Trading Reliability for Responsiveness in Remote Desktop

Somak Das
MIT CSAIL

Joshua Ma
MIT CSAIL

Abstract

Current versions of VNC are not resilient to round-trip time, packet loss, and bandwidth. We design and evaluate Octopus, a fork of TightVNC that addresses these issues by implementing server push, transport over UDP, and adaptive quality. The result is a significantly more responsive user experience. Octopus is able to maintain a steady frame rate with RTT over 500 ms and, under loss-prone conditions, improves response time 10× over standard VNC.

1. INTRODUCTION

Remote desktop enables a user to remotely view and control a computer's desktop screen in real-time. It is used by system administrators to manage computers, software engineers to collaborate on projects, and even doctors to control MRI scanners.

Virtual Network Computing (VNC) is a popular program that implements remote desktop. We attribute VNC's success to its use of a general, platform-independent protocol (Remote Frame Buffer) to synchronize server and client screens. RFB simply sends updates as pixel data over TCP. It relies on a *client pull* model, in which the server waits for an update request, sends an update, and then waits for the next update request. As a result, VNC is not resilient to high round-trip times, since the refresh rate (frame rate) is bounded above by 1/RTT. In addition, its dependence on TCP means that it responds poorly to packet loss, and it is unable to adapt to varying bandwidths since the client sets a fixed quality level.

Our contribution to persistently update the screen is ***Octopus***. It is a fork of TightVNC that improves responsiveness by actively pushing updates to the client, sending them over UDP, and adapting quality based on bandwidth. These features address the issues of RTT, packet loss, and bandwidth, respectively. We achieve better frame rates and response times than the original TightVNC, especially under high-RTT and loss-prone conditions. Our experience from Octopus suggests that networking performance of VNC (and real-time, interactive programs in general) can be dramatically enhanced with a better networking protocol.

2. BACKGROUND: VNC

VNC naturally has a client–server architecture. For each client, the VNC server tracks the areas of the screen (in terms of x,y coordinates) that have changed since the last update, called the modified region M. When it receives an update request, it reads the current pixels at M and sends them to the client, and resets M. When the client receives the update, it overwrites the areas of the screen with the new pixel data. Then it sends the next update request. Any keyboard/mouse input is also sent to the server.

Each update consists of a number of rectangles annotated with x,y coordinates, width and height, and pixel data. There exists different encodings for the pixel data, since sending it raw is too expensive. Examples include RLE, run-length encoding; ZRLE, RLE with `zlib` compression; and Tight, an encoding scheme that incorporates RLE, `zlib`, and other optimizations. TightVNC later augmented Tight encoding to include JPEG compression, and it exposes the degree of compression (more lossy vs. less lossy) as a parameter.

3. RELATED WORK

Round-trip time: For high-RTT conditions, most work has focused on generating extra update requests. The VNC-HL client tries to emulate server push by sending an update request every time interval T (e.g. 50 ms) [1]. Thus, it expects to receive an update every

T, regardless of RTT. The VNC-HL evaluation consists of the user passively watching a video (even though VNC doesn't support audio), and the authors do not evaluate how recent the received updates are. We address this issue in our evaluation (§5).

The Message Accelerator is a proxy between the server and client that sends extra update requests, similar to VNC-HL [2]. It then puts received updates into its own buffer and eventually forwards them to the client. Instead of emulating server push, our solution is to actually implement it. This way, the client does not send update requests when there are no updates from the server (supply-driven, not demand-driven).

Packet loss: All VNC distributions depend on reliable transport like TCP; when packet loss occurs, it keeps trying to retransmit the same update. In most variants, TCP also decreases its sending rate, slowing down recovery. Many VNC users note that packet loss causes the screen to hang and eventually disconnect [3]. Our evaluation shows that while VNC hangs (in some cases, for 10+ seconds), our system always recovers within one second.

Bandwidth: Since VNC is built on a pixel-based protocol, many encodings have been developed to compress the updates, which are essentially images. For example, TightVNC adapts JPEG compression for low-bandwidth conditions. Still, users must manually tune for the right encoding. We combine the problem of auto-tuning the encoding quality with doing flow control for server push, because the limiting factor is the same – bandwidth.

Adaptive delivery in general: There are several lines of work around making interactive, real-time programs (like VNC in our case) adapt to network conditions. Mosh is a mobile shell that runs State Synchronization Protocol over UDP to synchronize terminal screens between server and client [4]. SSP handles loss by skipping undelivered states and implements delay-based rate control for varying RTT. Presumably, it doesn't need to handle different bandwidths because terminal updates are small. We handle loss in the same way, but our sending rate is not delay-based because we aim to deliver a RTT-independent frame rate to the user. We also handle different bandwidths. Real-time video provides another example of handling RTT, loss, and bandwidth [5].

4. OCTOPUS

Our system, Octopus, allows the RFB protocol to run over unreliable transport. We make no changes to the server and client messages aside from adding sequence numbers and acknowledgments, since TCP has those built in and UDP does not. We examine our changes to the server and client themselves in three parts: server push, transport over UDP, and adaptive quality. Each component handles a separate challenge (RTT, packet loss, and bandwidth).

4.1 Server Push

The client pull model was intended to actually give VNC an adaptive quality, since a "slower" network results in slower updates [6]. However, this couples the issues of RTT and bandwidth together, which should be separate concerns. A high RTT should not necessarily cause a slower update rate, because it discounts the option to have multiple updates be in-flight.

Octopus removes update requests entirely. Instead, the server continuously pushes updates (if there are any, i.e. $M \neq \emptyset$) at an interval U. Thus, we are able to control the client's visible frame rate to be $\approx 1/U$, instead of 1/RTT. This makes our system resilient to RTT. Server push adds some challenges, because we must be careful not to send at too high a rate. We address this issue by making VNC adaptive in §4.3.

4.2 Transport over UDP

Octopus functions over unreliable transport, UDP, to improve resilience to packet loss as well as responsiveness. Conceptually, screen updates don't need to be sent over a byte stream like TCP; sending them over datagrams should work as well. If an update is lost, then the server can generate and send a newer update. This is possible because the client simply overwrites regions in its screen with the received pixel data.

However, datagrams are limited in size. The typical

maximum transmission unit (MTU) is 1500 bytes, while updates (essentially images) are often over 10–20 kB. If the network's loss rate is p and the update is fragmented into k packets, then the probability of successful delivery is $(1-p)^k$, not just $(1-p)$. Simply dividing the bytes of an update into 1 MTU packets and adding retransmissions would not help increase responsiveness – in nearly all encodings, the client cannot display the update until all the pixel data in the update is received.

Split and send: The Octopus server sends updates more intelligently by dividing updates into 1 MTU *blocks*, which can be individually received and displayed at the client. It uses a recursive method that first measures how large the update is, considering the current encoding's compression. If the measured size too large by a factor of k, we compute the bounding box of the region and split along the longer dimension into $\lceil k \rceil$ subregions.

Splitting a region into k subregions does not guarantee that each subregion is $1/k$. The encoding may do better compression in some regions than others. So, each subregion is recursively measured and split if necessary. Finally, whenever a 1 MTU block is generated, it is sent to the client over UDP.

Retransmission strategy: Since Octopus works over unreliable transport, it must have the ability to handle lost packets. To this end, we track a list of all regions that have been sent in an unacked queue (not the pixel data of the region that was sent, just the x, y coordinates that identify it). When the server sends a block with sequence number n, it puts a record containing the region, send time, and n into the queue. So, when the client receives the update, it sends an ack for n back to the server. When the server receives the ack, it deletes the corresponding region from the queue.

What if the update is lost? We use TCP's algorithm to calculate the retransmit timeout as smoothed RTT + 4 · RTT variation [7]. Suppose the region H at the head of the queue is not acknowledged within its timeout. Since the next update is computed from the modified region M, we simply union H into M, and delete H from the unacked queue. Note that the next update reflects the latest screen, not what was originally sent. Thus, we are able to retransmit the region, while TCP is only able to retransmit the stale pixel data. In our system, retransmissions often piggyback on actual updates.

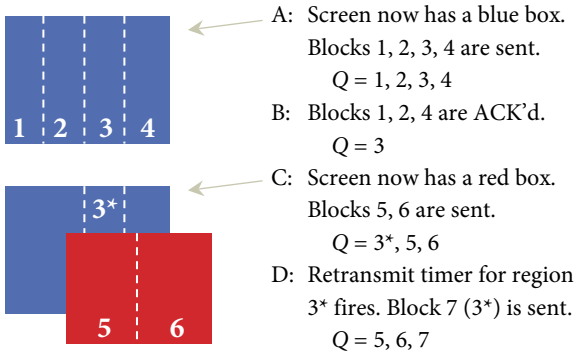

Figure 1: An example of Octopus' opportunistic retransmission strategy. Q represents the unacked queue. Region 3* is region 3 with regions 6, 7 subtracted out.

Each time the server sends an update for region R, it subtracts R from every region in the unacked queue. For example, if a region is in the queue but the next update completely covers it, the region now becomes $\emptyset$ and is removed from the queue. As a result, Octopus opportunistic in reducing the size of and eliminating possible retransmissions. Figure 1 shows another example of an update where this helps. In this case, block 3 was lost but the user opened a window obscuring most of region 3. When the retransmission timer fires, only the top part of region 3 must be sent.

Costs: Our specialized transport over UDP allows Octopus to perform well in loss-prone networks. There are a few costs, though: several encodings depend on sequential, reliable delivery of updates that our unreliable transport does not provide. The first is CopyRect, where the server instructs the client to copy a rectangle from the current location to a new location. It is useful for scrolling or moving a window, which a part of the screen just moves without changing. Since CopyRect assumes the client is at a certain state of pixel data (that we cannot describe since our sequence numbers only correspond to regions), we disable it.

Then there are encodings using `zlib`, such as ZRLE and Tight. These assume that the compressor

state is the same at the server and client due to sequential delivery of updates. This allows for better compression across multiple screen updates. Since we use unreliable transport, we cannot assume synchronized state. We considered using a stateless encoding like RLE or Hextile, but we found that Tight compresses over 20× better than them. Therefore, we made the Tight encoding stateless by resetting its `zlib` compressor after sending each block, though it results in suboptimal compression compared to standard VNC. We exclusively use the Tight encoding in Octopus.

4.3 Adaptive Quality

The Octopus server automatically tunes the quality for the client. We define quality as a combination of frame rate and encoding quality; the former gives the client a more responsive experience, while the latter provides a better image. A higher frame rate requires more updates per second, and a higher encoding quality requires more bytes per update. Hence, the quality is limited by bandwidth. Sending at a quality level too high or too low is undesirable. If it is too high, then we risk overflowing buffers; if it is too low, then we are underutilizing the network.

Since updates are typically sent in bursts of 1+ blocks with consecutive sequence numbers, Octopus uses packet pairs to maintain an EWMA-filtered estimate of the bandwidth (or more accurately, the bottleneck service rate along the path) [8]. The server gets an estimate by tracking when blocks are acked: it divides the size of block n by the interarrival spacing for blocks n and $n-1$. To avoid using consecutive blocks from separate updates for the estimate, we increment the sequence number by 1 before a new update.

Octopus also tracks its own sending rate by measuring the number of bytes sent over time. It compares the sending rate to the estimated bandwidth to make decisions about adapting the quality. Octopus tries to keep the sending rate between 90% and 100% of bandwidth. The quality is raised (lowered) if the sending rate is too low (too high).

When the quality needs to be raised (lowered), we decide whether to improve frame rate or encoding quality based on which one is currently worse (better). This decision depends on a scoring function, and we simply normalize frame rate (which can vary from 1 to 24 frames per second by changing U) and encoding quality (which can vary from Tight JPEG quality 1 to 7) into the interval [0,1]. For example, both 12 fps and Tight JPEG quality 4 have a score of 0.5. Lastly, we then adapt the chosen component by additive increase or decrease (specifically, ± 1 for encoding quality, ± 3 fps for frame rate).

5. EVALUATION

We implemented Octopus within TightVNC, server v1.3.10 (March 2009 release) and client v.2.7.0 (April 2013 release) [9]. The server is a Linux X server written in C89. Interestingly, that means that VNC on Linux does not show the physical desktop; instead, each running server starts a new virtual X desktop. The client is a platform-independent viewer in Java. In total, the program is 517,000 lines of code. To this, we added 1,300 lines to replace VNC's networking model (client pull over TCP) with ours (server push over UDP).

In our testing environment, the server ran on an Linux Mint desktop computer in Back Bay, Boston (TCP variant: CUBIC). The client ran on a Windows laptop computer at MIT (TCP variant: Compound). The unmodified network conditions were 1 ms RTT, 11 Mbps bandwidth, and <0.01% loss. We artificially modify the network conditions on ingress and egress traffic using `netem` on the server. Note that we varied each parameter individually (e.g., 20% loss with 1 ms RTT, not 20% loss with 100 ms RTT) so that we can observe the individual effect of each. We froze the U parameter (update interval) of Octopus to 50 ms.

Metrics: We quantify performance in two ways. One metric is the standard *frame rate* (in fps) while the user is watching a video. This is calculated as the number of updates received over time. However, since Octopus splits updates into blocks and there is packet loss, we also define the *partial frame rate* where we count the fraction of each update received (if it is at least 0.5) over time.

But while video allows measuring the frame rate, it

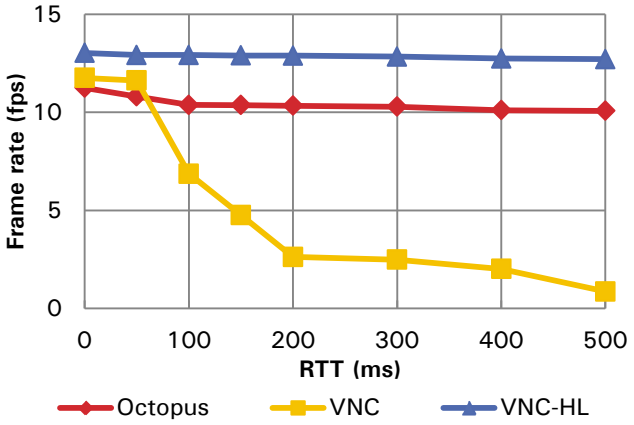

Figure 2: Average frame rate, varying RTT.

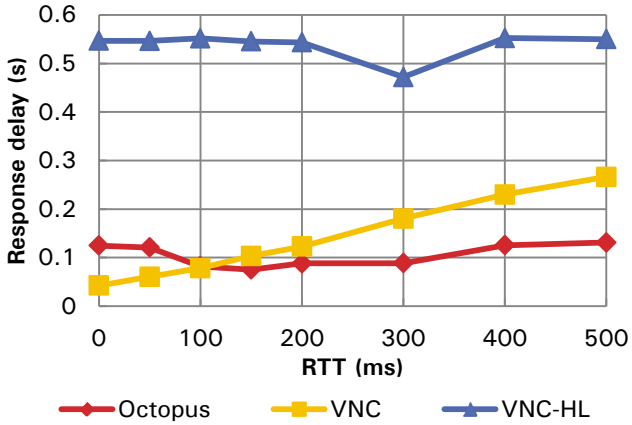

Figure 3: Average protocol-induced response delay.

is not a real use case for VNC (which doesn't have audio). So, the other metric is new for VNC studies: keyboard/mouse input event *response time* during a typical user session. We define this as the time between when the client sends an input event and when it receives a whole update incorporating the results of that event. Since the response time must take at least one round trip, we actually report the protocol-induced *response delay* (response time minus RTT) to measure the underlying protocol's effect.

To ensure repeatability, we recorded two tasks to measure frame rate and response time using ReMouse keyboard/mouse recorder, and then replayed them in real-time for each experimental run (3 runs per measurement) [10]. For frame rate, the user passively watched an OpenCourseWare video for 1 minute. For response time, the user actively worked with the terminal, word processor, and calculator for 3 minutes (1040 keyboard/mouse events).

5.1 Round-Trip Time

We tested RTT from 0 to 500 ms. We also implemented the VNC-HL method [1] to compare with Octopus, since it was designed for high-latency networks. The VNC-HL client sends an update request every T, and so it expects the VNC server to respond with an update every T. Since $U = 50$ ms, we also set $T = 50$ ms.

Shown in Figure 2, both Octopus (10.5 fps) and VNC-HL (12.8 fps) maintain a steady frame rate as RTT increases. As expected, VNC's frame rate degrades with increasing RTT. There is a performance gap between Octopus and VNC-HL and Octopus and VNC for <100 ms RTT due to the extra work Octopus does before sending updates (§5.4).

However, VNC-HL is not the ideal solution. Figure 3 shows that it always induces a response delay of 0.5+ s. This is because it fills the client's send buffer with update requests. It sends over 4440 update requests in the 3-minute client trace, which only has 1040 input events to send. As a result, the input events are queued behind many update requests, so the server is forced to respond to the update requests before receiving the (more important) input events. The VNC response delay increases with RTT because its frame rate decreases with RTT. Since the VNC server does not send an update until receiving an update request, it is slower to reply to the client. Since Octopus is agnostic to RTT, it has the best of both – it is fast to receive input events and fast to reply to the client. It consistently induces a response delay of only 0.1 s.

5.2 Loss

We tested loss rate from 0% to 30%. In Figure 4, VNC's frame rate degrades quickly, falling below 1 fps for loss rates greater than 10%. But Octopus maintains a steady frame rate around 10.8 fps. The partial frame rate decreases with increasing loss rate, but less so than VNC's. Interestingly, theory and reality match very

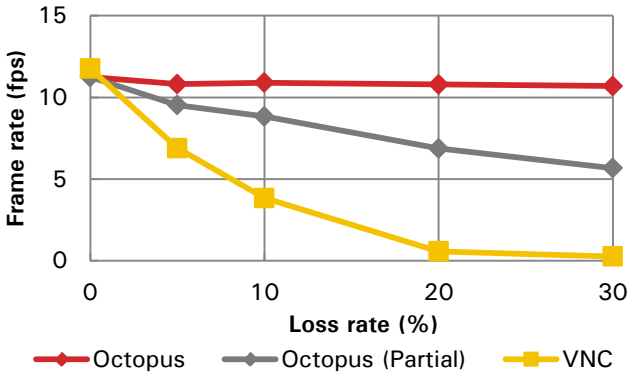

Figure 4: Average frame rate, varying loss rate.

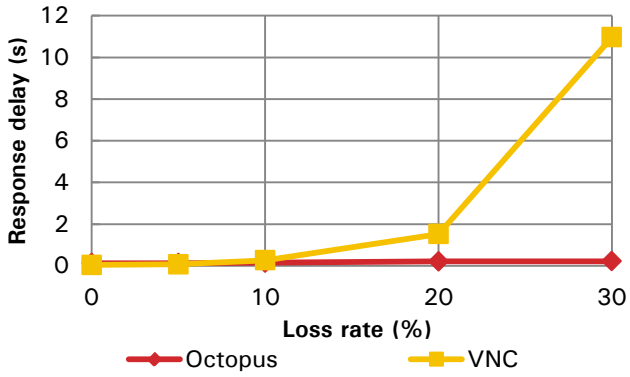

Figure 5: Average protocol-induced response delay.

well: we observe that Octopus' partial frame rate is proportional to $1 - \mathbb{P}(\text{loss})$ (packet success rate, correlation $R^2 = 0.97$) while VNC's frame rate is proportional to $1/\sqrt{\mathbb{P}(\text{loss})}$ (inverse square root of packet loss rate, just as in the TCP throughput formula, correlation $R^2 = 0.98$).

Octopus makes a case for *application-layer framing*, as opposed to only transport-layer resilience to loss. The TCP variant Westwood+ addresses non-congestive packet loss, but we found that it only achieves 2.7 fps for 30% loss. It is 10× better than CUBIC, but still less than half of Octopus' partial frame rate. Octopus' standout property is the ability to fast-forward the client to the server's current screen, which is only possible with unreliable transport. Figure 5 shows that while VNC's response delay decays to several seconds with increasing loss rate, Octopus always has a response delay less than 0.2 s.

5.3 Bandwidth

To analyze Octopus' ability to adapt the quality, we tested bandwidth from 100 kbps to 10 Mbps using the video task. Since the bandwidth was held constant, the measured bandwidth, sending rate, and quality converged to the values shown in Table 1.

Our packet pair measurements tracked the bandwidth allotted by the network relatively accurately (mean absolute error: 15%). Sending rate increased with bandwidth, along with an increase in encoding quality and frame rate.

Bandwidth (Kbps)	Estimated bandwidth (Kbps)	Sending rate (Kbps)	Encoding quality	Frame rate (fps)
100	116.9	330.9	1	1
500	511.5	455.3	4	4
1000	899.6	858.1	6	11
5000	6761.1	1207.3	7	24
10000	11415.4	1284.1	7	24

Table 1: Adaptive quality, varying bandwidth (red is minimum quality; blue is maximum quality).

5.4 Costs

Octopus uses more CPU time on the server because it uses the compressor many times per update. First, it first compresses the update to measure its size. In our current implementation, we recompress the update a second time to actually send it, if it fits in 1 MTU. If it does not, we split the region, compress the subregions a third time to measure, and recompress a fourth time to send the blocks. While it is possible to avoid recompressing the update when sending by saving the results of measuring, compressing both the regions and subregions cannot be avoided.

To quantify this overhead, we tested the 1-minute video task with Octopus and VNC (same frame rate and Tight encoding). Octopus spent 2.38 s compressing, 3.5× more than VNC's 0.69 s. Given the 4 rounds of compression required on a typical large update, this is expected. However, Octopus sent only 6% more bytes even though it resets compressor state on every block sent, so that cost is rather small.

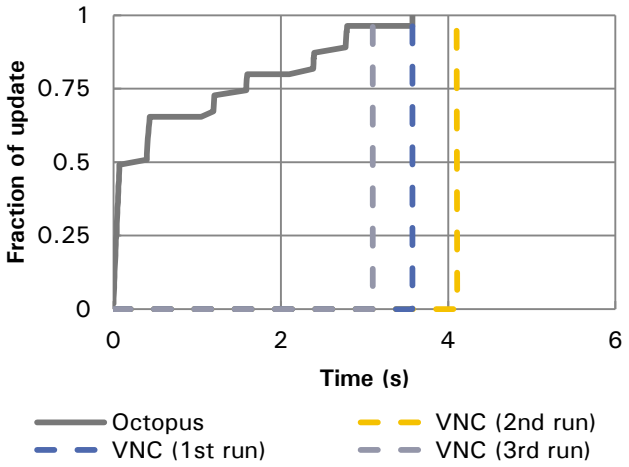

Figure 6: Fraction of large update delivered over time.

6. CONCLUSION

Our performance evaluation demonstrates that Octopus does improve the responsiveness of VNC under varying network conditions. With large enough RTT and loss, the performance of standard VNC degrades to the point where it ceases to respond while Octopus continues to deliver usable performance.

We believe that Octopus' partial updates are a step forward in responsiveness. Figure 6 shows the average fraction of a large update that the client receives over time on a loss-prone (30%) link. In this test, we opened the start menu under maximum JPEG quality to generate the 55 kB update. Under VNC, the entire update needs to be received before displaying it, resulting in a delayed but all-at-once update. In Octopus, the blocks are sent and displayed independently, so the end user is able to see parts of the new screen as they arrive. This makes for a more responsive user experience, because the user is able to immediately interact with the visible part of the start menu. If she does, then VNC will cancel the irrelevant retransmissions.

Overall, Octopus makes remote desktop with VNC more resilient to RTT, packet loss, and bandwidth. Our implementation of server push, transport over UDP, and adaptive quality within TightVNC allow for a more responsive user experience. Unlike the standard VNC, Octopus is able to maintain a steady frame rate with RTT over 500 ms and, under loss-prone conditions, keep response time under 0.2 s. To our knowledge, it is the first VNC fork to use unreliable transport. Future work for Octopus includes making it more congestion-aware and TCP-friendly.

Acknowledgments

We thank Keith Winstein, Anirudh Sivaraman, and the 6.829 staff for helpful comments on this work.

7. REFERENCES

[1] T. Tan-atichat and J. Pasquale. VNC in High-Latency Environments and Techniques for Improvement. In *GLOBECOM*, 2010.

[2] C. Taylor and J. Pasquale. Improving Video Performance in VNC under High Latency Conditions. In *CTS*, 2010.

[3] Examples: http://forum.ultravnc.info/viewtopic.php?t=9676; http://www.realvnc.com/pipermail/vnc-list/2009-March/059691.html; http://www.realvnc.com/pipermail/vnc-list/2007-February/057038.html

[4] K. Winstein and H. Balakrishnan. Mosh: An Interactive Remote Shell for Mobile Clients. In *USENIX ATC*, 2012.

[5] N. Feamster. Adaptive Delivery of Real-Time Streaming Video. M.Eng. thesis, MIT, 2011.

[6] VNC – how it works. http://virtuallab.tu-freiberg.de/p2p/p2p/vnc/ug/howitworks.html

[7] V. Paxson, M. Allman, J. Chu, and M. Sargent. Computing TCP's Retransmission Timer. RFC 6298, 2011.

[8] S. Keshav. Packet-Pair Flow Control. In *IEEE/ACM Transactions on Networking*, 1994.

[9] TightVNC. http://www.tightvnc.com/

[10] ReMouse. http://www.remouse.com/