Declarative Transport

A Customizable Transport Service for the Future Internet

Karim Mattar† Ibrahim Matta† John Day‡ Vatche Ishakian† Gonca Gursun†

†College of Arts & Science ‡Metropolitan College
Department of Computer Science, Boston University

{kmattar, matta, day, visahak, goncag}@bu.edu

ABSTRACT

We argue that in a clean-slate architecture, transport state is an integral part of the network state, which includes information for routing, monitoring, resource allocation, *etc.* Given the myriad of transport policies needed to support advanced functions such as in-network caching, in-network fair allocation, and proxying, these policies should be made programmable. We outline how flexible and generic transport policies can be specified in a declarative language to realize a transport service where distributed transport state is shared and manipulated using recursive queries.

1. INTRODUCTION

As the Internet continues to grow, network technologies (with new quality-of-service properties) and applications (with new requirements) continue to emerge. This has led to several transport paradigms and even more custom point-solutions that need to be continually adapted, but little in terms of a unified theory. There are efforts to standardize transport solutions and develop extensible platforms (see the work by Bridges *et al.* [4] and references therein) but none of which are generic across all environments[1]. In this position paper, we start by answering an important question:

What aspects of designing and programming transport functions make their generalization across different environments inherently difficult?

We believe that the introduction of congestion control mechanisms in TCP, after the congestion collapse in 1986, is a kludge. Since then TCP has become the de-facto transport protocol and transport functions (state) are executed (maintained) primarily at the end-hosts. End-hosts, however, may be the farthest points from the congestion event and the least suited to react. This does not mean that end-hosts should not perform end-to-end checks but makes one question whether they should be the only ones. In addition, the end-to-end principle [18] views any in-network modification (*e.g.,* router support or proxy) that breaks the end-to-end semantics as a "hack". This is the case because there is no support for the "ends" to explicitly coordinate with "middleboxes". This view of building networks is an artifact of the Internet's architecture. If an alternate organizing structure is considered, the solution concepts become fundamentally different.

[1]By environment we are referring to the underlying network technology and the class of applications being supported.

We are proponents of Robert Metcalf's view that networking is only inter-process communication (IPC). IPC in an operating system requires certain functions (*e.g.,* locating processes, checking permissions, passing information, scheduling, managing memory) to allow two processes to communicate. Similarly, two applications on different end-hosts should communicate by utilizing the services of a distributed IPC facility (DIF). A DIF is an organizing structure—what we generally refer to as a "layer". The functions that constitute this layer, however, are fundamentally different. A DIF is a collection of IPC processes (nodes) that communicate and share state information. *Each IPC process executes routing, transport and management functions.* The goal of a DIF (private network or overlay) is to provide a distributed service that allows application processes to communicate.

Two novel aspects of a DIF is that it *repeats* and is *relative.* As shown in Figure 1, two IPC processes A and B in an N-level DIF communicate by utilizing the services of an (N-1)-level DIF. Thus, IPC processes are the application processes requesting service from the lower-layer.

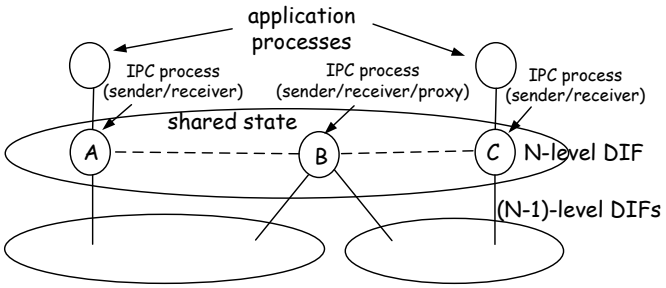

Figure 1: An N-level DIF consisting of IPC processes maintaining shared state and communicating via the (N-1)-level DIFs.

Our IPC-based architecture can be found in [11]. Here we only highlight two key aspects of this architecture that have a fundamental impact on how transport functions should be designed and programmed, namely:

- *The "ends" are not only end-hosts. Any two communicating nodes within a private network are considered the "ends" of communication over the underlying network that connects them.*

- *Transport is a service provided by all nodes in the network to support applications. Transport state is distributed. It can be collected, stored, manipulated and queried by all nodes.*

Our Contribution:

We show that our novel perspective on transport as a *service that manipulates distributed transport state* makes it possible to program flexible and generic transport functions across nodes in a network. Instead of programming end-hosts, one would program the transport functions executed by *each* IPC process, as well as how it interacts (*i.e.,* shares state) with other IPC processes within the same DIF. Programming transport as a distributed service allows us to:

1. Leverage declarative networking systems, that have been extensively used to implement routing protocols [14, 16], overlays [15] and applications [20], to specify the transport functions of an IPC process.

2. Query resource monitoring information made available by other protocols and applications that have already been specified declaratively.

Transport state may include packet sequence numbers sent, received, acknowledged, transmission rates used, as well as other monitoring information such as delay or bandwidth estimates. Since the size of a DIF is a design decision, transport state can be made quite manageable. Also, the *relative* nature of DIFs has a profound impact on the manageability of transport state. Generally, transport state is associated with a "flow" (*i.e.,* transport connection). In a relative architecture, what constitutes a flow is different at every layer. More specifically, many N-level flows may be aggregated into a few (N-1)-level flows. For example, a core network constituting a low-level DIF may maintain transport state for only a few *aggregate flows*.

Using a few examples, we show that by eliminating the notion of "end-hosts", transport functions such as those typically done by proxies (*e.g.,* snoop [2], ack-regulator [6]), in-network support mechanisms (*e.g.,* jtp [17], xcp [12]), or even multicast-specific mechanisms (*e.g.,* stair [5]) become significantly easier to realize. We also show that transport state is naturally represented as relations that can be manipulated and shared by executing recursive queries over the network.

Traditionally, transport constitutes data-intensive functions performed by end-hosts. Such functions may not be well-handled by a declarative language [20]. We show, however, that transport decomposes naturally into data transfer, control and management functions. This separation of concerns allows for an efficient implementation of data-intensive functions and a declarative manipulation of transport control and management information.

2. DECOMPOSING TRANSPORT

In our previous work [11] we showed that the functions executed by an IPC process can be broadly classified as transfer, control and management functions. These functions operate at different timescales as shown in Figure 2. Generally, control and management functions operate over a longer

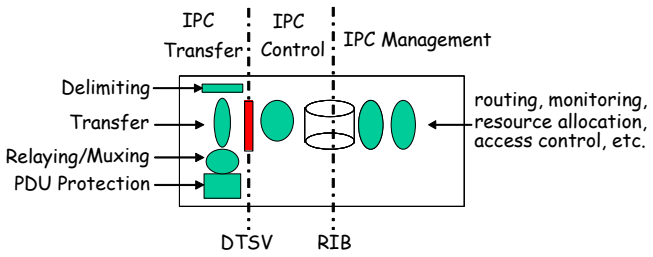

Figure 2: Functions executed by an IPC process over different timescales.

timescale. Here we outline how this is also the case for transport functions.

The two end-points of a transport connection maintain shared state by exchanging protocol data units (PDUs). A PDU consists of the user's data and the protocol control information (PCI)[2]. State information is either passed explicitly in the PCI (*e.g.,* available buffer space) or inferred from the exchange of PDUs over time (*e.g.,* round trip time). Analyzing the PCI in traditional transport solutions, one quickly realizes that there are two types of information:

1. Information that must be associated with the user's data (*e.g.,* checksums, sequence numbers) and must be transmitted with the data—the transfer PDU.

2. Information that does not have to be associated with the user's data (*e.g.,* bandwidth estimates) and can be transmitted in a separate PDU—the control PDU.

We refer to the mechanisms that produce these PDUs as tightly-coupled and loosely-coupled mechanisms, respectively. The degree of "coupling" is with respect to the user's data. This leads to a natural decoupling of transport into the data transfer protocol (DTP) and the data transfer control protocol (DTCP). DTP and DTCP share information via the data transfer state vector (DTSV). Our view is consistent with Clark *et al.* [9] who argued for the general separation of data manipulation and control functions.

It is important to note that this decoupling of transport is a fundamental property that is independent of any particular implementation. This decoupling, however, makes transport functions more amenable to a declarative specification where only transport control and management information need to be manipulated declaratively. Also, in principle the decomposition could in some instances be logical where information generated by DTCP is "piggybacked" onto DTP messages. Similarly, PDUs generated by different DTCP mechanisms could be automatically combined together into a single control PDU.

2.1 Data Transfer Protocol (DTP)

Every flow (*i.e.,* transport connection) must have a DTP instance associated with it. Service data units (SDUs) are enqueued by the application[3]. DTP is responsible for delimiting, fragmenting / concatenating SDUs to create trans-

[2]We use the term PCI as opposed to the traditional "header" to make clear that "information" is what the protocol understands as opposed to the "data" which it does not.
[3]An SDU is the unit of data handed by the N-level DIF to the (N-1)-level DIF.

fer PDUs whose size is less than the maximum transmission unit, as well as appending sequence numbers, addresses and checksum information. DTP is also responsible for reassembling / separating PDUs to recreate the original SDUs before handing them to the application. DTP only implements mechanisms that are tightly-coupled with the user's data over a short timescale (*i.e.,* packet-level) and generates a single PDU type—the transfer PDU. DTP's implementation could be made very efficient. Note that DTP represents a UDP-like transport connection.

2.2 Data Transfer Control Protocol (DTCP)

DTCP consists of all the loosely-coupled mechanisms that execute concurrently and are independent from the user's data. Each mechanism generates its own control PDU over a longer timescale (*i.e.,* flow-level). The existence of a DTCP instance depends on whether the supported flow requires any of the control mechanisms to be activated. Sample control mechanisms include: error, acknowledgment, retransmission, flow and rate control. While all loosely-coupled control mechanisms execute independently, in some instances they may affect the operation of DTP. For example, the policy instantiated in DTCP's error control mechanism may instruct DTP's PDU protection mechanism to use a particular forward error correction scheme.

2.3 Transport Management

Transport management provides support for the required performance monitoring applications. Performance monitoring can be done either passively (by observing transfer PDUs) or actively (by sending probes). All network performance monitoring information, including those collected by other protocols, is stored in a resource information base (RIB)[4]. Any information in the RIB can be queried by the transport control mechanisms and disseminated to other nodes periodically.

We do not require any explicit state management mechanisms (*e.g.,* TCP's handshaking mechanism) as Watson *et al.* [22] proved that timers are necessary and sufficient for maintaining transport state. We therefore rely exclusively on timers rendering explicit state management mechanisms unnecessary.

3. P2 SYSTEM

We use the declarative networking system P2 [1] to specify transport functions. In P2 users specify network protocols in NDlog, a declarative language based on extensions to Datalog. These specifications are then compiled into a dataflow graph where each declarative rule is converted to a strand of elements implementing the required operations to evaluate the rule. Rules query or update relations and trigger events to implement the desired logic. Tuples, representing PDUs and events, are sent and received over the network. Figure 3 outlines the dataflow graph that when executed results in the implementation of the specified protocol.

P2 does provide transport functionality in the Network-In and Network-Out modules. These modules implement functions for sending and receiving tuples, reliable transmission, and rate control. The dataflow architecture adopted by P2 also allows these modules to be re-ordered and configured dynamically [10]. P2's transport modules, however, only perform end-to-end transport functions. Tuples are retransmitted and acknowledgments are sent but only between a pair of communicating nodes. We, on the other hand, are interested in programming a *transport service across nodes in the network*. Nodes manipulate distributed transport state and can execute transport functions to act as relays, caches, proxies, *etc.*

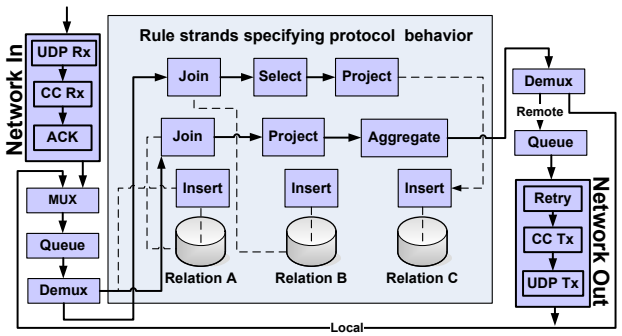

Figure 3: Dataflow in P2

In NDlog, a rule has the form `rulename <head>:-<body>`. The body consists of predicates and the head is triggered only when all these predicates evaluate to `true`. All predicates are relations which can either be hard-state, soft-state or event relations. Hard-state and soft-state relations are materialized relations containing tuples that have infinite and finite lifetimes, respectively. Event relations, on the other hand, are triggers. Each generated tuple is stored at the address associated with the *location specifier*, @. If the address is remote, the tuple is sent over the network.

We consider three sample declarative rules. We denote event relations by `eEventName` and materialized relations by `relationName`. The body of rule r1 is triggered when the event tuple is triggered (*i.e.,* exists and evaluates to `true`). Tuples are then *selected* from `table1` such that the values under the common fields `I`, `A` and `B` match those in `eEvent1`. Each matching tuple causes `eHead1` to be triggered.

```
r1 eHead1(@I,A,B) :- eEvent1(@I,A,B), table1(@I,A,B).
```

The body of rule r2 is triggered when either `table1` or `table2` is triggered. Materialized relations are triggered when tuples are inserted or updated. The materialized relations are first *joined*, then a *projection* on field `B` is done. All `eHead2` tuples are sent from node `I` to `J`.

```
r2 eHead2(@J,I,B) :- table1(@I,J,A,B), table2(@I,J,B,C).
```

In rule r3, the head contains `a_COUNT<*>`, an aggregation operator that returns the number of tuples in `table3`. The current time `T` is found using a built-in function `f_now()`.

```
r3 eHead3(@I,T,a_COUNT<*>) :- table3(@I,D), T := f_now().
```

[4]The RIB only signifies the existence of some "information base" without excluding the various notions of what it might mean to implement it.

4. TRANSPORT POLICIES IN P2

We illustrate the ease of declaratively specifying transport policies using a few examples. We show that by eliminating the notion of "end-hosts", transport functions such as those typically done by proxies, in-network support mechanisms or even multicast-specific mechanisms become significantly easier to realize. In our specifications, distributed transport state is manipulated and shared across nodes by executing recursive queries over the network. The distributed nature of transport state makes it an integral part of the network state (*i.e.,* routing, monitoring and resource allocation information). This integration of state allows for flexible / generic transport functions to be realized without requiring so-called "cross-layer" communication. From our perspective, a traditional end-to-end solution (*e.g.,* TCP, DCCP [13]) is a degenerate case where only two nodes in the network, the end-hosts, execute transport functions over a virtual link. Our sample specifications consider two scenarios where:

1. Nodes are located along a path, connecting a sender and receiver, to support a single application (*i.e.,* transport connection). The network provides a hop-by-hop transport service. Any transport tuples sent over the network are destined to the appropriate neighbor, on the forward or reverse path, by querying local routing information which populates the `nextHop` and `prevHop` relations.

2. Nodes are located on a tree, connecting a single source to multiple receivers, to support a layered multicast session. Any transport tuples sent over the network are destined to the appropriate upstream or downstream neighbor, by querying local routing information which populates the `upstreamNeighbor` and `downstreamNeighbors` relations.

In general, one would program any configuration of nodes supporting multiple transport connections based on their roles (*e.g.,* relay, cache, proxy, bandwidth estimator). One could also program a transport service for a wireless sensor network where all nodes potentially maintain transport state (see [21] for a survey of existing transport solutions). In the sample transport policies below, the role of a node should be clear.

4.1 Acknowledgment Control

One of the most common transport control functions is ack control. There are several ack policies that are commonly used. Cumulative acks inform previous nodes along the path of the last in-order correctly received packet. Selective acks, on the other hand, inform previous nodes of all (potentially non-contiguous) received packets. Other types of "exotic" ack policies may include: (1) a periodic cumulative ack that is transmitted, irrespective of lost packets, provided some minimum delivery rate for new packets is maintained, or (2) a periodic ack used primarily as a keep-alive message.

4.1.1 Cumulative Acknowledgments

When a node receives a transfer PDU its sequence number is enqueued in the `rcvSeq` relation maintained in DTSV[5] which triggers rule `ack1`. If the received sequence number `Seq` is expected then the `eIncrement` event is triggered to increment the expected sequence number in rule `ack2`. Every time the `expSeq` relation is updated, it indirectly triggers rule `ack4` via rule `ack3`. Rules `ack4` and `ack5` deal with the case where the received transfer PDU is indeed the expected one but subsequent PDUs were previously received. In this case, `expSeq` is incremented again (recursively) to reflect the sequence number following the last in-order correctly received sequence number.

```
ack1 eIncrement(@I, J) :- rcvSeq(@I,J,Seq),
     expSeq(@I,J,ExpSeq), Seq == ExpSeq.
ack2 expSeq(@I,J,NewExpSeq) :- eIncrement(@I,J),
     expSeq(@I,J,ExpSeq), NewExpSeq := ExpSeq + 1.
ack3 eIncremented(@I,J,ExpSeq) :- expSeq(@I,J,ExpSeq).
ack4 eMinSeq(@I,J,a_MIN<Seq>) :- eIncremented(@I,J,
     ExpSeq), rcvSeq(@I,J,Seq), Seq >= ExpSeq.
ack5 eIncrement(@I,J) :- eMinSeq(@I,J,MinSeq),
     expSeq(@I,J,NewExpSeq), MinSeq == NewExpSeq.
```

The ack control mechanism is triggered periodically at some appropriate update rate, $\frac{1}{T}$, to transmit a cumulative ack updating the transport state maintained by other nodes. Note that acks traverse nodes on the reverse path towards the source and the shared state is the sequence numbers received by each node.

```
ack6 ePeriodic(@I,J) :- periodic(@I,E,T),
     prevHop(@I,J).
ack7 eAckPDU(@J,I,Seq) :- ePeriodic(@I,J),
     expSeq(@I,J,ExpSeq), Seq := ExpSeq - 1.
```

When a node receives a cumulative ack it removes all records in its retransmission queue `rtxQ` such that the sequence number received in the ack is greater than or equal to the sequence number field in the transfer PDU's record. Each matching record triggers `eDel` to delete that record from `rtxQ`.

```
ack8 eDel(@I,J,TimeSent,Seq) :-
     eAckPDU(@I,J,RcvdSeq), rtxQ(@I,J,TimeSent,Seq),
     RcvdSeq >= Seq.
ack9 delete rtxQ(@I,J,TimeSent,Seq) :-
     eDel(@I,J,TimeSent,Seq).
```

4.1.2 Selective Acknowledgments

For selective acks, a node could send an ack for every transfer PDU that is received or for every *new* transfer PDU as shown in rules `ack10` and `ack11`, respectively.

```
ack10 eAckPDU(@J,I,Seq) :- rcvSeq(@I,J,Seq),
      prevHop(@I,J).
ack11 eAckPDU(@J,I,Seq) :- rcvSeq(@I,J,Seq),
      expSeq(@I,J,ExpSeq), Seq >= ExpSeq,
      prevHop(@I,J).
```

A less expensive approach would be similar to cumulative acks where the node periodically transmits a selective ack containing an encoding of the unacknowledged sequence numbers received so far.

4.1.3 Smart Acknowledgments

A "smart" ack policy at an intermediate node can realize the behavior of a *performance-enhancing proxy.* For example, an *ack regulator* proxy was introduced in [6] for cellular data

[5]Recall that DTSV allows DTP and DTCP to share information.

networks to alleviate the impact of the wireless channel on TCP. The proxy only forwards acks when it has sufficient buffer space to absorb incoming packets, while forcing TCP to operate in congestion avoidance by deliberately losing one packet per "congestion epoch". Such an ack policy can be easily specified declaratively.

4.2 Retransmission Control

Retransmission control is responsible for handling transfer PDUs whose information is currently stored in DTSV's retransmission queue, `rtxQ`. For simplicity, we only consider timeout-triggered retransmissions. When the timer for a transfer PDU expires, several retransmission policies are possible: (1) retransmit the PDU whose timer expired, (2) retransmit all PDUs in `rtxQ`, or (3) retransmit at most N PDUs where N could be a limit set by a rate control policy.

To retransmit only the PDU whose timer expired, we periodically check if the timer for any transfer PDU has expired. If so, that PDU is retransmitted. Assume that all PDUs have the same retransmission timeout `RTO` and that triggering event `eRtx` handles the retransmission task.

```
rtx1 eRtx(@I,J,Seq) :- periodic(@I,E,T),
     rtxQ(@I,J,TimeSent,Seq),
     Tnow := f_now(), Tnow - TimeSent > RTO.
```

Caching transfer PDUs and executing a retransmission control policy at an intermediate node can realize a performance-enhancing proxy that performs *local retransmissions*. For example, Snoop [2] is a proxy solution that locally retransmits packets on behalf of the TCP source. Snoop also hides duplicate acks from the source. This can be done declaratively by keeping track of transmitted ack tuples to make sure that an ack tuple is transmitted at most once.

4.3 Rate Control

In general, monitoring applications keep track of a wide range of performance metrics (*e.g.,* delay, throughput, available bandwidth). A rate control policy can query the RIB for the appropriate metrics to compute the maximum transmission rate that does not overload the network. More specifically, giving a source *in-network support* in the form of an explicit rate (*e.g.,* jtp [17], xcp [12]) can be done easily using a recursive query. In jtp, the source receives an estimate of the minimum available bandwidth on the path to the destination. Declaratively, this can be done by having the destination node execute rule `rc1` to periodically send a measurement tuple. All other nodes execute rule `rc2` to update the minimum bandwidth estimate and forward the measurement tuple towards the source. Here the shared transport state is the minimum available bandwidth on the path traversed by the supported transport connection.

```
rc1 bwPDU(@J,I,BW) :- periodic(@I,E,T), prevHop(@I,J).
rc2 bwPDU(@K,I,BWout) :- bwPDU(@I,J,BWin), bw(@I,J,BW),
    BWout = f_min(BWin,BW), prevHop(@I,K).
```

4.4 Flow Control

Consider a flow control policy executed by nodes in the network to support a *layered multicast session* connecting a single source to multiple receivers as shown in Figure 4. The stream to be disseminated utilizes a layered encoding where

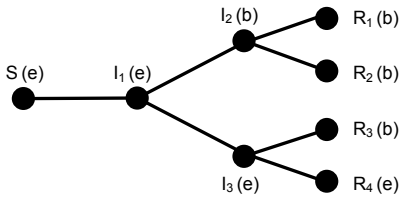

Figure 4: A sample multicast tree with a source S and four receivers R_1 to R_4. Nodes are marked with the highest layer they are subscribed to. The base and enhanced layers are denoted by b and e, respectively.

it is divided into two layers, namely, a *base layer* and an *enhanced layer*. Depending on the receiver's device type (*e.g.,* laptop, personal digital assistant, mobile phone) and its network connection (*e.g.,* broadband, LAN, 3G) it either subscribes to both layers or only the base layer. Depending on each receiver's capability, upstream nodes must disseminate the appropriate streams accordingly to avoid overloading the receivers. One example of a layered multicast solution can be found in [5].

In this simple example, each node keeps track of the highest layer it needs to subscribe to in the relation `layer` and periodically updates its upstream neighbor using the event tuple `eLayerUpdate`. The base and enhanced layers are encoded in the relations as 0 and 1, respectively.

```
m1 eLayerUpdate(@J,I,HighestLayer) :- periodic(@I,E,T),
   upstreamNeighbor(@I,J), layer(@I,HighestLayer).
```

Once a layer update from downstream neighbors is received, each node must update its view of the network. This view includes the highest layer downstream neighbors are subscribed to and in turn the highest layer that the node itself needs to support (*i.e.,* which streams need to be transmitted).

```
m2 downstreamNeighbors(@I,J,Layer) :-
   eLayerUpdate(@I,J,Layer).
m3 layer(@I,a_MAX<Layer>) :-
   downstreamNeighbors(@I,J,Layer).
```

Assume that the triggering of events `eSendBaseStream` and `eSendEnhancedStream` handles the task of disseminating the appropriate streams to downstream neighbors.

```
m4 eSendBaseStream(@J,I) :-
   downstreamNeighbors(@I,J,NeighborLayer),
   NeighborLayer ≥ 0.
m5 eSendEnhancedStream(@J,I) :-
   downstreamNeighbors(@I,J,NeighborLayer),
   NeighborLayer == 1.
```

5. POSSIBLE EXTENSIONS TO NDLOG

Based on our preliminary specifications of transport policies, we identified possible extensions to NDlog.

5.1 Support for Timers

Implementing transport policies in NDlog requires support for timers (*e.g.,* retransmission timers, state timers). Using

`periodic` to check if a timer expired triggers tuples unnecessarily which degrades performance. We are considering the following extensions:

1. Allowing each tuple in a materialized relation to have its own lifetime attribute[6].
2. Triggering a rule-level event containing the information in an expired tuple that is being removed from a materialized relation.

Given these extensions, consider the tuples in `rtxQ` that contain information about transfer PDUs that may need to be retransmitted. Each tuple has a lifetime that is equal to the PDU's retransmission timeout (*i.e.,* `RTO`). When a tuple expires, the triggering of the expired tuple allows the retransmission of the transfer PDU to be handled.

5.2 Support for Transactions

NDlog does not support multi-rule atomicity. This leaves specifications susceptible to race conditions. One may require rules to be executed sequentially and atomically to guarantee correct behavior. This can be crudely achieved by assigning priorities to rules as done by Chu *et al.* [8] to bias the scheduling of rule execution.

When dealing with transport state in DTSV, race conditions either degrade performance or threaten protocol correctness. Thus, in addition to implementing the data-intensive and time-sensitive DTP mechanisms in the underlying dataflow elements in P2, we are considering possible extensions to NDlog based on Transactional Datalog [3].

6. CONCLUSIONS

We argued that in a clean-slate architecture [11], transport state is an integral part of the network state, which includes information for routing, monitoring, resource allocation, *etc.* Given the myriad of transport policies needed to support advanced functions such as in-network caching, in-network fair allocation, and proxying, these policies should be made programmable. We outlined a few sample declarative transport specifications in P2. We are completing and testing our specifications to evaluate ease of programming, correctness, and performance. Our evaluation will include prototyping on heterogeneous environments that include different network technologies (*e.g.,* wireless, cellular) and various application requirements (*e.g.,* delay-tolerant, loss-tolerant).

Open problems include determining the appropriate tradeoff between maintaining transport state across nodes and passing state information in PCI fields (as done for TCP to improve its scalability and resilience properties [19]). Another open problem is in selecting optimal locations for maintaining transport state and/or invoking the appropriate transport functions (as done in sensor networks where query optimizations automated the selection of rendezvous and proxy locations [7]).

Acknowledgments

This work has been partially supported by National Science Foundation awards: CISE / CCF #0820138, CISE / CSR #0720604, CISE / CNS #0524477, CNS / ITR #0205294, and CISE / EIA RI #0202067.

[6]NDlog associates a single lifetime attribute with all tuples.

7. REFERENCES

[1] P2: Declarative Networking. http://p2.berkeley.intel-research.net/.

[2] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz. Improving TCP/IP Performance over Wireless Networks. In *ACM MOBICOM*, November 1995.

[3] A.J. Bonner. Workflow, Transactions, and Datalog. In *ACM PODS*, November 1999.

[4] P. G. Bridges, G. T. Wong, M. A. Hiltunen, R. D. Schlichting, and M. J. Barrick. A Configurable and Extensible Transport Protocol. *IEEE/ACM Trans. Netw.*, 15:1254–1265, 2007.

[5] J. Byers, G. Kwon, M. Luby, and M. Mitzenmacher. Fine-grained Layered multicast with STAIR. *IEEE/ACM Trans. Netw.*, 14(1):81–93, 2006.

[6] Mun Choon Chan and Ramachandran Ramjee. TCP/IP Performance over 3G Wireless Links with Rate and Delay Variation. In *ACM MOBICOM*, September 2002.

[7] D. Chu and J. Hellerstein. Automating Rendezvous and Proxy Selection in Sensornets. In *IPSN*, April 2009.

[8] D. Chu, L. Popa, A. Tavakoli, J. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The Design and Implementation of a Declarative Sensor Network System. In *International Conference on Embedded Networked Sensor Systems*, 2007.

[9] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *ACM SIGCOMM*, September 1990.

[10] T. Condie, J. Hellerstein, P. Maniatis, S. Rhea, and T. Roscoe. Finally, a Use for Componentized Transport Protocols. In *HotNets-IV*, November 2005.

[11] J. Day, I. Matta, and K. Mattar. Networking is IPC: A Guiding Principle to a Better Internet. In *Re-Architecting the Internet (ReArch)*, December 2008.

[12] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *ACM SIGCOMM*, August 2002.

[13] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion Control Without Reliability. In *ACM SIGCOMM*, September 2006.

[14] C. Liu, Y. Mao, M. Oprea, P. Basu, and B. T. Loo. A Declarative Perspective on Adaptive MANET Routing. In *ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of TOmorrow*, August 2008.

[15] B. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *ACM SOSP*, October 2005.

[16] B. Loo, J. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *ACM SIGCOMM*, August 2005.

[17] N. Riga, I. Matta, A. Medina, C. Partridge, and J. Redi. JTP: An Energy-conscious Transport Protocol for Multi-hop Wireless Networks. In *CoNEXT Conference*, December 2007.

[18] J. Saltzer, D. Reed, and D. Clark. End-to end Arguments in System Design. *ACM Trans. Comput. Syst.*, 2(4):277–288, 1984.

[19] A. Shieh, A. Myers, and E. Sirer. Trickles: A Stateless Network Stack for Improved Scalability, Resilience, and Flexibility. In *NSDI*, May 2005.

[20] A. Tavakoli, D. Chu, J. Hellerstein, P. Levis, and S. Shenker. A Declarative Sensornet Architecture. *SIGBED Rev.*, 4(3):55–60, 2007.

[21] C. Wang, K. Sohraby, B. Li, M. Daneshmand, and Y. Hu. A Survey of Transport Protocols for Wireless Sensor Networks. *IEEE Network*, 20:34–40, 2006.

[22] R. Watson. Timer-Based Mechanisms in Reliable Transport Protocol Connection Management. *Computer Networks*, 5:47–56, 1981.