Enabling High-Performance Internet-Wide Measurements on Windows

Matt Smith and Dmitri Loguinov*

Department of Computer Science and Engineering
Texas A&M University, College Station, TX 77843, USA
{matt, dmitri}@cse.tamu.edu

Abstract. This paper presents analysis of the Windows kernel network stack and designs a novel high-performance NDIS driver platform called IRLstack whose goal is to enable large-scale Internet measurements that require sending billions of packets and managing millions of outstanding connections on inexpensive commodity hardware available to any research lab. Our results show that with just 75% of one modern CPU core, IRLstack can saturate a gigabit link with SYN packets (i.e., 1.48M pps) and achieve 3.52 Gbps (i.e., 5.25 Mpps) with a quad-core CPU. IRLstack's transmission performance exceeds that of Winsock by a factor of 92-174, batch-mode WinPcap by a factor of 4.7-6.7, and the latest optimized PF_RING/TNAPI Linux kernel by up to 30%.

1 Introduction

With the expansion in size and popularity of the Internet, many distributed applications now require high-performance network stacks to sustain the scalability demands of their users. Traditional domains that exhibit a significant network burden in terms of bitrate and packets per second (pps) are massive Internet services with hundreds of millions of active users (e.g., Google, Facebook, Blogspot, root DNS servers, CDNs), whose main approach to solving scalability issues has been to acquire vast server clusters and distribute incoming requests across multiple geographic datacenters. While scaling the server side of network applications in commercial applications has a well-established solution, researchers often face scalability problems from the *client* side (i.e., issuing rather than receiving requests) and often do not have the resources to deploy dedicated clusters to conduct their Internet measurements. To overcome this problem, we investigate scalability issues arising during Internet-wide experimental studies, explore network-stack bottlenecks in the most-commonly deployed OS in the Internet (i.e., Microsoft Windows), and propose a solution that enables large-scale network measurements using a single inexpensive Windows host.

Due to its ever-growing size, diversity, decentralized nature, and enormous amount of information, the Internet is becoming more of a mystery every day (e.g., even Google does not know how big the web is [16]). Many Internet studies

*Supported by NSF grant CNS-0720571.

aim to shed light on its structure [2], [13], user behavior [14], [17], host availability [6], [11], and web content [7], [10]; however, accurately capturing Internet-wide metrics has long remained a challenging research problem. The main tradeoff involves the amount of available hardware and the delay the user is willing to tolerate. In many cases, measurements over a longer period of time are less desirable as they skew the obtained result, delay the corresponding analysis, and potentially impede future research. To provide additional motivation for developing large-scale measurement platforms, we next outline several of our projects that have experienced network-stack bottlenecks and then present our solution.

Our first project [17] involves measurement of P2P networks and modeling of various system properties (e.g., churn, lifetimes, topology) based on the obtained results. This process relies on a Gnutella crawler that contacts all alive ultra-peers in the system and obtains their neighbors via special requests. In order for the measurement to be unbiased [15], it is highly beneficial to capture Gnutella snapshots instantaneously; otherwise, a crawl of duration T samples a superposition of multiple Gnutella networks that exist during interval $[0, T]$. Given approximately 1.2M ultra-peers, connection rates on the order of 200K/sec are needed to guarantee cover times that would approximate an instantaneous snapshot (i.e., 10 seconds or less). While [17] was over 18 times faster than any previous P2P crawler, its coverage delay of 3 minutes could use a lot of improvement; however, various bottlenecks inside the Windows kernel leave no room for much speedup.

Our second project [7] is a high-performance web crawler IRLbot, whose main requirement has been keeping CPU utilization of the network stack close to 0% in order to leave room for computationally expensive processing related to spam control, HTML parsing, page decompression, calculation of domain reputation, and checking for duplicates. With one CPU core almost entirely dedicated to networking, IRLbot is usually CPU-limited during its crawls. Since Winsock does not scale very well to multiple cores (see below), achieving very high download rates is almost infeasible with a single host.

Our third project studies the DNS infrastructure for Internet-wide delay measurements [8] and various botnet-related anomalies, which requires traversing the DNS tree with over 650M DNS requests. Sending such a large number of small packets presents a problem for Winsock and limits the duration of the measurement to days instead of minutes. A slightly different, but related, measurement goal that requires high pps sending rates is discovery of open services using horizontal scanning [1], [3], [6], [11], where each IP address in the IANA (3.3B destinations) or BGP (2.1B) space is probed with a packet on a given port. Instead of using months to scan the Internet as in prior work [1], [3], [6], [11], our goal in another ongoing project is to accomplish this activity in several hours/days.

Other applications that are enabled by a scalable network stack are various Intrusion Detection Systems (IDS), firewalls, software routers, and network monitoring tools, all of which require line-rate capture of incoming packets and

sometimes certain processing on the fly. Leaving as much CPU as possible for processing and not dropping any packets are both of critical importance.

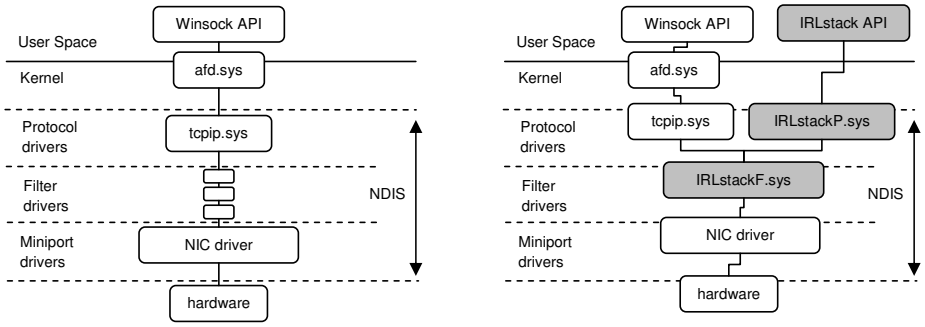
The novelty of this work lies not only in our approach to designing a high-performance *client-side* rather than server-side network stack, but also in our tackling of this problem in *Windows*, which has not been attempted before (see [4], [5], [12] for Linux approaches). The benefit of using just the client side of TCP is that it requires minimal functionality of performing the SYN handshake and sending one request packet, without tedious congestion-control functionality, management of complex timers and buffers, and retransmission overhead. As a result, a well-designed TCP stack can function at wire speed. The benefit of using Windows lies in its wide range of powerful APIs, outstanding support (in terms of software and hardware), and more ubiquitous deployment opportunities (i.e., finding a Windows host to conduct measurements is simpler than a Linux host, especially at remote locations). As there is a general perception that Windows is too slow for serious high-performance research work, we aim to dispel this myth and provide researchers with an additional platform option.

2 Overview of Windows and Linux Network Stacks

The structure of the Windows networking stack is illustrated in Fig. 1(a). Application packets are transmitted through a Winsock API into the kernel driver afd.sys whose main purpose is to manage the socket interface and interact with *protocol drivers* inside NDIS. Most normal Winsock exchange takes place with the default TCP/IP protocol driver tcpip.sys. Packet buffers created by TCP/IP are then sent down the stack to any *filter drivers* that are registered in the stack, which may do additional processing and/or filtering. The last step of this chain are *miniport drivers*, which are specific to each NIC and whose purpose is to directly interface with the hardware, set up DMA transfers, process interrupts, and manage the assigned adapter. Once the miniport has sent the frame (or queued it internally) and no longer needs the structure it received, it issues a callback up the stack indicating completion of the request, which causes the corresponding protocol driver to notify the user-space caller of the completion of their request. This process, described in terms of the synchronous send path, is similar on the receive side and for asynchronous operations.

Besides Winsock, network applications can use WinPcap [9], which is a popular tool for network capture and transmission on Windows. It is implemented as a filter driver with an API directly exported to a user-space library. Since it is located below tcpip.sys inside NDIS, it handles raw link-layer frames and bypasses most of the Windows network stack, which in theory should enable it to perform significantly faster than standard Windows sockets. However, as we will show in Section 3, this is not the case in practice.

The third alternative is a highly optimized Linux network stack such as the one developed by the ntop project [5] (the default Linux performance is lower and not studied here). Ntop makes use of a custom Linux kernel and modified network adapter drivers to exploit the features of the latest NICs. The first



(a) Winsock (b) IRLstack

Fig. 1. Windows network stack, NDIS, and IRLstack.

main modification is PF_RING, whose primary contribution is using DMA in combination with technologies such as Intel's I/OAT to directly expose kernel memory buffers (into which incoming packets have been transferred) to user-space processes. The second modification is TNAPI, which deserializes receive operations by exposing multiple RX (receive) queues as virtual adapters that can be used concurrently in user-space. This distributes load across several processors and allows the stack to scale in multi-core systems.

3 Performance of Winsock and WinPcap

Most of the issues discussed in the introduction arise from the poor *small-packet* performance of default Windows and Linux kernels. Our goal then is to achieve wire-rate transmission of arbitrary packets from user-space to many unique destinations, which translates into high rates of outgoing TCP connections/sec, fast horizontal scanning of the Internet, and low-overhead management of millions of concurrent connections to numerous remote servers (e.g., using multiple IPs aliased to the same interface with 64K ports each).

To calculate the target pps rate, we focus on gigabit Ethernet as one common example. The IEEE 802.3 Ethernet standards define the minimum frame size as 64 bytes, with smaller packets padded by the adapter as necessary. Taking into account the inter-grame gap (12 bytes), preamble (7 bytes), and the "start of frame" delimiter (1 byte), 84 bytes (672 bits) must be transmitted per minimum-size frame including overhead. We thus arrive at $1,000,000,000/672 = 1,488,095$ frames per second as the absolute upper limit for gigabit Ethernet, which is our performance goal. Using three handshake packets (SYN, SYN-ACK, ACK) and one RST for terminating connections, the absolute best performance of any TCP stack is 371K connections/sec. In applications that require graceful termination with four FIN packets, this number is 212K/sec.

Method	Destinations	Rate in pps (link utilization)		CPU
		1 port / 1 core	4 ports / 4 cores	
Winsock (default)	single	116,037 (7.7%)	193,142 (3.2%)	100%
	all unique	16,290 (1%)	30,110 (0.5%)	100%
Winsock (services off)	single	207,041 (14%)	244,196 (4.1%)	100%
	all unique	75,687 (5%)	153,232 (2.5%)	100%
WinPcap 4.1	single	49,349 (3.3%)	152,467 (2.5%)	75%
	all unique	49,493 (3.3%)	152,297 (2.5%)	75%

Table 1. Server 2008 SP2 raw SYN transmission performance.

3.1 Raw Packets

We now examine the performance of the Windows network stack to measure the maximum send rate of TCP SYN packets on a raw socket (ICMP and UDP results are nearly identical and thus omitted). All Windows tests in this paper are run on a dual AMD Opteron 2427 (2.2GHz, six cores per socket) system with 32GB of DDR2-667 RAM. The NIC is an Intel Pro/1000 PT Quad-Port Gigabit PCI-E NIC, and the OS is Windows Server 2008 SP2. We dedicate a single CPU core to each gigabit port and restrict the OS kernel, all drivers, and user-space programs to run on as many cores as there are ports being used during the test. All reported CPU utilization numbers later in the paper are relative to the number of active cores.

As shown in the first row of Table 1, Winsock can send packets to a single destination at rates between 116 Kpps (single core, single port) and 193 Kpps (quad-core, quad-port) at 100% CPU utilization. Winsock additionally drops its performance by a factor of 7 when each packet targets a unique IP address (demonstrated in the next line of the table). In order to alleviate the CPU overhead, we experimentally found that completely disabling (not just turning off) certain default Windows services (e.g., firewall and network list service) allowed Winsock to achieve a 25-80% speedup for a single destination and a five-fold rate increase for multiple destinations as shown in the next two rows of the table. However, this performance is still quite poor compared to the line rate of 1.48 Mpps and far from desirable in practice as no other processing can be done on the server due to the high CPU utilization. Furthermore, disabling critical Windows services (such as the firewall) causes installation of certain OS updates to fail and potentially leaves the host vulnerable to attack, which is undesirable. Another interesting result, shown in the last two rows of the table, is that WinPcap performs no better (and sometimes worse) than Winsock with disabled services.

As CPU usage is extremely high for the number of packets sent for both approaches above and multi-core scaling is rather poor due to various bottlenecks in the kernel, one must conclude that Winsock and WinPcap are unsuitable for truly high-performance applications.

Method	Rate (conn/sec)		CPU
	1 port / 1 core	4 ports / 4 cores	
connect/closesocket	16,656	39,462	100%
connectEx/disconnectEx	20,801	45,277	100%
WSK (kernel mode)	31,389	54,783	100%

Table 2. Server 2008 SP2 TCP connection performance to a single destination.

3.2 TCP Connections

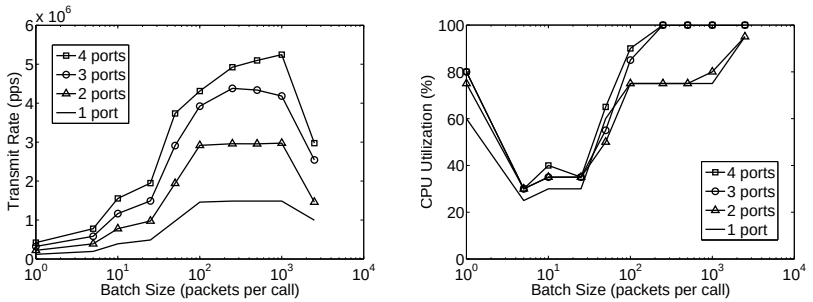
TCP connection performance to a single destination is summarized in Table 2. The standard approach using the Unix BSD socket interface (i.e., connect/closesocket) achieves between 16K and 39K connections/sec, which is slightly surpassed by the new Winsock APIs connectEx/disconnectEx with their 20 and 45K connections/sec, respectively. The performance gain is related to the fact that these APIs keep sockets open between subsequent connections. Finally, the new (i.e., Vista/Server 2008) kernel-level Winsock interface WSK is measurably faster at 31 and 54K connections/sec, but its multi-core scalability is again quite poor. Connection rates to multiple unique destinations are much worse and not shown here due to limited space.

4 IRLstack: Overcoming the Bottlenecks

Kernel stack traces indicate that the performance drop when sending raw packets to many unique destinations occurs in afd.sys and tcpip.sys in Fig. 1(a). Bypassing them completely and generating raw SYN packets entirely from within the kernel brings performance up to 289 Kpps (single-core) and 652 Kpps (quad-core), regardless of firewall settings and how many destinations are used. Nevertheless, this solution is hardly acceptable as it still consumes 100% of the CPU, stays well below link capacity, and requires writing kernel-level code for each high-performance application, which is cumbersome and prone to crashing the system.

Further profiling of NDIS shows that its path from protocol to miniport drivers in Fig. 1(a) has another major bottleneck in synchronization spinlocks and DMA transfers to the NIC. To overcome this problem, we developed a general-purpose suite of network drivers called IRLstack that accepts buffers of packets from user-space (using standard Windows API calls such as WriteFile) and transmits them in a single call to the miniport driver. Multiple outstanding asynchronous requests are supported via overlapped I/O. The buffer consists of multiple raw link-layer frames, each preceded by an IRLstack-specific header. Link-layer, IP, and TCP/UDP checksums may be omitted as they are calculated by the NIC using checksum offloading.

At the kernel level, the protocol driver scans through the buffer creating the appropriate auxiliary data structures for each encountered packet and proceeds to send the entire batch in a single call as allowed by NDIS. Batching multiple



(a) TX Performance (b) CPU Utilization

Fig. 2. IRLstack transmission performance and CPU utilization using 40-byte SYN packets.

packets maximizes useful work between acquisitions of kernel spinlocks, ensures that the send path remains zero-copy, and allows the NIC to perform large DMA transfers directly from user-space. In Fig. 1(b), protocol driver IRLstackP.sys handles raw application-layer packets through a special IRLstack API, while filter driver IRLstackF.sys intercepts return packets and channels those destined to IRLstack applications back to IRLstackP.sys. The remaining packets are sent to tcpip.sys as before. This is accomplished by redirecting any incoming traffic destined to non-default IP addresses on the NIC (assumed to be allocated for IRLstack's use) to our protocol driver.

4.1 Sending

The first issue we investigate is the transmit performance of IRLstack and the optimal batch size needed to saturate the link. Results of our testing can be seen in Fig. 2(a). All transmission rates are far from optimal until the burst size starts to exceed 50 packets, at which point IRLstack achieves 50-66% (depending on the number of cores) link utilization. For single and dual-core cases, full wire speed is reached with any batch size between 128 and 1,024 packets, while the 3-core setup has a unique peak at 256 packets and the 4-core case maxes out at 1,024. Interestingly, for very large batch sizes, performance actually drops due to bottlenecks in the Intel miniport driver, which for some reason is unable to efficiently handle large bursts of packets.

As seen in Fig. 2(b) at batch sizes 128-512, IRLstack can saturate a 1 Gbps link with just 75% CPU utilization of a single 2.2GHz core and two gigabit links using 75% of two cores. With multiple NIC ports, IRLstack scales much better than Winsock and achieves 5.25 Mpps as shown Table 3. This scaling is less than linear due to synchronization bottlenecks stemming from the common (single-threaded) miniport driver controlling all four ports, though the main sublinear dropoff only occurs when increasing from 3 to 4 ports.

Method	Rate in pps (link utilization)			
	1 port / 1 core	2 ports / 2 cores	3 ports / 3 cores	4 ports / 4 cores
IRLstack	1,487,298 (100%)	2,973,933 (100%)	4,379,999 (98%)	5,245,458 (88%)
WinPcap	319,814 (21%)	485,617 (16%)	648,370 (14%)	815,921 (14%)

Table 3. Send performance with SYN packets using optimal batch size (2.2 GHz Opteron 2427).

While WinPcap also exports a batch-mode interface to *user-space*, it does not fully utilize the interfaces provided in NDIS 5.x and later (e.g., NdisSendNetBufferLists in 6.x) for batching within the kernel. This makes its multi-packet performance significantly lower than it could be as also seen in Table 3. We thus note that IRLstack's in-kernel batching techniques could be easily implemented in WinPcap as well, benefitting those who seek higher pps performance in WinPcap-based tools on commodity PC platforms.

4.2 Receiving

While most of our projects have required high sending rates, additional research can be enabled by a network stack that allows high capture rates as well. We now turn our attention to receive performance in Table 4, where we only focus on IRLstack, with Linux PF_RING/TNAPI numbers [5] provided as a reference. (Winsock/WinPcap results are again vastly suboptimal and are thus omitted.) Observe in the table that the receive path in IRLstack is approximately 20-50% slower than the send path, which can be explained by two factors. First, our receive path is *not* zero-copy as it was during transmission, because IRLstack is able to directly export user-space buffers for DMA transfers *into* the NIC; however, no reverse functionality (i.e., *from* the NIC) is provided by NDIS unless specialized hardware is used. Second, the interrupt frequency is higher along the receive path than the send path since the miniport driver controls the former and IRLstack controls the latter. With the maximum miniport batch size equal to 64 packets, it is no wonder that it is unable to sustain the wire speed along the receive path. If future versions of Intel drivers remove this limitation, much higher receive rates are to be expected.

Nevertheless, IRLstack's receive performance compares quite favorably to the latest Linux numbers from a custom PF_RING/TNAPI kernel [5]. Specifically, both solutions achieve close to 3 Mpps with quad-cores and four independent RX queues (we use four gigabit ports, while [5] uses a single 10 GE adapter with four hardware queues). This is despite IRLstack's receive path not being zero-copy (which it is in [5] using Intel I/OAT), its use of rather frequent 64-packet interrupts, standard Intel NIC drivers, default NIC settings (e.g., adaptive interrupt moderation), and no kernel modifications (i.e., all drivers are loaded at run-time). Furthermore, while [5] posts the highest throughput numbers we've seen on Linux, it is meant for capture only and does not have a transmit path for general-purpose traffic.

Method	Rate (pps)			
	1 port / 1 core	2 ports / 2 cores	3 ports / 3 cores	4 ports / 4 cores
IRLstack	1,232,745 (82%)	1,526,460 (51%)	2,282,554 (51%)	2,946,707 (50%)
Linux [5]	$\sim$ 920,000 (61%)	–	–	$\sim$ 3,000,000

Table 4. Receive performance with SYN packets (IRLstack on a 2.2 GHz Opteron 2427 vs. Linux on a 2.4 GHz Xeon 54xx).

4.3 TCP Connections

IRLstack implements the client side of TCP in user space, which simultaneously allows for easy debugging and high-performance management of numerous outstanding connections – hiding the work of constructing link-layer frames that would otherwise be required of the user. All supported operations are performed using batching and include issuing outgoing connections with three handshake packets, ability to send requests in regular or ACK packets of the handshake (which is however not always supported by remote servers), and standard SACK TCP receiver functionality (i.e., selective ACKs, large windows, etc.). To avoid keeping the server in the time-wait state, the application has an option of terminating connections using RST packets, in which case the useful connection throughput in Gnutella-like applications is close to 250K/sec (i.e., four control packets, one request packet, one reply packet).

4.4 Latency

It should be noted that the receive-path interrupt batching provides notification from the miniport to NDIS every 64 packets and is not under control of IRLstack. The batching delay along the send path, however, is user-selectable based on the batch size passed down to IRLstack. Thus, applications that require accurate timestamps might need to trade off pps performance for lower latency by changing the miniport interrupt moderation and reducing the batch size during transmission.

5 Conclusions and Future Work

We have shown that while Windows is often overlooked as a platform on which to conduct serious networking research, perhaps due to impressions of inefficiency or low performance, this need not be the case. With a well-designed NDIS 6.x network stack, it is possible to achieve wire-rate transmission (and near wire-rate reception) on gigabit Ethernet using inexpensive commodity hardware. IRLstack achieves a nearly 100-fold increase in transmission performance over Winsock (when unique destinations are used), with lower CPU usage. Moreover, it can coexist with the default network stack on a single adapter so that other network applications may run as usual.

Future work involves expanding IRLstack's receive performance (e.g., using DMA remapping, multiple hardware queues) and evaluating its performance on 10 GE hardware.

References

1. D. Benoit and A. Trudel, "World's First Web Census," *Intl. Journal of Web Information Systems*, vol. 3, no. 4, pp. 378–389, 2007.
2. H. Chang, S. Jamin, and W. Willinger, "To Peer or not to Peer: Modeling the Evolution of the Internets AS-level Topology," in *Proc. IEEE INFOCOM*, Apr. 2006.
3. D. Dagon, N. Provos, C. P. Lee, and W. Lee, "Corrupted DNS Resolution Paths: The Rise of a Malicious Resolution Authority," in *Proc. NDSS*, Feb. 2008.
4. L. Degioanni and G. Varenni, "Introducing Scalability in Network Measurement: Toward 10 Gbps with Commodity Hardware," in *Proc. ACM IMC*, Oct. 2004, pp. 233–238.
5. L. Deri and F. Fusco, "Exploiting Commodity Multicore Systems for Network Traffic Analysis," July 2009. [Online]. Available: http://ethereal.ntop.org/MulticorePacketCapture.pdf.
6. J. Heidemann, Y. Pradkin, R. Govindan, C. Papadopoulos, G. Bartlett, and J. Bannister, "Census and Survey of the Visible Internet," in *Proc. ACM IMC*, Oct. 2008, pp. 169–182.
7. H.-T. Lee, D. Leonard, X. Wang, and D. Loguinov, "IRLbot: Scaling to 6 Billion Pages and Beyond," in *Proc. WWW*, Apr. 2008, pp. 427–436.
8. D. Leonard and D. Loguinov, "Turbo King: Framework for Large-Scale Internet Delay Measurements," in *Proc. IEEE INFOCOM*, Apr. 2008, pp. 430–438.
9. WinPcap: The Windows Packet Capture Library. [Online]. Available: http://www.winpcap.org/.
10. M. Najork and A. Heydon, "High-Performance Web Crawling," Compaq Systems Research Center, Tech. Rep. 173, Sep. 2001. [Online]. Available: http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-173.pdf.
11. Y. Pryadkin, R. Lindell, J. Bannister, and R. Govindan, "An Empirical Evaluation of IP Address Space Occupancy," USC/ISI, Tech. Rep. ISI-TR-2004-598, Nov. 2004.
12. F. Schneider, J. Wallerich, and A. Feldmann, "Packet Capture in 10-Gigabit Ethernet Environments Using Contemporary Commodity Hardware," in *Proc. PAM*, Apr. 2007, pp. 207–217.
13. N. Spring, R. Mahajan, and D. Wetherall, "Measuring ISP Topologies with Rocketfuel," in *Proc. ACM SIGCOMM*, Aug. 2002.
14. D. Stutzbach and R. Rejaie, "Understanding Churn in Peer-to-Peer Networks," in *Proc. ACM IMC*, Oct. 2006, pp. 189–202.
15. D. Stutzbach, R. Rejaie, N. Duffield, S. Sen, and W. Willinger, "On Unbiased Sampling for Unstructured Peer-to-Peer Networks," in *Proc. ACM IMC*, Apr. 2006, pp. 27–40.
16. The Official Google Blog, "We knew the web was big..." Jul. 2008. [Online]. Available: http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html.
17. X. Wang, Z. Yao, and D. Loguinov, "Residual-Based Estimation of Peer and Link Lifetimes in P2P Networks," *IEEE/ACM Trans. Networking*, vol. 17, no. 3, pp. 726–739, Jun. 2009.