

Declarative Transport

NO MORE TRANSPORT PROTOCOLS TO DESIGN, ONLY POLICIES TO SPECIFY*

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

[†]College of Arts & Science [‡]Metropolitan College
Computer Science
Boston University

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

Technical Report BUCS-TR-2008-014

August 26, 2008

This version revises an original version posted July 12, 2008

ABSTRACT

Transport protocols are an integral part of the inter-process communication (IPC) service used by application processes to communicate over the network infrastructure. With almost 30 years of research on transport, one would have hoped that we have a good handle on the problem. Unfortunately, that is not true. As the Internet continues to grow, new network technologies and new applications continue to emerge putting transport protocols in a never-ending flux as they are continuously adapted for these new environments.

In this work, we propose a clean-slate transport architecture that renders all possible transport solutions as simply combinations of policies instantiated on a single common structure. We identify a minimal set of mechanisms that once instantiated with the appropriate policies allows any transport solution to be realized. Given our proposed architecture, we contend that there are no more transport protocols to design—only policies to specify.

We implement our transport architecture in a declarative language, Network Datalog (NDlog), making the specification of different transport policies easy, compact, reusable, dynamically configurable and potentially verifiable. In NDlog, transport state is represented as database relations, state is updated/queried using database operations, and transport policies are specified using declarative rules.

We identify limitations with NDlog that could potentially threaten the correctness of our specification. We propose several language extensions to NDlog that would significantly improve the programmability of transport policies.

*This work has been partially supported by a number of National Science Foundation grants, including CISE/CCF Award #0820138 CISE/CSR Award #0720604, CISE/CNS Award #0524477, CNS/ITR Award #0205294, and CISE/EIA RI Award #0202067.

1. INTRODUCTION

As the Internet continues to grow, new network technologies (*e.g.*, wireless, cellular, ad hoc, sensor and mesh networks) as well as new applications continue to emerge. These new network technologies come with new Quality-of-Service (QoS) properties (delay/jitter, bandwidth variability, error/loss rates, *etc.*) while new applications come with new requirements (the need for rate, flow, loss/error control, *etc.*) This puts the state of transport solutions in a never-ending flux as they are continually adapted for new environments¹.

Over the last three decades, many transport *paradigms* (*e.g.*, reliable / connection-oriented, unreliable / connectionless, transaction-oriented and real-time) were adopted. Each new paradigm can be viewed as a different point in the spectrum of possible *requirements* that must be supported by the transport solution. This (somewhat) narrow view of transport has led to the development of many custom point-solutions (*e.g.*, UDP, TCP, RTP, DCCP, JTP [16]) but little in terms of a general framework or unified theory.

Recent work on configurable and extensible transport solutions (see the work by Bridges *et al.* [2] and references therein) proposes a general transport framework where all possible transport functions are implemented using smaller modules called *microprotocols*. The end goal is to enable the selection of an appropriate set of microprotocols that, when combined together, implements the desired transport behavior. While such a framework, as well as others described in [2], does indeed share our goals, it suffers from two main drawbacks. First, there is no way to reason about and decide on the number of microprotocols that will be required.

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

Instead, new microprotocols are created whenever more functionality is needed. While this provides what seems to be unlimited flexibility in protocol development, it results in increasingly complex implementations and integration problems over time [2]. Second, an ordering of these microprotocols is required to achieve the desired behavior. Given the potentially large number of microprotocols that need to be strung together, this is by no means trivial, which makes verifying protocol correctness a daunting undertaking [2].

Our Contribution:

Our work progresses along different fronts. First, we focus on developing a general transport architecture. We show that only a minimal set of mechanisms is required to realize the entire spectrum of possible solutions. The key idea is to *separate mechanisms from policies*. We avoid overloaded semantics whenever possible if it comes at a reasonable cost. The flexibility of our proposed architecture comes from its ability to allow different policies to be activated within each mechanism, realizing any possible transport solution, while enabling its dynamic configurability at no additional cost. Our transport architecture consists of two protocols, the Data Transfer Protocol (DTP) and the Data Transfer Control Protocol (DTCP), as well as a management application. This leads to a separation of concerns over different timescales. DTP operates over a short timescale (packet-level) and deals exclusively with data manipulation functions. DTCP and the management application, on the other hand, operate over a longer timescale (flow-level) to control and manage the supported flow. Our view is consistent with Clark and Tennenhouse’s seminal work [5] that argues for the separation of data manipulation functions from control. While Clark and Tennenhouse do consider in-band versus out-band communication as a way to separate data manipulation from control, we show that the complete decoupling of DTP from DTCP is a fundamental property that occurs naturally. In our architecture, DTCP control mechanisms and the policies associated with these mechanisms operate and evolve independently. This observation re-emphasizes that TCP and UDP, for example, should have never been separate protocols. Instead, they can be realized using a single structure. We contend that there are *no more transport protocols to design, only policies to specify*. Of course one may always need to make design decisions to support new requirements, but we believe that such extensions will only require specifying and implementing new transport policies without having to redesign the structure / protocol itself.

Second, we specify an initial prototype of our transport architecture in the declarative language Network Datalog (NDlog). While providing an absolute argument for using any particular formal description lan-

guage is inherently difficult, we note the primary reasons that motivated our choice. Declarative languages in general allow for the investigation of protocol behavior without worrying about the implementation. In other words, they allow one to specify the “what” and not the “how.” Declarative languages also allow one to leverage the work on provable correctness and protocol verification / testing (*e.g.*, [4, 11, 17]). NDlog in particular has been extensively used recently in the literature to provide specifications of routing protocols [12, 15], overlays [14], as well as sensor network architectures and applications [3, 18], just to name a few. The ease of programming, compactness of the specification, the reusability of the code, as well as the security provided by the restricted expressiveness of the language, have all been widely documented in the literature. In terms of specifying transport policies, NDlog utilizes relations (*i.e.*, tables) and database operations to implement the specified behavior, which as we show, proves to be a natural way to represent and manipulate transport state.

2. NEW TRANSPORT ARCHITECTURE

2.1 Components

In general, the two end-points of a transport connection (flow) maintain shared state by exchanging protocol data units (PDUs). A PDU consists of two parts, namely, the user’s data and the protocol control information (PCI)². State information is either passed explicitly in the PCI (*e.g.*, the receiver’s current available buffer space) or inferred from the exchange of PDUs over time (*e.g.*, an estimate of the connection’s round trip time). Analyzing the PCI in TCP, the de-facto transport protocol, one quickly realizes that there are two types of control information: 1) information that must be associated with the user’s data (*e.g.*, the checksum) and therefore must be transmitted with the data, and 2) information that does not have to be associated with the user’s data (*e.g.*, SACK blocks) and can be transmitted in a separate PDU. We call these PDUs Transfer and Control PDUs, respectively. This leads to a natural decoupling of transport into two separate protocols: the Data Transfer Protocol (DTP) and the Data Transfer Control Protocol (DTCP). DTP and DTCP are decoupled via the Data Transfer State Vector (DTSV) which contains all the shared state. This leads to a separation of concerns over different timescales. Our proposed architecture is outlined in Figure 1.

2.1.1 Data Transfer Protocol (DTP)

In general, every flow must have a DTP instance as-

²We 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.

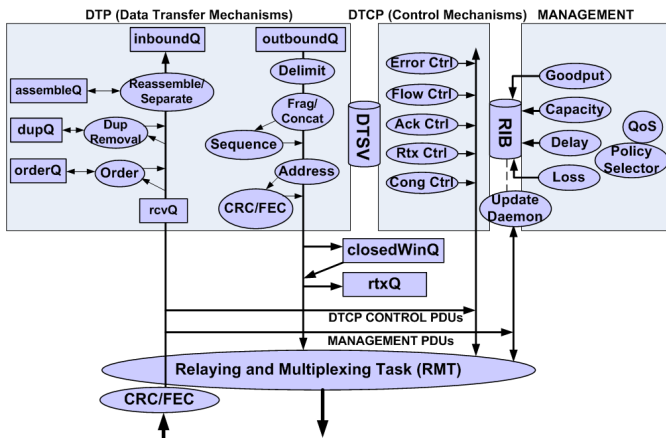


Figure 1: Our Transport Architecture

sociated with it. Service data units (SDUs)³ are enqueued in `outboundQ` by the application (or the layer above). DTP, at the sender, is responsible for delimiting the SDUs, performing any fragmentation / concatenation of SDUs to create transfer PDUs whose size is less than the Maximum Transmission Unit (MTU) of the layer below. DTP, at the receiver, is responsible for reassembling / separating these PDUs to recreate the original SDUs before handing them to the application (or layer above). This is consistent with the Application Level Framing concept introduced in [5] where the authors argue that transport solutions should provide services in terms of Application Data Units (ADUs). DTP is also responsible for tacking sequence numbers, addresses and checksum information onto the transfer PDUs. Hence, DTP only consists of mechanisms that are tightly coupled with the user’s data and only generates a single PDU type—the transfer PDU. Without a DTCP instance, DTP hardly contains any policies and its implementation could be made very efficient⁴. We defer the details regarding the policies associated with DTP, as well as its interaction with DTCP to Section 5 where we specify various transport policies in a declarative language.

2.1.2 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. DTCP is where most of the transport policies reside. The existence of a DTCP instance is a matter of policy and depends on whether the supported flow requires any of the control mechanisms to be activated. Sample control mechanisms include, error, acknowledgement, retransmission, flow and congestion control. It is important to note that while all loosely-

³An SDU is a unit of data handed by the application (or layer above) to the transport protocol.

⁴This would represent a degenerate case of a UDP-like flow.

coupled control mechanisms execute independently, in some limited instances they may affect the operation of DTP mechanisms. For example, the error control policy may instruct the CRC/FEC mechanism to use a particular Forward Error Correction scheme.

2.1.3 Management

Management provides the applications being supported with the necessary interfaces to specify their QoS requirements. It is then up to the policy selector to activate the appropriate mechanisms and instantiate suitable policies to satisfy these requirements. Management also provides support for all the required performance monitoring applications. Performance monitoring can be done either passively (by observing transfer PDUs) or actively (by sending probe packets). All monitoring information is stored in a Resource Information Base (RIB) and shared between the sender / receiver using an update daemon that periodically sends update / refresh messages.

We have not described any specific connection management mechanisms. In Watson *et al.* [9], it is proved that timers for maintaining state are necessary and sufficient. Hence, hard-state protocols, such as TCP, which require explicit control and removal of state also need timers. On the other hand, Delta-t [19], which is the solution we adopt, relies exclusively on timers for maintaining state which renders explicit connection management mechanisms unnecessary.

2.2 Transport in a Repeating IPC Layer

We would like to note that our proposed transport architecture is part of a larger general structure—a repeating layer. More specifically, its development was influenced by the fresh perspective that networking is not a layered set of different functions but rather a single layer of distributed Inter-Process Communication (IPC) that repeats over different scopes [7, 8]. In other words, the same set of functions / mechanisms repeat but are instantiated with policies that are tuned to operate over different ranges of the performance space (*e.g.*, capacity, delay, loss). In addition to scope, each repeating layer has a rank denoted by N . The transport architecture we describe operates at any (N) -layer. DTP, at the sender, receives SDUs from the $(N+1)$ -layer and the concatenation mechanism aggregates these SDUs to improve the Relaying and Multiplexing Task’s (RMT)⁵ performance—reduce switching overhead by processing larger units less often. We adopt this (N) -notation from hereon when describing our proposed transport architecture. More details on how our transport architecture

⁵The RMT is simply a module that multiplexes and schedules the PDUs associated with all supported N -layer flows on the appropriate outgoing interface towards their destination.

fits within the general structure of a repeating IPC layer can be found in [8] and is outside the scope of this paper.

3. BACKGROUND

This section provides an overview of NDlog (the declarative language) that we use to specify transport policies and P2 (the underlying system that provides us with a bare-bones communication pipe).

3.1 P2 System

P2 is a declarative networking system developed at Berkeley. Users specify network protocols in NDlog, a declarative language based on extensions to Datalog [13]. These specifications are then *compiled into a dataflow graph* similar to the one used by Click [10]. Each declarative rule specified in NDlog is converted to a strand of elements (*i.e.*, a rule strand) implementing the required relational database operations (joins, selections, projections, aggregations) to evaluate the rule. Rules query / update relations (*i.e.*, tables) and trigger events to implement the desired logic. Tuples, representing PDUs and events, are sent and received over the network via the Network-Out and Network-In modules. The network modules implement functionalities for sending and receiving messages, reliable transmission, and congestion control. The network modules, the queuing / multiplexing elements and the rule strands, as shown in Figure 2, constitute the dataflow graph that when executed results in the implementation of the specified protocol.

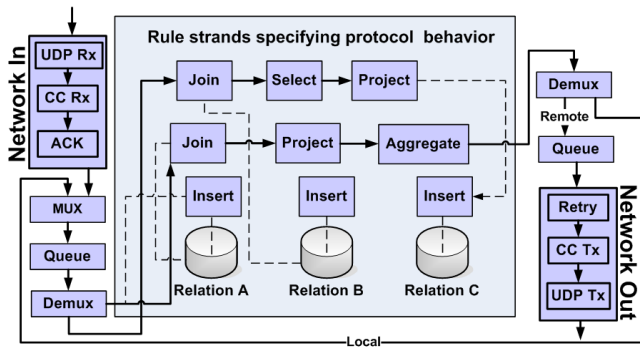


Figure 2: P2's Dataflow Architecture

3.2 Network Datalog

An NDlog program consists of a set of declarative rules. A rule has the form `rulename <head> :- <body>`, where the body consists of many predicates separated by commas indicating an implicit conjunction. The head is triggered only if all the predicates in the body evaluate to `true`.

In NDlog, all predicates are relations. A relation can be either a hard-state, soft-state or an event relation.

Hard-state and soft-state relations are materialized relations containing tuples⁶ that have infinite and finite lifetimes, respectively. Event relations, on the other hand, are treated as streaming tuples that serve as trigger events. When a tuple's time-to-live expires, it is removed from the relation. Materialized relations are declared using the `materialize` command where the name, tuple lifetime, maximum number of tuples and primary key fields of the relation are specified. Tuples in a materialized relation can be inserted, updated, deleted or queried. Each tuple generated by an NDlog program is stored at the address associated with the *location specifier* denoted with the `@` symbol. If the address is remote, the tuple is sent over the network. The declarative rules in NDlog are implemented using traditional database operations.

We consider below four sample declarative rules that highlight the key aspects of NDlog that will be used in our specification of transport policies. We denote event relations by `eEventName` and materialized relations by `relationName`. The body of rule `r1` contains one event relation `eEvent1` and one materialized relation `table1`⁷. The body is triggered when the event tuple is fired (*i.e.*, exists and evaluates to `true`). Tuples are then *selected* from `table1` such that all the values of identical field names in `table1` and `eEvent1` match. Each matching tuple causes the head of the rule to be triggered. All tuples will be generated and consumed by node `I`.

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

The body of rule `r2` contains two materialized relations, `table1` and `table2`. The body is triggered when either `table1` or `table2` is triggered. In general, materialized relations are triggered when tuples are inserted or updated. Having two (or more) materialized relations in the rule's body causes the relations to be *joined*. Finally, a *projection* on field `B` is done. All `eHead` tuples are sent from node `I` to node `J`.

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

In rule `r3`, the head contains an aggregation operator `a_COUNT<*>` that returns the number of tuples in `table1`. Other aggregation operators include `a_MIN<field>` and `a_MAX<field>` which return all tuples that have the minimum (maximum) value in the specified field.

`r3 eHead(@I,a_COUNT<*>) :- table1(@I,A,B).`

NDlog supports built-in functions. In rule `r4` the current time is returned using a built-in function, `f_now()`, when `eEvent1` is triggered. If the expression in the rule body is `true` a tuple with a matching time field is deleted from `table1`.

⁶It is important to note that we use the term *tuple* to denote both trigger events and records in materialized relations both of which could be transmitted over the network.

⁷The body of a rule can contain at most one event relation.

```

r4 delete table1(@I,Time) :- eEvent1(@I,Time),
    TNow := f.now(), TNow > Time.

```

4. DECLARATIVE TRANSPORT

This section motivates our choice to implement a specification of our transport architecture in the declarative language NDlog, and compares it to the existing transport implementation in P2.

4.1 Componentized Versus Declarative

The network elements in P2 implement functionalities for sending and receiving messages, reliable transmission and congestion control, as shown in Figure 2. In [6], the authors propose utilizing the configurability of the dataflow graph to organize and reorder these elements to implement a componentized transport solution that is able to more closely satisfy application requirements.

We take a different approach. Instead of implementing various transport elements in an imperative language, we implement a fine-grained specification of transport policies using declarative rules. The retry element in P2’s network module, for example, is replaced with rules specifying several possible retransmission policies. Thus, we view P2 as a system that provides us with a bare-bones communication pipe for exchanging tuples over which we build our transport architecture declaratively. Our end goal is to incorporate our transport architecture within the P2 system and allow other declarative specifications of applications and protocols to utilize it. In the following section, we outline the benefits of having such a fine-grained declarative specification.

4.2 Benefits of a Fine-Grained Declarative Specification in NDlog

In addition to the wide adoption of NDlog (as well as several extensions to it) and the documented advantages of its compact / potentially verifiable specifications (*e.g.*, see [3,12,14,15,18]), we believe that NDlog has several key properties that make it a natural fit for implementing transport policies.

NDlog utilizes relations (*i.e.*, tables) and database operations to implement the specified behavior. As we show in Section 5, this proves to be a natural way for representing and manipulating transport state. We show that representing transport state as database relations and specifying transport policies using declarative rules allows for the concise specification of transport behavior.

In terms of implementing the specified behavior, P2 provides us with a suitable and somewhat unique communication paradigm. Transfer and control PDUs are created by sending formatted tuples over the network. A formatted tuple contains fields that both transport end-points understand (*i.e.*, know how to process). The

exchange and/or firing of tuples allows different mechanisms to pass and share information (*e.g.*, PCI fields).

5. TRANSPORT POLICIES IN NDLOG

In this section, we declaratively specify a subset of our transport architecture outlined in Figure 1. We focus on DTP mechanisms and how they are affected by various DTCP policies, as well as a few independent control policies in DTCP. The sender and receiver are located at nodes I and J , respectively, while the connection between them is stored in `link(@I,J)`. For ease of exposition, we incrementally add DTCP control mechanisms. We first start by outlining what transport state is maintained as database relations.

5.1 Transport State as Database Relations

We first describe the relational schema for the materialized relations representing the queues in DTP. The DTP outbound mechanisms require `outboundQ(@I, J, TimeInserted, Data)` that holds SDUs inserted by the $(N+1)$ -layer. The `closedWinQ(@I, J, Seq, Data)` holds transfer PDUs to be scheduled for transmission by the DTCP flow control mechanism once the transmission window allows more data to be transmitted⁸. Copies of transmitted transfer PDUs are stored in `rtxQ(@I, J, TimeSent, Seq, Data)` whenever the DTCP retransmission control mechanism is activated. The DTP inbound mechanisms, on the other hand, require `rcvQ(@I, J, TimeRcvd, Seq, Data)` which holds transfer PDUs received from the $(N-1)$ -layer. Out-of-order transfer PDUs are enqueued in `orderQ(@I, J, Seq, Data)`. For ease of exposition, we assume that all relations representing queues have no limit on the number of tuples that can be inserted.

Next we describe the relational scheme for the materialized relations representing the state variables in the Data Transfer State Vector (DTSV). At the sender, the sequencing mechanism in DTP maintains the current sequence number in `curSeq(@I, J, Seq)` and the sequence number of the last *in-order* acknowledged packet in `lastAckRcvd(@I, J, LastAckRcvd)`. At the receiver, the expected sequence number is maintained in `expSeq(@I, J, ExpSeq)`. The maximum buffer space available at the receiver is stored in `winSize(@I, J, win)`. We focus on a single transport connection so all state relations contain at most one tuple⁹.

⁸For simplicity, we only consider window-based, rather than rate-based, transmission control mechanisms.

⁹More generally, any relation that stores information about a single variable will contain a single tuple (*i.e.*, record) indicating the current value of the variable.

5.2 DTP Data Transfer Policies

5.2.1 DTP Outbound + No Dup Removal + No DTCP Mechanisms

We start by specifying the outbound mechanisms (responsible for processing outgoing SDUs) associated with a flow that only has a DTP instance and does not require the duplicate removal mechanism. Neither sequencing nor error correction is required in this case, thus these mechanisms are deactivated by having a null policy. Addressing is provided by P2's location specifiers (cf. Section 3.2). For simplicity, we ignore delimiting and fragmentation / concatenation of SDUs to create transfer PDUs. Thus when an SDU is inserted in `outboundQ` by the (N+1)-layer, a transfer PDU (`eTransferPDU`) is constructed and sent to its peer.

```
snd01 eTransferPDU(@J, I, Data) :-
    outboundQ(@I, J, TimeInserted, Data).
```

5.2.2 DTP Outbound + DTP Dup Removal + DTCP Error Ctrl

Duplicate removal and error control require transfer PDUs to contain a sequence number and a checksum, respectively. When SDUs are inserted in `outboundQ`, the sequence number `curSeq` associated with the DTP instance is first incremented. Then the first SDU inserted in `outboundQ` is selected (in case multiple SDUs were inserted), a transfer PDU is constructed and sequenced. The transfer PDU's checksum is then computed using a built-in function `f_checksum()`. Once the sequence number and checksum are tacked onto the transfer PDU, it is sent to its peer.

```
snd01 eUpdateSeq(@I, J) :- outboundQ(@I, J, Time, Data).
snd02 curSeq(@I, J, NewSeq) :- eUpdateSeq(@I, J),
    curSeq(@I, J, Seq), NewSeq := Seq+1.
snd03 eSeqUpdated(@I, J) :- curSeq(@I, J, Seq).
snd04 eMinTime(@I, J, a_MIN<Time>) :- eSeqUpdated(@I, J),
    outboundQ(@I, J, Time, _).
snd05 eSequencedData(@I, J, Seq, Data) :- eMinTime(@I, J, Time),
    outboundQ(@I, J, Time, Data), curSeq(@I, J, Seq).
snd06 eData(@I, J, Seq, Data, Checksum) :-
    eSequencedData(@I, J, Seq, Data),
    Checksum := f_checksum(I, J, Seq, Data).
snd07 eTransferPDU(@J, I, Seq, Data, Checksum) :-
    eData(@I, J, Seq, Data, Checksum).
```

5.2.3 DTP Outbound + DTCP Rtx Control

When the flow has a DTCP retransmission control instance associated with it, the `rtxQ` relation is allocated and a copy of every transmitted transfer PDU is inserted in it. It is then up to DTCP to retransmit the inserted PDU when its retransmission timer expires. We will discuss retransmission policies in more detail in Section 5.4. For ease of presentation, the `Checksum` field is henceforth omitted.

```
#include(snd01, snd02, snd03, snd04, snd05).
```

```
snd06 rtxQ(@I, J, Tnow, Seq, Data) :-
    eSequencedData(@I, J, Seq, Data), Tnow := f_now().
```

5.2.4 DTP Outbound + DTCP Flow Control

When the flow has a DTCP flow control instance associated with it, the `closedWinQ`, `lastAckRcvd` and `winSize` relations are allocated. A transfer PDU is sent over the network only if the number of unacknowledged transfer PDUs computed by $(Seq - LastAckRcvd)$, where `Seq` denotes the sequence number of the PDU to be transmitted, does not exceed the window size allowed by the flow control mechanism. Otherwise, the PDU is buffered in `closedWinQ`. It is then up to DTCP to transmit the buffered PDUs when the window opens up again.

```
#include(snd01, snd02, snd03, snd04, snd05).
snd07 eTransferPDU(@J, I, Seq, Data) :- eSequencedData(@I, J, Seq,
    Data), lastAckRcvd(@I, J, LastAckRcvd), winSize(@I, J, Win),
    Win >= Seq-LastAckRcvd.
snd08 closedWindowQ(@I, J, Seq, Data) :- eSequencedData(@I, J,
    Seq, Data), lastAckRcvd(@I, J, LastAckRcvd),
    winSize(@I, J, Win), Win < Seq-LastAckRcvd.
```

5.2.5 DTP Inbound + DTP Ordering + No DTCP Mechanisms

Here we specify DTP inbound mechanisms, particularly the ordering mechanism, associated with a flow that only has a DTP instance. The receiver could potentially have several policies for buffering received transfer PDUs. When a transfer PDU is received it is placed in `rcvQ`. The receiver may process (and buffer) only in-order (expected) PDUs by triggering `eOrderedDataRcvd` while dropping all out-of-order PDUs.

```
ord01 eOrderedDataRcvd(@I, J, Seq, Data) :-
    rcvQ(@I, J, TimeRcvd, RcvdSeq, Data), expSeq(@I, J, ExpSeq),
    RcvdSeq == ExpSeq.
ord02 eUnexpDataRcvd(@I, J, Seq, Data) :-
    rcvQ(@I, J, TimeRcvd, RcvdSeq, Data), expSeq(@I, J, ExpSeq),
    RcvdSeq != ExpSeq.
On the other hand, the receiver may choose to enqueue at most an entire window of out-of-order transfer PDUs in orderQ. PDUs that do not have the expected sequence number are considered out-of-order10.
#include(ord01).
ord02 eUnexpDataRcvd(@I, J, Seq, Data) :- rcvQ(@I, J, TimeRcvd,
    RcvdSeq, Data), expSeq(@I, J, ExpSeq), winSize(@I, J, Win),
    (RcvdSeq >= ExpSeq+Win || RcvdSeq < ExpSeq).
ord03 orderQ(@I, J, Seq, Data) :- rcvQ(@I, J, TimeRcvd, RcvdSeq,
    Data), expSeq(@I, J, ExpSeq), winSize(@I, J, Win),
    RcvdSeq >= ExpSeq, RcvdSeq < ExpSeq + Win.
```

5.3 DTCP Acknowledgement Policies

There are several acknowledgement policies that are commonly used. Cumulative acknowledgements inform the sender of the last in-order correctly received packet (or byte). Selective acknowledgements, on the other hand, inform the sender of all, potentially non-contiguous, packets received.

¹⁰Rule `ord02` below uses `||` which denotes a logical OR operator.

5.3.1 Cumulative Acknowledgements

As the receiver enqueues transfer PDUs in `rcvQ`, the expected sequence number `expSeq` is maintained by DTP and stored in DTSV. If the received PDU is expected and subsequent PDUs were previously received, `expSeq` is incremented *recursively*. Once the expected sequence number has been maintained (and `eExpSeqReady` is triggered), the ack control mechanism uses it in the acknowledgement PDU transmitted. For simplicity, we assume that transfer PDUs with sequence numbers less than the expected sequence number are deleted from `rcvQ` once processed.

```

ack01 eIncrementExpSeq(@I, J) :- rcvQ(@I,J,_,Seq,_),
    expSeq(@I,J,ExpSeq), Seq == ExpSeq.
ack02 eExpSeqReady(@I, J) :- rcvQ(@I,J,_,Seq,_),
    expSeq(@I, J, ExpSeq), Seq != ExpSeq.
ack03 expSeq(@I,J,NewExpSeq) :- eIncrementExpSeq(@I,J),
    expSeq(@I,J,ExpSeq), NewExpSeq := ExpSeq + 1.
ack04 eExpSeqIncremented(@I,J,ExpSeq) :- expSeq(@I,J,ExpSeq).
ack05 eMinSeq(@I,J,a_MIN<Seq>) :- eExpSeqIncremented(@I,J,
    ExpSeq), rcvQ(@I, J, _, Seq, _), Seq >= ExpSeq.
ack06 eIncrementExpSeq(@I, J) :- eMinSeq(@I, J, MinSeq),
    expSeq(@I, J, NewExpSeq), MinSeq == NewExpSeq.
ack07 eExpSeqReady(@I, J) :- eMinSeq(@I, J, MinSeq),
    expSeq(@I, J, NewExpSeq), MinSeq != NewExpSeq.
ack08 eAckPDU(@J, I, Seq) :- eExpSeqReady(@I, J),
    expSeq(@I, J, ExpSeq), Seq := ExpSeq - 1.

```

The sender handles cumulative acknowledgements by removing all records in `rtxQ` such that the sequence number received in the acknowledgement is greater than or equal to the sequence number field in the PDU's record. Each matching record triggers `eDelAckedPDUs` to delete a tuple.

```

ack09 eDelAckedPDUs(@I,J,TimeSent,Seq,Data) :-
    eAckPDU(@I,J,RcvdSeq), rtxQ(@I,J,TimeSent,Seq,Data),
    RcvdSeq >= Seq.
ack10 delete rtxQ(@I, J, TimeSent, Seq, Data) :-
    eDelAckedPDUs(@I, J, TimeSent, Seq, Data).

```

5.3.2 Selective Acknowledgements

Selective acknowledgements are simpler. Every time the receiver enqueues a transfer PDU in `rcvQ`, the `eDataRcvd` event is triggered and the DTCP ack control mechanism sends an acknowledgement with the sequence number of that PDU over the network.

```

ack01 eDataRcvd(@J, I, Seq) :- rcvQ(@I, J,_,Seq,_).
ack02 eAckPDU(@J, I, Seq) :- eDataRcvd(@I, J,Seq).

```

The sender handles the selective acknowledgement by only removing records from `rtxQ` that match the received sequence number `Seq`.

```

ack03 eDelAckedPDUs(@I, J, TimeSent, Seq, Data) :-
    eAckPDU(@I,J,RcvdSeq), rtxQ(@I,J,TimeSent,Seq,Data).
#include ack10.

```

5.4 DTCP Retransmission Policies

For simplicity, we only consider timeout-triggered retransmissions. DTCP's retransmission control mechanism might have several policies associated with it. Upon the timeout of a transfer PDU, either only the PDU that timed out is retransmitted, all unacknowledged PDUs are retransmitted or at most N unacknowledged PDUs are retransmitted.

5.4.1 Retransmit Expired Transfer PDU Only

NDlog does not have support for timers. We consider this issue in more detail in Section 6. NDlog does, however, have a `periodic` command that could infinitely trigger a tuple at node `I` every `T` seconds. We use `periodic` to continuously check if any transfer PDUs have timed out. If so, the PDU is retransmitted and reinserted in `rtxQ`.

```

rtx01 eData(@I, J, Seq, Data) :- periodic(@I, E, T),
    rtxQ(@I, J, TimeSent, Seq, Data),
    Tnow := f_now(), Tnow - TimeSent > RT011.
rtx02 eTransferPDU(@J,I, Seq, Data) :- eData(@I,J, Seq, Data).

```

5.4.2 Retransmit All Unacknowledged Transfer PDUs

Here we first need to detect if any transfer PDU experienced a timeout by checking the first transmitted PDU (using the `MIN` aggregate) stored in `rtxQ`. If a timeout occurred, all PDUs in `rtxQ` are selected and retransmitted.

```

rtx01 ePeriodic(@I, J) :- periodic(@I, E, T), link(@I,J).
rtx02 eCountUnackedPDUs(@I,J, a_COUNT<*>) :-
    ePeriodic(@I, J), rtxQ(@I,J,_,_,_).
rtx03 eRtxOnCount(@I, J) :-
    eCountUnackedPDUs(@I,J,Count), Count > 0.
rtx04 eMinTimePDU Sent(@I,J, a_MIN<TimeSent>) :-
    eRtxOnCount(@I,J), rtxQ(@I,J,TimeSent,_,_).
rtx05 eTimeout(@I,J) :- eMinTimePDU Sent(@I,J, TimeSent),
    TNow := f_now(), TNow - TimeSent > RT0.
rtx06 eData(@I,J, Seq, Data) :- eTimeout(@I,J),
    rtxQ(@I,J, TimeSent, Seq, Data).
rtx07 eTransferPDU(@J,I, Seq,Data) :- eData(@I,J, Seq,Data).

```

5.5 DTCP Rate Control Policies

Monitoring applications keep track of a wide range of connection performance metrics such as throughput, goodput, loss rate, available capacity, delay, *etc.* Thus, any rate control policy, which is arguably the hardest policy in a transport protocol, degenerates to simply querying the required metrics from the Resource Information Base and using any controller or algorithm that is suitable for the environment. In other words, specifying rate control algorithms becomes relatively easy.

6. EXTENSIONS TO NDLOG

6.1 Support for Timers

Implementing transport policies in NDlog requires

support for timers¹² (e.g., retransmission and/or state maintenance timers). We are currently considering a few alternatives. One possibility involves two extensions to NDlog: 1) allowing each tuple in a materialized relation to have its own lifetime attribute¹³, and 2) triggering a rule-level event containing all the information associated with a tuple being removed from a materialized relation due to the expiration of its lifetime.

Consider the tuples in `rtxQ`¹⁴. Each tuple would have a lifetime that is equal to the packet’s retransmission timeout. When the tuple expires, the triggering of the expired tuple event allows the packet retransmission to be readily scheduled.

6.2 Support for Transactions

NDlog does not support multi-rule atomicity. This leaves specifications susceptible to race conditions. Imagine two rules, `r1` and `r2` that operate as follows. Rule `r1` reads from a materialized relation and checks if a particular condition is satisfied. Rule `r2` writes a new value to the relation. One may require rules `r1` and `r2` to be executed atomically to guarantee correct behavior. This can be crudely achieved by assigning priorities to rules as done in [3] to bias the scheduling of rule execution.

When dealing with transport state in DTSV, race conditions lead to either performance degradation or threaten protocol correctness—something which cannot be tolerated. For example, rejecting a received packet because the expected sequence number was not updated correctly causes the transport protocol to induce unnecessary losses. On the other hand, sending two consecutive packets with the same sequence number, threatens the reliability of the protocol. We are currently considering possible extensions to NDlog based on Transactional Datalog [1].

7. ONGOING AND FUTURE WORK

In addition to specifying our proposed transport architecture and evaluating it, our ongoing work involves producing a full specification of a repeating IPC layer (cf. Section 2.2). Such a layer would combine transport and routing, as well as layer management functions for performing enrollment, authentication, resource allocation, address assignment, access control, *etc.* Our goal is to build repeating IPC layers in P2, as part of our effort to realize and evaluate a clean-slate Internet architecture. This architecture is scalable in that it can repeat indefinitely¹⁵ where the scope of the lowest IPC

layer is a physical (shared or dedicated) link. In this regard, we view P2 as a bare-bones communication pipe for exchanging tuples at any IPC layer, and NDlog as the declarative language in which all the elements in the IPC layer, including our transport architecture, can be specified.

8. REFERENCES

- [1] A.J. Bonner. Workflow, Transactions, and Datalog. In *ACM Symposium on the Principles of Database Systems*, pages 294–305, November 1999.
- [2] 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.
- [3] 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.
- [4] W. Citrin. Simulation of Communications Architecture Specifications using Prolog. *SIGAPP Appl. Comput. Rev.*, 1(1):10–17, 1993.
- [5] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. *Computer Communication Review*, 20, September 1990.
- [6] T. Condie, J. Hellerstein, P. Maniatis, S. Rhea, and T. Roscoe. Finally, a Use for Componentized Transport Protocols. In *HotNets-IV*, 2005.
- [7] J. Day. *Patterns in Network Architecture: A Return to Fundamentals*. Prentice Hall, 2008.
- [8] J. Day, I. Matta, and K. Mattar. Networking is IPC: A Guiding Principle to a Better Internet. Technical report, Boston University, August 2008.
- [9] J. G. Fletcher and R. W. Watson. Mechanisms for a Reliable Timer-Based Protocol, February 1978.
- [10] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.
- [11] I. E. Liao and M. T. Liu. Incremental Protocol Verification using Deductive Database Systems. In *International Conference on Data Engineering*, pages 216–223, February 1989.
- [12] 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 (PRESTO)*, August 2008.
- [13] B. Loo, T. Condie, M. Garofalakis, D. Gay, J. Hellerstein, P. Maniatis, R. Ramakrishnan,

ture does not impose any limits. There may, of course, be physical limits and other constraints.

¹²Using `periodic` to check if the timer expired triggers tuples unnecessarily and degrades performance.

¹³NDlog associates the same lifetime attribute with all the tuples in a relation.

¹⁴Contains copies of packets that may need to be retransmitted by DTCP.

¹⁵By “indefinitely” we mean that the nature of the architec-

- T. Roscoe, and I. Stoica. Declarative Networking: Language, Execution and Optimization. In *ACM SIGMOD International Conference on Management of Data*, pages 97–108, 2006.
- [14] B. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *20th ACM Symposium on Operating Systems Principles*, October 2005.
- [15] B. Loo, J. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *ACM SIGCOMM*, August 2005.
- [16] 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.
- [17] D. P. Sidhu and T. K. Leung. Formal Methods for Protocol Testing: A Detailed Study. *IEE Trans. on Software Engineering*, 15(4):413–426, April 1989.
- [18] A. Tavakoli, D. Chu, J. Hellerstein, P. Levis, and S. Shenker. A Declarative SensorNet Architecture. *SIGBED Rev.*, 4(3):55–60, 2007.
- [19] R. W. Watson. The Delta-t Transport Protocol: Features and Experience. In *Local Computer Networks*, pages 399–407, October 1989.