit *source* type
  - here: ubyte, unsigned byte
- Type may constrain the value space
  - Explicit permitted values
  - Named choices act as alias
- May directly alias source
- May override source descriptor

sender-settle-mode – restricted type schema

```
<type name="sender-settle-mode" class="restricted" source="ubyte">
   <choice name="unsettled" value="0"/>
   <choice name="settled" value="1"/>
   <choice name="mixed" value="2"/>
</type>
```

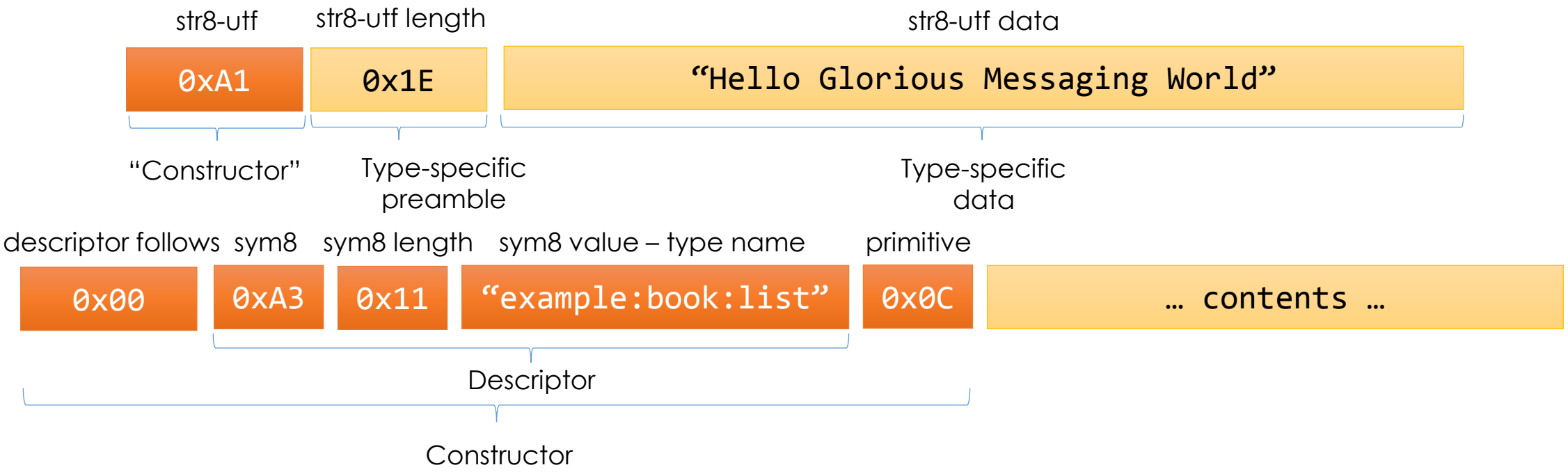
seconds – restricted type schema

```
<type name="seconds" class="restricted" source="uint"/>
```

delivery-annotations – restricted type schema

```
<type name="delivery-annotations" class="restricted“
     source="annotations" provides="section">
    <descriptor name="amqp:delivery-annotations:map“
                code="0x00000000:0x00000071"/>
</type>
```

Data on the wire



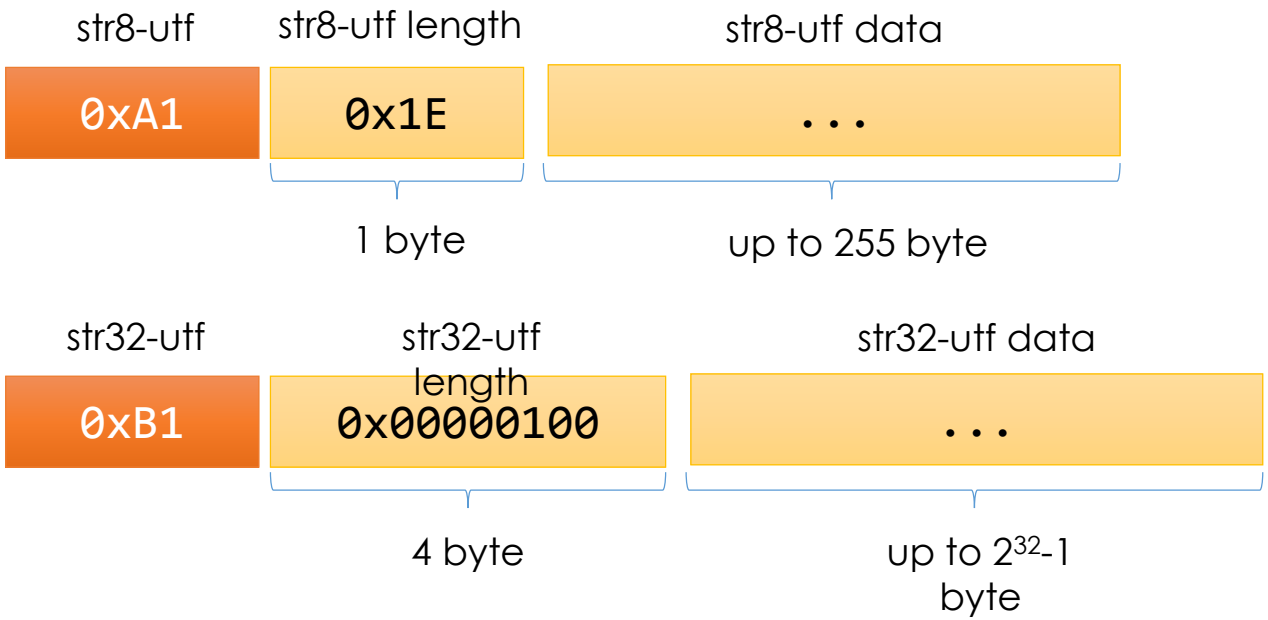
- All data elements are prefixed with a “constructor” indicating type
  - Built-in, primitive types use a single octet (0xA1 designates str8-utf)
  - Composite and restricted types use a symbolic or numeric

Fixed Size Primitives

- Fixed size primitives are organized by encoding size
- *null*, Boolean *true* and *false*, and zero-valued *uint* and *ulong* can be expressed with a constructor (no data byte)
- *uint* and *ulong* valued < 256 encodes into 1 data byte
- *int* and *long* valued from -128 to 127 encodes into 1 data byte

	data bytes occupied by type					
	0	**1**	**2**	**4**	**8**	**16**
null	0x40					
boolean	0x41: true 0x42: false	0x56				
ubyte		0x50				
ushort			0x60			
uint	0x43: v=0	0x52: v<256		0x70		
ulong	0x44: v=0	0x53: v<256			0x80	
byte		0x51				
short			0x61			
int		0x54: -128<v<127				
long		0x55: -128<v<127			0x81	
float				0x72		
double					0x82	
decimal32				0x74		
decimal64					0x84	
decimal12						0x94
char				0x73		
timestamp					0x83	
uuid						0x98

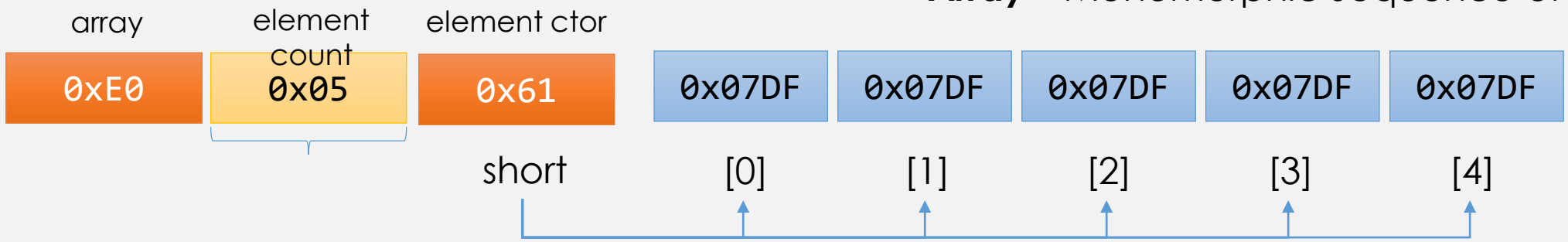
Variable Size Primitives



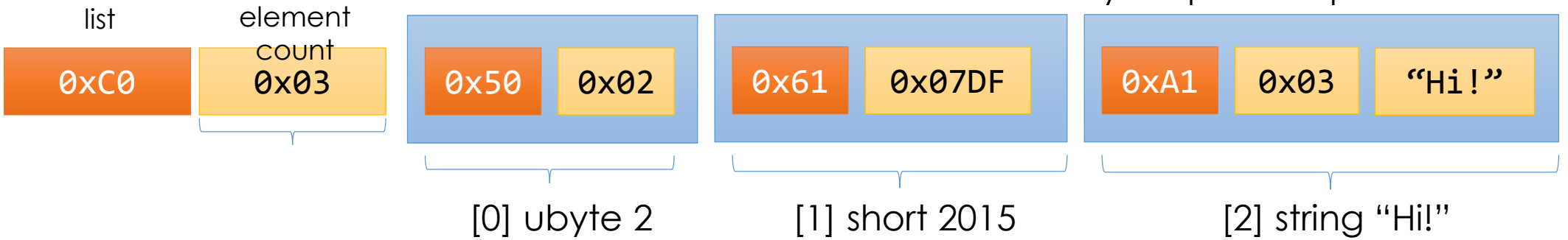
	preamble size	
	1	4
binary – vbin8	0xA0	
binary – vbin32		0xB0
string – str8-utf	0xA1	
string – str32-utf		0xB1
symbol – sym8	0xA3	
symbol – sym32		0xB3

- Variable size primitive encodings have a 1 or 4 byte preamble. Which encoding is used depends on the actual data size.
- UTF-8 strings, symbols, and binary data of up to 255 bytes has a single byte preamble, otherwise a 4 byte preamble is used

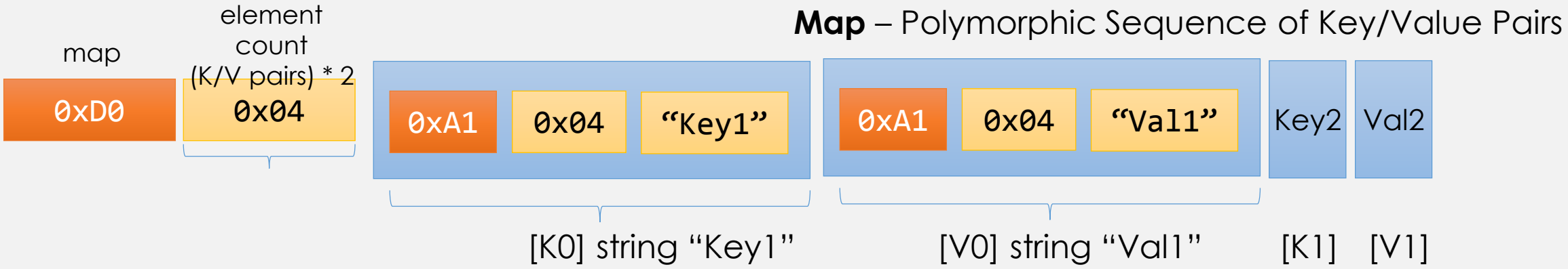
Array – Monomorphic Sequence of Elements



List – Polymorphic Sequence of Elements



Map – Polymorphic Sequence of Key/Value Pairs



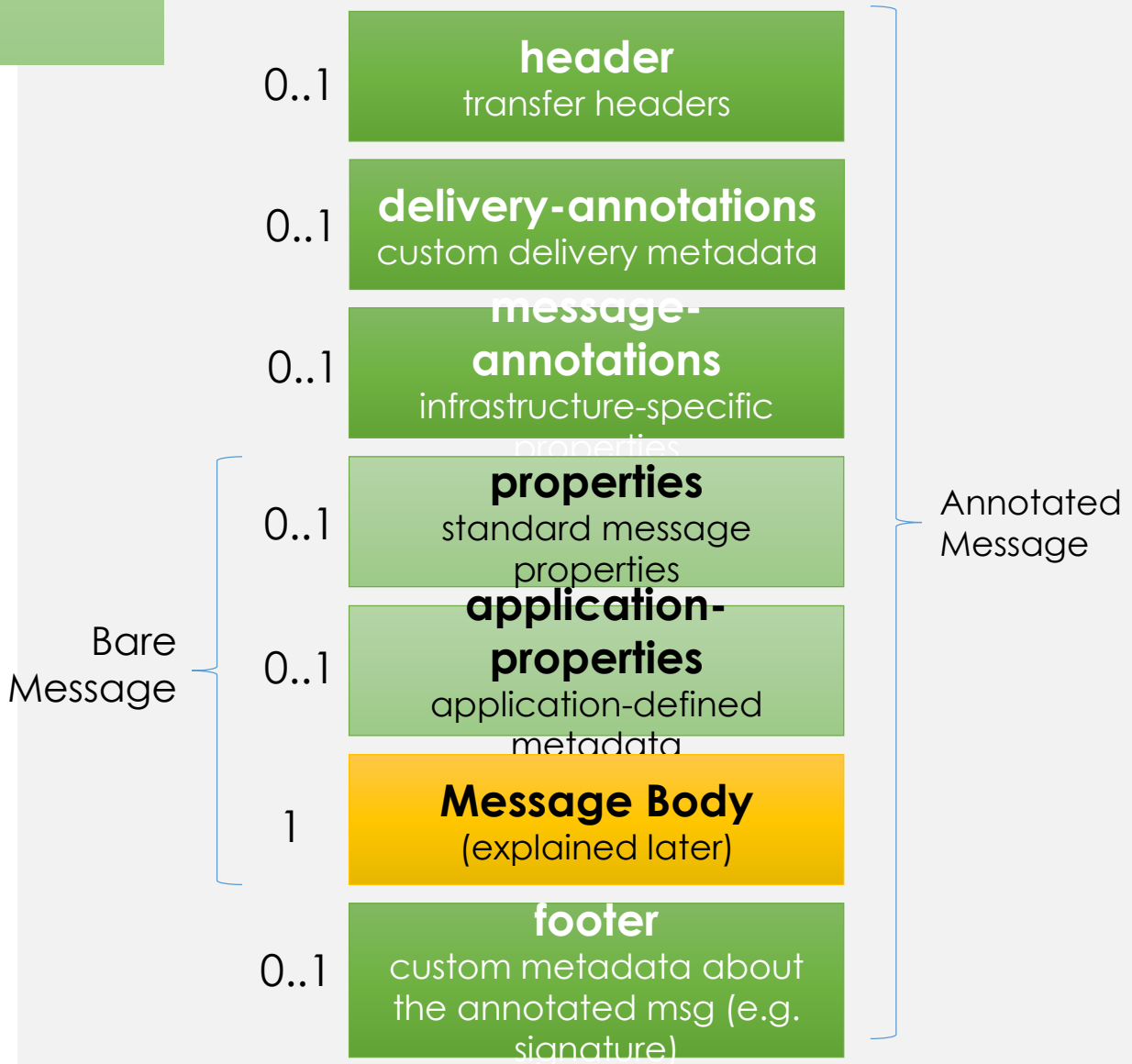
AMQP
Advanced Message Queuing Protocol

AMQP on the Wire

Composite Types and Messages

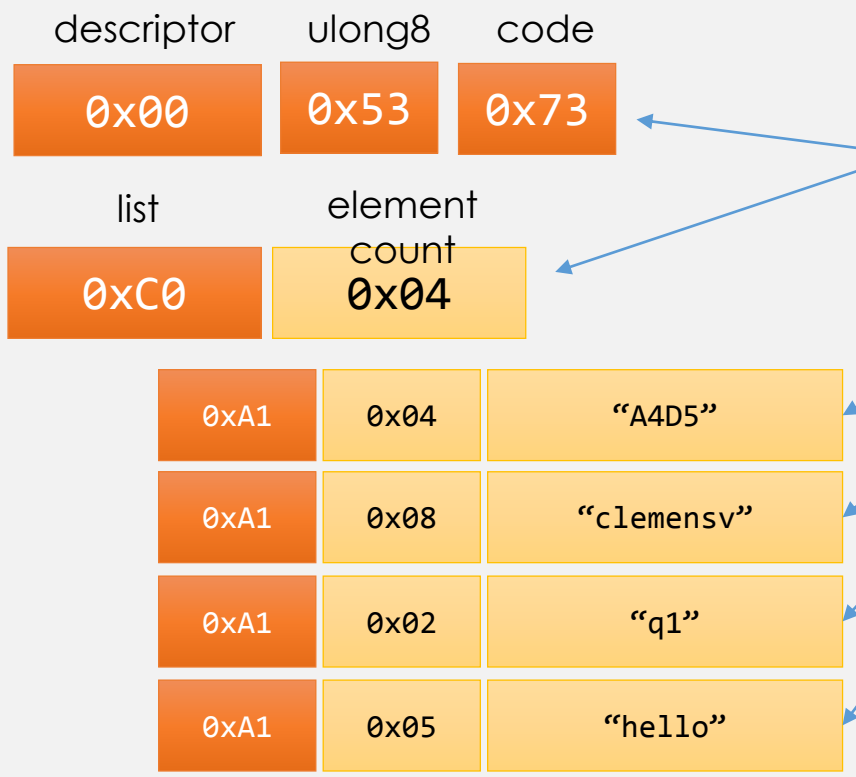
AMQP Messages

- The standard AMQP message format shown here is replaceable/extensible.
  - Message format kind and version is selected on TRANSFER frame. Default format is zero.
- *Bare Message* core is immutable from sender to ultimate receiver through all intermediaries
- *Annotated Message* frame may be altered by intermediaries
- All sections defined as composite types (archetype "section")



Composite Type Encoding

- Prefixed with type descriptor
- Encoded as list
- List contains elements in schema order
- Trailing elements can be omitted



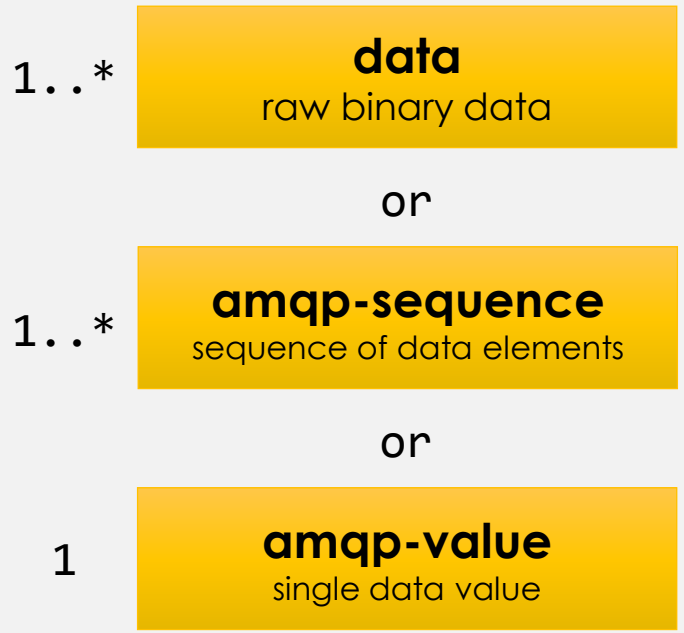
properties – composite type schema

```
<type name="properties" class="composite"
      source="list" provides="section">
   <descriptor name="amqp:properties:list"
               code="0x00000000:0x00000073"/>
    <field name="message-id" type="*" requires="message-id"/>
    <field name="user-id" type="binary"/>
    <field name="to" type="*" requires="address"/>
    <field name="subject" type="string"/>
    <field name="reply-to" type="*" requires="address"/>
    <field name="correlation-id" type="*" requires="message-id"/>
    <field name="content-type" type="symbol"/>
    <field name="content-encoding" type="symbol"/>
    <field name="absolute-expiry-time" type="timestamp"/>
    <field name="creation-time" type="timestamp"/>
    <field name="group-id" type="string"/>
    <field name="group-sequence" type="sequence-no"/>
    <field name="reply-to-group-id" type="string"/>
</type>
```

Application Data – AMQP Message Body

The AMQP message body consists of one of the following three choices:

- One or more **data** sections
  - Binary data up to remaining maximum frame size
  - Hints to interpret this data can be carried in the standard *content-type* and *content-encoding* properties.
  - JSON, MsgPack, Avro, Thrift, XML, etc. payloads
- One or more **amqp-sequence** sections
  - A sequence is a list of AMQP encoded values
- A single **amqp-value** section
  - A single AMQP value (including null)
  - Can be a composite type value



JSON to AMQP - Schemaless

```
{
 “books” : [
  {
    “title” : “Normal Accidents”,
    “authors” : [“Charles Perrow”],
    “isbn” : “978-0-691-00412-9”
  },
  {
    “title” : “Rockets and People, V3”,
    “authors” : [“Boris Chertok”,
                 “Asif Siddiqi”],
    “isbn” : “978-0-160-81733-5”
  }
 ]
}
```

Identical in shape, easy from/to mapping

nearly identical in concrete footprint, but AMQP with deeper type system

JSON to AMQP – Using Schema

```
{
 “books” : [
  {
    “title” : “Normal Accidents”,
    “authors” : [“Charles Perrow”],
    “isbn” : “978-0-691-00412-9”
  },
  {
    “title” : “Rockets and People, V3”,
    “authors” : [“Boris Chertok”,
                 “Asif Siddiqi”],
    “isbn” : “978-0-160-81733-5”
  }
 ]
}
```

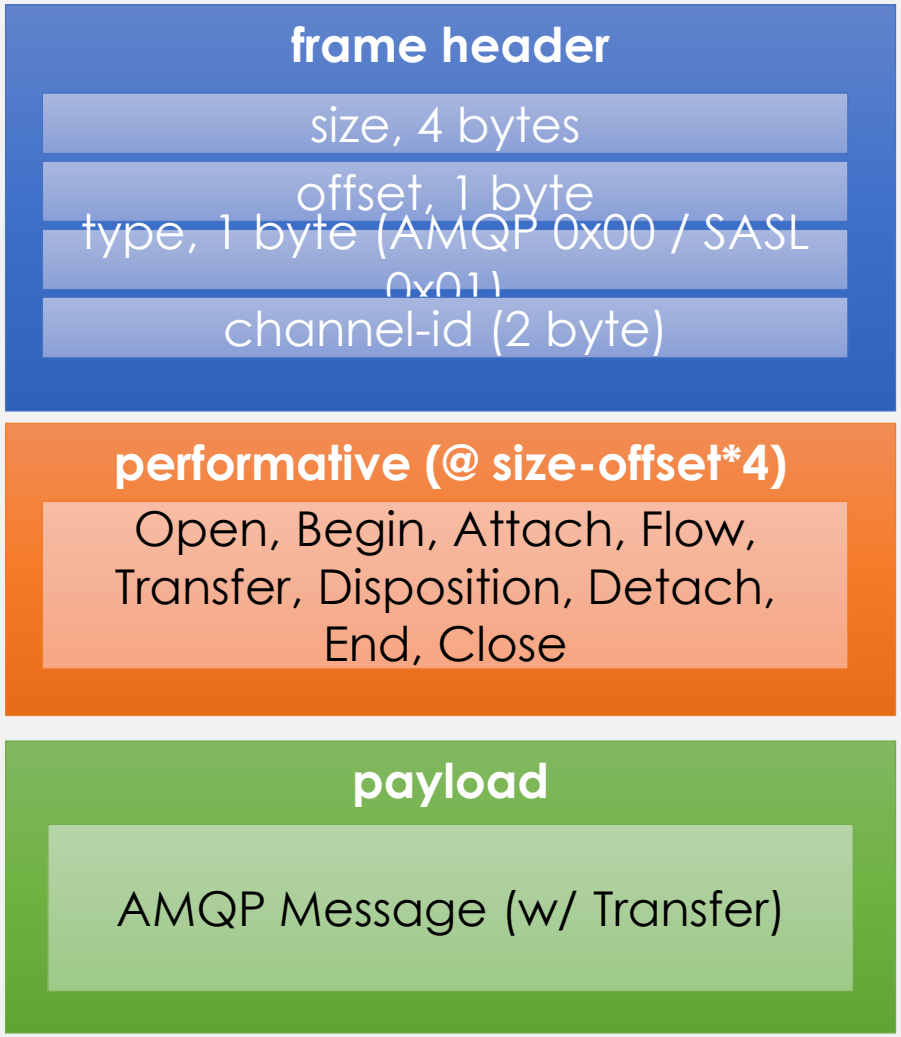
```
<type class="composite" name="book" label="example composite type">
  <descriptor name="example:book:list" code="0x00000003:0x00000002"/>
  <field name="title" type="string" mandatory="true" />
  <field name="authors" type="string" multiple="true"/>
  <field name="isbn" type="string"/>
</type>
```

```
map 2
str8-utf 5 “books” list 2
   desc ulong 0x0000000300000002 list
     strf8-utf 16 “Normal Accidents”,
     array 1 str8-utf 14 “Charles Perrow”
     strf8-utf 16 “978-0-691-00412-9”
    desc ulong 0x0000000300000002 list
     strf8-utf “Rockets and People, V3”
     array 2 str8-utf
          13 “Boris Chertok”,
          12 “Asif Siddiqi”
     str8-utf 16 “978-0-160-81733-5”
```

AMQP more compact as descriptive metadata is externalized into schema

AMQP Frames

- Frame header (8 byte)
  - Overall frame size (ulong)
  - Data offset (count of 4-byte words, byte)
  - Frame type indicator (byte)
  - Channel identifier (ushort)
- Performative
  - Composite type describing the AMQP operation
  - Open, Begin, Attach, Flow, Transfer, Disposition, Detach, End, Close
- Payload
  - Performative-dependent payload, immediately following the performative.
  - Currently only defined to be used for Transfer'



Wire Footprint?

- Minimal AMQP transfer footprint for a message containing the string “Hello!”
- Just 35 bytes transfer overhead.
  - plus IP, TCP, TLS (~100 byte)

```
Frame Header                          ..... 8 bytes
Transfer Performative       5b
     link-handle     (1b) 1+4b
     delivery-id     (1b) 1+4b
     delivery-tag (2+1b) 2+4b
     message-format         1b
   (theoretical min 11 bytes)     .... 22 bytes
AMQP Message
     amqp-value             3b
         “Hello!”         2+6b     .... 11 bytes
                                   =============
                                        41 bytes
```

AMQP
Advanced Message Queuing Protocol

… there's more

Transactions
Addressing
Management
Claims Based Security

… next time …