UNIVERSITY OF WATERLOO

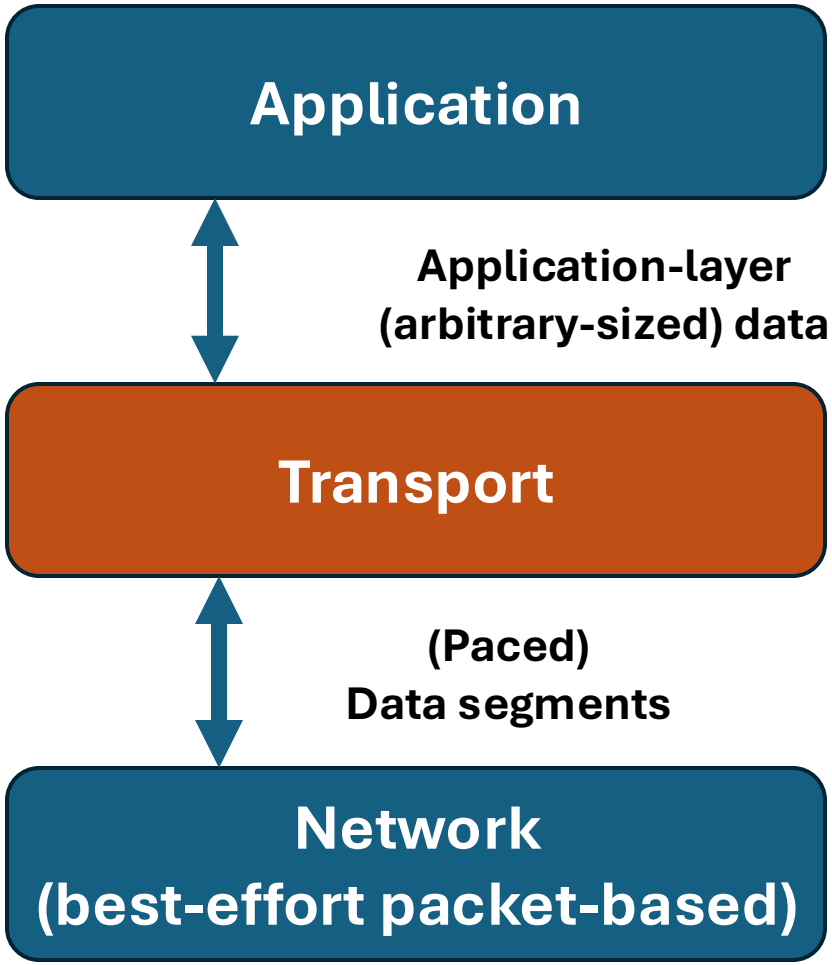
High-Level and Target-Agnostic Transport Programs

Mina Tahmasbi Arashloo

University of Waterloo

P4 Workshop 2025

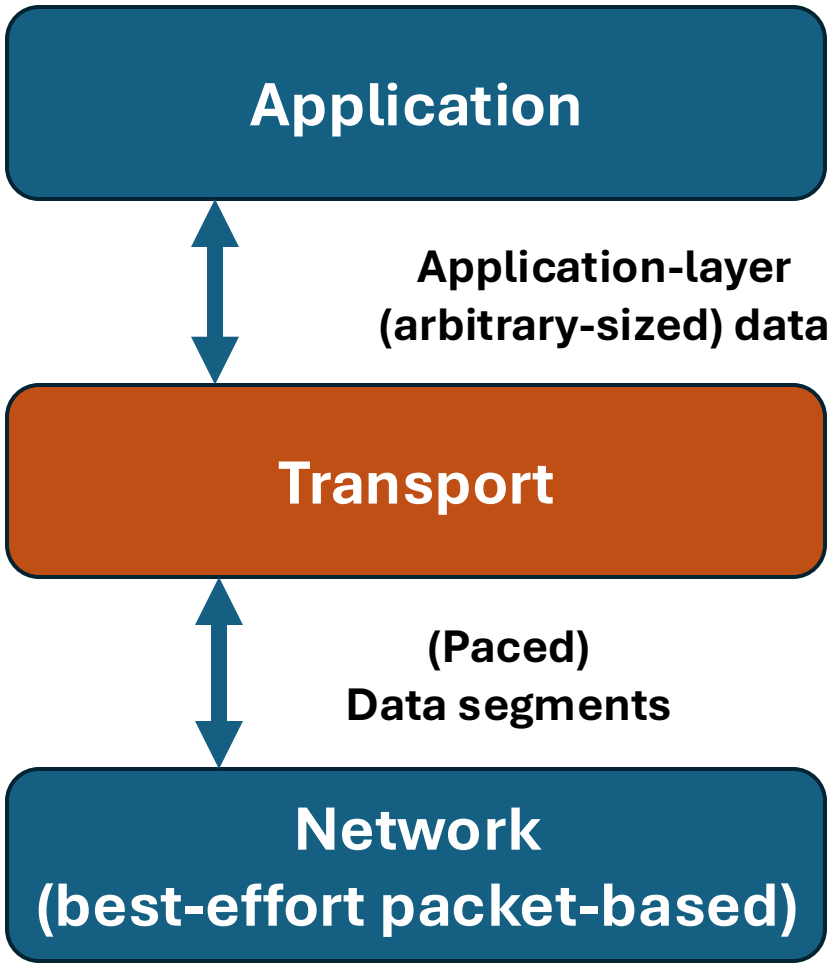
No “one-size-fits-all” transport protocol



which data segment to send and when such that

- Data is reliably delivered to the receiver
- as fast as possible
- w/o overwhelming the network and receiver

No “one-size-fits-all” transport protocol



which data segment to send and when

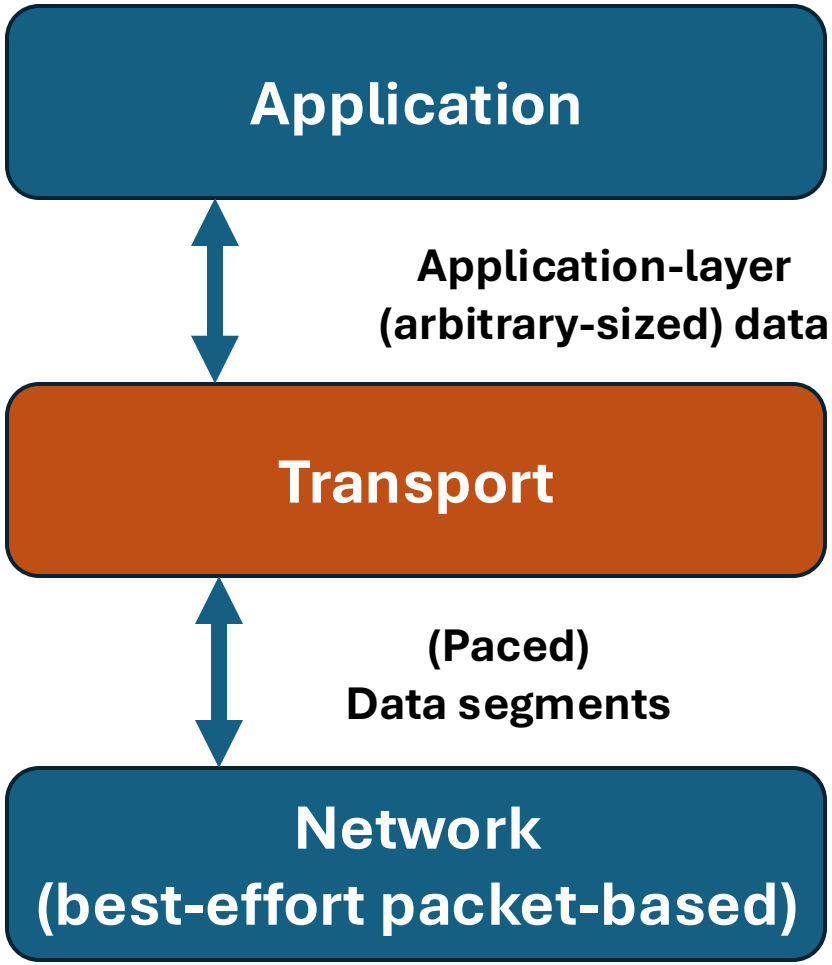
such that

- Data is reliably delivered to the receiver

Depends on

- Network characteristics
  - Wide area? Data center?
- Applications
  - Traffic patterns: small flows? Bursty?
  - Requirements: low latency? High throughput?

No “one-size-fits-all” transport protocol

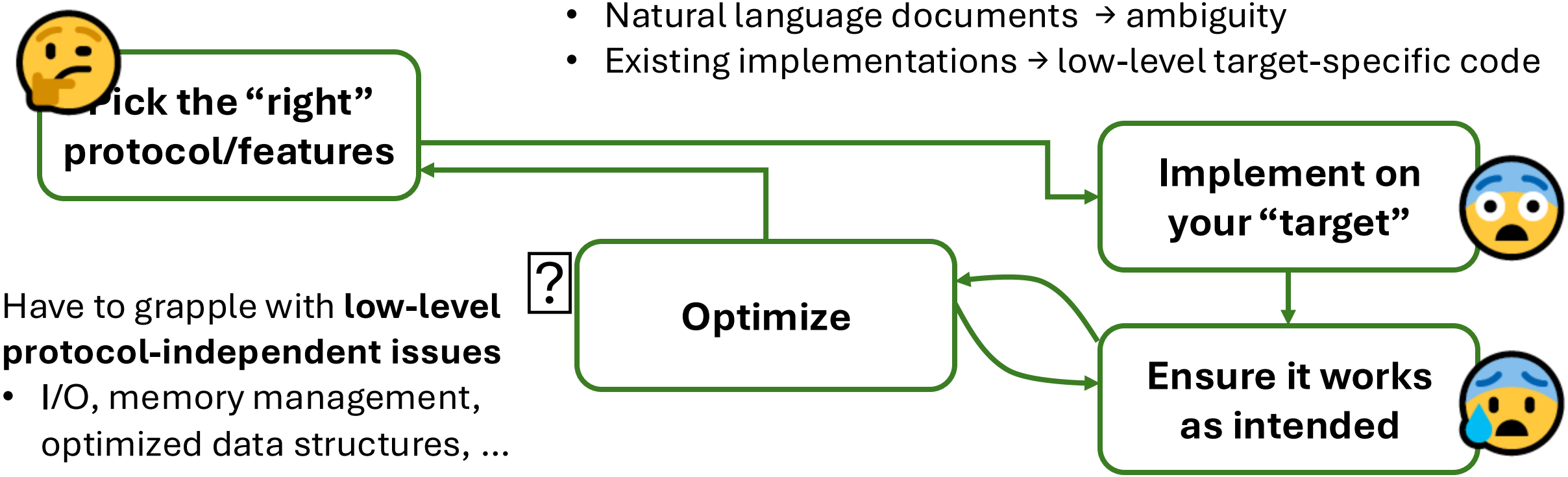


TCP and its many many variants
QUIC
NDP
Homa
RoCEv2
…

The transport protocol development cycle today

No high-level specification with well-defined semantics

- Natural language documents → ambiguity
- Existing implementations → low-level target-specific code



Have to grapple with **low-level protocol-independent issues**

- I/O, memory management, optimized data structures, …

No high-level specification with well-defined semantics

- Intended behavior is not always clear
- Pick and choose scenarios to test
- No automated high-coverage analysis and testing

The transport protocol development cycle today

No high-level specification with well-defined semantics

- Natural language documents → ambiguity
- Existing implementations → low-level target-specific code

Pick the "right" protocol/features

Implement on your "target"

?

Optimize

Ensure it works as intended

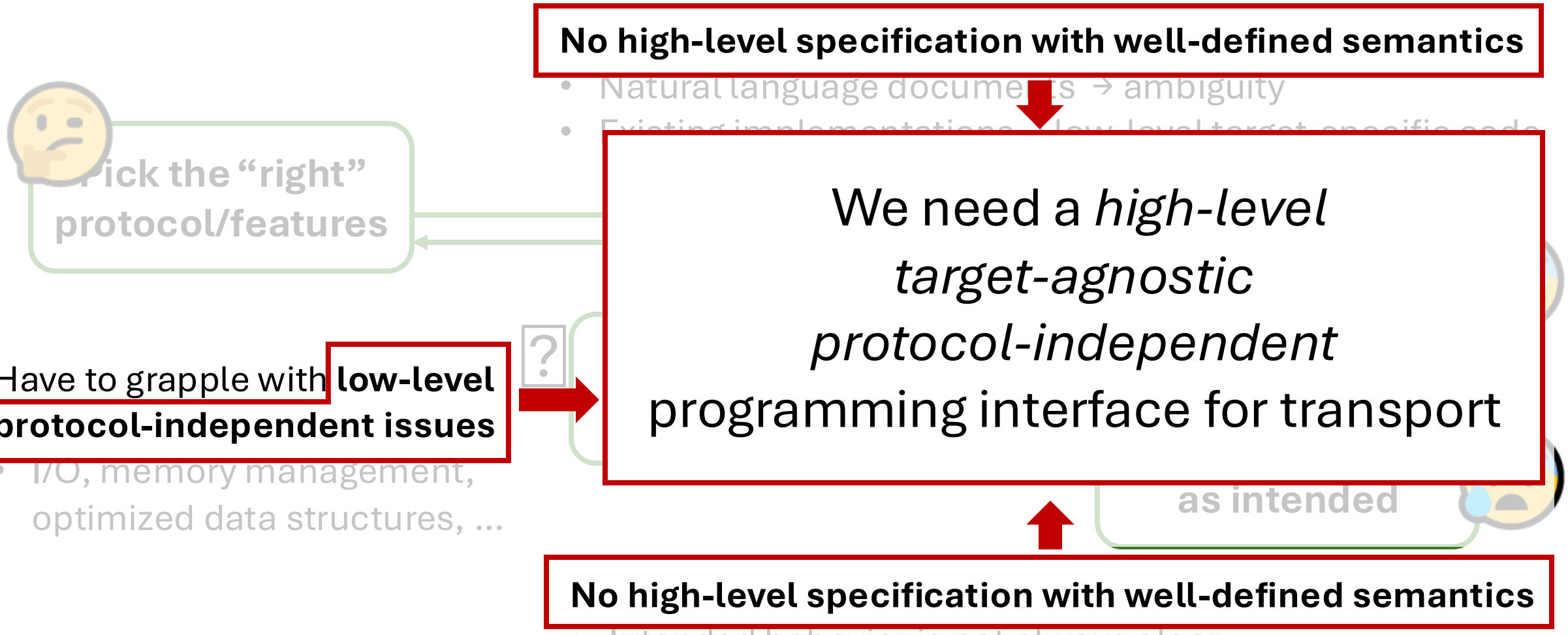
Have to grapple with **low-level protocol-independent issues**

- I/O, memory management, optimized data structures, …

No high-level specification with well-defined semantics

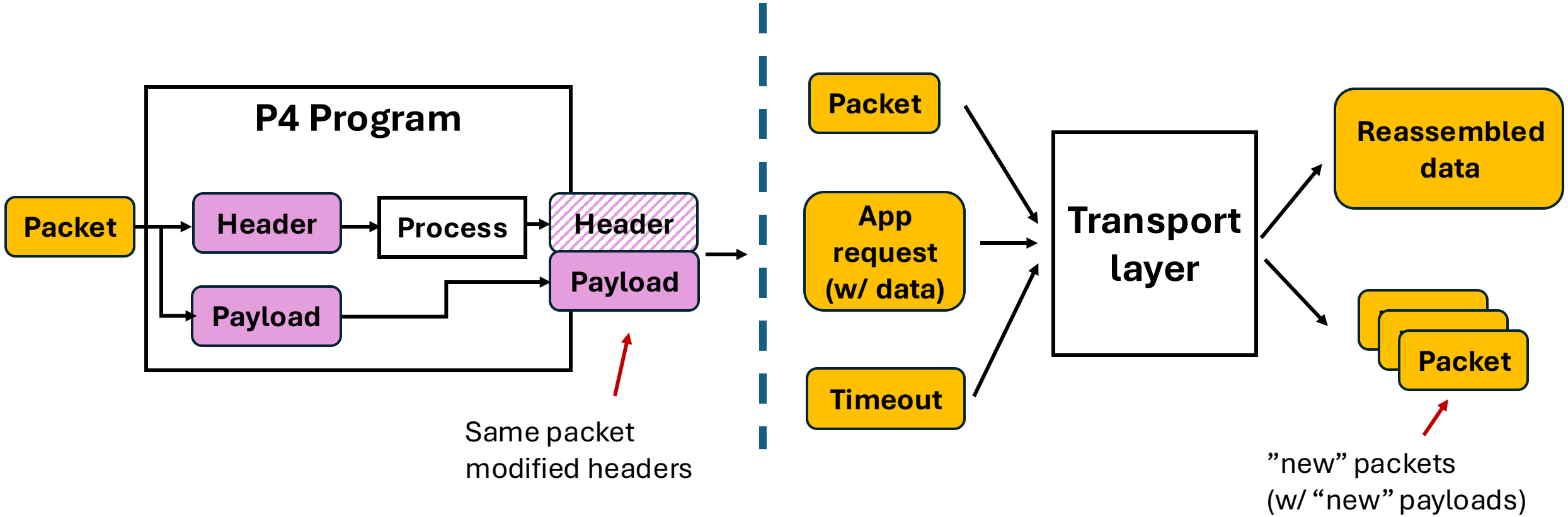
- Intended behavior is not always clear
- Pick and choose scenarios to test
- No automated high-coverage analysis and testing

The transport protocol development cycle today

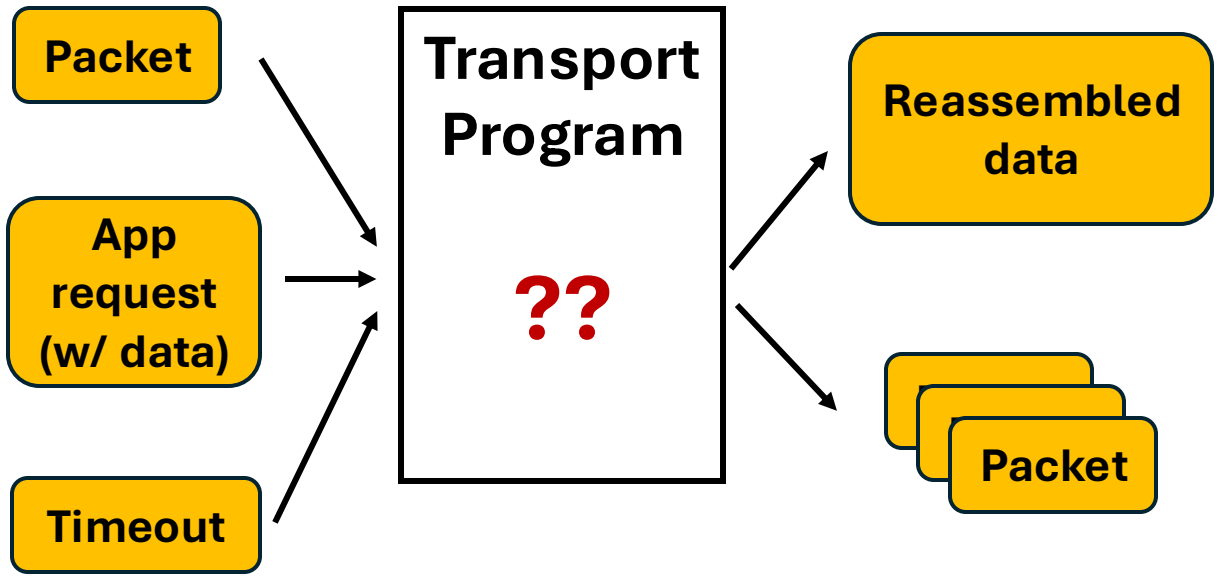


Can’t we use P4?

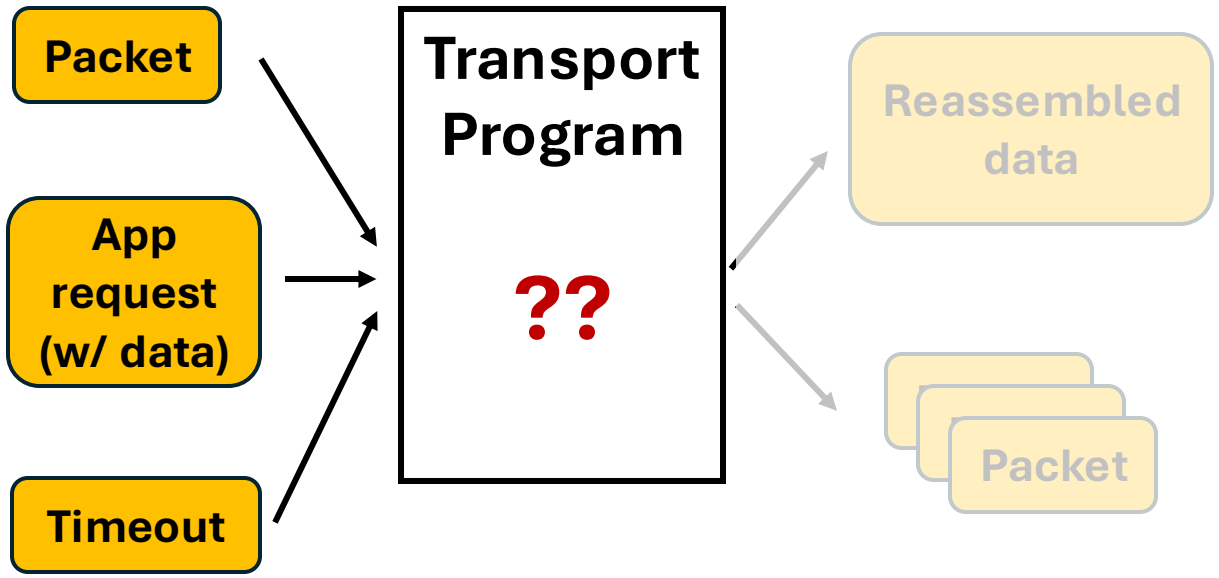
- P4 is the most widely-used high-level network programming language
- … but for L2/L3 network functionality (i.e., routing and forwarding)



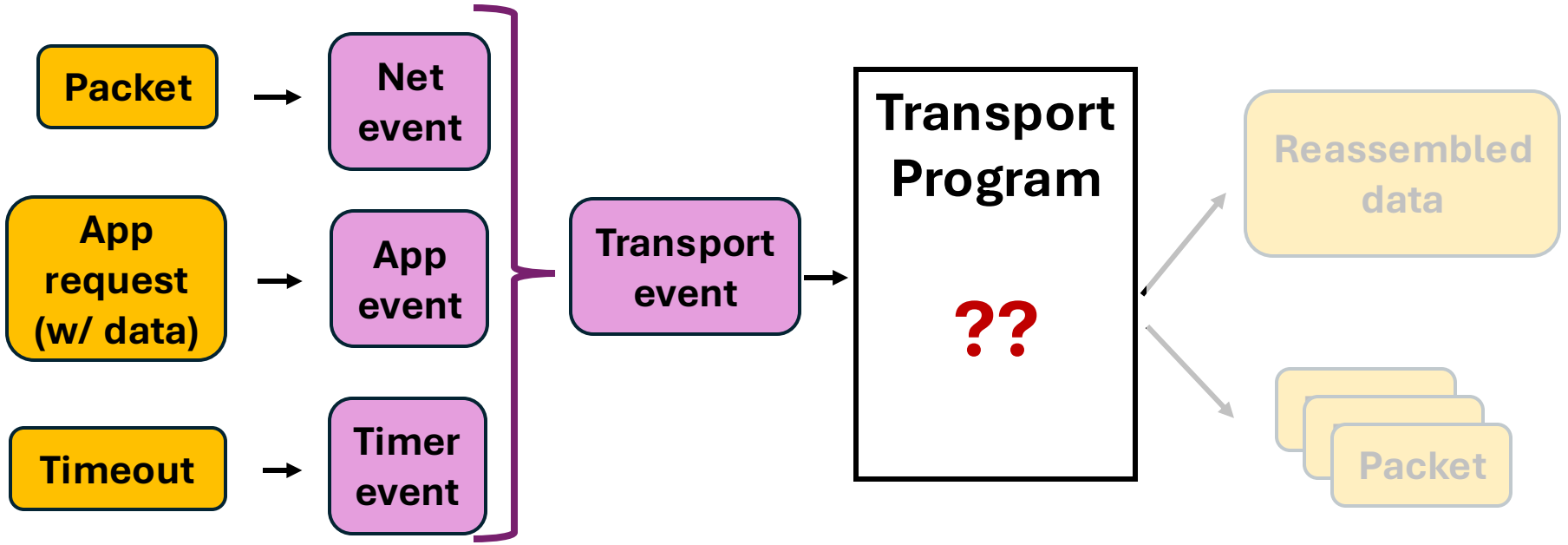
What should a transport program look like?



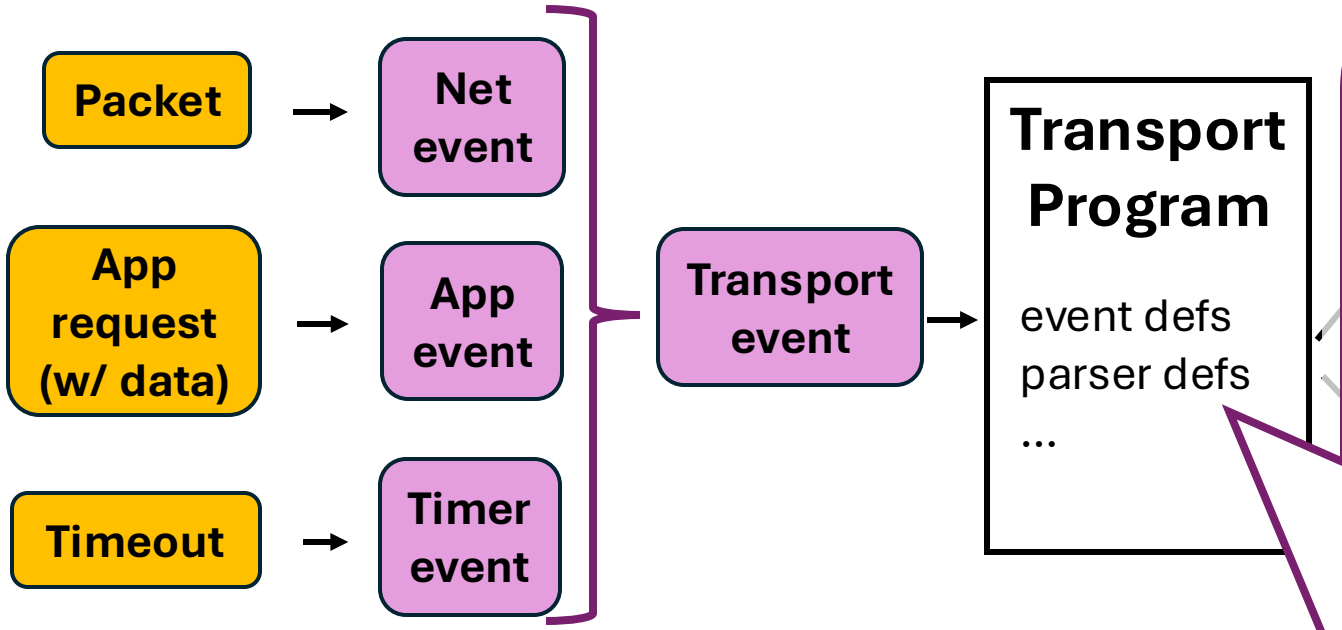
What should a transport program look like?



Transport events



Transport events



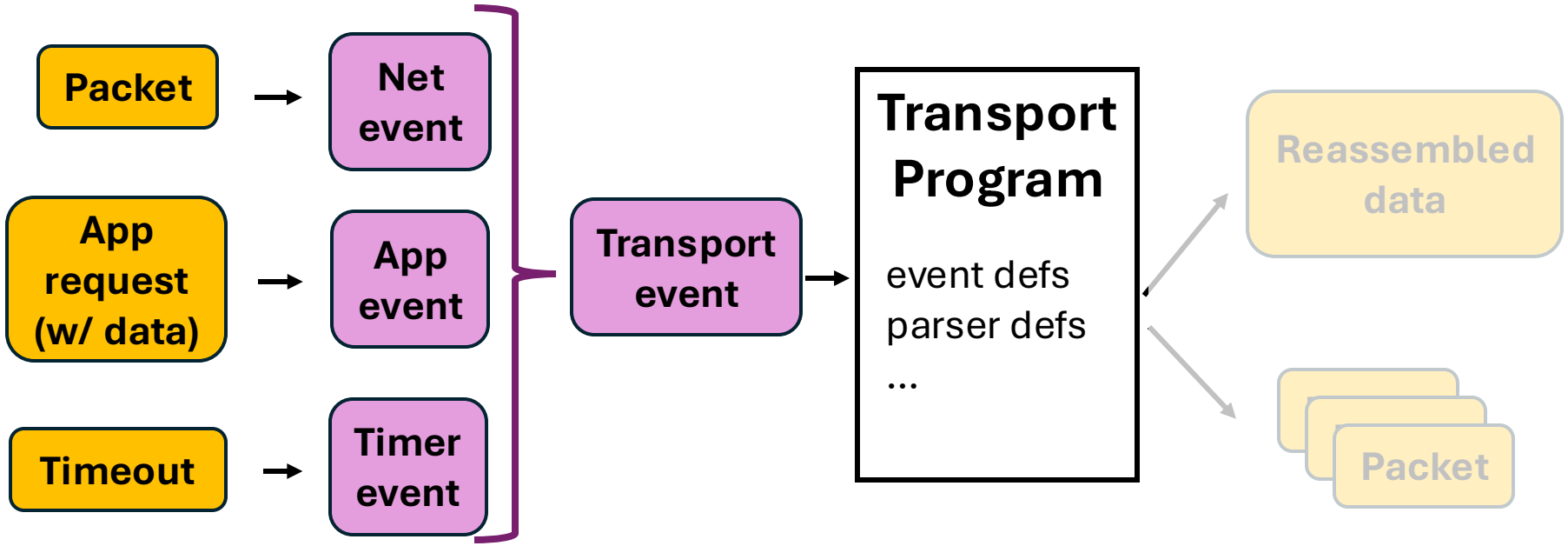
- Specifies what events it expects:

```
event tcp_snd : APP {
  uint32 data_size;
  addr_t user_buff_addr;
  ...}

event tcp_data_pkt : NET {
  uint32 seq_num;
  uint32 payload_size;
  addr_t payload_addr;
  ...}
```

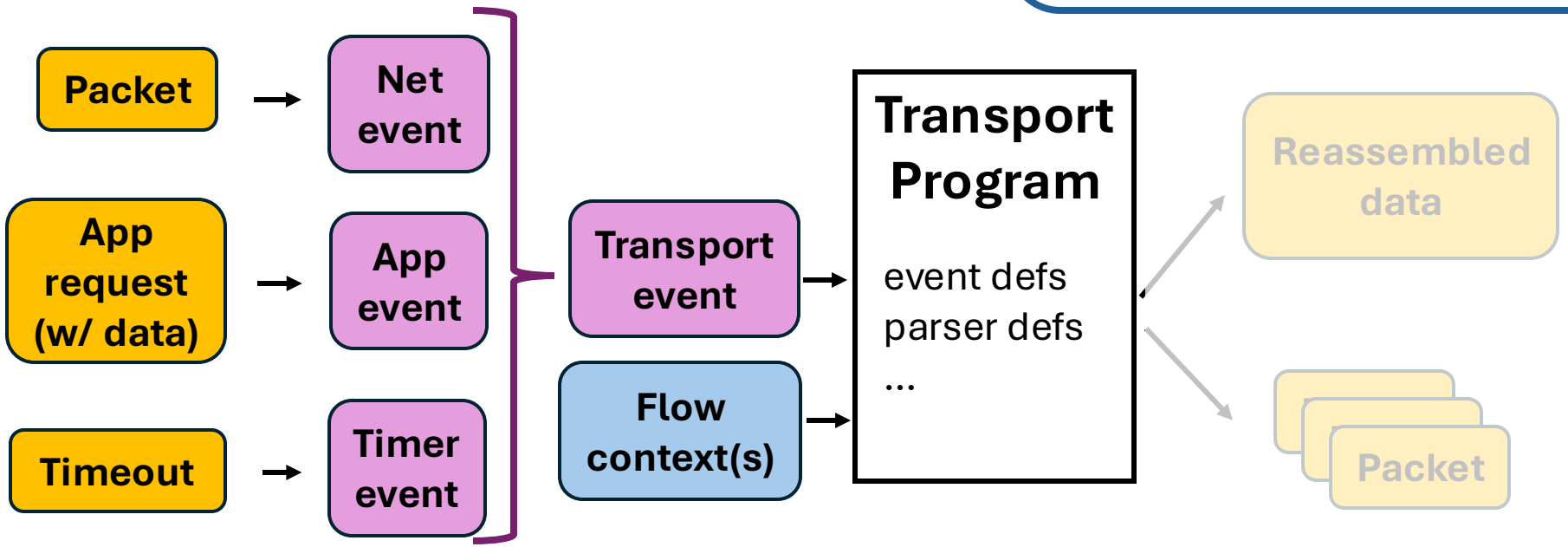
- Specifies how to create events from packets and app requests
- Syntax similar to other network languages

Transport events



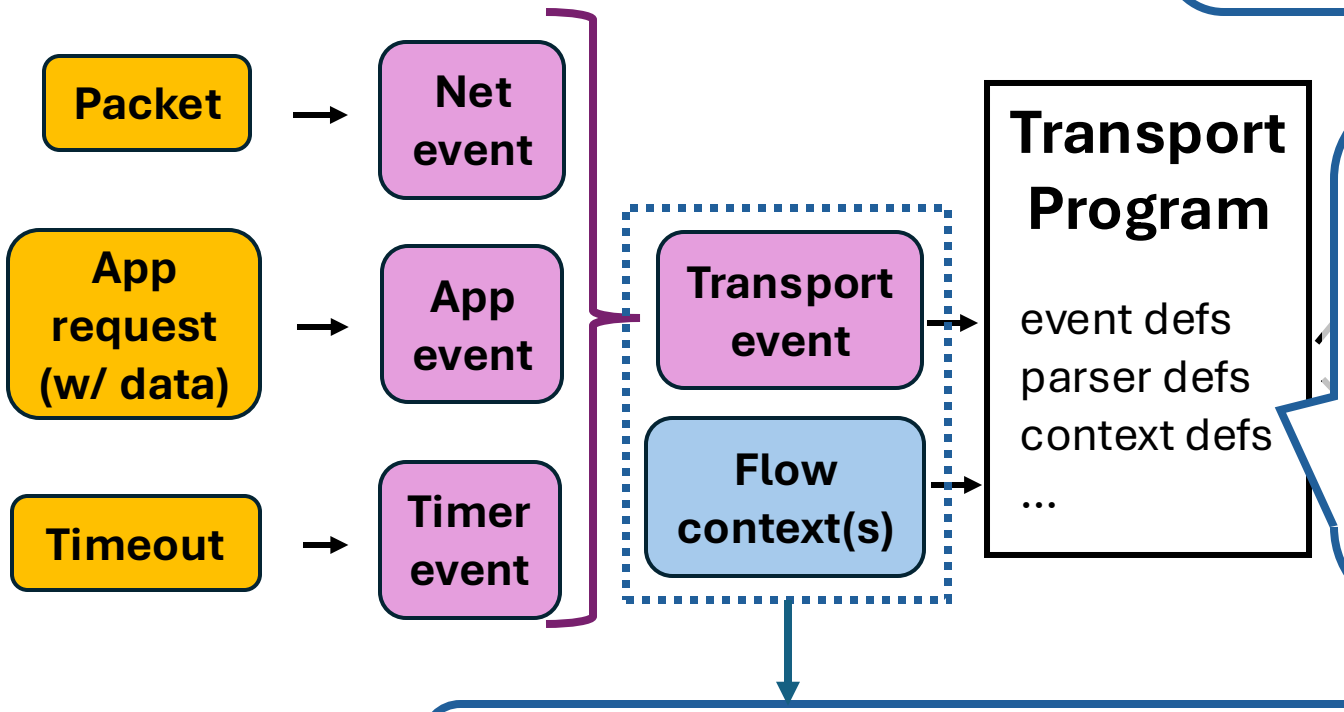
Flow contexts

Each flow has some **state (or context)** that is
- used in event processing
- **maintained across events**
- E.g., sliding window start and end in TCP



Flow contexts

Each flow has some **state (or context)** that is
- used in event processing
- **maintained across events**
- E.g., sliding window start and end in TCP

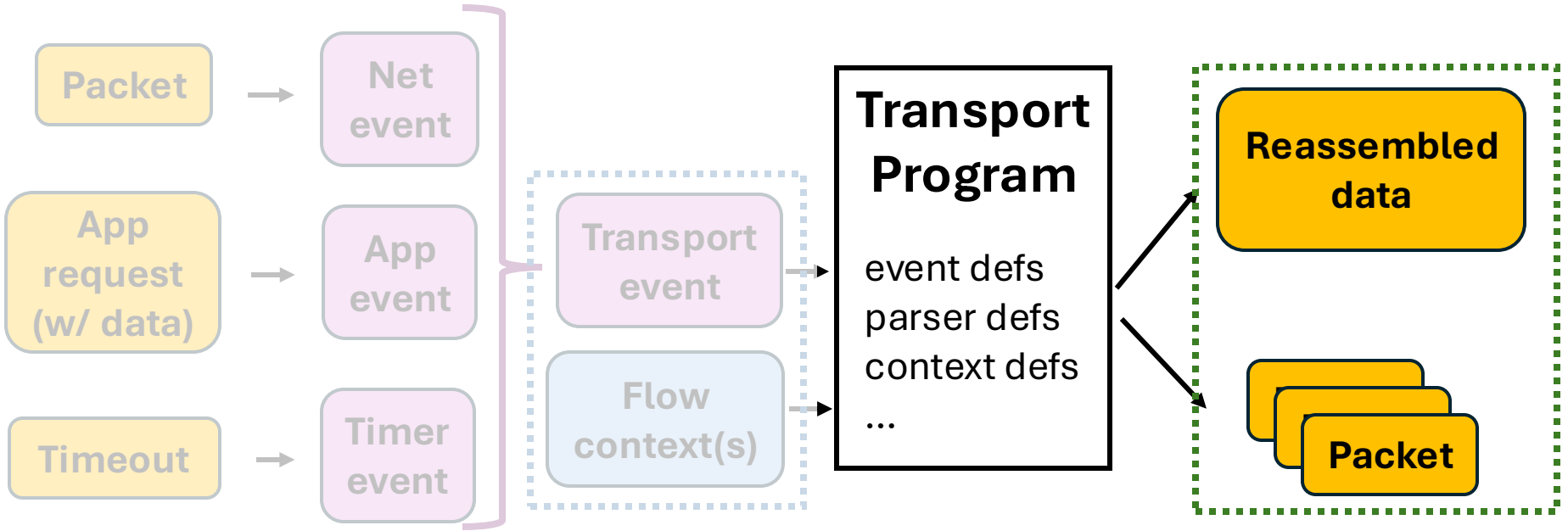


- Specifies what information to keep in the context.

```
context tcp_context {
  uint32 send_una;
  uint32 send_nxt;
  uint32 cwnd_size;
  ...
}
```

- Each event is associated with a specific flow
- Programs attach look-up keys to events during parsing

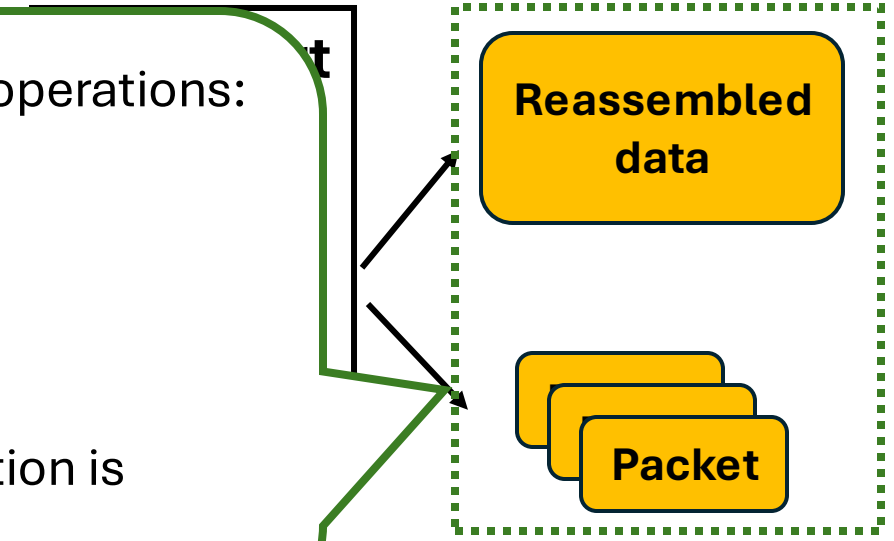
Output: ??


Packet
App request (w/ data)
Timeout
Net event
App event
Timer event
Transport event
Flow context(s)
Transport Program
event defs
parser defs
context defs
...
Reassembled data
Packet

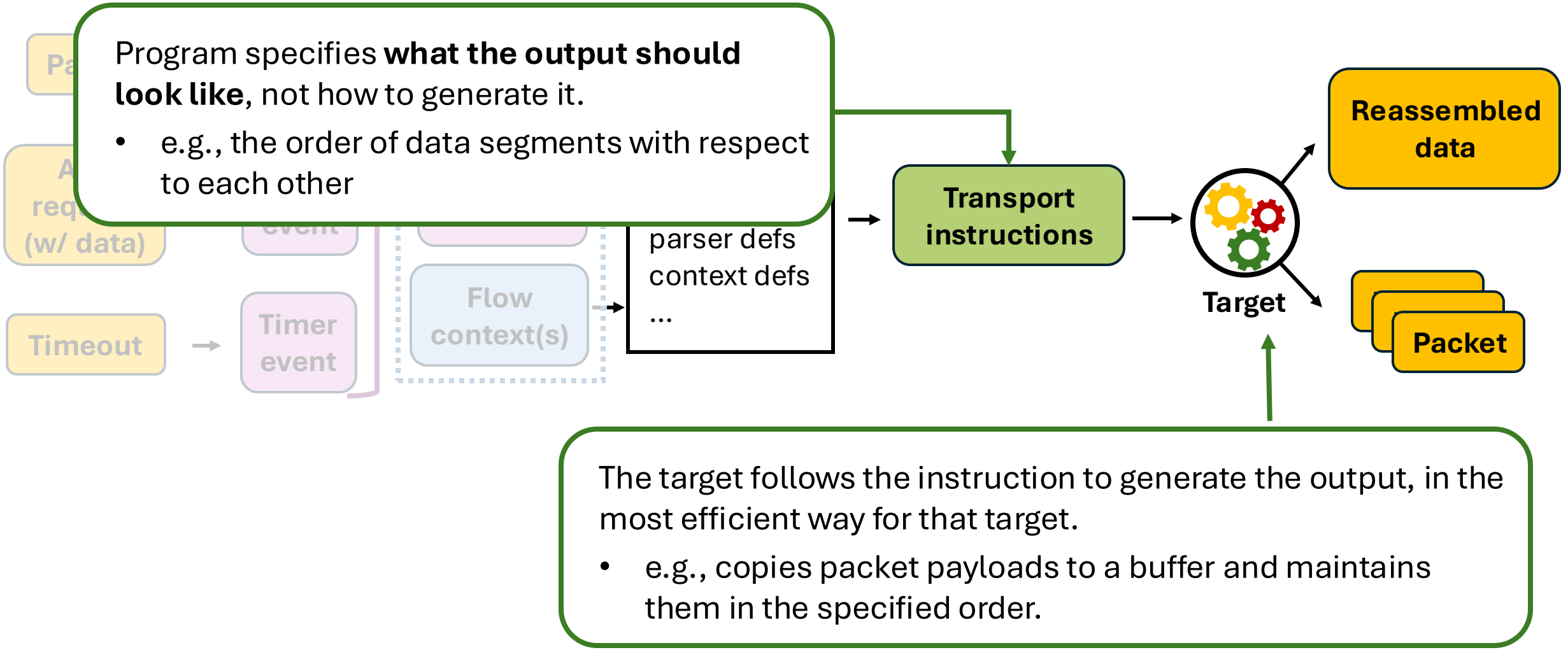
Output: ??

How do we **decouple protocol logic** for reassembly and packet generation from target-specific **implementation details**?

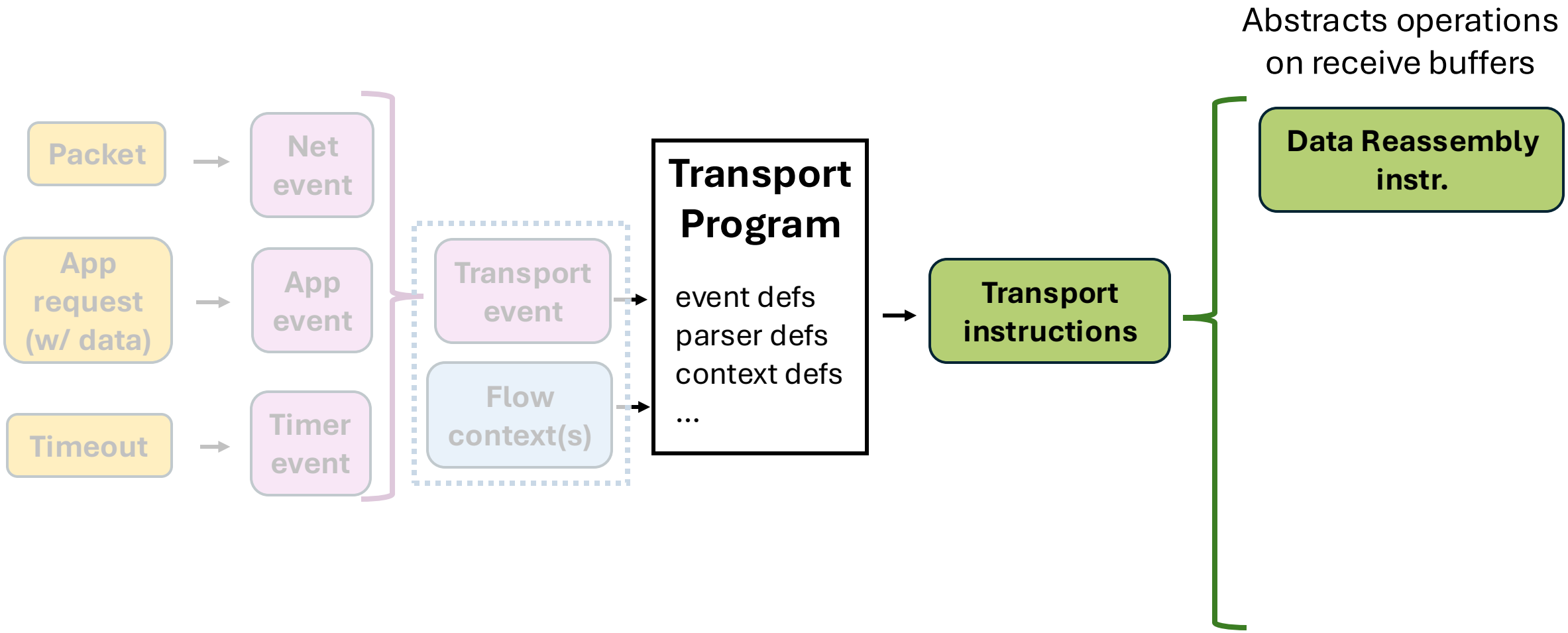
- Involves performance-sensitive operations:
  - Data movement
  - Buffer management
  - Packet pacing
  - …
- The most “optimal” implementation is **target-dependent**



Transport instructions



Transport instructions



Transport instructions – Data Reassembly

Transport instructions issued by the program

new_rx_ordered_data(uid, size[, addr])

- I expect to receive *size* bytes of consecutive data
  - *size* can be *INF* for byte streams
- The identifier for this "unit" is *uid*
- The data should eventually be available at *addr*

What the target should do

- Allocate memory accordingly
  - Dynamic allocation?
  - Pool of buffers?
  - Zero copy *(addr)?*
  - …
- Maintain a mapping between *uid* and the allocated space

Transport instructions – Data Reassembly

Transport instructions issued by the program

add_rx_data_seg(addr, len, uid, offset)

- I want *len* bytes starting from *addr* to be at index *offset* of the consecutive data unit *uid*
  - *addr* → where incoming packet's payload is stored

What the target should do

- Find the right "destination" memory locations based on *offset* and *uid*
- Copy data from *addr*

Transport instructions – Data Reassembly

Transport instructions issued by the program

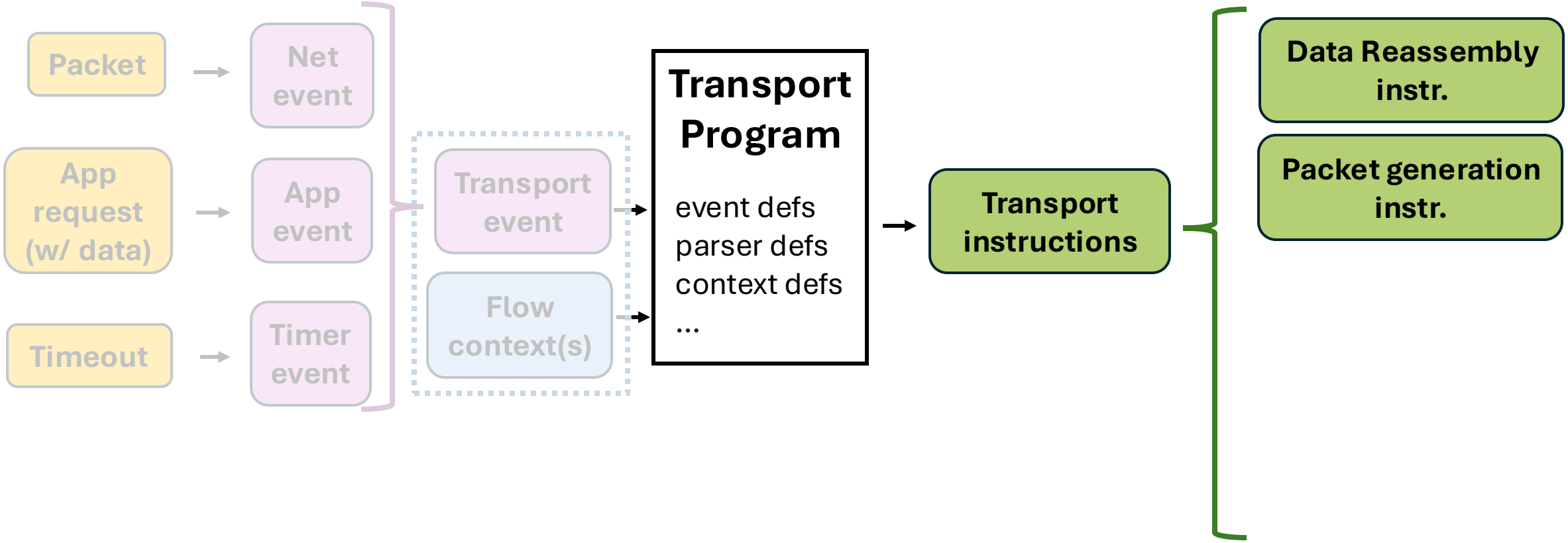
rx_flush_and_notify(uid, len, addr)

- I want *len* more bytes from *uid* to be made available to the application at *addr*
  - *addr* → user's buffer address

What the target should do

- Keep track of how far into *uid* has been "flushed" to the app
- Find the right "source" memory locations accordingly
- Move data to *addr*

Transport instructions



Transport instructions – Packet Generation

Transport instructions issued by the program

new_tx_ordered_data(uid, size[, addr])

add_tx_data_seg(addr, len, uid)

tx_flush_and_notify(uid, len)

- Similar to the “rx” counter-parts
- Abstracts operations on send buffers

What the target should do

- Allocate memory for *uid*
- Append app data to *uid*
- Remove data from *uid*
- …

Transport instructions – Packet Generation

Transport instructions issued by the program

pkt_gen(pkt_bp[, seg_rule_id, …])

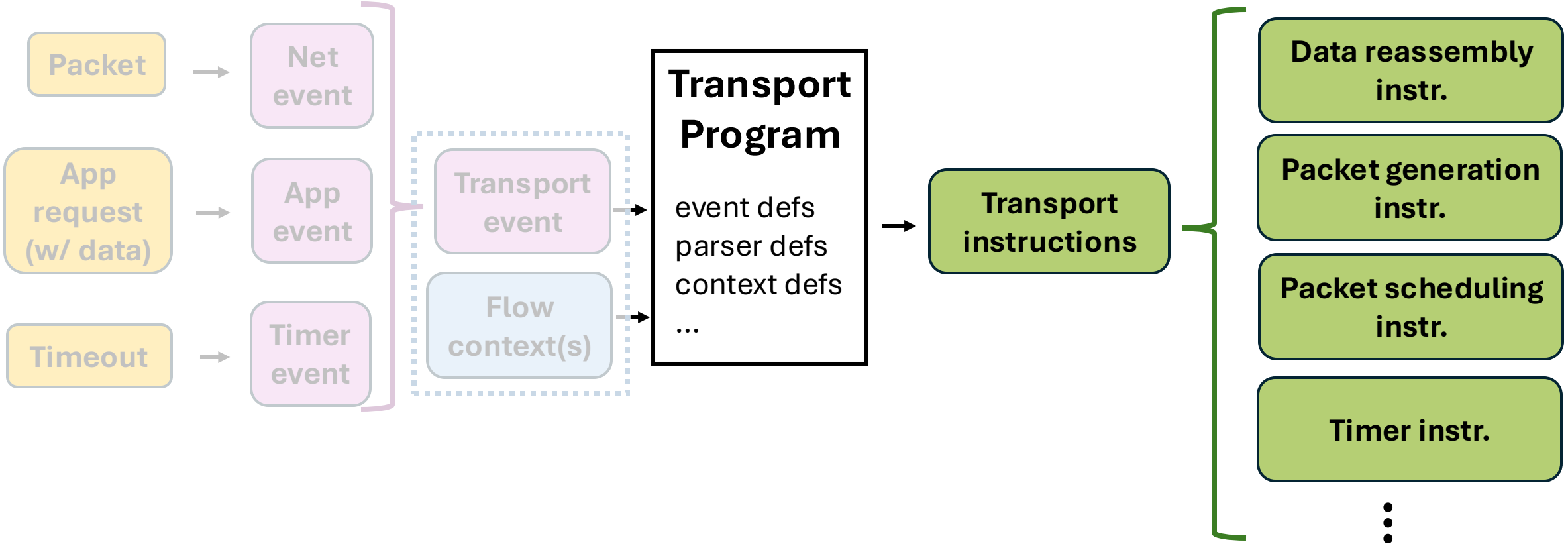
- I want packets looking like this *blueprint*
- blueprint:
  - header
  - data address and size for payload
- If data does not fit in one packet, segment it:
  - Update headers based on *seg_rule_id*

What the target should do

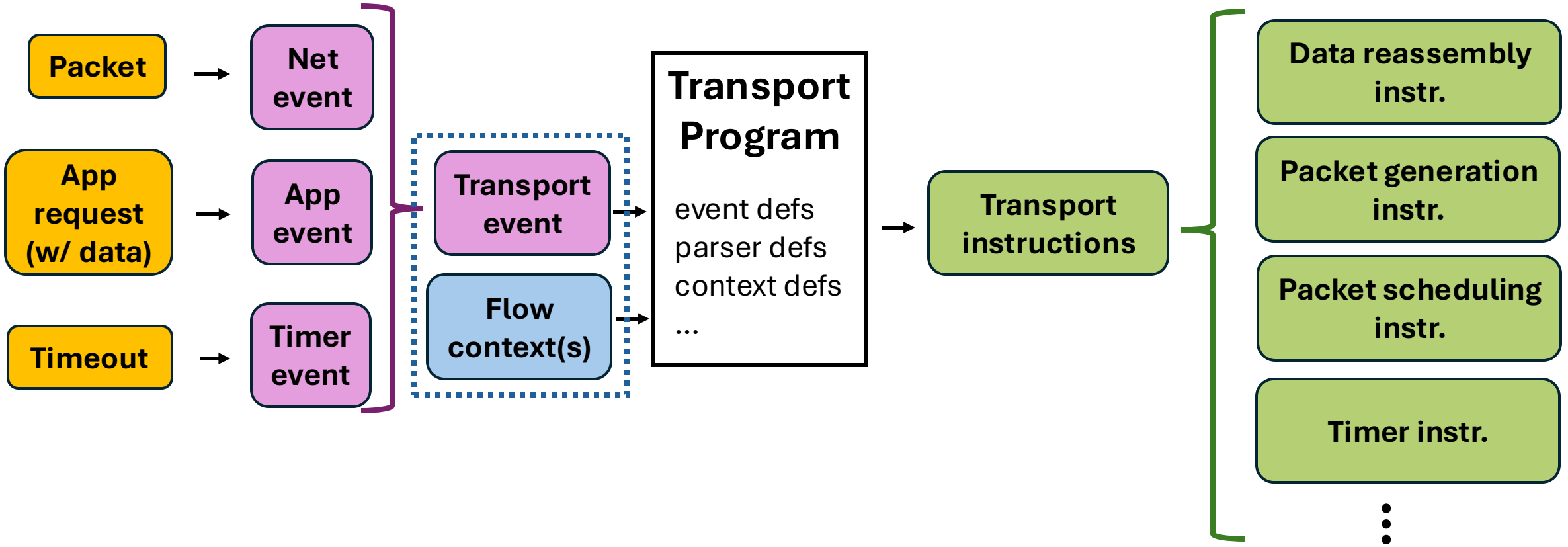
Generate the actual packets:

- Allocate packet memory
- Fill out headers
- Move data for payload
- …

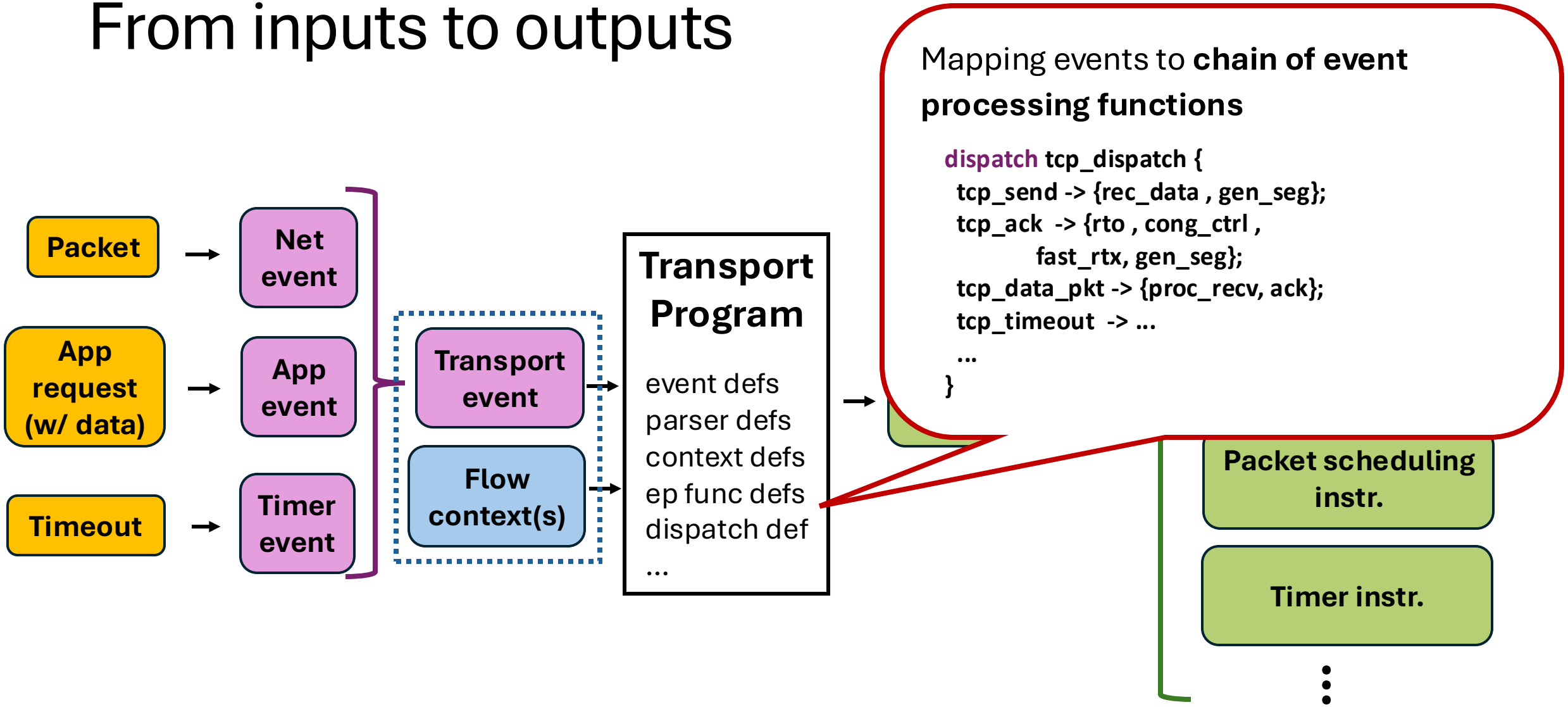
Transport instructions



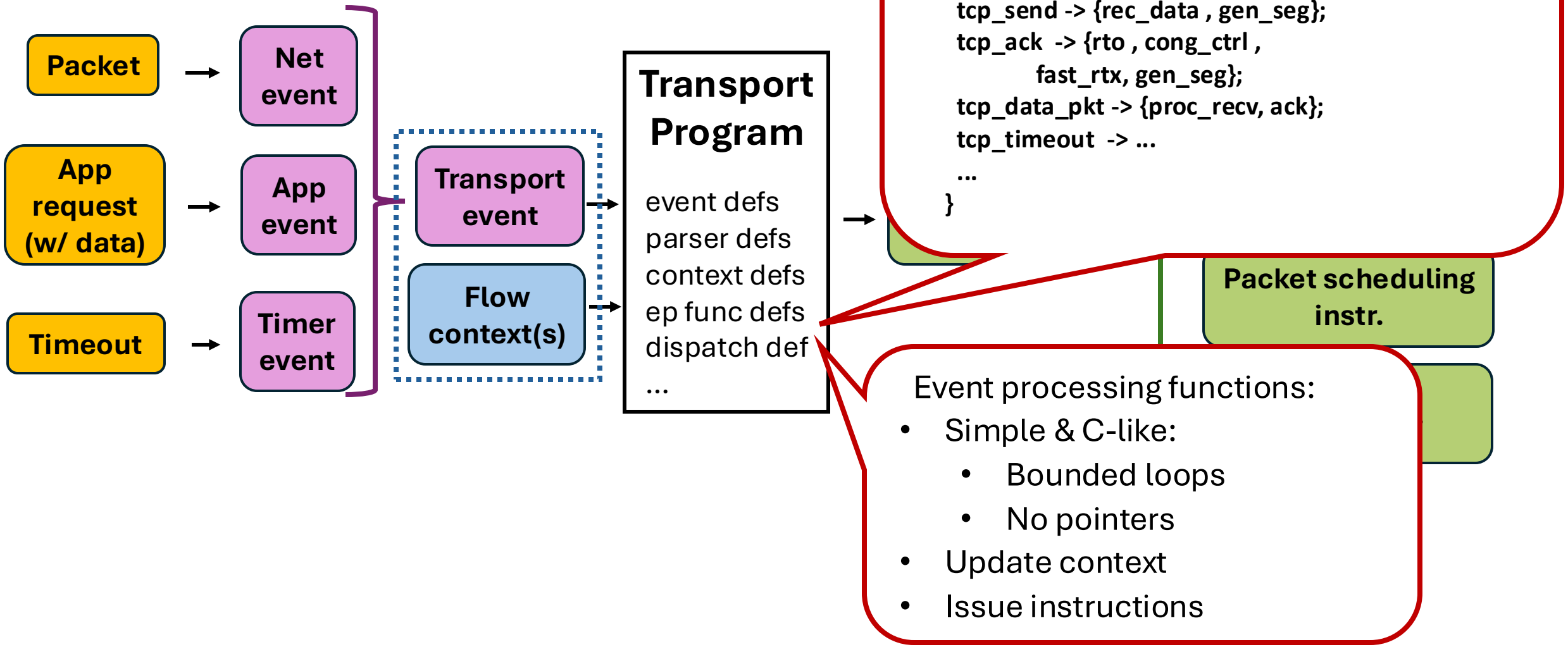
From inputs to outputs



From inputs to outputs



From inputs to outputs



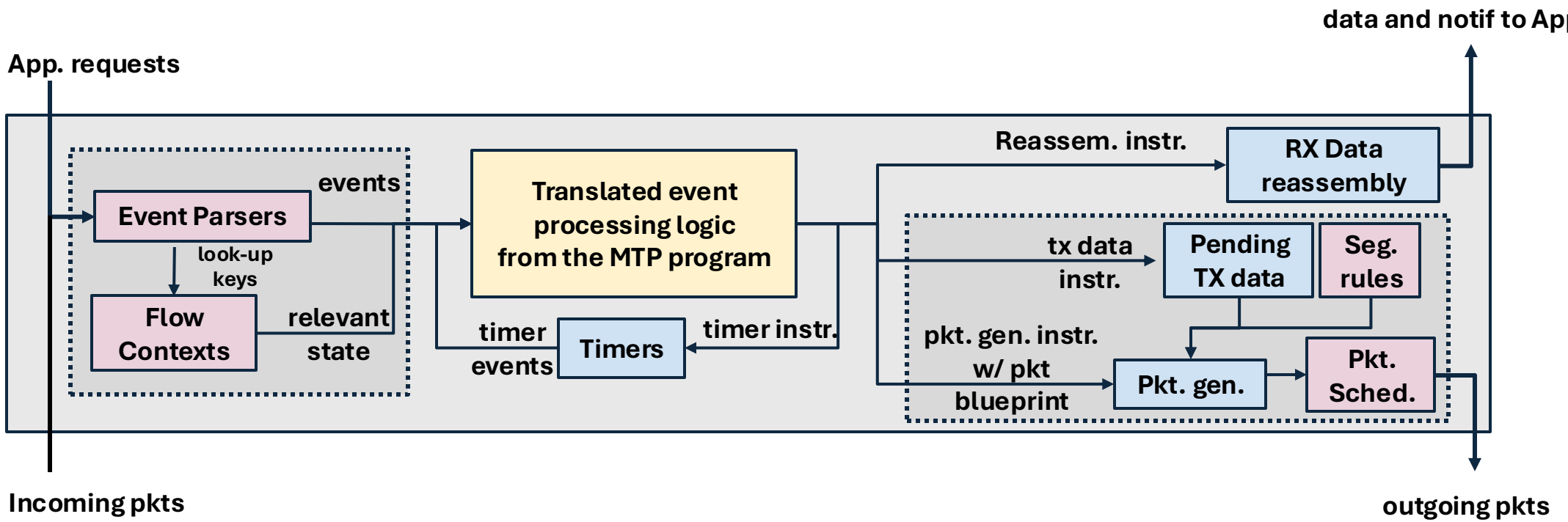
Modular Transport Programming (MTP)

MTP Program

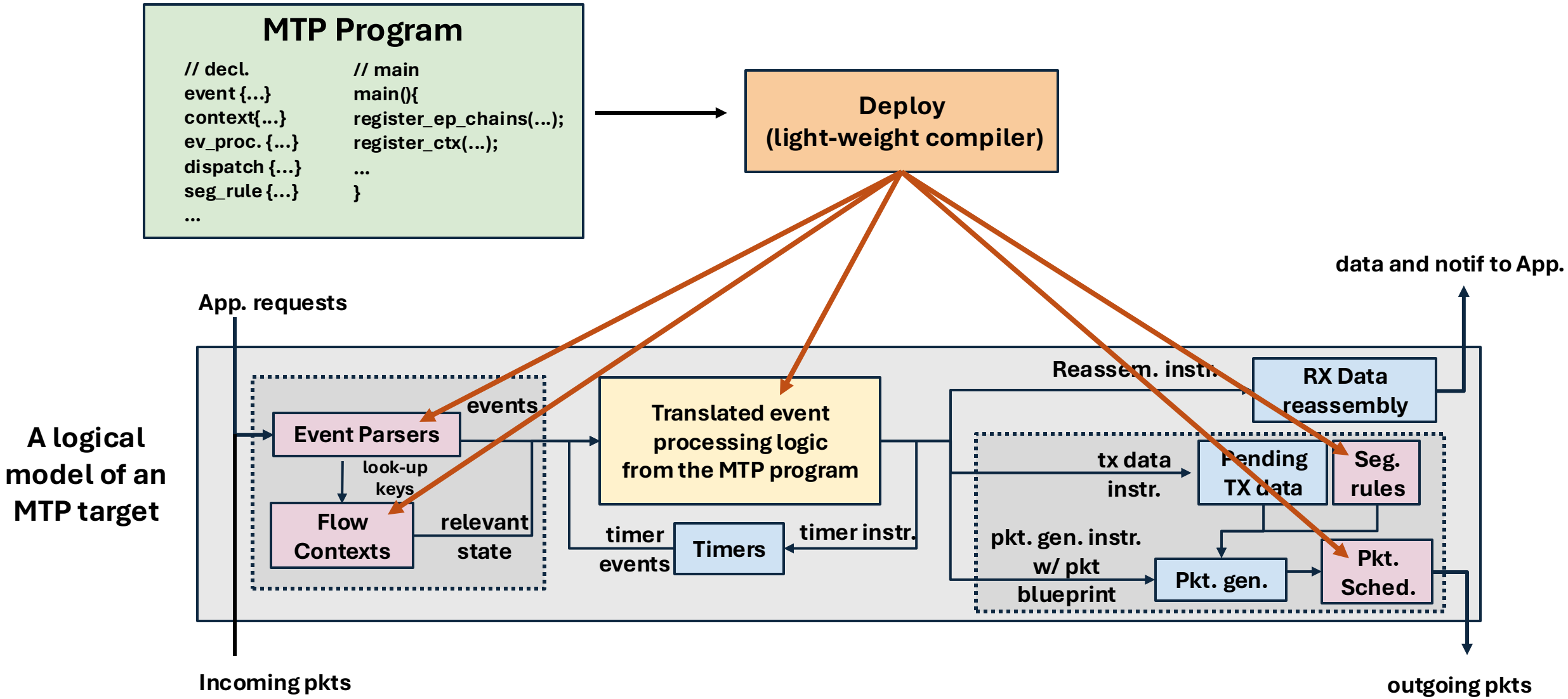
```
// decl.                 // main
event {...}              main(){
context{...}             register_ep_chains(...);
ev_proc. {...}           register_ctx(...);
dispatch {...}           ...
seg_rule {...}           }
...
```

→ **Deploy (light-weight compiler)**

A logical model of an MTP target



Modular Transport Programming (MTP)



Expressiveness

Ultra Ethernet Transport in MTP?
Short-term future work

- ✓ TCP
- ✓ QUIC-Lite

- Stream-based
  - Applications append data to byte streams to be sent
  - TCP: one per connection
  - QUIC-Lite: multiple parallel ones per connection
- Sender-side congestion control

- ✓ Homa
- ✓ NDP

- Message-based
  - Application message size is known (e.g., RPC)
- Receiver-driven

- ✓ RoCEv2

- Message-based
- Queue pairs as “connections”
- Designed for hardware

What about performance?

Observation:

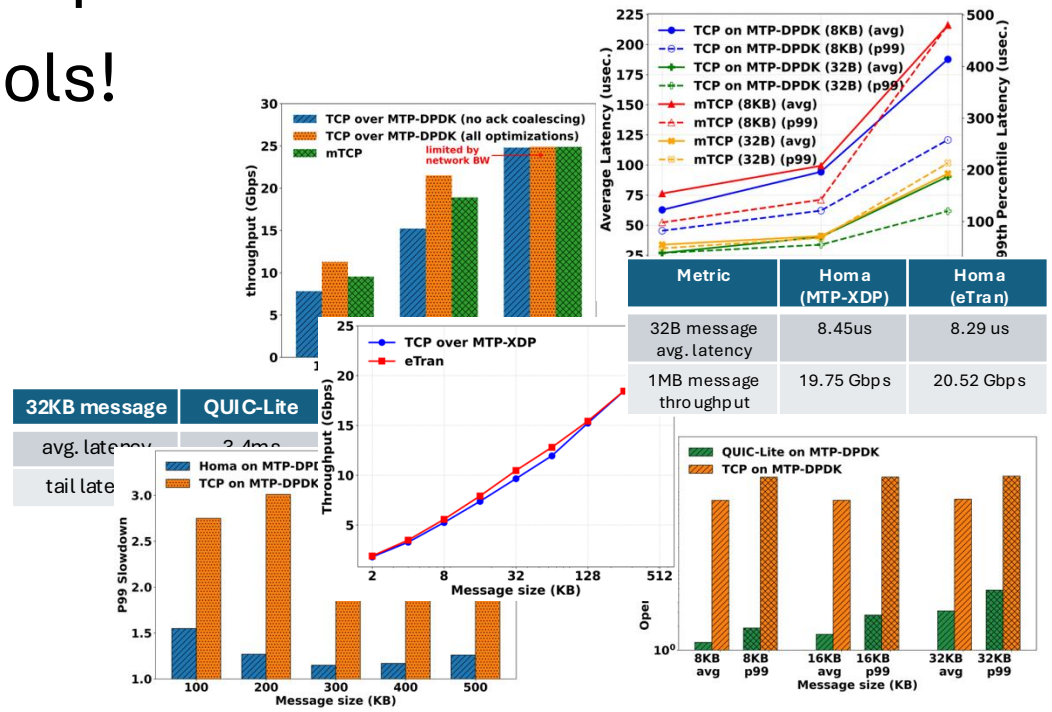
Existing protocol implementations already know how to do transport tasks *efficiently* in a specific execution environment

- e.g., buffer management, packet I/O, per-flow state tracking, …

We can “refactor” them to expose these tasks via MTP’s high-level unifying interface.

MTP-compliant targets offer comparable performance

- MTP-DPDK and MTP-XDP
  - Refactored an existing TCP implementations over DPDK/XDP
    - mTCP (NSDI'14) and eTran (NSDI'25)
  - To implement MTP's API
- TCP over MTP targets has comparable performance
- It is possible to swap in other protocols!
  - Homa and QUIC-Lite
- See paper for
  - Each target's implementation details
  - Experiment details
  - And plots!



Metric	Homa (MTP-XDP)	Homa (eTran)
32B message avg. latency	8.45us	8.29 us
1MB message throughput	19.75 Gbps	20.52 Gbps

32KB message	QUIC-Lite
avg. late	
tail late	

Takeaways from implementing MTP targets

- MTP's API is at the right level of abstraction
  - abstracts away enough details to be target-agnostic
  - implementable with already existing efficient mechanisms
- Different targets' impl. of transport tasks vary in non-trivial ways
  - Confirmed our decision to abstract them as instructions
- The heavy lifting is in implementing the instructions
  - Abstract away most of the complexity
- Translating the event chains can be done with a light-weight compiler

Reduction in development effort

MTP Programs
Target-independent
Written once

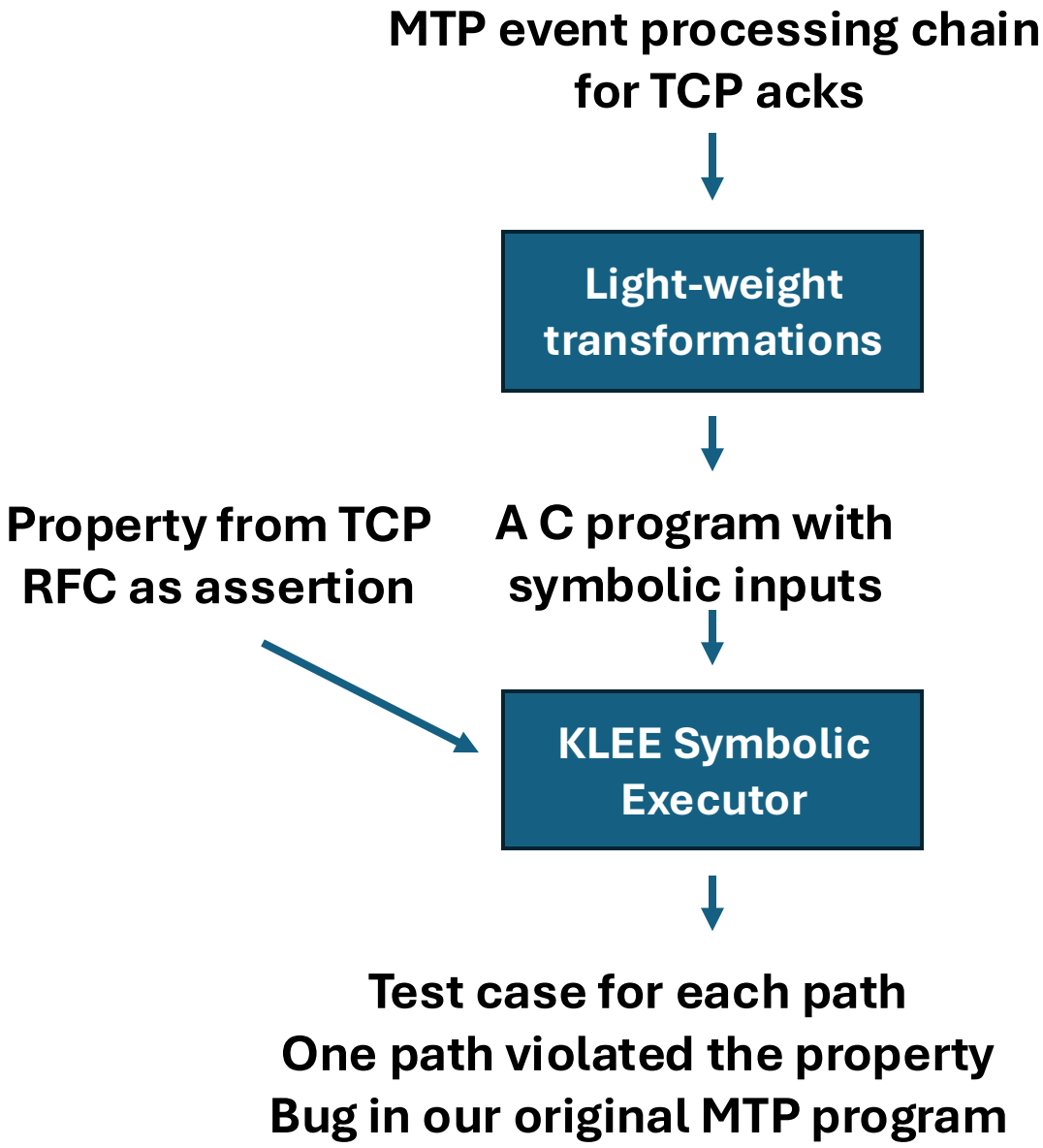
TCP	753 LoC
Homa	1205 LoC
QUIC-Lite	920 LoC

MTP-Compliant Targets
Protocol-independent
Developed once per target

MTP-DPDK	15,593 LoC
MTP-XDP	14,837 LoC

Automated analysis

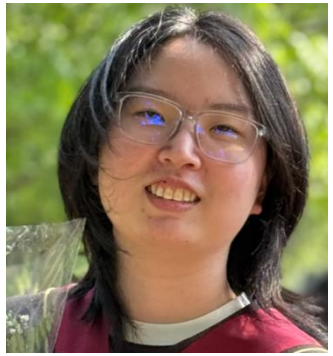
- MTP programs are amenable to automated analysis
  - Constrained C-like language
    - no pointers
    - Bounded loops
    - Constrained data structures
  - target-agnostic instructions hiding low-level details



A shout-out to the team!

Pedro Mizuno
UWaterloo

Kimiya Mohammadtaheri
UWaterloo



Linfan Qian
UWaterloo

Joshua Johnson
UWaterloo

Danny Akbarzadeh
UWaterloo

Chris Neely
AMD

Mario Baldi
NVIDIA

Nachiket Kapre
UWaterloo

Mina Tahmasbi Arashloo
UWaterloo

Summary and looking forward

- Transport protocols will continue to evolve
- Their execution environments will continue to evolve
  - Software: Kernel, Kernel-bypass, eBPF
  - Hardware accelerators
- This diversity calls for a language abstraction that is *high-level, target-agnostic, and protocol-independent* …
  - MTP takes a significant step in this direction.
- … that can unlock a myriad of benefits:
  - Seamlessly swapping in new protocols and add features on a target
  - Automated functional and performance verification
  - Automated testing
  - Write-once run-anywhere
  - ….