Erlang Solutions Ltd.

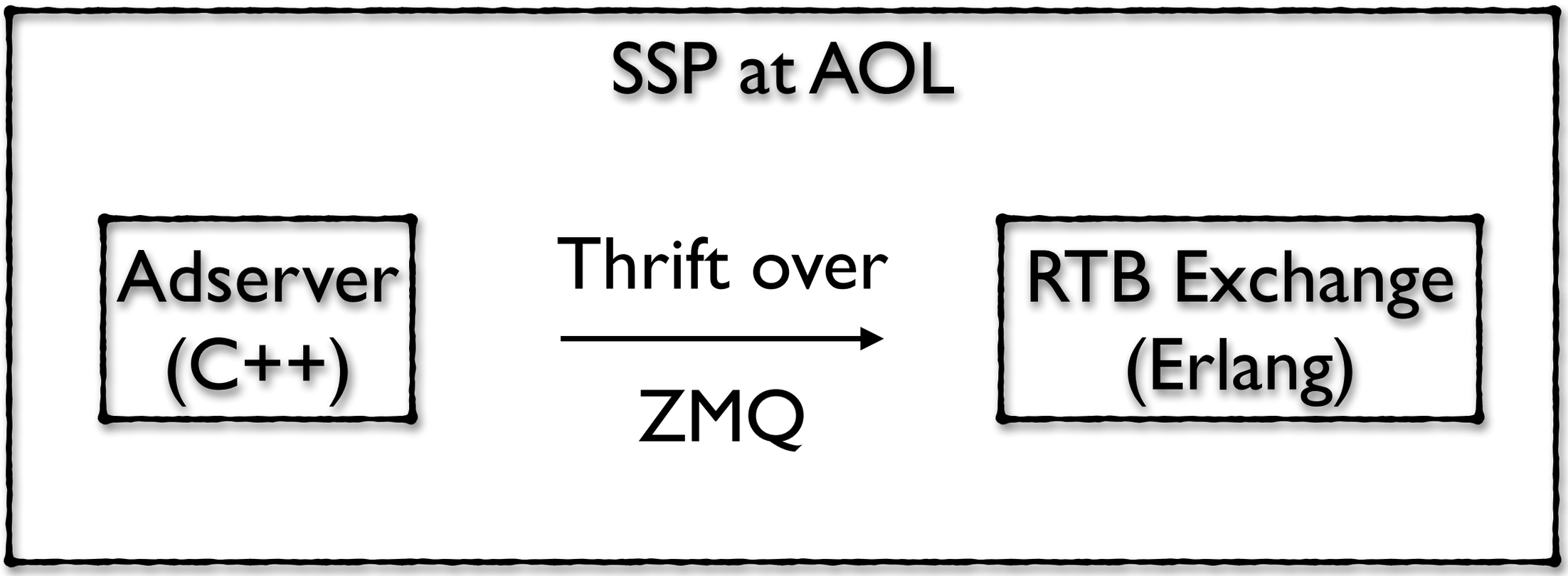
The fun part of writing a Thrift codec

Agenda

- The context
- What is Thrift
- Binary handling in Erlang
- The new library

The context

- Thrift is mostly used for serialisation between CPP and Erlang.
- Adserver considered as blackbox -> change to other serialisation technique (e.g. protobufs) is not an option



What is Thrift

What is Thrift

- Apache Thrift is a software framework for scalable cross-language services development.
- IDL: define data types and service interfaces in a simple definition file
- Compiler: from IDL files generates code to be used to easily build RPC clients and servers that communicate seamlessly across programming languages.

Thrift - Types

Base Types

- bool: a boolean value (true or false), one byte
- byte: a signed byte
- i16: a 16-bit signed integer
- i32: a 32-bit signed integer
- i64: a 64-bit signed integer
- double: a 64-bit floating point number
- string: encoding agnostic text or binary string

Container Types

not heterogeneous - all items in a container must be of the same type

- list<t1>: ordered
- set<t1>: unique values
- map<t1, t2>: unique key

Structs (and Exceptions)

field: numeric unique id, type, name, optional default value

Thrift - Services

- service (RPC interface): collection of methods
- method: has a return type, arguments and optionally a list of exceptions that it may throw.
  (Note that argument lists and exception list are specified using the exact same syntax as field lists in structs. Actually the argument list is passed as an anonymous struct on the wire, and the result is also an anon struct with only field 0 holding the result data.)

- oneway method: asynchronous, no response sent back
- returns void: result is an empty struct

Thrift - Network Stack

- Transport (TCP, HTTP, file): stream based API
- Protocol (JSON, XML, binary, compact): data structures
- Processor (compiler generated in many languages, not really in erlang): messages/methods
- Server/Client

- idea: squash together these layers

Why write a new Thrift codec

The old

- too flexible protocols/ transports: layers, callbacks, encapsulated states
- stream based transport whereas ZMQ has packet based API (buffering hack in our thrift_zmq_transport.erl)
- erlang IDL compiler only generates code that returns data types as simple terms

The new

- no callbacks
- direct binary access
- simpler decoder/encoder API
- generated code to use all info available from the IDL at compile time – generate the actual encoder/decoder code

Binary handling in Erlang

Binary handling in Erlang

- Bit Syntax
  - introduced in R7B http://www.erlang.org/euc/00/R7B.html
  - talk of the year from EUC 2000 (congrats :))
- Binaries are cheap to append to the end and read from the beginning

Internal binary types

- Containing binary data
  - heap binaries (up to 64 bytes)
  - RefC binaries (ProcBin reference + off-heap binary object)
- References to a part of a binary
  - sub binaries: split_binary or binary matched out of a binary; references a heap or RefC binary (never another SubBin)
  - match context: interim data structure similar to sub binaries but optimised for matching (contains more info)

Constructing binaries

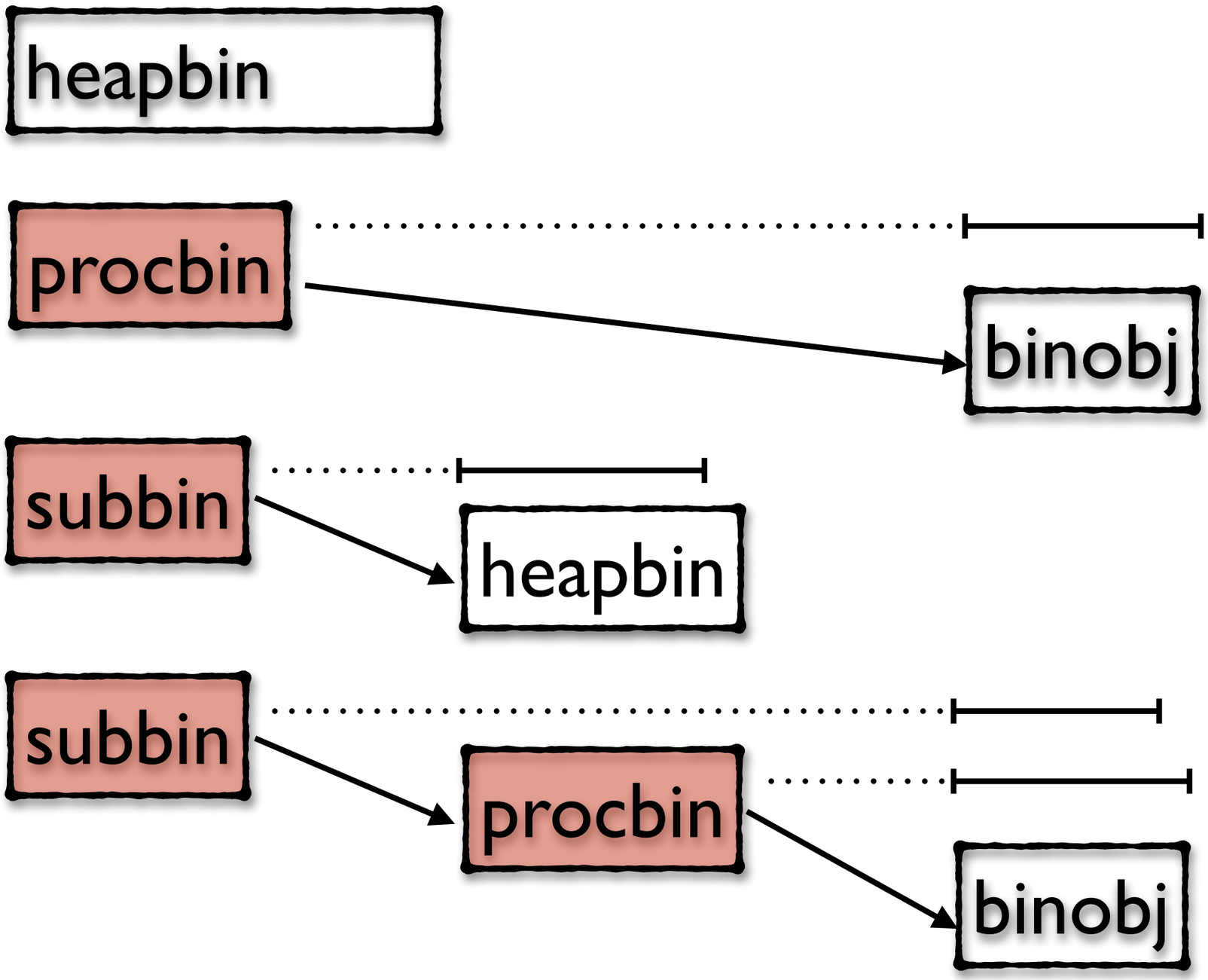
- creating new binary: two steps
  1. allocate empty heap or RefC binary based on the required size (bs_init, bs_init_writable beam ops)
  2. write new data into it bs_put_{integer,float,string,binary}
- appending to existing: two steps
  1. make sure there is space in the end for new data (maybe expand binary and leave some extra space) bs_append, bs_private_append beam ops
  2. write new data to the end bs_put_{integer,float,string,binary}

Binary append example

```
Bin0 = <<1>>, %% 1.

Bin1 = <<Bin0/binary,2,3,4>>, %% 2.




Bin2 = <<Bin1/binary,5,6,7>>, %% 3.


Bin3 = <<Bin1/binary,17>>, %% 4

{Bin2,Bin3} %% 5.
```

1. assigns a heap binary to the Bin0 variable.
2. append operation. As Bin0 has not been involved in an append operation
    1. a new RefC binary is created and the contents of Bin0 is copied into it.
    2. the ProcBin part of the RefC bin has size set to the size of the data stored (4 bytes)
    3. the binary object has extra space allocated. its total size is either twice the size of Bin1 (the new binary) or 256, whichever is larger. (now 256 bytes)
3. Bin1 has been used in an append operation, and it has 256-4 bytes of unused storage at the end, so the 3 new bytes are stored there.
4. Bin1 is not the result of latest append. Bin1 has to be copied and expanded

Binary init in details



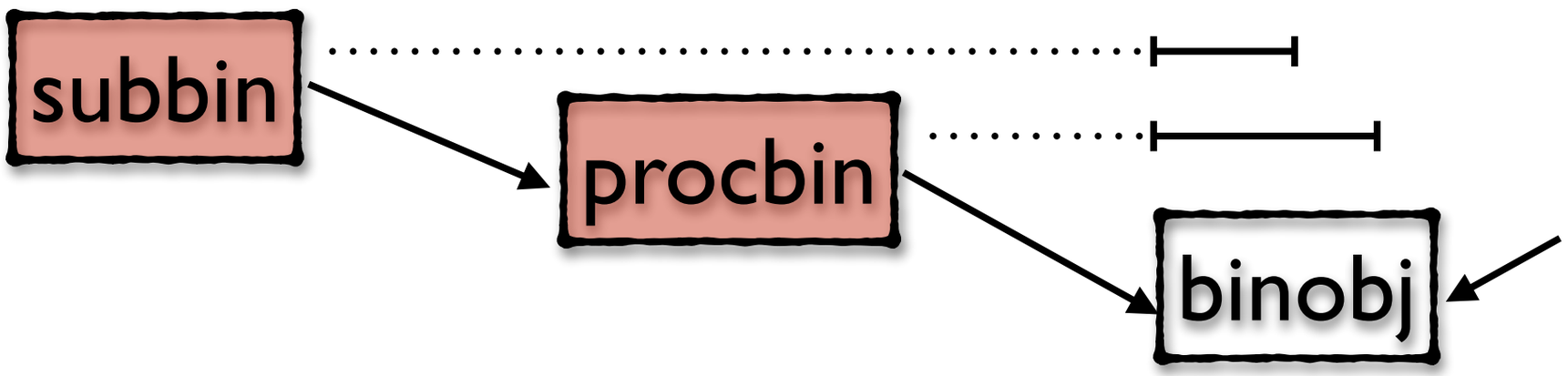
1. heap binary
   (bytesize =< 64)
2. RefC binary
   (bytesize > 64)
3. heap bitstring
   (bitsize mod 8 /= 0)
4. RefC bitstring
   (bitsize mod 8 /= 0)

Binary append in details

1. check if binary is writeable (can be modified in place)

   If SubBin or ProcBin is not writable

   allocate binaries:



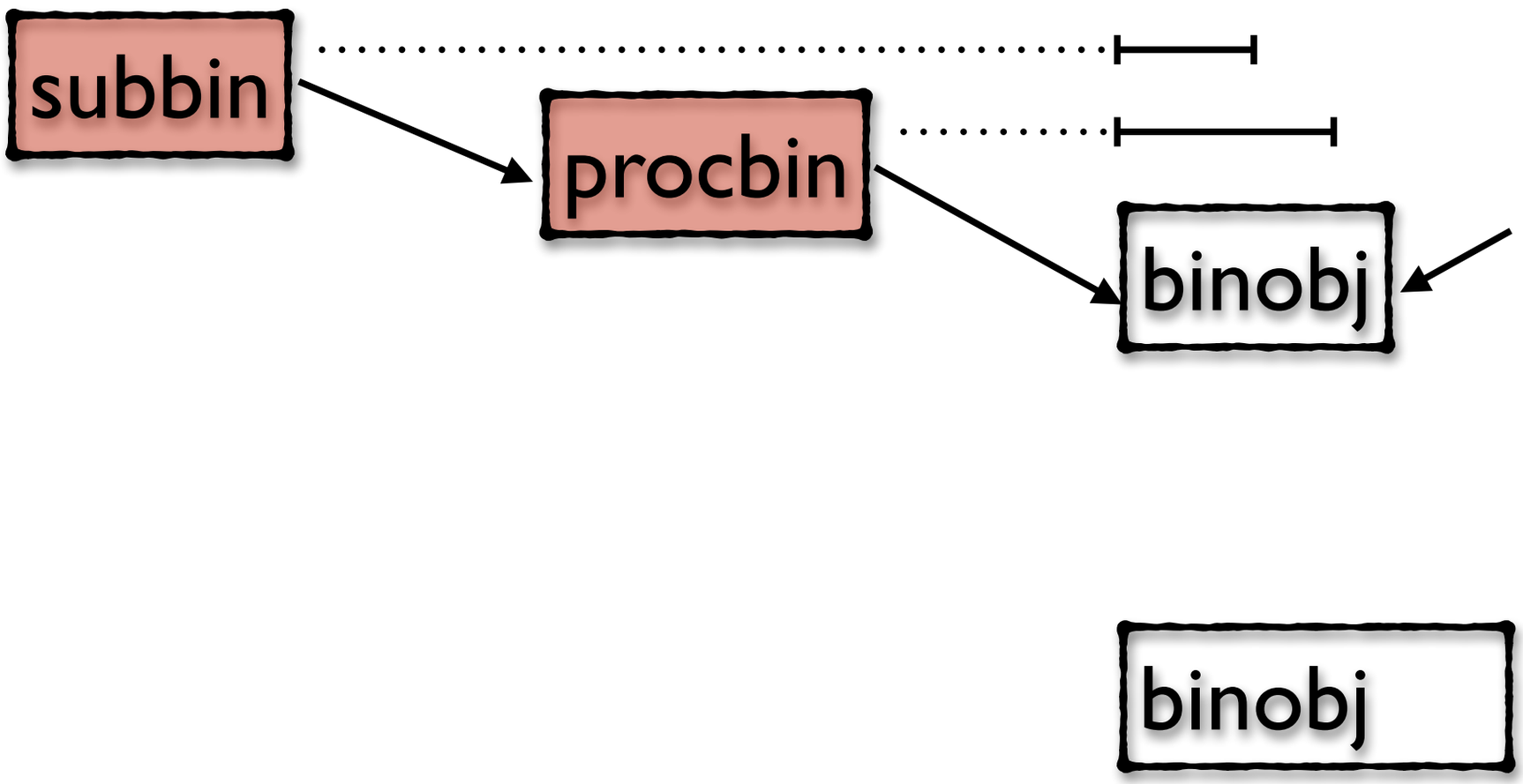
- SubBin is part of a newly created binary (not appended yet)
- SubBin is not the last append
- RefC was copied to other process (it's not the only reference)

Binary append in details

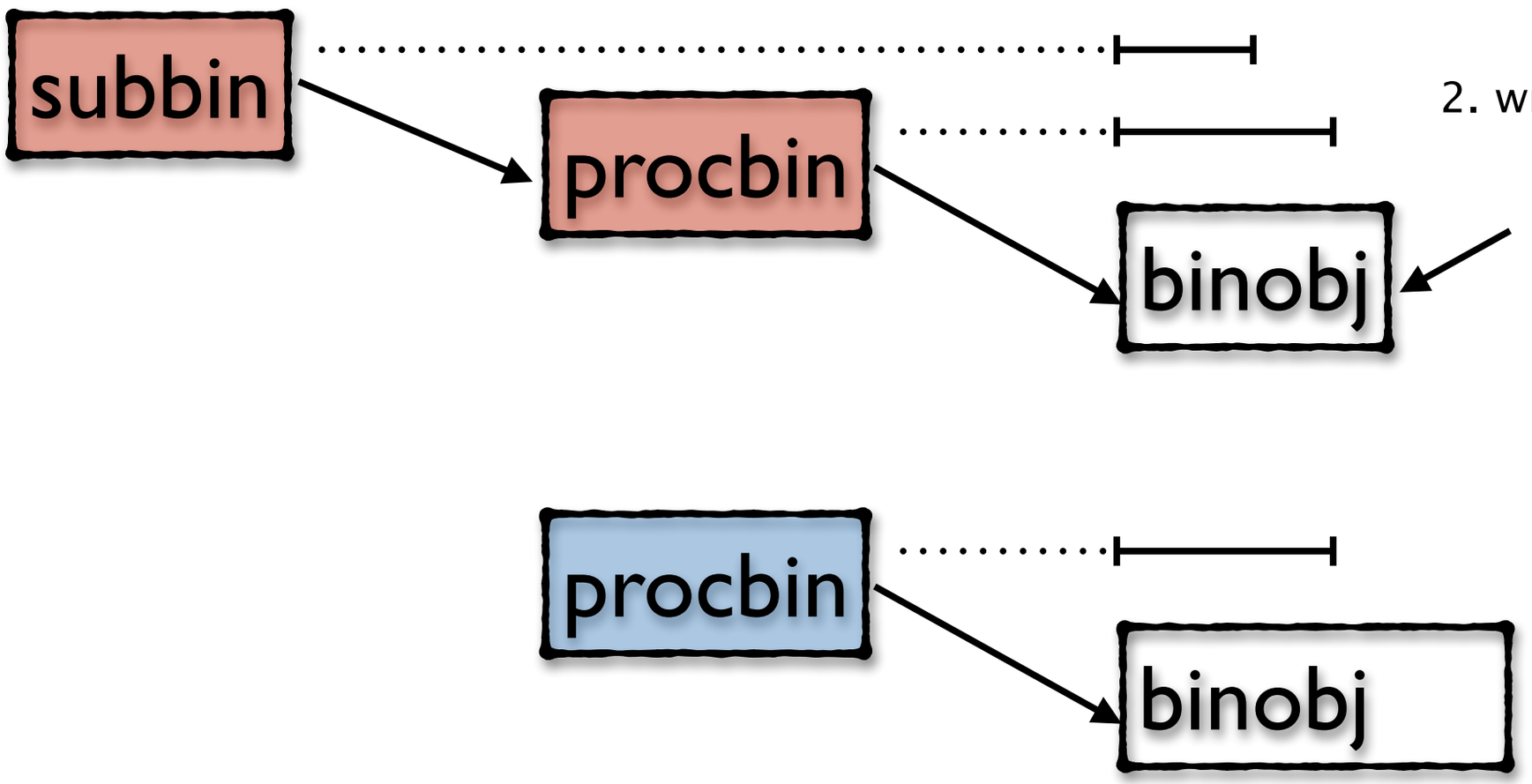
If SubBin or ProcBin is not writable

allocate binaries:

1. new binary object with size = max(256, 2 * (old bin size + size of data to be appended))



Binary append in details

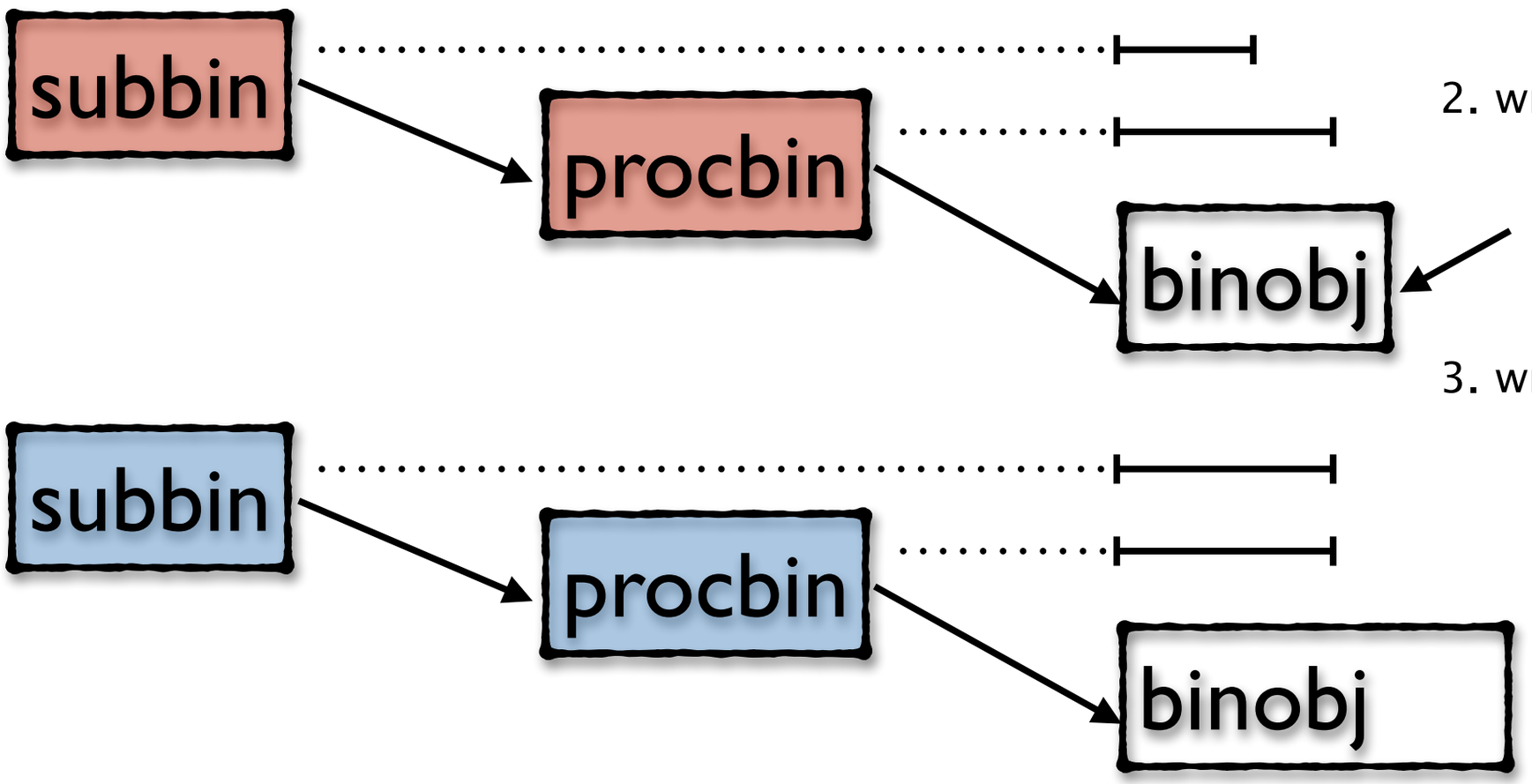


If SubBin or ProcBin is not writable

allocate binaries

1. new binary object with size = max(256, 2 * (old bin size + size of data to be appended))
2. writable ProcBin pointing to the binary object

Binary append in details

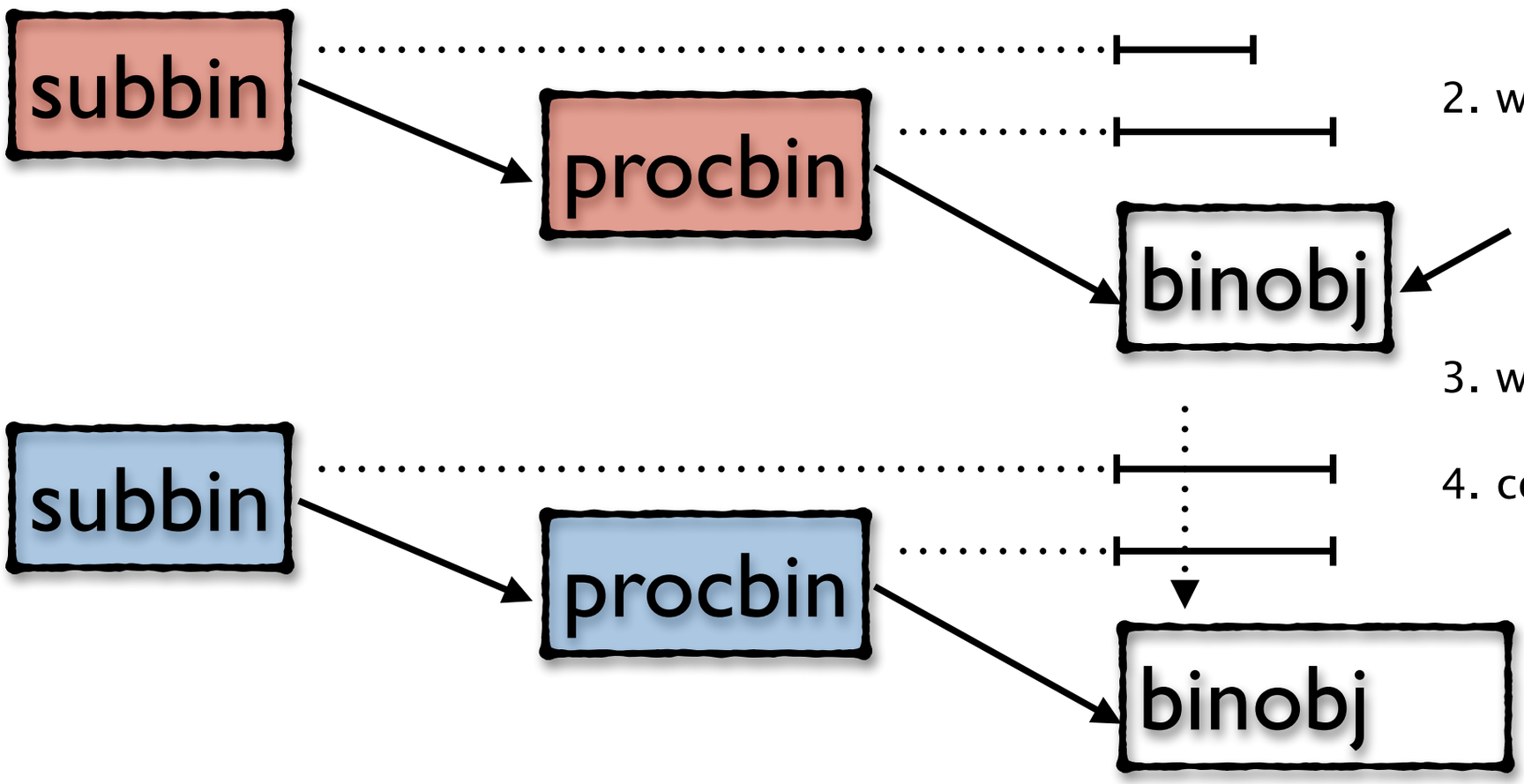


If SubBin or ProcBin is not writable

allocate binaries

1. new binary object with size = max(256, 2 * (old bin size + size of data to be appended))
2. writable ProcBin pointing to the binary object
3. writable SubBin pointing to the ProcBin

Binary append in details

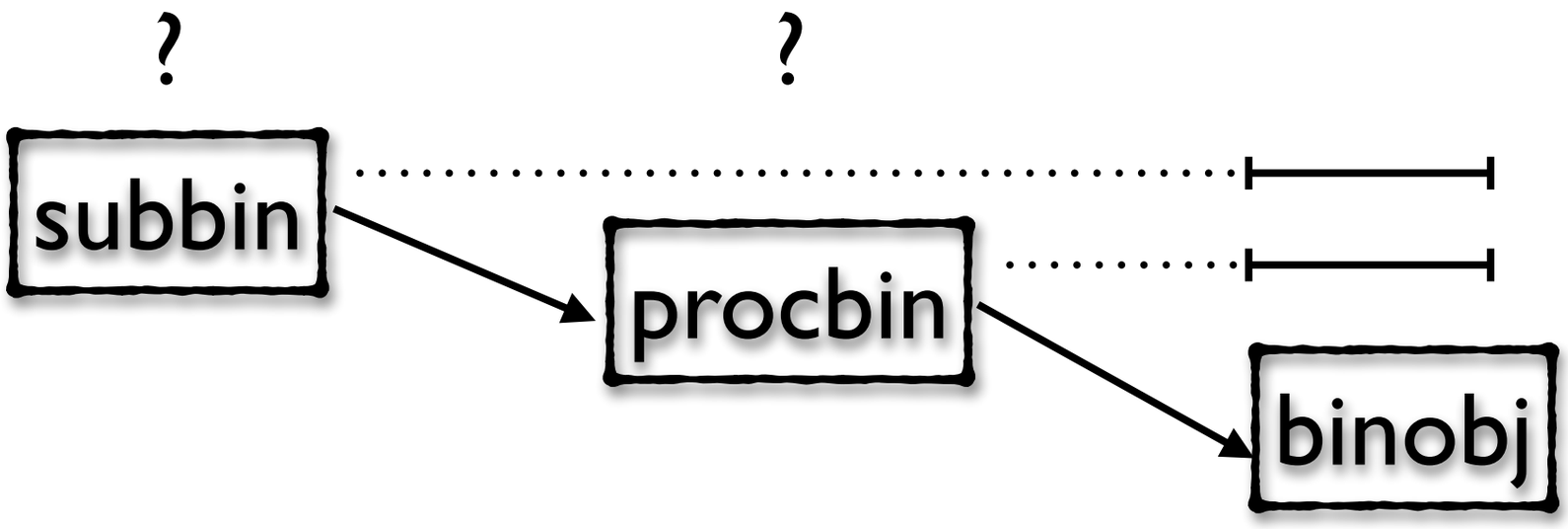


If SubBin or ProcBin is not writable

allocate binaries

1. new binary object with size = max(256, 2 * (old bin size + size of data to be appended))
2. writable ProcBin pointing to the binary object
3. writable SubBin pointing to the ProcBin
4. copy the original data into the new binary object

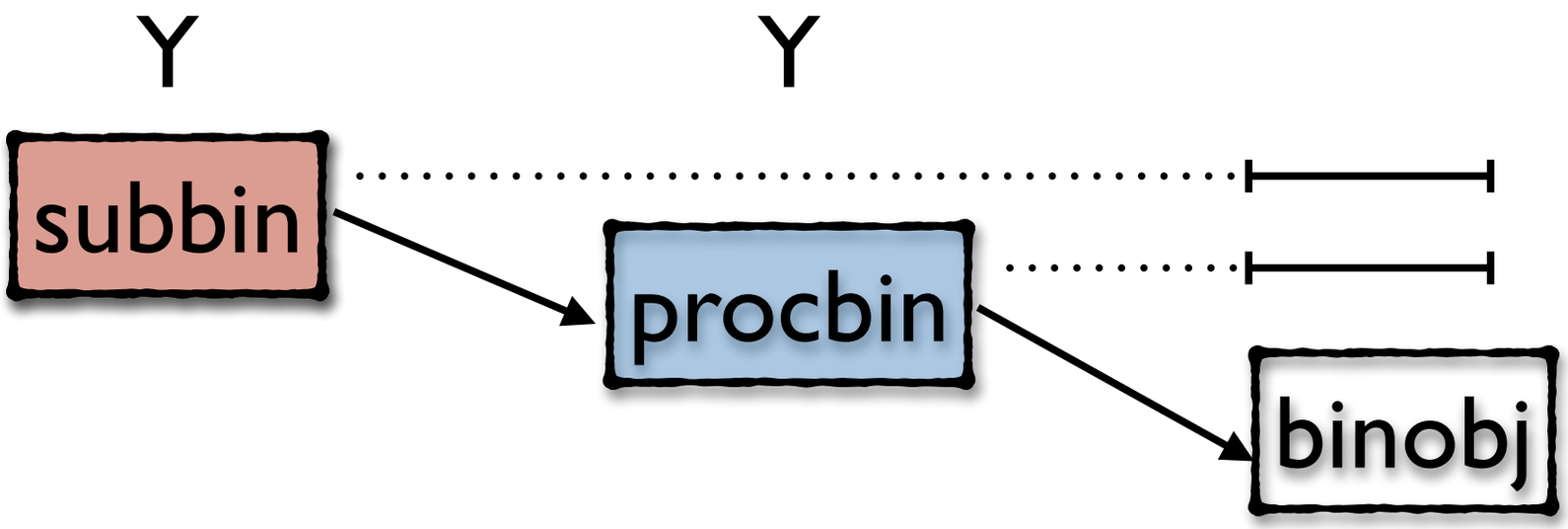
Binary append in details



1. check if binary is writeable (can be modified in place)

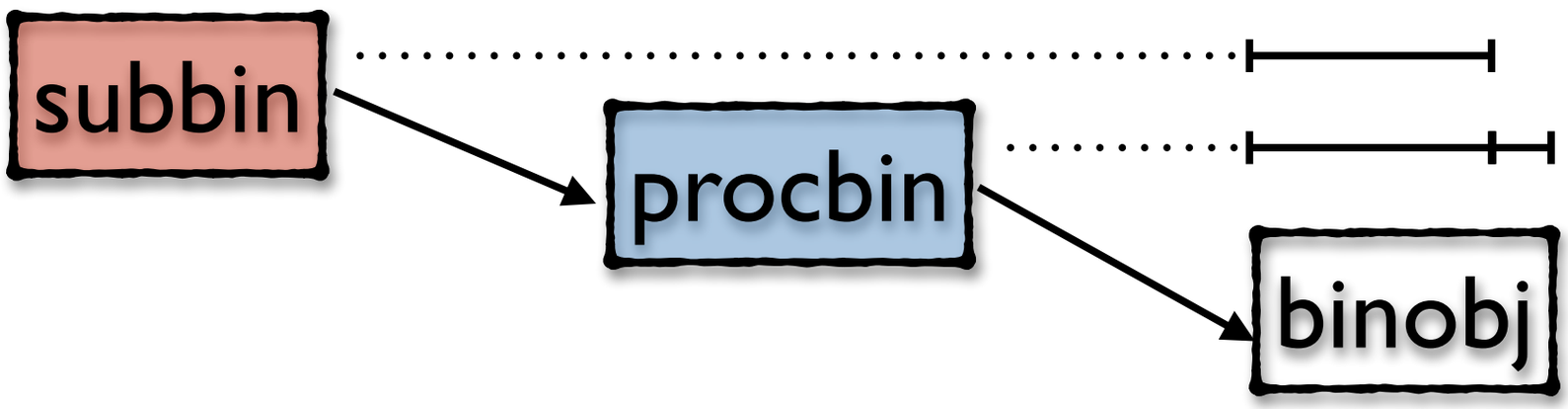
(ProcBin is writable only if there is only one reference to the binobj ie. only this process has a copy)

Binary append in details



if YES then all is good
mark SubBin as not writeable
(immutability)

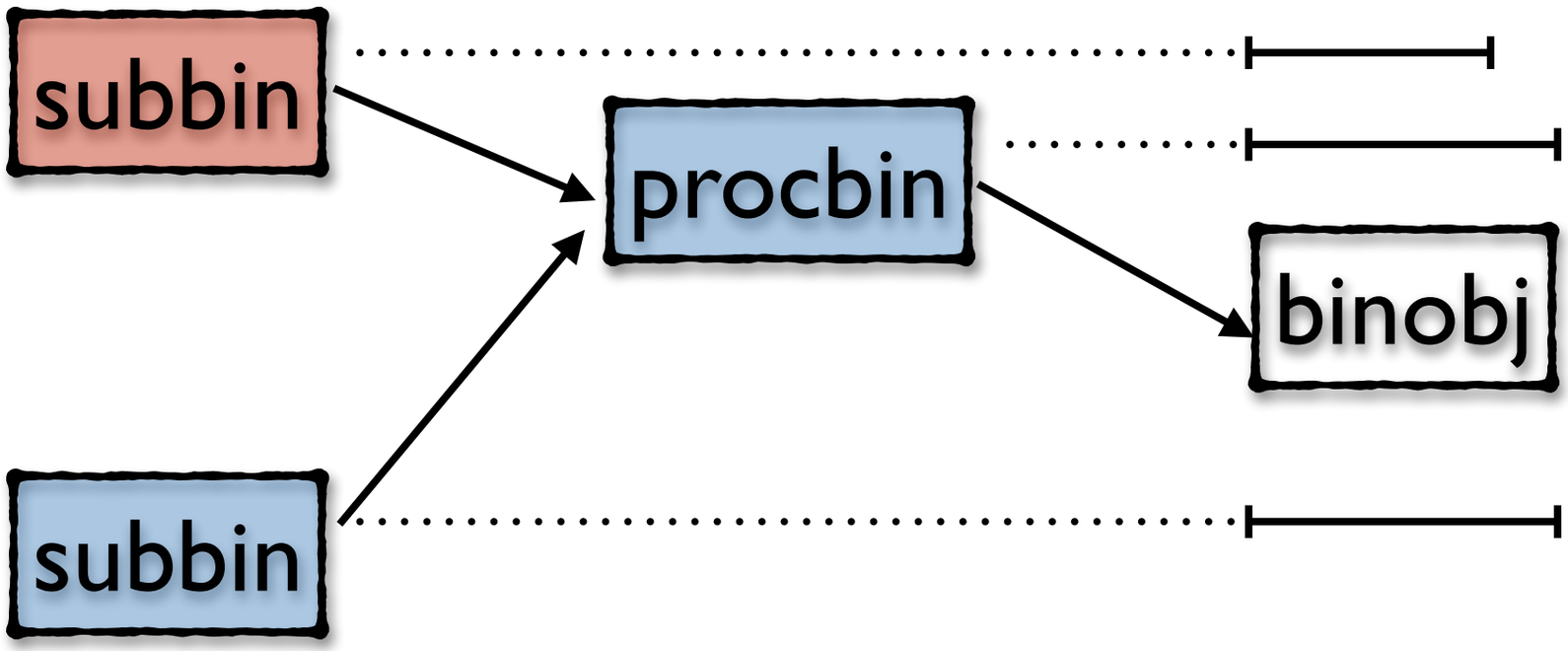
Binary append in details



2. check if there is enough room (after updating ProcBin size

if YES then all is good

Binary append in details

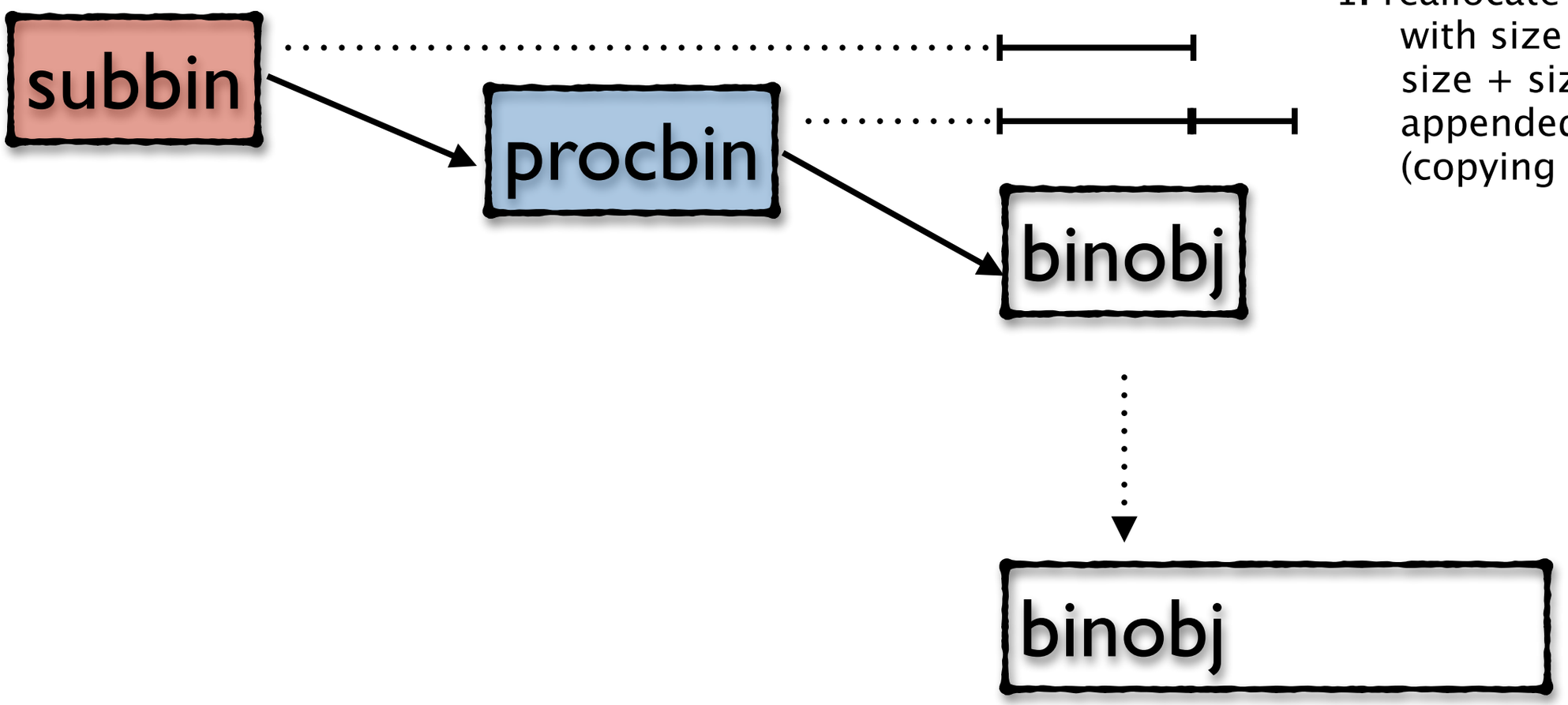


2. check if there is enough room (after updating ProcBin size

if YES then all is good

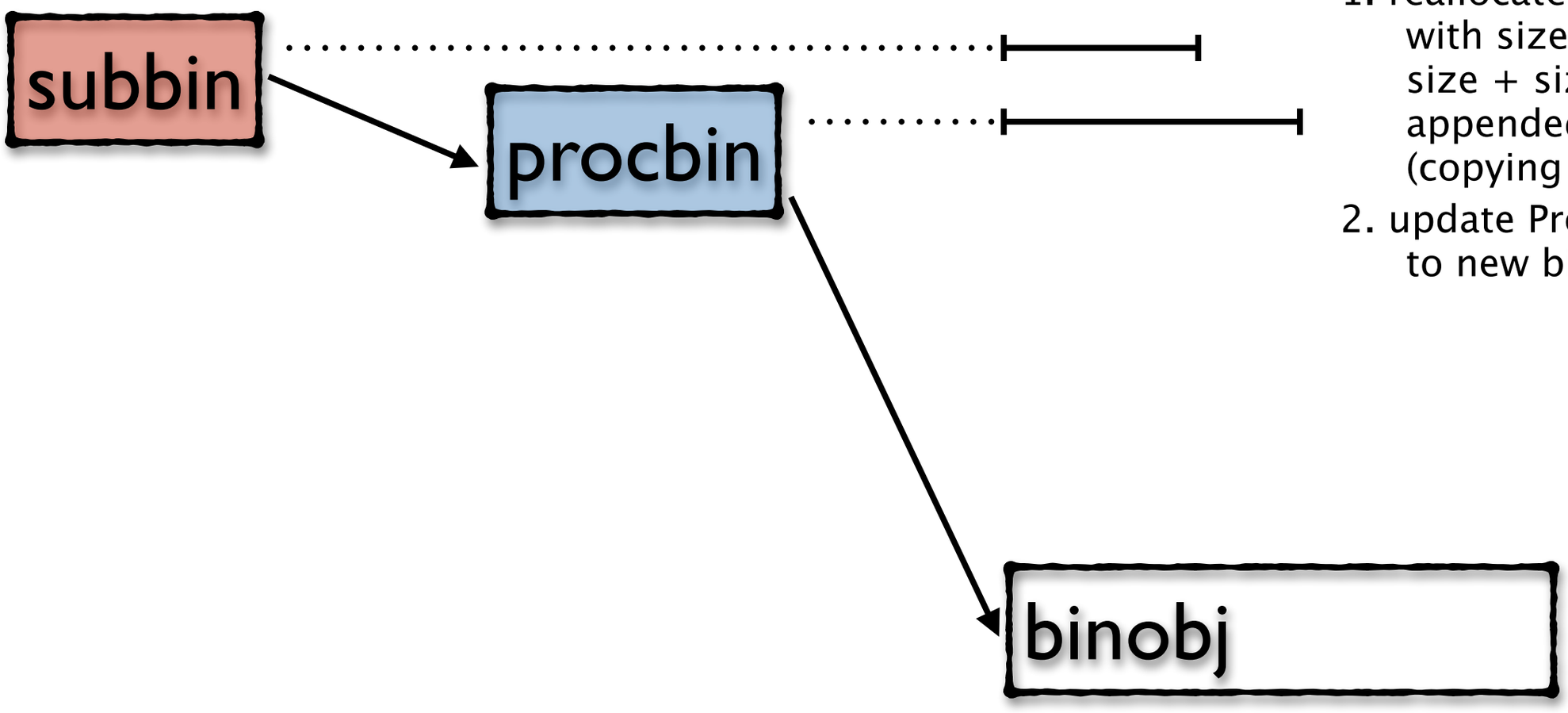
3. finally create new writable SubBin pointing to ProcBin and return it

Binary append in details



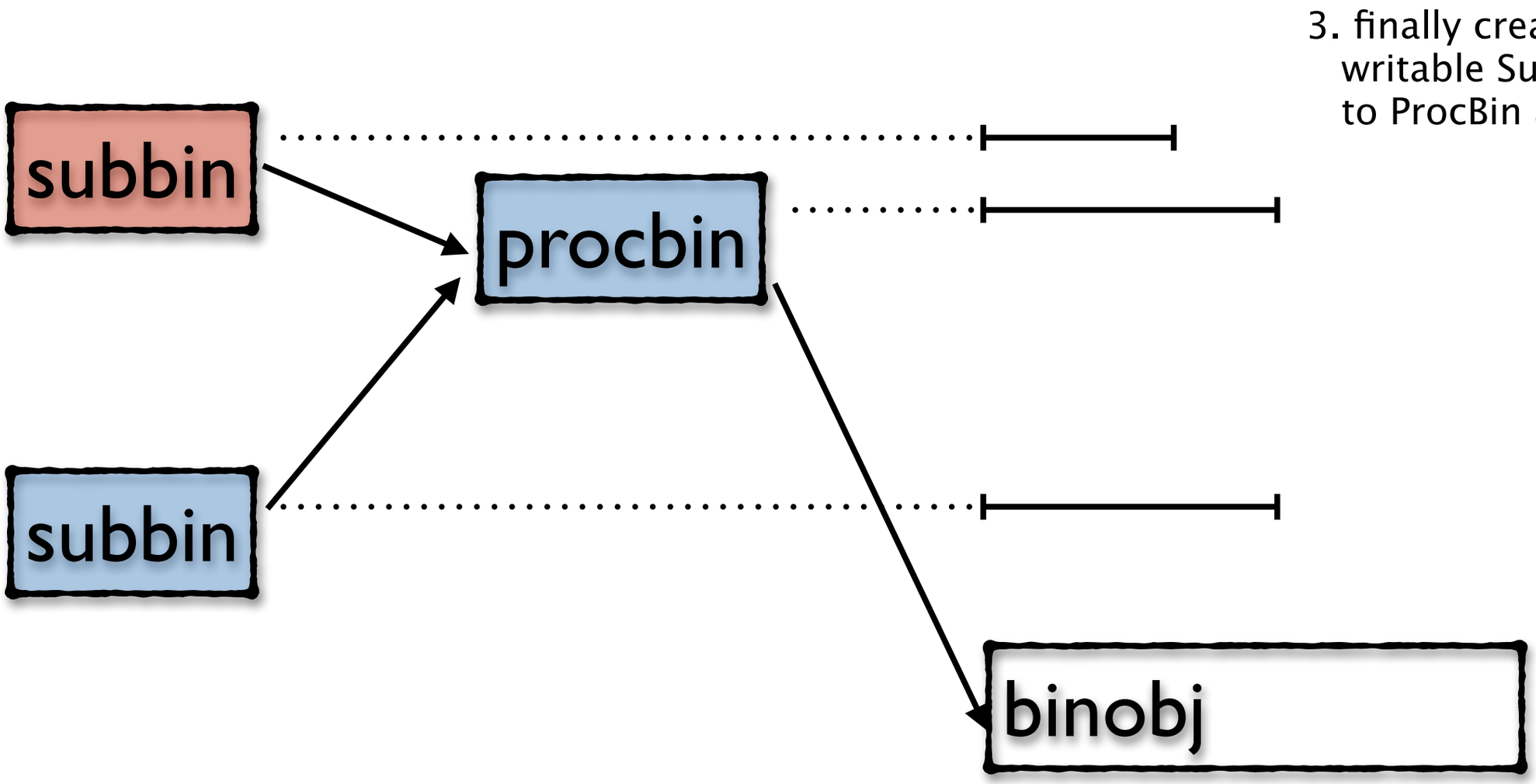
2. if NOT enough room
   1. reallocate binary object with size = 2 * (old bin size + size of data to be appended) (copying its old content)

Binary append in details



2. if NOT enough room
   1. reallocate binary object with size = 2 * (old bin size + size of data to be appended) (copying its old content)
   2. update ProcBin to point to new binary object

Binary append in details



3. finally create new writable SubBin pointing to ProcBin and return it

Constructing binaries - worth noting

- "As the runtime system handles the optimisation (instead of the compiler), there are very few circumstances in which the optimisation does not work."
  Less room for code generation-time optimisations
- When a binary is marked as not writable, it will be shrunk at the same time to reclaim the extra space allocated for growing.
- Appending to a binary always creates a RefC binary with size of at least 256 bytes (even if just 2 bytes are concatenated)
- Tracing sends binaries (in arguments) which marks them as non-writable enforcing more copying and allocation for the same code

Binary matching

```
skip_byte(<<_, Bin1/binary>>) ->
    Bin1.
```

1. bs_start_match2 – a match context is created (The match context points to the first byte of the binary)
2. bs_skip_bits2 – 8 bit skipped (The match context is updated to point to the second byte in the binary)
3. bs_get_binary – match out the remainder of the binary creating a SubBin match context is not used anymore, it is garbage_collected at next gc run

Binary matching

```
skip_byte(<<_, Bin1/binary>>) ->
    match_byte(Bin1).

match_byte(<<Byte, _/binary>>) ->
    Byte.
```

1. bs_start_match2
2. bs_skip_bits2
3. ~~bs_get_binary~~ – no SubBin created (only size is checked that Bin1 is indeed a binary and not a bitstring – bs_test_unit)
4. bs_start_match2 in match_byte – basically does nothing when it sees that it was passed a match context instead of a binary.
5. bs_get_integer2 – 8 bit matched into an integer (The match context is updated)
6. bs_context_to_binary – in case of function_clause (3., 5. fails) the match context is converted back to a binary (SubBin is created pointing to the binary object)

Delayed sub binary optimisation

- Since R12B the compiler sees that there is no point in creating a sub binary, because there will soon be a call to a function that immediately will create a new match context and discard the sub binary.
- Happens at a late stage in the compiler on beam asm code
- Removing or replacing op bs_get_binary
- All places must be binary matching where the bin/match context is used
- Enable verbose info with compiler option `bin_opt_info`
  - Compiler emits warnings where it could/couldn't optimise
  - Emits info messages for local functions which cannot handle match_context as arg

Bin opt cannot be applied when

- SubBin is used to create a term or returned

```
read_struct(Bin0, Struct) ->
    {Bin1, Type} = prot_read(Bin0, byte),
    {Bin2, Field} =
        case Type of ?TYPE_I16 ->
              prot_read(Bin1, i16);
        …
        end,
    read_struct(
      Bin2, set_field(Struct, Field));
…

prot_read(<<I16:16, Bin/binary>>, i16) ->
  % Warning: NOT OPTIMIZED:
  % sub binary is used or returned
    {Bin, I16};
prot_read(<<Byte, Bin/binary>>, byte) ->
  % Warning: NOT OPTIMIZED:
  % sub binary is used or returned
    {Bin, Byte}.
```

```
read_struct(<<?TYPE_I16, Field:16,
              Bin/binary>>, Struct) ->
  % Warning: OPTIMIZED:
  % creation of sub binary delayed
    read_struct(
      Bin, set_field(Struct, Field));
…
```

Bin opt cannot be applied when

- SubBin is used to create a term or returned
- SubBin is passed to a remote function
  (if its passed to an erlang:split_binary call it can be replaced with a matching)
- SubBin is matched or used in more than one place
- different control paths use different positions in the binary
- catch or try/catch is used

Bin opt cannot be applied when

- called local function does not begin with a suitable binary matching instruction
  - not only plain variables as args to the left of binary pattern (change arg order, make binary to be matched the first arg)
  - or one of them is used in a guard
  - the whole binary or a matched out binary is used in a guard
  - in consecutive clauses non-variable after a variable (underscore is also a variable!) (changing clause order might help)

```
read(<<I:16, _/binary>>, i16) ->
  I;
read(Bin, struct) ->
  a:read_struct(Bin);
read(<<I:32, _/binary>>, i32) ->
  %% INFO: matching non-variables after a previous clause matching a variable
  %% will prevent delayed sub binary optimization
  I.
```

The new library

The new library

- provides two codecs (only for the binary protocol)
  - generic (just squash together: zmq transport + binary protocol)
  - generated (same API)
- generator
  - uses the output erlang modules of original thrift compiler
  - generates one codec module per service
  - supports custom decoder for given type – can be manually optimised

Old usage example

- cascading style
- how to pass additional args to Handler
- how to measure encode/decode time

```
ProtoGen = fun() ->
  {ok, Transport} =
    thrift_zmq_transport:new(BinRequest, Pid, ZmqHeaders),
  {ok, Protocol} =
    thrift_binary_protocol:new(Transport),
  {ok, Protocol}
end,
thrift_processor:init(
  {server_name, ProtoGen, Service, HandlerCB}).
```

New simple API

```
-spec decode_request(BinRequest :: binary(),
                     Service :: atom())
        -> {Seqid :: integer(),
            Function :: atom(),
            Params :: tuple()}.

-spec encode_result(Seqid :: integer(),
                    Function :: atom(),
                    Result :: {reply, term()} | ok,
                    Service :: atom())
        -> {send, BinReply :: binary()} | nosend.
```

Performance results

- encoding (append - VM optimised)
  - generic/simple squashing: big gain (~8x)
  - generated: small additional gain (~16x)
- decoding (matching - compiler optimised)
  - generic/simple squashing: small gain (~2x)
  - generated: big additional gain (~16x)

- why the difference in gain between encoding and decoding?

The generic codec

- in the old lib a SubBin is created for every base type data
- can be avoided by merging the two layers together (no code generation needed)

```
%% thrift_binary_protocol
write(#bin_prot{transport = Trans}, {i32, I32}) ->
    thrift_transport:write(Trans, <<I32:32/big-signed>>);

%% thrift_zmq_transport
write(#zmq_trans{buf = Buf}, Data) ->
  #zmq_trans{buf = <<Buf/binary, Data/binary>>}.
```

What code to generate

- Binary arg always the first
- Avoid returning the tail-binary, chain of function calls
- Use IDL info: from field id we know field type, from field type we know how to match the field value
- lists of base types (fixed-length) can be encoded/decoded in batches let's say decoding every 10 element at once
- similarly encoding/decoding consecutive base type struct fields in one go (decoder caution: field order is not fixed)

Issues

- variable-length fields (list of struct with list field)

Generated code example

```
decode_structA(<<?TypeI32:16, 9, Val:32, Bin/binary>>,
               StructA) ->
  decode_fields_structA(Bin, setelement(9, StructA, Val);

decode_fields_structA(
<<?TypeList:16, 10, ?TypeI32:16, Size, Bin0/binary>>,
StructA) ->
 {Bin, Val} = decode_list_i32(Bin0, Size, []),
 decode_fields_structA(Bin, setelement(10, StructA, Val);
```

The new library

- Addresses a specific scenario
- Performance gain
- Simpler API
- Was a good learning as well

Questions, Comments…