Shadow: Running Tor in a Box for Accurate and Efficient Experimentation

Rob Jansen*
U.S. Naval Research Laboratory
rob.g.jansen@nrl.navy.mil

Nicholas Hopper
University of Minnesota
hopper@cs.umn.edu

Abstract

Tor is a large and popular overlay network providing both anonymity to its users and a platform for anonymous communication research. New design proposals and attacks on the system are challenging to test in the live network because of deployment issues and the risk of invading users' privacy, while alternative Tor experimentation techniques are limited in scale, are inaccurate, or create results that are difficult to reproduce or verify. We present the design and implementation of Shadow, an architecture for efficiently running accurate Tor experiments on a single machine. We validate Shadow's accuracy with a private Tor deployment on PlanetLab and a comparison to live network performance statistics. To demonstrate Shadow's powerful capabilities, we investigate circuit scheduling and find that the EWMA circuit scheduler reduces aggregate client performance under certain loads when deployed to the entire Tor network. Our software runs without root privileges, is open source, and is publicly available for download.

1 Introduction

Tor [8] is the most popular application providing anonymity for privacy-conscious Internet users. To achieve anonymity for its *clients*, Tor forwards communication between sources and destinations through a tunneled *circuit* of several volunteer *relays* located around the world. Data is encrypted using Onion Routing [11, 37] so that no single relay in the circuit can learn *both* the true source *and* the true destination of any forwarded message.

Tor's goal to provide *low-latency* anonymity for its clients has led to an enormous amount of research on topics including, but not limited to, anonymity attacks and defenses [3, 9, 13, 24, 32], system design, performance, and scalability improvements [2, 36, 40, 42, 47], and the economics of volunteering relays to the Tor network [1, 15, 28]. Most Tor research – whether implementing a new design approach or analyzing a potential attack – either requires or would benefit from access to the live Tor network and the data it generates. However, such access might invade clients' privacy or be infeasible to provide – testing a small design change in the real network requires propagating that change either to hundreds of thousands of Tor clients or to thousands of volunteer relays, and in some cases, both. Therefore researchers often use alternative strategies to experiment and test new research proposals.

Tor Experimentation. One approach for experimenting with Tor outside of the live public network is to configure a parallel private test network deployment [3, 42] either using machines at a university or a platform such as PlanetLab [7]. Since live deployments run real software over real hardware, the results are generally accepted. However, PlanetLab and other private deployments do not accurately reflect the same network conditions of the public Tor network, are difficult to manage, and do not scale well – PlanetLab has only around one thousand nodes of which roughly half are usable at any time. Therefore researchers often experiment through simulation [15, 25, 28, 33]. Simulating particular Tor mechanisms may increase scalability, but also harms accuracy: the Tor software and protocols are continuously updated by several Tor developers, causing simulators to become outdated and unmaintained. Moreover, since simulators tend not to be reused, the results of one group may be inconsistent with or can not be verified by other groups.

Tor in a Box. To increase consistency, accuracy, and scalability of Tor experiments, we design and develop a new and unique simulation architecture called Shadow. Shadow allows us to run a private Tor network on a single machine while controlling all aspects of an experiment. Results are repeatable and easily verifiable through independent analysis. Although Shadow *simulates* the network layer, it *links to and runs real Tor software*, allowing us to experiment with new designs by implementing them directly into the Tor source code. This strategy expedites the process of incorporating proposals into Tor since software patches can be submitted to the developers. Shadow is capable of simulating a *large* and *diverse* private Tor network, requiring little to no modification to the numerous supported Tor software

*Work partially performed at the University of Minnesota and the U.S. Naval Research Laboratory

versions. Shadow's focus on usability and commitment to open source software[1] improves accessibility and promotes community adoption.

Shadow is a discrete-event simulator that utilizes techniques allowing it to run real applications in a simulation environment. Real applications are encapsulated in a plug-in wrapper that contains functions necessary to allow Shadow to interact with the application. Although the application is only loaded into memory once, the plug-in registers memory addresses for all variable application state and Shadow manages a copy of these memory regions for each node in the simulation. Similar to a kernel context switch, Shadow swaps in the current node's version of this state before passing control to the application, and swaps out the state when control returns. Function interposition allows Shadow to intercept function calls, e.g. socket and event library calls, and redirect them to a simulated counterpart. As detailed in Section 3, we run Tor using these techniques and symbol table manipulations without modifying the source code.

Accurate Simulation. We validate Shadow's accuracy against a 402-node PlanetLab deployment, testing network performance using HTTP file transfers both directly and through a private Tor network deployment. Although private Tor networks on PlanetLab do not consistently represent the live Tor network well, they allow us to test our ability to model a real, diverse network (i.e. how well we can "shadow" PlanetLab conditions). We find that our results are within reason although PlanetLab exhibits highly variable behaviors because of overloaded CPUs caused by co-location and resource sharing.

To validate Shadow's ability to accurately and consistently represent the real, live Tor network, we simulate a 1051-node topology with bandwidth and relay characteristics taken from a live Tor network consensus. We model the Internet using network latency measurments taken between all PlanetLab nodes. We find that client performance in Shadow closely matches live statistics gathered by the Tor Project [45], with download time quartiles within 15 percent of the live statistics for various download sizes. Our results in Section 5 indicate that Shadow can accurately measure Tor client performance.

Improving Client Performance. Tor's popularity has lead to network congestion and performance problems. Tor's hundreds of thousands of clients [20] send data over a few thousand bandwidth limited relays, causing network bottlenecks that impair client performance. Using Shadow, we investigate scheduling as a technique to improve client performance. In Section 6 we explore the EWMA circuit scheduler [42] which prioritizes bursty circuits ahead of bulk circuits. We confirm previous results by re-evaluating EWMA when enabled on *small* bottleneck topologies consisting of three relays – similar to those tested by Tang and Goldberg. However, our results from a *network-wide* deployment of the scheduler in a scaled topology indicate that performance benefits are highly dependent on network load and a properly tuned half-life. We found that the scheduler *reduces* performance for Tor clients under certain network loads, a significant result since the EWMA scheduler is currently enabled by default for all sufficiently updated Tor relays.

[1] Shadow source code is publicly available under the GPLv3 [38, 39].

2 Requirements

Accuracy. In order to produce results that are consistent and representative of the live Tor network, Shadow should run a minimally-modified version of the native Tor software. Running the Tor software in our simulator will ensure that Tor's behavior in our simulated Tor network will closely represent the behavior of Tor running on a real machine in the live network.

In addition to running the Tor software, Shadow should also have accurate models of system-level interactions. Tor is mostly concerned with buffering, encrypting/decrypting cells, and sending and receiving large amounts of network traffic with non-blocking I/O. Inaccurate models of these mechanisms would lead to inaccurate results and measurements of Tor's behavior. Therefore, we are required to model the system-level network stack of an operating system by simulating TCP and UDP, correctly managing network-level buffers and buffer sizes, and simulating non-blocking event-driven I/O. Since a large amount of Tor's run-time is spent performing cryptography and processing data, Shadow should avoid execution of expensive cryptographic operations while instead modeling the CPU delays that would have occurred had the cryptography actually been performed.

Finally, accurate software and an accurate system will not function correctly without an accurate network. First, Shadow requires models for network characteristics including latency and reliability of network links, complex AS-level topologies, and upstream/downstream capacities for end-hosts. Second, Shadow must accurately model the network characteristics of Tor, including relay-contributed bandwidth, faithful bandwidth distributions among entry, middle, and exit relays, and geographical distribution of relays. Shadow must also incorporate network traffic from Tor clients and model accurate distributions of that traffic from live Tor traffic patterns.

Usability and Accessibility. A simulator that produces accurate results characteristic of the live Tor network will be of little use to the community without a usable simulation framework. Shadow should therefore do the following to increase usability and promote community adoption.

First, Shadow should be simple to obtain, build, and configure to allow for rapid deployment. Users should be able to run a simulation with minimal overhead and little or no

configuration. However, advanced users should be able to easily modify a simulation, generate new topologies, and configure network and system parameters. Simulation results should be easy to gather and parse to produce visualizations that allow the analysis of the network state. Second, Shadow should run completely as a user-level process on a single machine with inexpensive hardware to minimize overhead costs associated with obtaining, configuring, and managing multiple machines or clusters. Shadow should be accessible to anyone worldwide so results can be easily compared and verified.

3 Design

Shadow is a discrete event simulator that can run real applications as plug-ins while requiring minimal modifications to the application. Plug-ins containing applications link to Shadow libraries and Shadow dynamically loads and natively executes the application code while simulating the network communication layer. Shadow was originally a fork of Distributed Virtual Network (DVN) simulator [10], adding roughly 18,000 lines of code (including example plug-ins). An overview of Shadow's design is depicted in Figure 1 and details about Shadow's core simulation engine are given in Appendix A.

Shadow dynamically loads plug-ins and instantiates virtual nodes as specified in a simulation script. Communication between Shadow and the plug-in is done through a well-defined callback interface implemented by the plug-in. When the appropriate callback is executed, the plug-in may instantiate and run its non-blocking application(s). The application will cause events to be spooled to the scheduler by executing a system call that is intercepted by Shadow and redirected to a function in the node library. The interceptions allow integration of the application into the simulation environment without requiring modification of application code. Virtual nodes communicate with each other through a virtual network which spools packet and other network related events to the scheduler. Each virtual node stores only application-specific state and loads/unloads the state as necessary during simulation execution. We now describe the main architectural components that enable Shadow to realize the above functionality and fulfill our design requirements discussed in the previous section.

3.1 Simulation Script

Each simulation is bootstrapped with a simulation script written in a custom scripting language. This script gives the user access to commands that allow Shadow to dynamically load multiple plug-ins, create and connect networks, and create nodes. Valid plug-ins are loaded by supplying a filepath while parameters such as latency, upstream and downstream bandwidth, and CPU speed are specified by either loading a properly formatted CDF data file or generating a CDF using a built-in CDF generator. Hostnames may be specified for each node and are otherwise automatically generated to facilitate support for a Shadow name service. The script also specifies which plug-in to run and when to start each node.

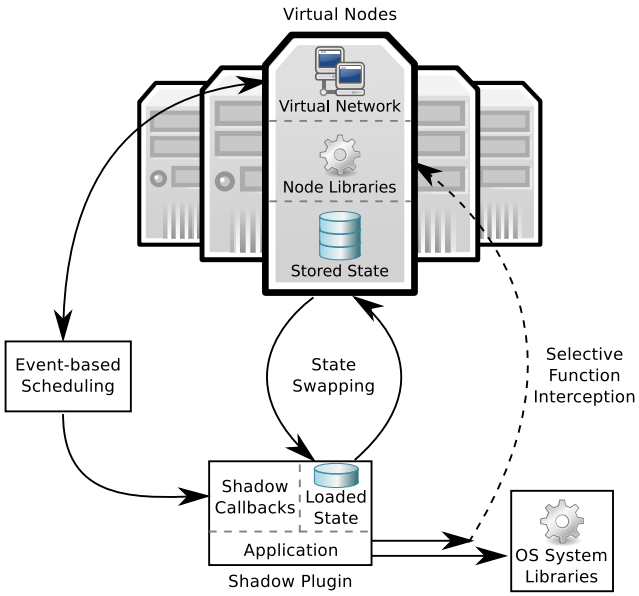


Figure 1: Shadow's architectural design. Using a plug-in wrapper, real-world applications are integrated into Shadow as virtual nodes while system and library calls are intercepted and replaced with Shadow-specific implementations.

Events are extracted from a properly formatted simulation script and spooled to the event scheduler using the times specified with each command. After the script is parsed, the simulation begins by executing the first extracted event and runs until either there are no events remaining in the scheduler or the end time specified in the script is reached. Each node creation event triggers the allocation of a virtual node and its network and culminates in a callback to a Shadow plug-in for application instantiation.

3.2 Shadow Plug-ins

A Shadow plug-in is an independent library that contains applications the user wishes to simulate and a wrapper around these applications allowing integration with the Shadow simulation environment. Each Shadow plug-in wrapper implements the plug-in interface – a set of callbacks that Shadow uses to communicate with the plug-in. Plug-ins may also link to a special Shadow plug-in utility libraries to, e.g. obtain an IP address or log messages.

Application. To run in Shadow, an application must be asynchronous, i.e. non-blocking, to prevent simulator dead-

locks during the execution of application code. We note that asynchronicity may be achieved with a small amount of code in the plug-in wrapper that utilizes the built-in Shadow callbacks or by writing the application using the `libevent-2.0` asynchronous event library [17], as Shadow supports a subset of this library.

Next, the application must be run as a single process and in a single thread. Child processes or threads forked or spawned by an application will not be properly contained in the simulation environment and are therefore currently unsupported. In most cases forking or spawning children will lead to undefined behavior or undesirable results. We note that most multi-threaded applications have a single-threaded mode and the difficulty in porting those that do not is application-specific.

Finally, the plug-in must register all variable application state with Shadow to facilitate multiple virtual nodes running the same application. Plug-ins fulfill this requirement by passing pointers to node-specific allocated memory chunks and their sizes to a Shadow library function. Therefore each variable must be globally visible during the registration process. However, we note that a plug-in may use standard tools to scan and globalize symbols present in the binary after the linking process. As in our Tor plug-in (Section 4), this technique may be used to dynamically generate variable registration code and eliminates the requirement of modifying variable definitions inside the application.

Shadow Callbacks. The Shadow plug-in interface allows Shadow not only to notify the plug-in when it should allocate and deallocate resources for running the application(s) contained in the plug-in, but also to notify the plug-in when it may perform network I/O (reading and writing) on a file descriptor without blocking. The I/O callbacks are crucial for asynchronicity as they trigger application code execution and prevent applications from the need for polling a file descriptor. The Shadow plug-in library also offers support for a generic timer callback so plug-ins may create additional events throughout a simulation. Note that callbacks may also originate from the virtual event library, as described in Section 3.3.2 below, if the application uses the `libevent-2.0` library.

3.3 Virtual Nodes

In Shadow, a virtual node represents a single simulated host. A virtual node contains all state that is specific to a host, such as addressing and network information that allows it to communicate with other hosts in the network. Virtual nodes also contain Shadow-specific implementations of system libraries that promote homogeneity between existing interfaces. Function interposition allows for seamless integration of applications into Shadow by redirecting calls to system functions to our Shadow implementation. Virtual nodes store their own application-specific state and swap this state into the plug-in's address space before passing control of code execution to the plug-in.

3.3.1 Virtual Network

In Shadow, the virtual network is the main interface through which virtual nodes may communicate. Upon creation, each node's virtual network interface is assigned an IP address and receives upstream and downstream bandwidth rates as configured in the simulation script. Each virtual network contains a transport agent that implements a leaky bucket (i.e. token bucket) algorithm that allows small traffic bursts but ensures average data rates conform to the configured rate. The transport agent handles both incoming and outgoing packets, allowing for asymmetric bandwidth specifications. The agent provides traffic policing by dropping (and causing retransmission of) all non-conforming packets. Conforming incoming packets are passed to the virtual socket library (discussed below) for processing, while events are created for conforming outgoing packets and spooled to the scheduler for delivery to another node after incorporating network latency.

3.3.2 Node Libraries

Each virtual node implements several system functions as well as network, event, and cryptography libraries. *Function interposition* is used to redirect standard system and library functions calls made from the application to their Shadow-specific counterparts. Function interposition is achieved by creating a *preloaded* library with functions of the same name as the target functions, and setting the environment variable `LD_PRELOAD` to the path of the preload library. Every time a function is called, the preload library is first checked. If it contains the function, the preloaded function is called – otherwise the standard lookup mechanisms are used to find the function. No additional modifications are required to hook into Shadow.

Virtual System. The virtual system library implements standard system calls whose results must be modified due to the simulation environment. Functions for obtaining system time are implemented to return the simulation time rather than the wall time and functions for obtaining hostname and address information are intercepted to return the hostnames as defined in the simulation script configured by the user.

The virtual system also contains a virtual CPU module in an attempt to consider processing delays produced by an application. Using a virtual CPU and processing delays improves Shadow's accuracy since without it, all data is processed by the application at a single discrete instant in the simulation. When a virtual node reads or writes data between the application and Shadow, the virtual CPU produces a delay for processing that data. This delay is "ab-

sorbed" by the system by delaying the execution of every event that has already been scheduled for that virtual node. As virtual nodes read and write more data, the wait time until the next event increases.

We determine appropriate CPU processing speeds as follows. First, throughput is configured for each virtual CPU – the number of bytes the CPU can process per second. Modeling the speed of a target CPU is done by running an `OpenSSL` [31] speed test on a real CPU of that type. This provides us with the raw CPU processing rate, but we are also interested in the processing speed of an application. By running application benchmarks on the same CPU as the `OpenSSL` speed test, we can derive a ratio of CPU speed to application processing speed. The virtual CPU converts these speeds to a time per-byte-processed and delays its events appropriately.

Virtual Sockets. The Shadow virtual socket library, the heart of the node libraries, implements the most significant features for a Shadow simulation. The virtual socket library implements all system socket functionality which includes: creating, opening, and closing sockets; sending, buffering, and receiving data; network protocols like the *User Datagram Protocol* (UDP) [34] and the *Transmission Control Protocol* (TCP) [35]; and other socket-level functionalities. Shadow's tight integration of socket functionalities and strong adherence to the RFC specifications results in an extremely accurate network layer as we'll show in Section 5.

Shadow intercepts and redirects functions from the system socket interface to the Shadow-specific virtual socket library implementation. When the application sends data to the virtual socket library, the data is packaged into packet objects. The packaging process copies the user data only twice throughout the lifetime of the packet, meaning the same packet object is shared among nodes. Only pointers to the packet are copied as the packet travels through various socket and network buffers, although buffer sizes are computed using the full packet size.

Our virtual socket libraries implement socket-level buffering, data retransmission, congestion and flow control mechanisms, acknowledgments, and TCP auto-tuning. TCP auto-tuning is required to correctly match buffer sizes to connection speeds since neither high bandwidth connections with small network buffers nor low-bandwidth connections with large network buffers will achieve the expected performance. TCP auto-tuning allows network buffers to be dynamically computed on a per-connection basis, allowing for highly accurate transfer rates even when endpoints have asymmetric bandwidth.

Virtual Events. Shadow supports the use of `libevent-2.0` [17] to facilitate asynchronous applications while easing application integration. While applications are not required to use `libevent-2.0`, doing so will likely reduce the complexity of the integration process. Shadow intercepts and redirects functions from the `libevent-2.0` interface to the Shadow-specific virtual event library implementation. The virtual event library consists of two main components: an event manager and a virtual I/O monitor. The event manager creates and tracks events and executes event callbacks while the I/O monitor tracks the state of Shadow buffers, informing the manager when a state change may require an event callback to fire for a given file descriptor.

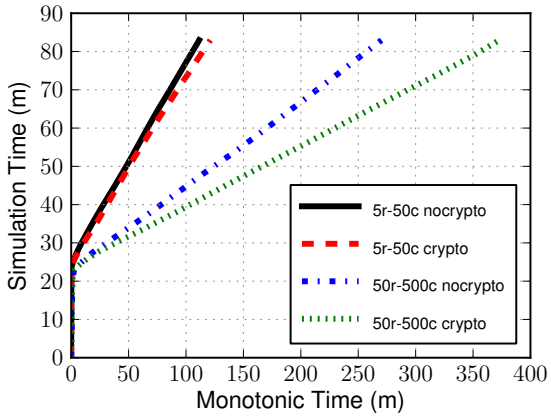


Figure 2: Simulation vs. wall clock time. Skipping expensive cryptographic operations results in a linear decrease in experiment run-time – nearly a one-third reduction in run-time for a small, 550-node Tor experiment.

Virtual Cryptography. Simulating an application that performs cryptography offers a chance for reducing simulation run-time. As data is passed from virtual node to virtual node during the simulation, in most cases it is not important that the data is encrypted: since we are not sending data out across a real network, confidentiality is not necessarily required. Therefore, applications need not perform expensive encryption and decryption operations during the simulation, saving CPU cycles on our simulation host machine.

Shadow removes cryptographic processing by preloading the main `OpenSSL` [31] functions used for data encryption. The `AES_encrypt` and `AES_decrypt` functions are used for bulk data encryption and the `EVP_Cipher` function is used to secure data on SSL/TLS connections. These functions only perform the low-level cipher operations: all other supporting cryptographic functionality is unmodified. When preloading these functions, Shadow will not perform the cipher operation during encryption and decryption. Our virtual CPU already models application processing delays, and skipping the cipher operations will not affect application functionality.

Figure 2 shows the time savings Shadow realizes using this technique with the Scallion plug-in (discussed below in Section 4) for various Tor network sizes. Larger savings in real running time are realized as experiment size increases.

3.3.3 Stored State

Multiple virtual nodes may run the same plug-in. Rather than duplicating the entire plug-in in memory for each virtual node, Shadow only duplicates the variable state – the state of an application that will change during execution. Registration of this variable state with Shadow happens once for each plug-in. The plug-in registration procedure allows Shadow to determine which memory regions (beginning address and length) in the current address space will be modified by each virtual node running the plug-in.

Following registration, Shadow possesses pointers to each memory region that may be changed by the plug-in or application. Multiple nodes for each plug-in are supported by allocating node-specific storage for each registered memory region and maintaining a copy of each plug-in's state. For transparency, Shadow loads a node's state before every context switch from Shadow to the plug-in, and saves state back to storage when the context switches back to Shadow. This process minimizes the total memory consumption of each plug-in, and results in significant memory savings for large simulations and large applications.

4 The Scallion Plug-in: Running Tor in Shadow

Shadow was designed especially for running simulations using the Tor application. Therefore, Shadow design choices were made in support of "Scallion"[2], a Tor plug-in implementation. Each virtual node running the Scallion plug-in represents a small piece of the Tor network. Since Shadow supports most functionality needed by Scallion, the plug-in implementation itself is minimal (roughly 1500 lines of code). Here we describe some of the specific components necessary for the Tor application plug-in.

State Registration. Recall that Shadow requires all variable application state to be registered for replication among virtual nodes. Scallion must find and register all Tor variables, including static and global variables. Unfortunately, static variables are not accessible outside the scope in which they were defined. Therefore, scallion uses standard binary utilities such as `objcopy`, `readelf`, and `nm` to dynamically scan, rename, and globalize Tor symbols. Registration code is then dynamically generated based on the symbols present in the Tor object files, and injected into the plug-in before compilation. Note that the size of each variable is also extracted with the binary utilities.

Bandwidth Measurements. TorFlow [44] is a set of scripts that run in the live Tor network, continuously measuring bandwidth of volunteer relays by downloading several files through each. TorFlow helps determine the bandwidth to advertise in the public consensus document. Scallion contains a component that approximates this functionality. However, Scallion need not perform actual measurements since the bandwidth of each virtual node is already configured in Shadow. Scallion queries for these bandwidth values through a Shadow plug-in library function and writes the appropriate file that is used by the directory authorities while computing a new consensus. The `V3Bandwidth` file is updated as new relays join the simulated Tor network.

Tor Preloaded Functions. In an effort to minimize the amount of changes to Tor, Scallion utilizes the same function interposition technique as Shadow. Scallion may intercept any Tor function for which it requires changes and implement a custom version. Changes in Tor are required only if the target function is static, in which case Tor can be modified to remove the static specifier. We now discuss some functional differences between Tor and Scallion.

The Tor socket function wrapper is one function that is intercepted by Scallion and modified to pass the `SOCK_NONBLOCK` flag to the socket call since Shadow requires non-blocking sockets. Another modification involves the Tor main function, which is not suitable for use in Scallion since it contains an infinite loop. This function is extracted to prevent the simulation from blocking, and Scallion instead relies on event callbacks from Shadow to implement Tor's main loop functionality.

Tor is a multi-threaded application, launching at least one CPU worker thread to handle onionskin tasks – peeling off or adding a layer of encryption – as they arrive from the network. Scallion implements an event-driven version of Tor's CPU worker since Shadow requires a single-threaded, single-process application. This is done by intercepting the Tor function that spawns a CPU worker and relying on the virtual event library to execute callbacks when the CPU worker has data ready for processing. The CPU worker performs its task as instructed by Tor, and communicates with Tor using a socket pair (a virtual pipe) as before. The virtual event library simplifies the implementation of the CPU worker functionality.

Finally, Scallion intercepts Tor's bandwidth reporting function. Each Tor relay reports its recent bandwidth history to the directory authorities to help balance bandwidth across all available relays. However, relays' reports are based on the amount of data it has recently transferred, and the reported value is updated every twenty minutes only if it has not changed significantly from the last reported value. This causes relays to be underutilized when first joining the network, and causes bootstrapping problems in new networks since every node's bandwidth will be zero for the first twenty minutes of the simulation. Without appropriate bandwidth values, clients no longer perform weighted relay selection and instead choose relays at random. To mitigate these problems, Scallion intercepts the bandwidth reporting function and returns its configured BandwidthRate no mat-

[2] Scallions are onion-like plants with underdeveloped bulbs.

ter how much data it has transferred. This improves bootstrapping and path selection for the simulated Tor network.

Configuration and Usability. There are several challenges in running accurate Tor network simulations with the Scallion plug-in and Shadow. Although Shadow minimizes the memory requirements, running several instances of Tor still requires an extremely large amount of memory. Therefore, simulations must generally run with scaled-down versions of Tor network topologies and client-imposed network load.

Correctly scaling available relay bandwidth and network load is complicated. For example, several relays with smaller bandwidth capacities will not result in the same network throughput as fewer relays with larger bandwidth capacities, even if the total capacities are equal. Further, correctly distributing this bandwidth among entry, middle, and exit nodes can be tricky. Although live Tor consensus documents may be used to assist in network scaling, two randomly generated consensus topologies can have drastically different network throughput measurements. Network throughput also depends on the number of configured clients and the load they induce on Tor. Published results about client-to-relay ratios [20] and protocol-level statistics [22] can only be used as a rough guide to creating clients and inducing the correct load. When generating a scaled topology, it is essential that performance measurements of simulations be compared to live Tor statistics for accuracy.

Due to these challenges, we implemented a script to generate and run simulations given a network consensus document. The script parses the consensus document and randomly selects relays based on configurable network sizes. Configurable parameters include the fraction of exit relays to normal relays, number of clients, and client type distributions. The script eases the generation of accurate scaled topologies and drastically improves simulator usability.

5 Verifying Simulation Accuracy

Many aspects of Shadow's design (discussed in Section 3) were chosen in order to produce accurate simulations. Therefore, we perform several experiments to verify Shadow's accuracy.

5.1 File Client and Server Plug-ins

`HTTP` client and server plug-ins were written for Shadow in order to provide a mechanism for transferring data through the Shadow virtual network. These plug-ins also include support for a minimal `SOCKS` client and proxy. The client may download any number of specified files with configurable wait times between downloads while the server supports buffering and multiple simultaneous connections. These plug-ins are used to test network performance during a simulation. Stand-alone executables using the same code as the plug-ins are also compiled so that client and server functionality on a live system and network is identical to Shadow plug-in functionality.

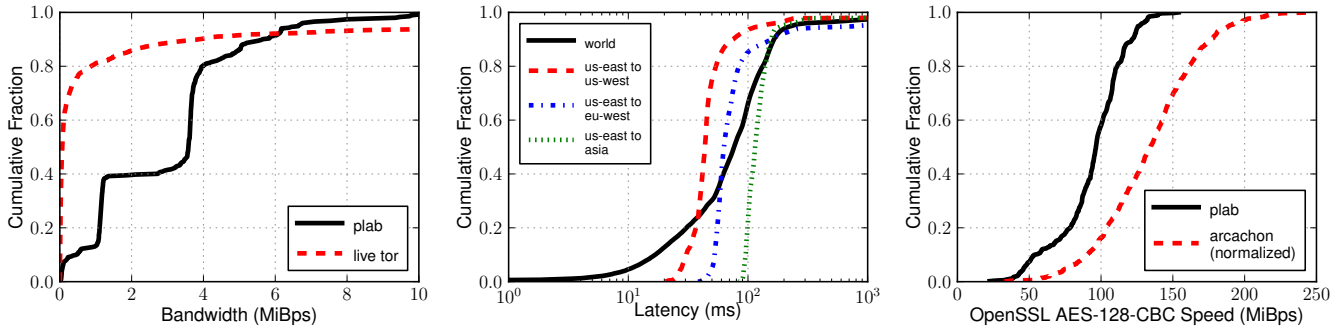
5.2 PlanetLab Private Tor Network

In order to verify Shadow's accuracy, we perform experiments on PlanetLab. Our experiments consist of file clients and servers running the software described above in Section 5.1. In our first PlanetLab experiment, each of 361 `HTTP` clients download files directly from one of 20 `HTTP` servers, choosing a new server at random for each download. 18 of the 361 clients approximate a bulk downloader, requesting a 5 MiB file immediately after finishing a download while the remaining 343 clients approximate a web downloader, pausing for a short time between 320 KiB file downloads. The length of the pause is drawn from the UNC think-time distribution [12] which represents the time between clicks for a user browsing the web (the median pause is 11 seconds). Clients track both the time to receive the first byte of the data payload and the time to receive the entire download. We selected the fastest PlanetLab nodes (according to the bandwidth tests described below) as our `HTTP` servers to minimize potential server bottlenecks, although we note that fine-grained control is complicated by PlanetLab's dynamic resource adjustment algorithms.

Our second PlanetLab experiment is run exactly like the first, except all downloads are performed through a private PlanetLab Tor network consisting of 16 exit relays, 24 non-exit relays, and one directory authority. All `HTTP` clients also run a Tor client and proxy their downloads through Tor using a local connection to the Tor `SOCKS` server.

Shadowing PlanetLab. To replicate the PlanetLab experiments discussed in Section 5 in Shadow, we require measurements of PlanetLab node bandwidth, latency between nodes, and an estimate of node CPU speed. These measurements allow us to configure virtual nodes and a virtual network that approximates PlanetLab and network conditions typical of the Internet. First, we estimate PlanetLab node bandwidth by performing Iperf [14] bandwidth tests from each node to every other node. We estimate a node's bandwidth as the maximum upload rate to any other node.

Figure 3a shows the results of our measurements compared with available bandwidth from Tor relays extracted from the Tor network status consensus. Notice the sharp increase in the number of nodes with 1.25 MiBps (10 Mbps) and 3.75 MiBps (30 Mbps) connections. PlanetLab rate-limiting is the likely reason: the most popular node-defined limit is 10 Mbps while PlanetLab also implements a fair-sharing algorithm by distributing opportunistic fractions of bandwidth to active slices. Also notice that our PlanetLab distribution does not approximate the live Tor distribution well, which means that our measurements in this experi-



(a) (b) (c)

Figure 3: Network and CPU measurements used for Shadow experiments. (a) Bandwidth measurements of PlanetLab nodes and live Tor relays. Relay bandwidth values were taken from a live consensus. (b) Latency between PlanetLab nodes, shown as aggregate ("world") and inter-region latency measurements. (c) Measured CPU speeds for PlanetLab nodes and our Intel Core2 Duo lab machine *arcachon*. The results from arcachon were normalized to create a distribution usable in Shadow.

ment will not provide a good indication of the performance of the live Tor network. Recall, however, that our focus here is *accurately shadowing PlanetLab*: re-creating a network consistent with live Tor is explored below in Section 5.3.

To model network delays due to propogation and congestion, we perform latency estimates between all pairs of nodes using the Unix command `ping`. The aggregate results of world latencies are shown in Figure 3b. Deriving a network model and topology from the latency measurements is a bit more complex since it depends on the geographical location of the source and destination of a ping. We approximate a network model by creating nine geographical regions and placing each node in a region using a GeoIP lookup [21]. We then create a total of 81 CDFs representing all possible inter- and intra-region latencies. We configure nine virtual networks in Shadow and connect them into a complete graph topology, where latencies for packets traveling over each link are drawn from the corresponding CDF. Latencies for a few selected regions are also shown in Figure 3b.

Finally, we measure CPU speed of each node in order to accurately configure delays for Shadow's virtual CPU system described in Section 3.3.2. As in our previous description, `OpenSSL` speed tests are run to get raw CPU throughput for PlanetLab nodes. Since PlanetLab nodes are often constrained, we also created a normalized distribution based on the CPU speed of *arcachon* – a standard desktop machine in our lab. CPU throughput is shown in Figure 3c. Tor application throughput – measured by benchmarks in which the middle relay is configured with a bandwidth bottleneck – is combined with raw CPU throughput measurements to configure each node's virtual CPU delay.

Client Performance. Figure 4 shows the results of our PlanetLab and Shadow experiments. We are mainly interested in two metrics: the *time to receive the first byte* of the data payload (ttfb) and the *time to complete* a download (dt). The ttfb metric provides insight into the delays associated with sending a request through multiple hops and the responsiveness of a circuit, and also represents the minimum time a web user has to wait until anything is displayed in the browser. The dt metric captures overall performance.

Figures 4c and 4e show the ttfb metric for web and bulk clients with direct and Tor-proxied requests both in PlanetLab and Shadow. Downloads through Tor take longer than direct downloads, as expected, since data must be processed and forwarded by multiple relays. Shadow seems to closely approximate the *network conditions* in PlanetLab, as shown by the close correspondence between the lower half of each CDF. However, PlanetLab exhibits slightly higher variability in ttfb than Shadow as seen in the tail of the plab and shadow CDFs – a problem that is exacerbated when downloads are proxied through Tor. Higher variability in results is likely caused by increased PlanetLab node delay due to resource contention with other co-located experiments.

Figures 4d and 4f show similar conclusions for the dt metric. Shadow results appear off by a small factor while we again see higher variability in download completion times for PlanetLab. However, inaccuracies in download times appear somewhat independent of file size. As shown in Figure 4a, statistics gathered from Tor relays support our conclusions about higher variability in delays. Shown is the number of processed cells for each relay over the one hour experiment and the one-minute moving average. The moving average of processed cells is slightly higher for Shadow because of PlanetLab's resource sharing complexity while the individual relay measurements also show higher variability for PlanetLab. Figure 4b shows that Shadow queue times are very close to those measured on PlanetLab, and again shows PlanetLab's high variability. While we are optimistic about our conclusions, we emphasize that PlanetLab results should be analyzed with a careful eye due to the issues discussed above.

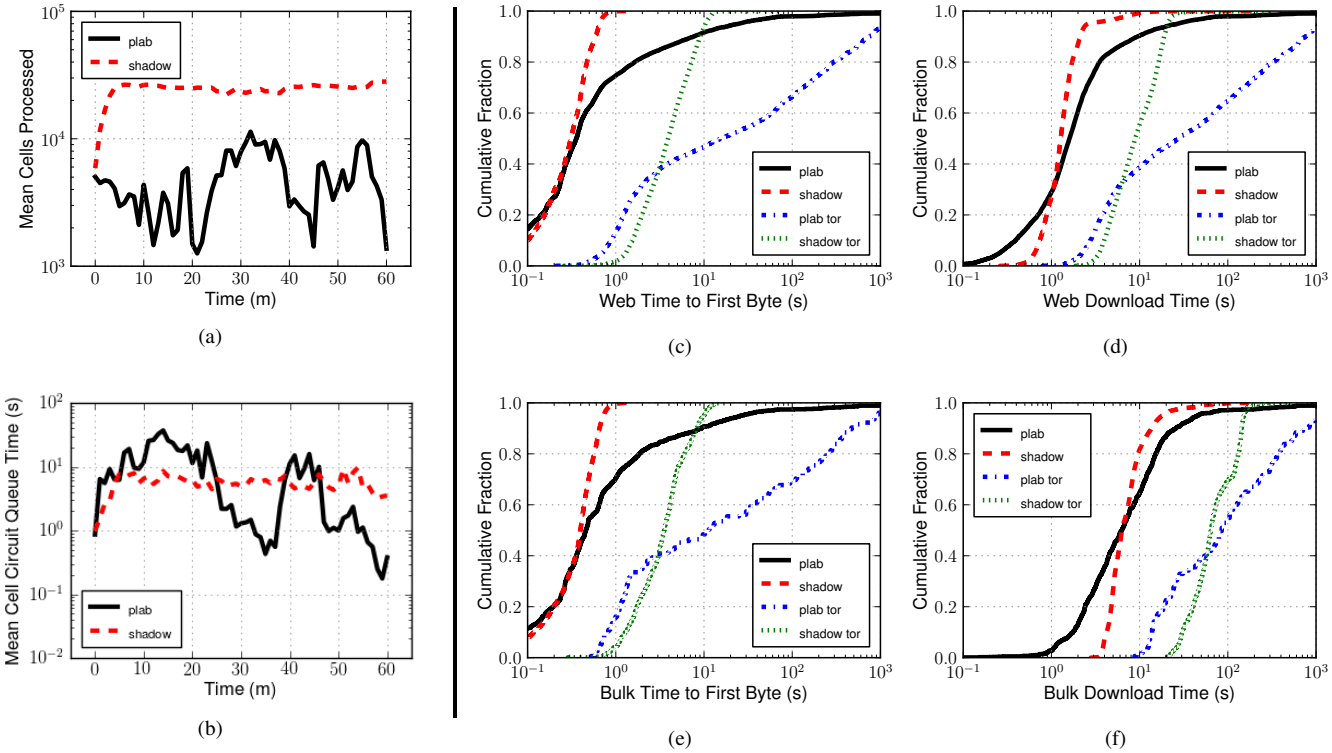


Figure 4: Shadow and PlanetLab network performance. PlanetLab download experiments were run with and without Tor and mirrored in Shadow. As shown in (a) and (b), Shadow approximates PlanetLab performance reasonably well while PlanetLab results show higher variability due to co-location and network/hardware interruptions. Also shown are CDFs of the number of elapsed seconds until the first byte of a 320 KiB file (c) and a 5 MiB file (e) is received, and the time to complete a download of the same files (d), (f).

5.3 Live Public Tor Network

Although the PlanetLab results show how Shadow performance compares to that achieved while running on PlanetLab and a *private* Tor network, they do not show how accurately Shadow can approximate the live *public* Tor network containing thousands of relays and hundreds of thousands of clients geographically distributed around the world. Therefore, we perform a separate set of experiments to test Shadow's ability to approximate live Tor network conditions as documented by The Tor Project [45]. Comparing results with statistics from Tor Metrics gives us much stronger evidence of Shadow's ability to accurately simulate the live Tor network.

The experiments are similar to those performed on PlanetLab: web and bulk clients download variable-sized files from servers through a private Tor network. However, file sizes are modified to 50 KiB, 1 MiB, and 5 MiB as used by TorPerf while configuration of Shadow nodes is also slightly modified to approximate resources available in live Tor. In these experiments, we use a directory authority, 50 relays, 950 web clients, 50 bulk clients, and 200 servers. We use a live Tor consensus[3] to obtain bandwidth limits for Tor relays and ensure that we correctly scale available bandwidth and network size, while client bandwidths are estimated with 1 MiB down-link and 3.5 MiB up-link speeds (not over-subscribed). Each relay is configured according to the live consensus: a `CircuitPriorityHalflife` of 30, a 40 KiB `PerConnBWRate`, and a 100 MiB `PerConnBWBurst`. Geographical location and latencies are configured using our PlanetLab dataset [39].

Figure 5 shows Shadow's accuracy while simulating a shadow of the live Tor network. CDFs of Shadow download completion times for each file size are compared with download times measured and collected by The Tor Project. The gray area represents the first-to-third quartile stretch and the dotted line shows the median download time extracted from live Tor network statistical data available at The Tor Metrics Portal [45] (gathered during April 2011 – the same month as our consensus). To maximize accuracy, the left edge of the gray area should intersect the CDF at 0.25, the right edge at 0.75, and the dotted line at 0.5. Our results show that the median download times are nearly identical for 50 KiB and 1 MiB downloads and within ten percent for 5 MiB downloads while the first and third quartiles are within 15 percent in all cases. We believe these results provide strong evi-

[3]The consensus was retrieved on 2011-04-27 and valid between 03:00:00 and 06:00:00.

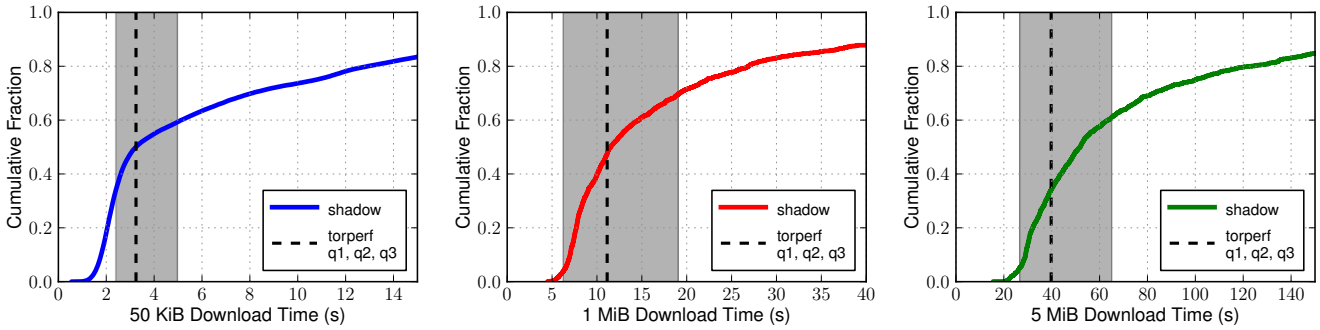


Figure 5: Shadow-Tor compared with live-Tor network performance. TorPerf represents live Tor network performance statistics available at `metrics.torproject.org`. The gray area shows TorPerf first to third quartile stretch while the dotted line represents the TorPerf median. Shadow closely approximates Tor performance for all file sizes.

dence of Shadow's ability to accurately simulate Tor. Further, we've shown that we can correctly scale down the Tor network in our simulations while maintaining the performance properties of the live Tor network.

6 Prioritizing Circuits

We now demonstrate Shadow's powerful capabilities by exploring a Tor circuit scheduling algorithm recently proposed and integrated into the Tor software. In Tor, whenever there is room in an output buffer, the circuit scheduler must make a decision about which circuit to flush. Tor's original design used a round-robin algorithm for making such decisions. Recently, an algorithm based on the Exponentially-Weighted Moving Average (EWMA) of cells sent in each circuit was proposed and incorporated into Tor, and has since become the default scheduling algorithm used by Tor relays. This section attempts to validate the results originally obtained by Tang and Goldberg [42].

EWMA in Bottleneck Topology. The EWMA scheduler chooses the circuit with the lowest cell count, effectively prioritizing bursty web connections over bulk transfers. Tang and Goldberg evaluated the EWMA algorithm by creating a congested circuit on a synthetic PlanetLab network and measuring performance of web downloads. Since the middle node was a circuit bottleneck, the benefits of EWMA for reducing web download times were clear. However, results for bulk downloads during this experiment were not given.

We perform a similar bottleneck experiment in Shadow. We configure a circuit consisting of a single entry, middle, and exit relay. Two bulk clients continuously download 5 MiB files to congest the circuit. Ten minutes after booting the "congestion" clients, two "measurement" clients are started and download for an hour: a third bulk client and a web client that waits 11 seconds (the median think-time for web browsers [12]) between 320 KiB file downloads. The middle relay is configured as a circuit bottleneck with a 1 MiBps connection while all other nodes (relays, clients, and server) have 10 MiBps connections.

We run the above experiment modifying only the scheduling algorithm. We test both the round-robin scheduler and the EWMA scheduler with a `CircuitPriorityHalfLife` of 66 as in [42]. Relay buffer statistics [43] are shown in Figures 6a and 6b. Notice a significant increase in traffic at the ten-minute mark, at which point the "measurement" clients start downloading. Figure 6a shows that the number of processed cells is similar for all relays, except occasionally the exit relay processes fewer cells due to middle relay congestion. Figure 6b shows that the circuit queues increase for the exit and middle relay while the entry relay's circuit queues are empty due to sufficient bandwidth to immediately forward data to the client.

Figures 6c and 6d show the performance results obtained from the web client for both schedulers. As expected, the time to the first byte of the data payload and the time to complete a download are both reduced for the web client, since bursty traffic gets prioritized ahead of the bulk traffic. The time to first byte for the "measurement" bulk downloader in Figure 6e also improves for a large fraction of the downloads because each new download originating from a new circuit will be prioritized ahead of the "congestion" bulk downloads. However, after downloading enough data, the "measurement" bulk client loses its priority over the "congestion" bulk clients and the time to first byte converges for each scheduler.

Tang and Goldberg claim that, according to Little's Law [19], bulk transfers will not be negatively affected while using the new circuit scheduler. While this may be theoretically true, it is not clear that it will hold in practice. The authors find that Little's Law holds when a single relay in the live Tor network uses the EWMA scheduler: their results show that bulk download times are not signifi-

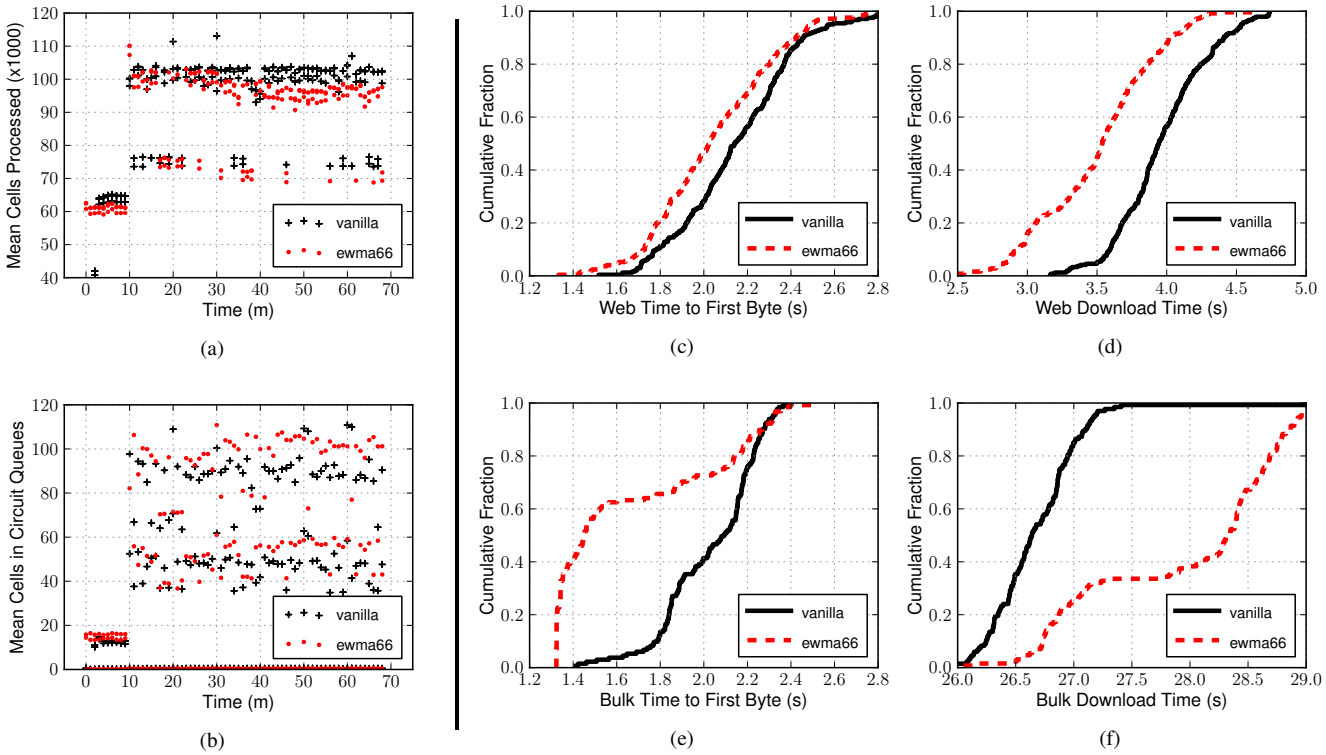


Figure 6: Seven-node bottleneck experiment similar to that performed by Tang and Goldberg [42]. The number of cells processed (a) and queued (b) increases at Time=10, when the measurement clients begin downloading. The EWMA scheduler improves responsiveness for bursty traffic (c), (d), and (e) but, contrary to the author's claims, decreases performance for bulk downloads (f).

cantly different for each scheduler. However, our results in Figure 6f indicate otherwise. Bulk download times are noticably worse for the EWMA scheduler, with a significant increase at around the 40th percentile. This increase again happens when the "measurement" bulk client loses its priority over the "congestion" bulk clients, suggesting a deeper analysis of the EWMA scheduling algorithm is appropriate.

EWMA in Network-wide Deployment. Tang and Goldberg's experiments suffer from a major limitation of scale: the experiments were run either on three-node PlanetLab topologies, or in the live Tor network with only a single relay scheduling with the EWMA algorithm. Although they provide results for what a single relay might expect when switching scheduling algorithms, they do not consider the network-wide effects of a full network deployment.

We explore the performance gains possible with the EWMA scheduler through a network-wide deployment in Shadow. We test the EWMA circuit scheduler with a range of half-life configurations and compare performance to the round-robin scheduler used in vanilla Tor. As in Section 5.3, we use 200 servers, 50 relays and 950 web clients for our experiments. To analyze the effects of various network loads on the scheduler, we run separate experiments configured with each of 25, 50, and 100 bulk clients. The adjusted load is significant since bulk clients account for a large fraction of network traffic. To reduce random variances, we run each experiment five times and show the cumulative results of each configuration by aggregating the results of all five experiments. Our results are shown in Figure 7.

Under a load of 25 bulk clients, Figures 7a–7c show that the EWMA circuit scheduler reduces performance over vanilla Tor for all clients, independent of the configured half-life. Bulk download times seem to be affected the most (7c), but our experiments indicate there is also a significant reduction in responsiveness for web clients (7a). As load increases to 50 bulk clients, Figures 7d–7f show that there are half-life configurations that still reduce performance when compared to vanilla Tor. The 30 and 90 second EWMA half-life configurations appear to improve performance for web clients (7d, 7e), but performance for bulk clients is either reduced or shows less improvement (7f). Performance is reduced for all clients when using a 3 second half-life. Finally, Figures 7g–7i show performance under the load of 100 bulk clients. Under heavy load, the EWMA scheduler appears to perform the best for web clients (7g, 7h) while bulk clients see no improvements over vanilla Tor and the round-robin scheduler (7i). Note that we also tested the schedulers under a lighter load than shown in Figure 7, but performance when the network is too lightly loaded is nearly identical regardless of the selected circuit scheduler.

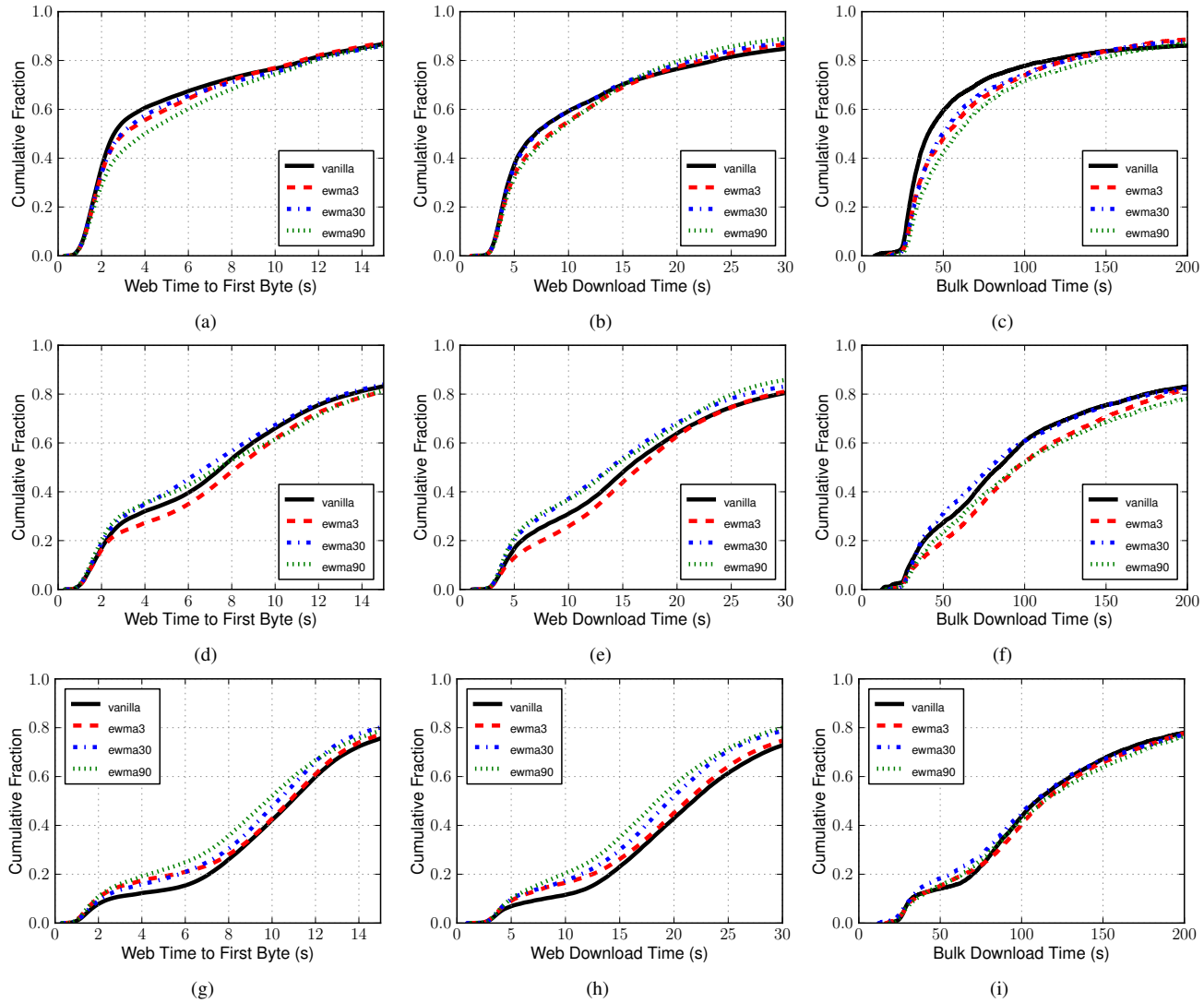


Figure 7: Performance of a full-network deployment of the EWMA circuit scheduler and vanilla Tor using a round-robin scheduler. Load is generated with 950 web clients and varied using (a)–(c) 25 bulk clients (d)–(f) 50 bulk clients, and (g)–(i) 100 bulk clients. While the EWMA circuit scheduler works best under heavily loaded networks, there are EWMA half-life configurations that lead to reduced client performance.

For these results, and responsiveness for bulk clients under the loads decribed above, please see Appendix B.

We conclude from our results that the EWMA scheduler should not necessarily be used under all network conditions since it is not clear that performance will always improve. When improvements over the round-robin scheduler are possible, they may be insignificant or depend on a correctly configured half-life. Tang and Goldberg find that low half-life values close to 0 and high values close to 100 result in little improvement when compared to unprioritized, vanilla Tor. We find this to be true under lighter loads, but Figure 7 shows that larger half-life values result in better performance for more heavily loaded networks. Our results illustrate that performance benefits are heavily dependent on network traffic patterns, and we stress the importance of frequently assessing the network to assist in determining appropriate half-life values over time. We suggest that more analysis is required to determine if the EWMA scheduler actually improves performance in the live Tor network, and if relays should enable it by default.

7 Related Work

This section reviews several experimentation techniques that have been used to test Tor's performance and resistance to various attacks. A test environment that accurately reflects Tor's behavior is crucial to produce meaningful results. We now briefly explore experimentation techniques

chosen by researchers to evaluate Tor proposals. We note that: Kiddle [16] provides a comprehensive analysis and discussion of system simulation and emulation techniques; Naicken *et al.* [26, 27] provide details on several generic simulators; Bauer *et al.* [4] provide an in-depth survey of experimental approaches historically used in Tor-related research; and the EWMA circuit priority scheduler from Tang and Goldberg [42] is discussed in detail in Section 6.

Simulation. Simulation typically involves creating abstract models of system processes and running multiple nodes in a single unified framework. Experiment management is simplified since there are many fewer simulation host machines (typically one) than simulated nodes. By abstracting system processes, simulators can run much more efficiently and are not required to run in real time. However, the abstraction process has the potential to reduce accuracy since the simulator may not encompass complex procedures that may in fact be important to system interaction. Although generic simulation platforms exist [18, 29, 30, 41], they are not capable of running unmodified versions of the Tor software.

Simulation has often been employed for Tor research, but simulators tend to be written for a specific problem and may be difficult to apply to a generic context: Murdoch and Watson explore Tor path selection strategies and algorithms [25], O'Gorman and Blott simulate packet counting and stream correlation attacks [33], Ngan *et al.* study the effects of their gold-star priority scheme on Tor performance [28], and Jansen *et al.* simulate queuing models and traffic prioritization mechanisms [15]. These simulators have either become unmaintained or are not publicly available, making published results challenging to validate.

Emulation. A competing and fundamentally different experimentation approach involves emulation. An emulator "tricks" an application or operating system that it is running on its own physical machine, when in fact it is virtualized in software. Emulators require a large amount of overhead to ensure the emulated software runs in real time while providing the virtualization layers needed to emulate an entire system. Therefore, emulation is potentially more accurate than simulation, but *much less scalable*: emulators typically run hundreds of nodes while simulators run thousands.

Due to intensive resource requirements, emulation platforms often utilize a large testbed of geographically distributed physical hardware. PlanetLab [7] and DETER [5] are examples of whole-system emulation testbeds. Both of these frameworks only supply a few hundred nodes to a user. Several Tor studies have utilized the PlanetLab and DETER testbeds for experimenting with traffic analysis attacks [3, 6, 13], attacks on Tor bridges [23], and relay circuit scheduling [42]. Due to resource consumption and colocation of nodes on each physical machine, results on these testbeds often suffer from a reduced and false sense of accuracy. Further, distributed experiments like those run on PlanetLab are challenging to manage and control while results are difficult to recreate.

A Tor emulation testbed has recently been simultaneously and independently proposed by Bauer *et al.* [4] based on the ModelNet emulation platform [46]. The emulation testbed, called ExperimentTor, works by configuring multiple host machines with new operating system installations. Some of these host machines run a version of ModelNet link emulators while the remaining machines run Tor and other application instances. Tor nodes are given IP addresses from separate virtual interfaces to allow multiple nodes per machine while sending all traffic over the ModelNet hosts to emulate configured network properties.

Shadow has several advantages over ExperimenTor despite having similar goals and motivations. First, Shadow is more *usable* than ExperimenTor, which requires multiple physical machines, kernel modifications, and complex configuration. Shadow can be run as a stand-alone user application without root privledges and requires little configuration, leading to an extremely small barrier to entry and improving accessibility to students, developers, and researchers around the world. Second, Shadow is more *efficient* and *scalable* than ExperimenTor. Shadow implements a discrete-event simulator which allows full utilization of computational resources while eliminating the requirement of running in real time: experiments may run either faster or slower than real time without affecting accuracy. Conversely, ExperimenTor suffers from both CPU and bandwidth bottlenecks: the CPUs on the machines running the ExperimenTor testbed must run at far less than 100 percent utilization and the aggregate traffic load from all application instances must not exceed the capacity of the physical network connecting the host machines. Both requirements must be met to ensure the emulated applications do not lag, since lag would skew and invalidate results obtained in an experiment. Shadow also minimizes the memory overhead of running multiple applications on a single machine with its "state swapping" approach to memory management whereas ExperimenTor duplicates entire copies of the application in memory. Finally, Shadow allows for a richer *customization* of the experimental process, e.g. adversarial entities could easily be added to links between nodes to allow monitoring of network level traffic. Similar customizations would be difficult to add to an ExperimenTor testbed.

8 Conclusion

In this paper, we presented the design and implementation of a large scale discrete event simulator called Shadow, and a plug-in called Scallion that is capable of linking to and running the Tor software over a simulated network. In addition to an explanation of Shadow's non-trivial design, we performed an extensive experimental analysis to verify the

accuracy of Tor simulations. We found that client performance for simulated Tor clients is surprisingly congruent to performance achieved through the live public Tor network. High accuracy is achieved by "shadowing" the Tor network, considering relay characteristics from a live Tor consensus and inter-node latency characteristics from PlanetLab ping measurements. As an example of the powerful capabilities of our simulation approach, we explore the EWMA circuit priority scheduler recently proposed and currently used in Tor to validate previous results and determine the effects of a network-wide deployment. We found that correct half-life configurations are highly network and load dependent, and that EWMA actually reduces performance for clients under certain network conditions. Although enabled by default, it is unclear if the scheduler improves performance in the live Tor network.

Limitations. Shadow is a discrete event simulator. By definition, Shadow imitates the behavior of system and network processes. These imitations were done by exploring and measuring real systems and real networks to produce models of real behaviors suitable for our study of performance in Tor. This modeling approach fundamentally limits the ability to adapt to highly dynamic environments, potentially reducing simulation accuracy. We now briefly discuss our modeling approaches in the context of their effects on simulations to emphasize the importance of analyzing and verifying results so that future researchers may make informed decisions regarding experimentation with their algorithms and protocols.

In addition to simulating TCP, Shadow models the Internet by utilizing real delays gathered from `ping` measurements on PlanetLab. The measurements give us a distribution of pairwise delays between nodes. We then categorize all nodes into geographical "regions" and aggregate node distributions between each region. This approximation forms our model of the Internet and is given as input to a simulation: when sending a packet from a node in one region to a node in another, we sample the distribution corresponding to the link between those regions. This model is based on specific measurements between specific points at a specific time. If Internet congestion at that time was uncharacteristically high or low, our model would skew results. Although this approach seemed to provide an adequate delay model in our simulations, future work should consider if their experiments could benefit from something more robust. In particular, experiments that depend on side-channels stemming from latency, throughput, or the inner workings of a specific TCP implementation may require more accurate simulations.

Scallion models system processing delays by gathering PlanetLab `OpenSSL` speed and application performance test measurements. Processing delays are then accumulated as the application reads and writes data, as this allows Shadow to quantify the amount of work the application is performing. Delays are incorporated into the event scheduling mechanism. This is a crude approximation of the real processing delays experienced on a system and is application specific: different applications may have very different processing delay models. For example, our processing delay model would be inadequate for an application that reads and writes data one percent of the time and spends the remaining time performing expensive operations. Inaccurate delay models would potentially skew results. Therefore, new application performance testing is required to appropriately model processing delays. Further, experiments that attempt to introduce variablilty in processing times as a key feature of an algorithm or as part of an attack will require more accurate simulations.

In our analysis of Tor performance, our modeling approaches were suitable to obtain realistic and consistent results. In particular, our Tor experiments were run using the same models while each experiment varied only a single configuration option. Therefore, each experiment was subject to the same system and network conditions and the same systematic biases that Shadow potentially introduces during simulation. As a result, we can be reasonably convinced that the *relative* differences between experimental results are due to our configured change and not to some systematic inaccuracy inherent to simulation, irrespective of how well the *absolute* simulation results match those obtained from real systems and networks.

Future Shadow users should be aware of the above limitations and recognize that adequate models are application dependent. It may be the case that more robust models are needed to effectively analyze particular algorithms or protocols. Future research should adapt models as appropriate to their work in order to draw meaningful conclusions, and validate results with other experimentation methods.

Future Work. Shadow may be used to explore a wide range of problems in Tor, including UDP transport mechanisms and alternative scheduling approaches. Shadow may also be used to validate previous work and analyze Tor attacks under various network configurations and client models. Finally, an exploration of more robust modeling techniques would reduce potential systematic biases introduced by our simulator and improve overall simulation accuracy.

There are several architectural modifications that can improve Shadow's run-time performance. The most significant improvement will enhance Shadow's ability to run in parallel environments, leading to faster experiments and better utilization of hardware resources. Shadow is open-source software that is available for download and includes useful tools for generating and running experiments. We feel Shadow is invaluable for understanding and evaluating Tor, hope that it will be useful to others in their research, and hope it will make a lasting impact on the community.

Acknowledgments. We thank John Geddes for assistance with Shadow plug-ins; Eric Chan-Tin, Denis Foo Kune, and Max Schuchard for discussions about Shadow's design; Chris Wacek for usability feedback; Roger Dingledine for insightful analysis of results; and Yongdae Kim, Nick Mathewson, and Paul Syverson for wisdom and encouragement. This research was supported by NFS grant CNS-0917154, ONR, and DARPA.

References

[1] A. Acquisti, R. Dingledine, and P. Syverson. On the economics of anonymity. In *Proceedings of the 7th International Conference on Financial Cryptography (FC'03)*, 2003.

[2] M. AlSabah, K. Bauer, I. Goldberg, D. Grunwald, D. McCoy, S. Savage, and G. Voelker. DefenestraTor: Throwing out Windows in Tor. In *Proceedings of the 11th International Symposium on Privacy Enhancing Technologies (PETS'11)*, 2011.

[3] K. Bauer, D. McCoy, D. Grunwald, T. Kohno, and D. Sicker. Low-resource routing attacks against Tor. In *Proceedings of the 6th ACM Workshop on Privacy in the Electronic Society (WPES'07)*, pages 11–20, 2007.

[4] K. Bauer, M. Sherr, D. McCoy, and D. Grunwald. Experimentor: A testbed for safe and realistic tor experimentation. In *the 4th Workshop on Cyber Security Experimentation and Test (CSET'11)*, 2011.

[5] T. Benzel, R. Braden, D. Kim, C. Neuman, A. Joseph, K. Sklower, R. Ostrenga, and S. Schwab. Design, deployment, and use of the DETER testbed. In *Proceedings of the DETER Community Workshop on Cyber Security Experimentation and Test*, 2007.

[6] S. Chakravarty, A. Stavrou, and A. Keromytis. Traffic analysis against low-latency anonymity networks using available bandwidth estimation. In *Computer Security – ESORICS*, pages 249–267, 2010.

[7] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Computer Communication Review*, 33:3–12, 2003.

[8] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[9] N. Evans, R. Dingledine, and C. Grothoff. A practical congestion attack on Tor using long paths. In *Proceedings of the 18th USENIX Security Symposium*, pages 33–50, 2009.

[10] D. Foo Kune, T. Malchow, J. Tyra, N. Hopper, and Y. Kim. The Distributed Virtual Network for High Fidelity Large Scale Peer to Peer Network Simulation. Technical Report 10-029, University of Minnesota, 2010.

[11] D. M. Goldschlag, M. G. Reed, and P. F. Syverson. Hiding Routing Information. In *Proceedings of Information Hiding Workshop (IH'96)*, pages 137–150, 1996.

[12] F. Hernandez-Campos, K. Jeffay, and F. Smith. Tracking the evolution of web traffic: 1995-2003. In *The 11th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer Telecommunications Systems (MASCOTS'03)*, pages 16–25, 2003.

[13] N. Hopper, E. Vasserman, and E. Chan-Tin. How much anonymity does network latency leak? *ACM Transactions on Information and System Security (TISSEC'10)*, 13(2):1–28, 2010.

[14] The Iperf bandwidth measurement tool. http://iperf.sourceforge.net/.

[15] R. Jansen, N. Hopper, and Y. Kim. Recruiting new Tor relays with BRAIDS. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, pages 319–328, 2010.

[16] C. Kiddle. *Scalable network emulation*. PhD thesis, University of Calgary, 2004.

[17] The Libevent event notification library, version 2.0. http://monkey.org/~provos/libevent/.

[18] S. Lin, A. Pan, Z. Zhang, R. Guo, and Z. Guo. Wids: an integrated toolkit for distributed system development. In *Proceedings of the 10th conference on Hot Topics in Operating Systems (HOTOS'05)*, 2005.

[19] J. D. Little and S. C. Graves. Little's Law. web.mit.edu/sgraves/www/papers/Little's%20Law-Published.pdf, 2008. Accessed July, 2011.

[20] K. Loesing. Measuring the Tor network: Evaluation of client requests to directories. Technical report, Tor Project, 2009.

[21] The MaxMind GeoIP Lite country database. http://www.maxmind.com/app/geolitecountry.

[22] D. Mccoy, K. Bauer, D. Grunwald, T. Kohno, and D. Sicker. Shining light in dark places: Understanding the Tor network. In *Proceedings of the 8th International Symposium on Privacy Enhancing Technologies (PETS'08)*, pages 63–76, 2008.

[23] J. McLachlan and N. Hopper. On the risks of serving whenever you surf: vulnerabilities in Tor's blocking resistance design. In *Proceedings of the 8th ACM Workshop on Privacy in the Electronic Society (WPES'09)*, pages 31–40, 2009.

[24] S. Murdoch and G. Danezis. Low-cost traffic analysis of Tor. In *IEEE Symposium on Security and Privacy*, pages 183–195, 2005.

[25] S. Murdoch and R. Watson. Metrics for security and performance in low-latency anonymity systems. In *Proceedings of the 8th International Symposium on Privacy Enhancing Technologies (PETS'08)*, pages 115–132, 2008.

[26] S. Naicken, A. Basu, B. Livingston, S. Rodhetbhai, and I. Wakeman. Towards yet another peer-to-peer simulator. In *Proceedings of the 4th International Working Conference on Performance Modelling and Evaluation of Heterogeneous Networks (HET-NETs'06)*, 2006.

[27] S. Naicken, B. Livingston, A. Basu, S. Rodhetbhai, I. Wakeman, and D. Chalmers. The state of peer-to-peer simulators and simulations. *SIGCOMM Computer Communication Review*, 37(2):95–98, 2007.

[28] T.-W. J. Ngan, R. Dingledine, and D. S. Wallach. Building incentives into Tor. In *The Proceedings of Financial Cryptography (FC'10)*, 2010.

[29] The ns-2 Network Simulator. http://www.isi.edu/nsnam/ns/.

[30] The ns-3 Network Simulator. http://www.nsnam.org/.
[31] The OpenSSL cryptographic library. http://www.openssl.org/.
[32] L. Overlier and P. Syverson. Locating hidden servers. In *IEEE Symposium on Security and Privacy*, 2006.
[33] G. OGorman and S. Blott. Large scale simulation of Tor: Modelling a Global Passive Adversary. In *Proceedings of the 12th Conference on Advances in Computer Science – ASIAN*, pages 48–54, 2007.
[34] J. Postel. User Datagram Protocol. RFC 768, http://www.ietf.org/rfc/rfc768.txt, August 1980.
[35] J. Postel. Transmission Control Protocol. RFC 793, http://www.ietf.org/rfc/rfc793.txt, September 1981.
[36] J. Reardon and I. Goldberg. Improving Tor using a TCP-over-DTLS tunnel. In *Proceedings of the 18th USENIX Security Symposium*, 2009.
[37] M. Reed, P. Syverson, and D. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 16(4):482–494, 1998.
[38] Shadow Development Repositories. http://github.com/shadow/.
[39] Shadow Resources. http://shadow.cs.umn.edu/.
[40] R. Snader and N. Borisov. A tune-up for Tor: Improving security and performance in the Tor network. In *Proceedings of the 16th Network and Distributed Security Symposium (NDSS'08)*, 2008.
[41] Scalable Simulation Framework, SSFNet. http://www.cc.gatech.edu/computing/compass/pdns/index.html.
[42] C. Tang and I. Goldberg. An improved algorithm for Tor circuit scheduling. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 329–339, 2010.
[43] Including network statistics in extra-info documents. https://gitweb.torproject.org/torspec.git/blob_plain?f=proposals/166-statistics-extra-info-docs.txt.
[44] The TorFlow measurement tools. https://gitweb.torproject.org/torflow.git/.
[45] The Tor Metrics Portal. http://metrics.torproject.org/.
[46] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. *SIGOPS Operating Systems Review*, 36(SI):271–284, 2002.
[47] C. Viecco. UDP-OR: A fair onion transport design. In *Proceedings of Hot Topics in Privacy Enhancing Technologies (HOTPETS'08)*, 2008.

Appendices

A Core Simulation Engine

Shadow is a fork of the Distributed Virtual Network (DVN) Simulator [10]. DVN is a discrete event, multi-process, scalable UDP-based network simulator written in C that can simulate hundreds of thousands of nodes in a single experiment. DVN takes a unique approach to simulation by running UDP-based user applications as modules loaded at run-time. Among DVN's core components are the per-process event schedulers, a process synchronization algorithm, and a module subsystem. We describe the main components but note that Foo Kune *et al.* [10] provide details in much greater resolution.

Discrete-event Scheduler. DVN implements a conservative, distributed scheduling algorithm (see Figure 8) that utilizes message queues to transfer events between workers. The scheduling algorithm consists of three phases: importing events initiated from remote nodes, synchronizing worker processes, and executing local node events. During the import phase, workers process incoming messages containing events and store them in a custom local event priority queue. After all messages are imported, workers send synchronization messages (discussed below) to other workers and finally process local events in non-decreasing order. Incoming messages are buffered while processing local events and handled during the next import phase.

Multi-process Synchronization. Messages between the master and workers enable global time synchronization throughout the simulation. Synchronized time is vital to ensure events are executed in the correct order since a conservative scheduling algorithm can not revert events. By exchanging messages, each process tracks the local time of all other processes. A *barrier* is computed by taking the minimum local time of each process and adding the minimum network latency between any two network nodes in the simulation. The barrier represents the earliest possible time that an event from one process may affect another process. Each process may execute events in its local event queue as long as the event execution time is earlier than the barrier. This is called the *safe execution window*: any event in this window may be safely executed without compromising the order of events (i.e. time will never jump backwards to execute a past event). Barriers are dynamically updated as new synchronization messages update local times. Future events are allowed to execute as the barrier progresses through time. This synchronization approach allows the distribution of events to multiple processes.

Module Subsystem. DVN contains a subsystem for dynamically loading modules. Modules, pieces of code that are run by nodes, are generally created by porting application code to use DVN network calls and implementing special functions required by DVN. The special functions allow modules to receive event callback notifications from DVN. Although each module may be run by several nodes, module libraries are only loaded into memory once. In order to support multiple nodes running the same module, DVN requires each module to register all variable application state. Using the registered memory addresses, DVN may properly load variables before passing execution control to the module, and unload and save variables after regaining control.

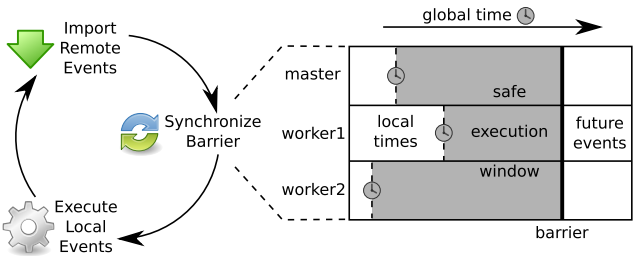
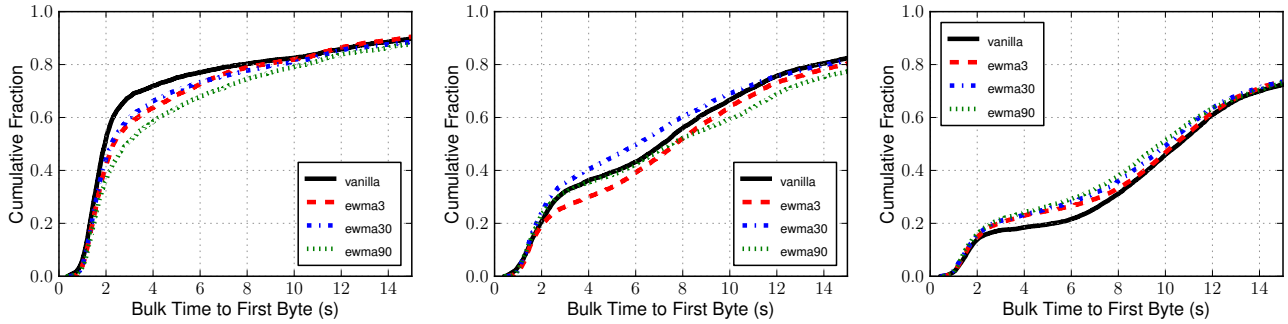


Figure 8: Main loop and conservative multi-process synchronization using dynamic barriers. Safe execution windows are calculated using the minimum local worker time plus the minimum simulated latency between nodes. The barrier is dynamically pushed as local times advance.

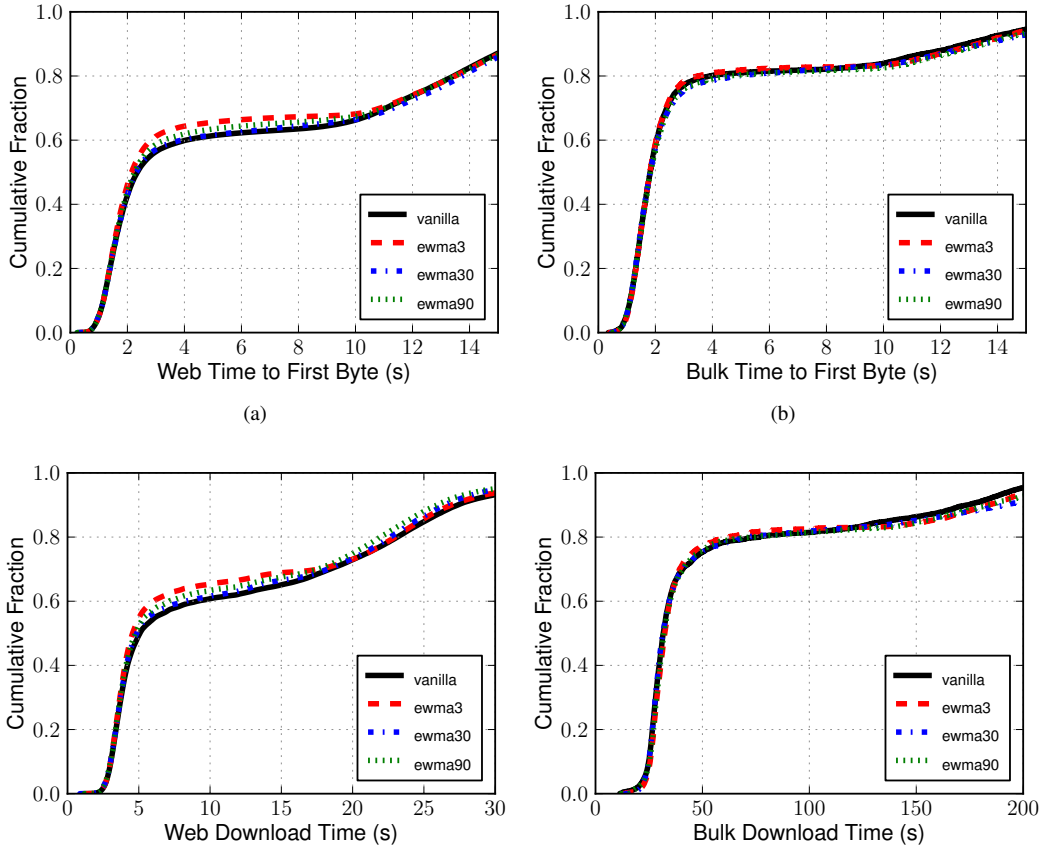
B Circuit Scheduler Performance

In Section 6, we showed results for web client responsiveness and overall performance for both web and bulk clients with different circuit schedulers under different network loads. In Figure 9 we show that responsiveness for bulk clients follow the same pattern as previously shown in Figure 7. (The results were obtained from the same experiments described in Section 6.) Although time to first byte is less important for bulk clients, the results support our conclusion that the EWMA circuit scheduling algorithm reduces performance both under lighter loads and when the half-life is not set correctly. Figure 10 shows performance under an extremely lightly loaded network of 475 web and 25 bulk clients. The results support our claims in Section 6 that choice of circuit scheduler is insignificant for client performance when the load on the network is too light.



(a) (b) (c)

Figure 9: Responsiveness for bulk clients under a varying network load of 950 web clients and (a) 25 bulk clients, (b) 50 bulk clients, and (c) 100 bulk clients. As in Figure 7, the network is less responsive under lighter loads when using the EWMA circuit scheduler.



(a) (b) (c) (d)

Figure 10: Network performance under an extremely light load of 475 web and 25 bulk clients. When the network load is too light, the circuit scheduling algorithm has an insignificant impact on performance.