

# Programming Transport Layer with Galvatron

Ze Xia

Xi'an Jiaotong University  
Xi'an, China

Yihan Dang

Xi'an Jiaotong University  
Xi'an, China

Hao Li

Xi'an Jiaotong University  
Xi'an, China

## ABSTRACT

This paper introduces Galvatron, a domain-specific language designed to simplify transport layer programming. Galvatron obscures the underlying data structures and operations, and exposes parametric processing stages to the developer, thus significantly reducing coding effort. Our evaluation demonstrate Galvatron's ability to encapsulate complex semantics of 4 real-world transport protocols using merely 2% of the code compared to native implementations.

## CCS CONCEPTS

• Networks → Transport protocols; Programming interfaces.

### ACM Reference Format:

Ze Xia, Yihan Dang, and Hao Li. 2024. Programming Transport Layer with Galvatron. In *The 8th Asia-Pacific Workshop on Networking (APNet 2024)*, August 3–4, 2024, Sydney, Australia. ACM, New York, NY, USA, 3 pages. <https://doi.org/10.1145/3663408.3665807>

## 1 INTRODUCTION

TCP has dominated the transport layer for decades, and evolved into a colossal protocol defined by 34 RFC spanning thousands of pages. Despite its array of robust features, TCP still falls short of meeting all design goals in today's network. Therefore, application developers and network operators are left with two options: (1) meticulously tailor TCP to their needs or (2) implement entirely new transport protocols from scratch, both of which, however, require huge coding efforts. As a reference, the kernel module of Homa extension consists of ~10K LOC [1], and MsQuic uses ~54K LOC to implement QUIC at user-space [2]. This raises an intuitive question: *can we program the transport layer with minor coding efforts?*

The root cause of the huge coding effort is the rich set of features and tight coupling between them in transport layer protocols. Figure 1 shows the four typical modules in such a layer: (1) a receiver to handle incoming packets, (2) a sender for transmitting packets, (3) a collection of data maintaining connection states and (4) a suite of APIs available to application developers. The implementation of these modules is complex due to the intricate interconnections among them, as shown by the grey arrows. For example, the retransmit queue storing sent while not acknowledged data makes a lot of interaction with other modules, e.g., retrieving data from the send buffer, marking data inside as delivered or lost, bookkeeping for congestion control, etc. Our insight from this workflow is that while the purposes vary among protocols, the set of operations on connection states remains uniform and could be parameterized. To this end, we introduce Galvatron, a domain specific language (DSL)

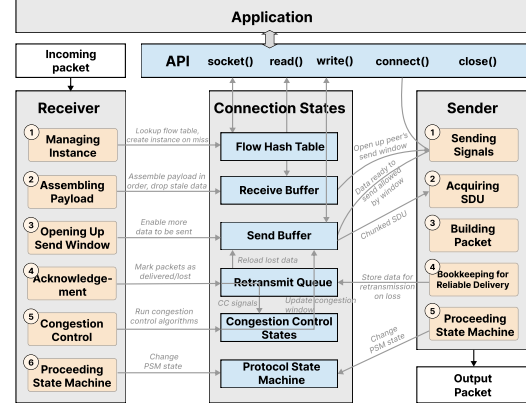


Figure 1: Workflow of a transport layer.

to program the transport layer with minimal effort. Galvatron only exposes the *processing stages* of the receiver and sender (yellow boxes in Figure 1), such that the operators focus on *what features are expected from the transport layer*, instead of *how to implement such features*. The backend compiler of Galvatron would translate the DSL program into native code implementing the underlying connection states and operations (blue boxes in Figure 1), which can be integrated into I/O runtimes such as DPDK, and run as a complete transport layer.

## 2 GALVATRON OVERVIEW

We use Figure 2 to overview how to write a simple SwitchML [8] transport layer with Galvatron, where basic reliable transmission is required but cannot afford TCP due to limitation of programmable switches. We walk through the code in a feature-oriented style.

**Packet format.** Line 4-9 defines the packet format with three Bit fields and a Payload field that fills the rest of the packet. These fields could be later accessed by receiver, e.g., `ml.layout.pld` (line 31), or filled by sender, e.g., `ml.layout(pld=...)` (line 49-54).

**Connection data.** Each connection hold a set of user-defined data (line 11-17), e.g., `ml.data` stores `pool_id` (line 14) which is used to manage on-switch memory in SwitchML.

**Protocol state machine.** Line 20 specifies a state machine for each connection with two states: the *connection established* state EST, and the *transfer done* state CLOSED. If an incoming packet holds the last piece of payload, i.e., `ml.data.roffset==ml.data.fin_offset`, then the state machine should proceed from EST to CLOSED (line 34).

**Connection Identification.** Line 22 tells Galvatron to match an existing connection according to `ml.layout.slot_id` of an incoming packet, or create a new connection if not found.

**Buffer Management.** The receiving buffer is filled with `ml.layout.pld` at the position indicated by `Offset` (line 31). The sending buffer is consumed to yield a piece of payload no longer than `ml.data.frag_siz` (line 40), which is filled into outgoing packets (line 51).

```

1  # Declare the ml layer
2  ml = TransportLayer()
3  # SwitchML packet format
4  class ml_pkt(Struct):
5      seq_num = Bit(32)
6      pool_id = Bit(1)
7      slot_id = Bit(15)
8      pld = Payload()
9      ml.layout = ml_pkt
10 # Connection data fields
11 class ml_data(Struct):
12     frag_siz = Bit(32, init=248)
13     slot_id = Bit(16, init=0)
14     pool_id = Bit(1, init=0)
15     fin_offset = Bit(32, init=-1)
16     roffset = Bit(32)
17     ml.data = ml_data
18
19 # Protocol State Machine
20 ml.psm = PSM([EST, CLOSED])
21
22 # Identify a SwitchML instance
23 ml.rcv.flowid = [ml.layout.slot_id]
24 ml.rcv.prep = Assign(ml.data.ROffset,
25                     ml.layout.seq_num+ml.layout.pld.Len())
26 # Acknowledge until roffset
27 ml.rcv.ack = AckUntil(ml.data.ROffset)
28 # Update sending window
29 ml.rcv.send_wnd = Window(-INF,
30                         ml.data.ROffset + ml.data.frag_siz)
31 # Input data, drop if data is stale
32 ml.rcv.payload = RecvPayload(
33     ml.layout.pld) @ Offset(ml.layout.seq_num)
34 # PSM transitions as in Rubik
35 ml.rcv.psm.teardown = (EST >> CLOSED) \
36     + Pred(ml.data.ROffset == ml.data.fin_offset)
37
38 # Signals that triggers sending
39 ml.snd.signals = [ml.send_ready]
40 # Prepare data for sending
41 ml.snd.payload = SendPayload(ml.data.frag_siz)
42
43 # Maintain pool id
44 ml.snd.prep = If(ml.snd.payload.is_rexmit == 0) \
45     >> Assign(ml.data.pool_id, 1-ml.data.pool_id)
46
47 # Record final sequence number
48 ml.snd.prep += If(ml.close & ml.send_empty) >> Assign(ml.
49     data.fin_offset, ml.snd.payload.offset + ml.snd.
50     payload.len)
51
52 # Fill in fields of the packet.
53 ml.snd.packet = ml.layout(
54     seq_num=ml.snd.payload.offset,
55     pool_id=ml.data.pool_id,
56     slot_id=ml.data.slot_id,
57     pld=PayloadFrom(ml.snd.payload)
58 )
59
60 # Store the Payload for reliable delivery
61 ml.snd.reliable = ReliableInfo(ml.snd.payload) @ Index(
62     off=ml.snd.payload.offset, siz=ml.snd.payload.len)

```

Figure 2: SwitchML transport layer in Galvatron: layer declaration (green), receiver (blue) and sender (red).

**Flow and congestion control.** SwitchML keeps the flow control window as one packet on purpose to ease on-switch memory management. Line 28-29 achieves this by advancing the window endpoint to `data.ROffset+data.frag_siz` upon incoming packet, allowing one more packet to be sent. This window take effect by suppressing the `ml.send_ready` event when no data in window could be sent. This event is monitored in Line 38, among other *signals* that triggers the sender side to build and send a packet. Galvatron relies on CCP [7] to program congestion control logic, while this is not needed for SwitchML due to its small, fixed flow control window.

**Reliable Delivery.** The payload is stored as `ReliableInfo` at index `[payload.offset, payload.offset+payload.len]` (line 56), allowing it to be retransmitted upon lost. The stored `ReliableInfo` can be acknowledged by an incoming packet (line 26), where all `ReliableInfo` with index less than `ml.data.ROffset` is acknowledged and thus freed. Galvatron maintains a RTT-based timer for loss detection. Upon timeout, the stored payload is detected as lost and becomes in-window data, thus triggering the `ml.send_ready` event and invoking the sender (line 38). `SendPayload` would then re-obtain this payload(line 40), causing it to be sent again.

**Application Interface.** Galvatron would compile the DSL program into native C code, providing a set of `epoll`-like APIs, which could be linked with the application.

**Summary.** In sum, we implemented SwitchML’s end-host logic in 56 LOC, without additional code to define API or runtime environments. As a comparison, original SwitchML implementation in DPDK took about 700 LOC to implement the transport layer, and ~1000 extra LOC to get a complete runtime environment, requiring 30× more code to write.

### 3 PRELIMINARY EVALUATION

We evaluate Galvatron with 4 transport protocols: TCP, QUIC, TDTCP, and SwitchML. Table 1 shows that developers only need to write a mere hundred of LOC with Galvatron, a ~47× reduction compared with native approaches. This LOC is also greatly reduced compared with the generated one, thanks to the concise interface of DSL. An echo application built upon the generated TCP stack can run at the speed of 1.8Gbps with a single core.

### 4 SUMMARY AND FUTURE WORK

We propose Galvatron, a DSL addressing the need of programming the end-host transport layer, which could reduce the coding effort

Table 1: LOC comparison of DSL code, Galvatron-generated code and native implementations.

Protocol	Lay.	Rcv.	Snd.	Total	Gen.	Native
TCP	211	143	148	502	11063	16481 [6]
QUIC	390	245	121	766	22490	38436 [2]
TDTCP	37	33	36	106	12295	11798 [5]
SwitchML	25	20	26	71	2884	1915 [8]

of complex transport layers into hundreds of LOC. That said, there exists many future work, as outlined below.

**Customizing Underlying Building Blocks.** Currently, Galvatron only provides a fixed set of underlying operations (blue boxes in Figure 1), e.g., RTT-based timeout and fast-retransmit for loss detection. Programming those logic is complementary to Galvatron, we could rely this work to external modules, e.g., Tonic [3].

**Multiple runtime environments.** Galvatron is independent of low-level implementation, thus the same DSL code could run in multiple runtimes, e.g., as a Linux kernel module, a DPDK application or a hardware design [4], i.e., *write once, run anywhere*.

### ACKNOWLEDGMENTS

This paper is supported by the National Key Research and Development Program of China (No. 2022YFB2901403) and NSFC (No. 62172323). Hao Li is the corresponding author.

### REFERENCES

- [1] 2022. PlatformLab/Homa. <https://github.com/PlatformLab/Homa>.
- [2] 2024. microsoft/msquic. <https://github.com/microsoft/msquic>.
- [3] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghebadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *USENIX NSDI*.
- [4] Serhat Arslan, Stephen Ibanez, Alex Mallery, Changhoon Kim, and Nick McKeown. 2021. NanoTransport: A Low-Latency, Programmable Transport Layer for NICs. In *ACM SOSR*.
- [5] Shawn Shuoshuo Chen, Weiyang Wang, Christopher Canel, Srinivasan Seshan, Alex C. Snoeren, and Peter Steenkiste. 2022. Time-division TCP for reconfigurable data center networks. In *ACM SIGCOMM 2022*.
- [6] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *USENIX NSDI*.
- [7] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. 2018. Restructuring endpoint congestion control. In *ACM SIGCOMM*.
- [8] Amedeo Sapia, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *USENIX NSDI*.