Toward a Composable Transport: Separating Loss Detection from Retransmission

Hoi-Sheung Wilson So*


Abstract

Transport protocols are traditionally designed and implemented as a monolithic entity. As a result, it is extremely difficult for researchers to modify a small portion of it and evaluate it independently. This barrier encourages the re-use of existing transport protocols out of convenience even when performance is sub-optimal. In this paper, we explore the challenges of separating different transport functions into individual pieces. Specifically, we investigate the separation of loss detection from automatic retransmission in a transport protocol. We propose an architecture to logically separate loss detection, congestion control, and automatic repeat request (ARQ). We implemented a simple experimental framework to allow researchers to mix and match different acknowledgment schemes, congestion control algorithms, and retransmission schemes. The fundamental challenge here is to separate loss detection from loss action. A special ack scheme using ack vector is proposed to detect losses for unreliable flows.

1 Introduction

1.1 Motivation

The design and implementation of transport protocols is a very complicated subject. Usually, a transport protocol is designed and implemented as a monolith. As a result, it is very difficult for researchers to experiment with a different control algorithm of a small part without understanding the implementation of the entire transport protocol stack. This intrinsic difficulty in modifying transport protocols discourages the use of application specific transport protocols which can deliver better performance. Ideally, we would like to have a composable transport protocols whose features such as ARQ, in-order delivery, and duplicate-removal that can be turned on or off depending on application needs. In addition, the framework would allow substitution of different algorithms. In reality, defining such a general transport framework is often impossible to achieve. Instead of attacking the general problem of coming up with a composable transport protocol from scratch, we search for a unified framework that allows researchers and programmers to easily write a new transport protocol tailored to specific needs using components from existing protocols. We demonstrate how this framework is general enough to cover TCP, a widely-deployed transport protocol and Web/TP[?], a new transport protocol we developed. At the heart of this problem of separation is a new way to detect losses for packets in an unreliable flows. We implemented the framework in Java and connected it to a Java-based discrete event simulator called SimJava[SimJava]. With this modular framework, researchers can easily study the behavior and performance of different protocol components and control algorithms in isolation.

*Email: so@cs.berkeley.edu
1.2 Overview

In section 2 we first describe our framework used to describe a transport protocol. As mentioned above, the most important feature of this frame is the separation of loss detection from congestion control and ARQ algorithms. We will describe the components of a sender, followed by those of the receiver. In section 2.3, we describe how one can fit TCP into our framework and yield a module implementation of TCP. In section 2.4, we describe how WebTP fits naturally in this framework. Even though TCP and WebTP are vastly different in their design philosophy, their structures can be readily described using this framework. For simplicity, we will only consider unicast transport protocols with sender driven control. In order words, loss detection, congestion control, and ARQ are assumed to be responsibilities of the sender. In addition, we make the simplifying assumption that all packets have the same size. The extension to cover the case with variable packet lengths is easy. We also ignored the issues of flow control because the mechanisms required to support it is very similar to that of congestion control.

2 Modular Transport Framework

2.1 Sender

2.1.1 Send Window

The Send Window (swnd) is the core data structure of the sender. It keeps track of two things: the edges of its control windows (see Figure 2) and the status of each packet (see Figure 3). Each Send Window object keeps track of three edges. They mark the first outstanding packet (snd_una), the next brand new data packet to be sent(snd_max), and the first packet that cannot be sent because of congestion control (snd_limit). The following statements always hold:

1. $\text{snd\_una} \leq \text{snd\_max} \leq \text{snd\_limit}$
2. \( \text{snd} \_\text{una} = \text{snd} \_\text{max} \), if and only if no packet is outstanding

3. \( \text{snd} \_\text{max} = \text{snd} \_\text{limit} \), if and only if the congestion controller does not allow more packets to be send.

Any packets between \( \text{snd} \_\text{una} \) and \( \text{snd} \_\text{limit} \), as shown in Figure 2 are said to be within the send window because it has been sent, or can be sent immediately. Any packets between \( \text{snd} \_\text{una} \) and \( \text{snd} \_\text{max} \) are said to be within the ack window because these packets are in transit to the destination and so acknowledgments are expected for these packets.

When a new data packet is sent by the application, it is in the unborn state. As soon as the congestion control allows the packet to be sent, the state is changed to new. A new packet is eligible for immediate sending, but it might not be sent immediately because the output network interface is busy, for example. When a packet is first sent, the packet enters the sent state. Once a packet is sent, either an ack for the packet is received which causes the packet to be put in the 

<table>
<thead>
<tr>
<th>D</th>
<th>D</th>
<th>S</th>
<th>L</th>
<th>A</th>
<th>N</th>
<th>N</th>
<th>U</th>
</tr>
</thead>
</table>

\( \text{snd} \_\text{una} \) \hspace{1cm} \( \text{snd} \_\text{max} \) \hspace{1cm} \( \text{snd} \_\text{max} \)

---

![](image.png)

**Figure 2:** Edges of the Send Window: \( \text{snd} \_\text{una} \), \( \text{snd} \_\text{max} \), and \( \text{snd} \_\text{limit} \)

---

**Figure 3:** Status of packets in Send Window data structure.

---

2. \( \text{snd} \_\text{una} = \text{snd} \_\text{max} \), if and only if no packet is outstanding

3. \( \text{snd} \_\text{max} = \text{snd} \_\text{limit} \), if and only if the congestion controller does not allow more packets to be send.

Any packets between \( \text{snd} \_\text{una} \) and \( \text{snd} \_\text{limit} \), as shown in Figure 2 are said to be within the send window because it has been sent, or can be sent immediately. Any packets between \( \text{snd} \_\text{una} \) and \( \text{snd} \_\text{max} \) are said to be within the ack window because these packets are in transit to the destination and so acknowledgments are expected for these packets.

When a new data packet is sent by the application, it is in the unborn state. As soon as the congestion control allows the packet to be sent, the state is changed to new. A new packet is eligible for immediate sending, but it might not be sent immediately because the output network interface is busy, for example. When a packet is first sent, the packet enters the sent state. Once a packet is sent, either an ack for the packet is received which causes the packet to be put in the acked state or the packet is inferred to be lost due to acks for subsequent packets or due to timeouts. If the packet is inferred to be lost, it is put into the lost state. Depending on whether ARQ is enabled for a flow (i.e. reliable), a packet is moved to the dead state under one of the two conditions. Either the flow is reliable and the packet at the left edge of the send window is acked, or the flow is unreliable and the packet on the left edge is either acked or lost. Intuitively, a packet should be considered dead if it is assumed to have permanently left the network. For reliable flows, due to retransmissions, a packet cannot be considered dead until it has been acked. On the other hand, a packet from an unreliable flow is never retransmitted and so is dead as soon as it is acked or lost. Each module has read/write access to the Send Window object as shown by the dashed lines.
2.1.2 Loss Detector

Although named a Loss Detector, it actually reports successful delivery of packets in addition to inferring packet losses using methods other than timeouts. The Loss Detection module has only one method:

```c
void input(AckHeader ack);
```

The ack header contains feedback received from the receiver. At the very least, the ack header contains an acknowledgment number. However, the exact semantics of the ack number is not specified by the framework. The ack header might contain some other information as well. The operation of the Loss Detector is closely related to the specific acknowledgment scheme used in a protocol. One requirement is that the Loss Detector be able to generate ack event as soon as the a packet is known to have been delivered to the receiver. The sequence number of the packet is contained in the ack event. Another requirement is that Loss Detector generates a loss event as soon as there is enough evidence that a packet has been lost. This should be done using methods other than timeout. Loss detection due to timeouts are handled by the Timeout module separately.

2.1.3 Feedback Action

The role of the Loss Detector is to detect successful packet delivery or losses. The action that should be taken upon acks and losses are carried out in the Feedback Action module. Feedback Action has three methods:

```c
void ackAction(long seqnum);
void lossAction(long seqnum);
void timeoutAction(long seqnum);
```

As their name suggests, they are invoked on ack, loss, and timeout events. Loss Detector and Timeout modules must guarantee that for every packet sent, one of the 3 methods must be called eventually with that specific sequence number. The primary job function of these methods is to update the Send Window object so that the status of the packets are up to date. The rest of the actions are protocol specific. Also depending on the transport protocol, the ack/loss/timeout events might have duplicates. In such case, a packet might be reported as loss by a loss detector more than once. However, loss action should not be taken more than once.

2.1.4 Congestion Control

Congestion Control is triggered by ack, loss, and timeout actions only. Each time the Congestion Control is triggered, it calculates the right edge of the congestion control window. The right edge controls how much the sender can send at any time\(^1\). The interface of congestion control is:

```c
void ack(long seqnum);
void loss(long seqnum);
void timeout(long seqnum);
long getRightEdge();
```

It is our design choice that Congestion Control does not explicitly specify a “congestion window” within which the packet can be sent. It is implicit in our framework that the left edge of the congestion control window coincides with snd_una of the Send Window object. Hence only a right edge needs to be specified. Because congestion control windows can grow or shrink, it is possible that the Congestion Control shrinks a window enough that packets that were already sent now falls off the right edge of the send window.

\(^1\)Currently, Congestion Control needs to know if a flow is reliable or not so that it can report the right edge correctly. In the next version, the Congestion Control reports a number of credits instead so it does not have to know whether the flow is reliable or not. Output can decide whether to transmit old or new packets using these credits.
2.1.5 Timeout

The timeout modules allows an alarm to be associated with each outstanding packet. The interface for setting the timeout is:

\[
\begin{align*}
\text{void setTimeout}(\text{long seqnum, long millisec}); \\
\text{void cancelTimeout}(\text{long seqnum});
\end{align*}
\]

When the timer expires, a loss event will be generated and FeedbackAction will be triggered. A timeout should be canceled when an ack is received.

2.1.6 Output

The Output module is responsible for deciding which packets should be sent which is protocol dependent. For reliable flows, the retransmission of lost packets should take precedence over transmission of new packets. However, for unreliable flows, lost packets are simply not retransmitted. The only restriction imposed by our framework is that it should always update the send window so that the ack window is up to date. The output module has a single method:

\[
\text{void getNextPkt}(\text{PktHdr pkthdr});
\]

This method blocks until the congestion control allows at least one packet to be sent. PktHdr contains the sequence number of the packet to be sent.

2.2 Receiver

Because we make the assumption that congestion control, retransmission, and loss-detection are done by the sender, the receiver side is very simple (see Figure 4). A receive window tells the receiver the range of packets that it is expecting. This window is similar to the send window. Received packets that are outside the receive window should be dropped silently. The Input module decides whether the packet should generate an ack immediately or after some delay. In any case, the ack is generated by the output module which generates the ack using the information stored in the Receive Window object.
2.3 TCP

In this section, we want to demonstrate how to retrofit TCP into our framework. In particular, we will use TCP Reno as our example.

- **Receiver** — Receiver sends an ack for every other packet received (delayed ack). Each ack carries the largest sequence number that was received in sequence.
- **Loss Detector** — It generates a loss event on receiving 3 duplicate acks. The sequence number of the loss event is the number in the duplicate ack.
- **Feedback Action** — Update the send window pointers (snd_una, snd_max, snd_limit). Cancels timeout on receiving an ack. Reverts a timeout on a timeout.
- **Congestion Control** — Depending on slow-start or congestion avoidance, the left edge is incremented by either 1 or 1/\text{wnd\_size} per RTT. \text{Wnd\_size} is reduced to half on a loss event, and to 1 on a timeout.
- **Timeout** — TCP sets one alarm for the packet at the left edge of the send window. Therefore, to mimic TCP, Timeout module should only set a timeout for a packet with the smallest sequence number.
- **Output** — Output always picks the first lost packet, if available, to send; otherwise, it picks a new packet to send.

2.4 WebTP

WebTP also fits well in this transport framework. One special feature of WebTP is that it has separate packet sequence numbers and ADU numbers. As a result, a retransmitted ADU has the same ADU number, but different sequence number. The reason for doing so is to completely separate loss detection and congestion control which uses the packet sequence numbers from retransmission of ADUs which use the ADU number.

- **Receiver** — The feedback given by WebTP receivers is an ack number followed by an ack vector. The ack number tells the sender which packet was received correctly. The ack vector is a sequence of bits, each of which can take the value of 0 or 1. 0 indicates a lost, while 1 indicates correct receipt. When the receiver acks, it also tells the sender of the status of the n packets there are immediately before the packet it just received.
- **Loss Detector** — An ack event is generated if an ack is received with the matching ack number, or if an appropriate bit in the ack vector is turned on. A loss event is generated if there are at least 3 ack vectors that signal the same packet is lost.
- **Feedback Action** — Same as TCP’s Feedback Action.
- **Congestion Control** — WebTP also uses an additive increase and multiplicative increase scheme for congestion control similar to that of TCP. TCP keeps track of a congestion control window which moves only when the packet at the left edge has been acked. In WebTP, because packets are not retransmitted, the left edge stops at the first packet unacked and not lost. WebTP then calculates the number of extra packets that can be sent by adding the congestion control window to the left edge, then subtracting the number of outstanding packets (unack’d and not lost).
- **Timeout** — WebTP sets one timer per packet so that packets can timeout more quickly in the case of multiple losses. This is easily done by setting a timeout every time a packet is sent.
- **Output** — Output always sends a data packet with a new sequence number and so it only needs to keep track of a single counter that increments by one each time a packet is sent.
3 Software Implementation

3.1 Implementation in Java

We implemented the framework described in Section 2 in Java. Each module is implemented as an object with public methods listed above. The internal states are stored as private variables. The Sender object is a composition of all the modules on the sender side while the Receiver object is a composition of objects on the receiver side. The flexibility of the framework builds on the object-oriented features of Java. For example, once the TCP Sender and TCP Receiver are programmed, anyone can replace a component of protocol with their own with relative ease. For example, if one wants to experiment with Selective ACK (SACK) in TCP, she only need to define a subclass of Loss Detector on the sender side and a subclass of Output on the receiver side. These new sub-classes can implement the SACK rather than Cumulative ACK. The rest of the modules do not have to be modified because the public interface of the Loss Detector and Output still uses the same notification models for losses and timeouts, and need not be changed at all.

3.2 Simulator in Java

To make our framework more useful, we integrated it with a discrete-event Simulator developed by University of Edinburgh called SimJava [SimJava]. In SimJava, we can define our own Sender Class and Receiver Class implementing the protocol we want to investigate. In addition, we must also define some Gateway Classes so that we can form a network simulator. Each object in SimJava is a subclass of SimEntity which automatically inherits the necessary methods to send and receive messages between different objects in the simulation. In the simple setup where we want to study effects of severe ACK losses on WebTP, we can setup the simulator as shown in 5. First we setup the WebTP Sender which continuously send packets of 1500 bytes, subject to congestion control. Next, we setup a WebTP Receiver (on the right) to generate WebTP acks. In between, we created a forward Gateway class with a drop tail buffer of 10 packets, and a reverse Gateway class similar to the forward Gateway except that the link between and the Sender and the reverse Gateway incurs independent random losses with a fixed drop probability. When the simulation starts, the Sender, Receiver, and the two Gateways all run as individual threads sending messages to each other. Using multiple threads in a simulator is not the most most efficient approach in discrete-event simulator but it is much easier to program than purely event driven simulators. With the use of Java Just-in-Time (JIT) compilers from Sun[SunJDK], the performance of SimJava appears reasonable. We were able to simulate 20 seconds of virtual time with the sender sending about 50,000 packets in less than 10 seconds of real time even with exhaustive tracing of events at every hop. For simple simulations, the time to program and debug the simulator code dominates the actual running time of the simulation.
4 A Simple Loss Detection Protocol

4.1 Loss Detection

One of the most difficult problems in separating loss detection from the rest of the protocol is that existing transport protocols (e.g., TCP) providing loss detection also assume that packets will be retransmitted. We must therefore find a new loss detection protocol that does not assume packets are retransmitted. Next, we propose such a new acknowledgment scheme using the composite architecture and the simulator framework described above. Below, we describe some preliminary results of the performance of the ack scheme and experiences we have using the composite architecture.

The ack scheme that WebTP uses is a combination of positive ack and selective ack. When the receiver receives a packet with sequence number \( N \), it returns an ack for the packet just received containing the sequence number of the received packet, together with a vector of \( k \) bits that represents the status of the \( k \) packets immediately before the one received in the sequence number space. Bit \( i \) of the vector is set to 1 if the packet \( N - i \) has been received and zero otherwise. Therefore, this scheme provides redundant information about the status of previous packets.

WebIP detects losses in three ways. In the first method, if an ack vector contains a gap in the ack bits, the gap is remembered as a hint of loss and pushed onto a FIFO of these hints. If a set number (currently 3) of subsequent ack vectors suggest the same packet is lost, the FIFO overflows, and we infer that this packet is lost. The second way of detecting loss is that if an ack for packet \( X \) is received, then any outstanding packets with sequence numbers smaller than or equal to \( X - C \) are considered lost. \( C \) is a constant (currently 3) that bounds the degree of reordering in the network. This covers the case when there is heavy ack loss in the reverse path and hence we might not receive enough ack vectors with gaps to infer losses quickly. The third way of detecting loss uses a timeout per packet outstanding. When a packet is sent, a timeout is set for the packet. If a packet is received, the timeout is canceled. When the packet timeout occurs, the packet is inferred to have been lost.

4.2 Congestion Control

Congestion control in WebTP is very similar to that of TCP. There is a slow-start phase at the beginning of a flow, followed by a congestion avoidance phase. TCP uses a congestion control window to bound the number of packets that can be outstanding at any time. WebTP also uses a window, but the window semantics are slightly different for reasons explained in section 2.1.1. In WebTP, a packet is allowed to leave the window if either it is acked or lost. WebTP congestion control algorithm is modeled after the TCP one. In the slow-start phase, the window increase by 1 packet per ack received. The receiver sends 1 ack per packet received, so the window size grows exponentially, until the first packet is dropped. The window size is halved and congestion control enters the congestion avoidance phase. In this phase, window size grows by one packet per RTT. The simulation setup is similar to that shown in 5, except that the delay imposed by gateway is 5ms instead of 50ms plus queuing delay. The retransmission timeout is set to be 1.0 second. In figure 6, we can see the window evolution of WebTP. As seen from the figure, the slow start phase in the first several RTT quickly ramps up the window to about 24 packets, and drops by half and enter into congestion avoidance phase. During the entire simulation period of 20 seconds, the window repeatedly increases the window linearly and halves it later, maintaining a steady flow of data. Losses are detected quickly after the router drops a packet.

4.3 WebTP Behavior under Ack Losses

In this section, we want to investigate the performance of our ack vector scheme in the cases of ack losses. Cumulative acks used in TCP are fairly robust to ack losses because each subsequent packet carries a superset of information contained in a previous ack. Therefore, occasion ack losses have little impact on losses. In
WebTL, cumulative losses cannot be used because the packets are not retransmitted with the same sequence number. As a result, we must consider the performance of the ack scheme under ack losses.

In our simulation, we generated ack losses by simulating a lossy link between the reverse gateway and the sender. Acks are dropped independently with a fixed probability. When the ack drop probability is at 10% (fig. 7), WebTP can easily recover from the ack losses and there is very little impact on the throughput. WebTP relies entirely on fast loss detection methods using the ack vector gaps, without resorting to the more costly timeouts.

However, when ack loss probability increases to 25% (fig. 8), not all packet losses are detected through fast loss detection mechanisms. When timeout happens, the throughput of the WebTP flow suffers. For example, when timeout happens around 13.5 seconds, the flow sends only 1 packet per RTT for several seconds. Figure 9 summarizes the effects of ack losses on the overall throughput of WebTP. The throughput is normalized to the link capacity. With losses below 15%, the effect of ack losses on throughput is negligible. When ack loss probability increases to 25%, the throughput drops to only about 70% of the link capacity. Beyond 25% losses, the throughput of WebTP drops significantly. We can therefore conclude that the ack vector scheme is quite resilient to ack losses up to 15% which is fairly heavy in today’s Internet. At higher losses however, the effectiveness of fast loss detection decreases because it takes several acks with the same gap to detect that a packet is lost. As the ack loss ratio increases, there are more cases where fast loss detection does not have enough acks to infer losses. Coarse grain timeouts are much more expensive not only because it takes a long time to detect the loss, but also because when it timeouts, it resorts to slow start.

Figure 10 shows the packet loss ratio as the ack loss probability increases. It also shows the break down of losses detected using fast loss detection and by timeouts respectively. When less than 20% of acks are lost, most packet losses are detected by fast loss detection. At higher ack loss probability, the effectiveness of fast loss detection decreases and more timeouts are incurred. Nonetheless, even with very high ack losses, the amount of data packets dropped in the forward direction remains small (<1%). The increased burstiness of the sender does not increase the packet loss significantly.
Figure 7: Behavior of WebTP under 10% ack losses

Figure 8: Behavior of WebTP under 25% ack losses
An interesting artifact of figure 10 is that the packet loss ratio increases as the ack loss ratio increases. The reason is that when acks are lost, subsequent acks carry information about the delivery of multiple packets. This causes the increase of congestion window to be less smooth. In our simulation, packets are sent as soon as the right edge of the congestion control window moves. Therefore, if the right edge moves in large jumps, the sender becomes more bursty and causes more drops to happen. One possible solution is to pace out the sending of packets at the rate of $\frac{1}{\text{send} \times RTT}$.

5 Conclusion

5.1 Future Work

One possible advantage of ack vector over cumulative acks and selective acks is that it stores information of received packets in a highly compressed form. The sender can easily reconstruct the sequence in which the packets arrive at the receiver. It allows the sender to detect possible network reordering and avoid mistaking reordered packets as signs of losses. We hope to investigate the performance of ack vector with packet and ack reordering. We also want to investigate the effects of different ack vector lengths and different loss detection algorithm parameter. Our framework currently only support window-based congestion control. For multimedia applications, it is beneficial to extend our current framework to include rate-based controls. Finally, we are in the process of comparing the performance of TCP versus WebTP using simulation.

5.2 Related Work

Congestion Manager[BR99] is an architecture for explicit congestion management. It allows different protocols and different applications to share congestion and bandwidth information to achieve integrated congestion control across different flows to the same destination. Congestion Manager relies on application to detect
Figure 10: Effects of ack losses on packet losses

losses on their own and report them to the CM which is different from our goal.

TCP SACK [MMFR96] is an option in TCP to use a different acknowledgement scheme that include acks blocks for continuous bytes in addition to the cumulative ack number. This scheme is similar to our ack vector except that our scheme is more suitable for losses that are non-consecutive. Because SACK blocks store ranges of sequence numbers in TCP option header which is limited to 20 bytes, our scheme is more compact.

TCP Friendly Rate Control [FHPW00] is a congestion control mechanism by which a unreliable flow changes its sending rate according to the measured packet loss rate and round-trip time. It does separate congestion control from ARQ, but it does not provide application with a feedback as to which packets are lost. Loss detection is done at the receiver, and only a loss rate is transmitted back to the sender. The sender does not know which packets are lost or received.

5.3 Summary

Loss detection, automatic retransmission, and congestion control are tightly coupled in monolithic transport protocols such as TCP. In this paper, we present a unifying framework for a modular implementation of these functions. Our framework proves that it is possible to separate these functions and allow them to operate independently. A modular design allows individual algorithms to be replaced without affecting the rest of the protocol. Another advantage is that application can specify whether to retransmit a particular packet and the order in which they are transmitted without having to write the code to detect losses or do congestion control. Separating loss detection from ARQ and congestion control poses a new challenge not present in TCP, namely the ability to detect losses in an unreliable stream. We propose a new ack scheme that uses a combination of positive ack and ack vectors to solve the problem. Preliminary results show that ack vector is capable of detecting losses reliably and quickly without timeouts for ack losses of up to about 20%. At higher ack loss rates, throughput drops due to increased timeouts and burstiness of sender. It might be
possible to monitor ack losses in the reverse direction to dynamically adjust the sensitivity of loss detection to sequence number gaps. Overall, the modular framework enables us to easily experiment with different algorithms for different components. It also allows us to re-use components across different protocols. The modular approach is promising for those who need to design transport protocols that tailor to their specific needs in short time.

5.4 Acknowledgments

The work is in collaboration with Jing Lung. More details on congestion control algorithms can be found in Lung’s thesis [Lun00]. The author would also like to thank Dr. Jean Walrand for his patience and support.

References


