The Distributed Virtual Network for High Fidelity, Large Scale Peer to Peer Network Simulation

Denis Foo Kune, Tyson Malchow, James Tyra, Nick Hopper, Yongdae Kim
University of Minnesota - Twin Cities

ABSTRACT

The ability to analyze the behavior of large distributed systems can be limited by the modeling tools used. The Distributed Virtual Network (DVN) is a discrete event network simulator providing a platform for realistic, high fidelity, scalable and repeatable simulations of large distributed systems. With a global view of the network, it provides the ability to quantify the behavior of the system under stress and attack conditions. We present the architecture of the simulator along with the simulation results from a real world P2P protocol implementation ported to DVN. We also compare DVN with another similar tool, outlining the benefits of our contribution.

1. INTRODUCTION

Modeling of peer-to-peer systems such as the Kad network has proven to be challenging [30][32][33]. As a result, simulation has been frequently used, but general-purpose simulation has presented several challenges in terms of scalability, fidelity, node diversity support and portability of simulated nodes. Faced with those challenges, many groups developed their own ad-hoc simulators [24].

Simulators can be grouped into two categories based on how much of the underlying communication layers they support [12]. Low level simulators that can support a large portion, if not all of the OSI layers of a protocol stack tend to be very accurate but suffer from resource consumption problems [27][28]. For example, they may test working code by running several instances on a single machine, producing results that are faithful to the implementation but are typically limited to a thousand or fewer nodes [24]. High level simulators supports high level functionality or aspects considered to be important of protocols, and thus tend to simulate down to the network or transport layers (OSI layers 3 and 4). Since they abstract away the lower communication layers, they consume lower resource allowing them to run large scale experiments with upwards of 10^5 nodes [24], but may suffer from a lack of fidelity.

Fidelity is important to the security analysis of a distributed system, as vulnerabilities can reside in subtle behaviors of the protocol. Similarly, efficiency has an important role to allow simulations to scale up to the same order as currently deployed networks. Thus the typical design choices involve a trade-off between the scalability and fidelity of a simulation. Our simulator was developed as a general-purpose packet-based simulator at the network layer with a simple datagram transport layer API for protocol plugins. Those plugins can be ported directly from released code or developed from the ground up on DVN and cross compiled to interface with a real network stack. By running the actual code for protocols above the OSI layer 4, the simulator provides high fidelity simulations while allowing large network sizes on the same order as those deployed in the real world. Abstracting the resource intensive lower layers was a critical design choice for DVN, which is fit for overlay network studies for which it was originally conceived.

1.1 Design Requirements

From the challenges outlined above, we derived the following requirements for our overlay network simulator.

Scalability and Efficiency The simulator should support experiments consisting of a large number of nodes and messages, bounded by the actual hardware such as memory available or CPUs avaialble. It should avoid limits including thread number limits or maximum port numbers. It is important to be able to use multiple CPUs in parallel on a single machine to leverage the current trend of multicore machines. The simulator should allow independent events to be executed in parallel. To support the goal of simulating a deployment of aroung 10^6 nodes, the simulator would benefit from distributed computations in order to take advantage of multiple machines when the resources required exceed the capacity of a single machine.

Fidelity The simulator should be able to run the same protocol code as the actual implementation to minimize risks of bug introduction in the released code. Moreover, the simulator should allow code from real implementations to be ported from current active projects to run on the virtual network thereby allowing accurate modeling of the actual network. The porting effort should be significantly smaller than the re-implementation effort. The code designed for the simulator should also be easily "exported" so that it can be used on a real implementation. Additionally, the simulator should provide a means to support the following secondary fidelity goals:

Network Model. The architecture should provide support for realistic network conditions encountered by large deployments such as non-transitive connectivity and network partitions.

Event scheduling. The simulator should support scheduling of a series of events — such as node addition, deletion, network merge or partitioning — at predetermined times, to

enable replicable experiments.

Node diversity. The architecture should support nodes running with different settings, different versions of a protocol stack, or completely altered nodes. This allows the modeling of situations like incrementally deployed upgrades, networks with "super peers," or the effects of malicious nodes on a network.

1.2 Our Contribution

This paper describes our hybrid design that instead of doing a traditional fidelity and scale tradeoff, provides both with plugin modules on top of a layer 3 simulator with a simple datagram transport layer. For improved fidelity, code developed for the simulator are built as libraries and can be cross compiled on a real implementation. At the same time, by eliminating the simulation at layer 3 and below, it reduces the resource consumptions when compared to more complete simulators such as NS2 and allows simulations to scale up to hundreds of thousands of simulated nodes on a single machine. With some porting effort, real implementations can run inside of the simulator to ensure maximum fidelity. We also built a distributed architecture to spread the load across multiple machines. We named our simulator the "Distributed Virtual Network", or DVN for short.

We compared our simulator to the state of the current state of the art for peer to peer simulation in the form of the WiDS[20] toolkit and show a factor of 4 in improved performance when DVN was compared to WiDS. We also ported the aMule[1] client version of Kad nodes to simulate a large Kademlia[21] network on DVN.

We ran large Kad simulations with DVN on different platforms including single and multicore machines with different configuations, and Amazon Elastic Computing Cloud (EC2). We compare the simulation running times on those platforms and measure DVN's performance on each one.

2. RELATED WORK

Large scale networks simulation is a hard problem [34] but a necessity to understand the behavior of massively distributed systems such as peer to peer networks. Simulation is an invaluable tool not only for designing the system but for understanding its behavior in the presence of adversaries and evaluating mitigation strategies.

2.1 Simulation techniques

Parallel and distributed simulators based on discrete events have been developed previously [35, 7, 13]. Parallel Discrete Event Simulation can be divided into two categories, conservative and optimistic [11] [12]. In conservative engines, the logical time is advanced in a coordinated fashion. Optimistic simulators try to move ahead with the risk of having to roll back in time. DVN uses a conservative engine to guarantee the chronological order of events. However, it uses the network delay to build a safe window to allow independent parallel processes to move ahead if they can. DVN uses a master-worker relationship similar to WiDS [19] with the other nodes to avoid flooding the physical network with synchronization messages [36]. In the WiDS toolkit [20], the authors introduce Slow Message Relaxation (SMR), a variant of optimistic scheduling where the scheduler for a simulator worker might be ahead of the schedulers for other simulator workers. If it happens that a message is destined to a virtual node running on the worker who is ahead, that worker treats the message as a "slow" message, analogous to one that has experience severe network delays. This technique is a tradeoff between performance and accuracy since slow messages might skew the statistical distribution of message delays. Given DVN's performance on a single machine using worker node decoupling, there was no need for such a tradeoff.

2.2 Related Simulators

WiDS [20] was designed to allow development of protocol stacks from scratch that could be ported to real applications. We wanted to do the reverse. We wanted to take the real code and run it on a simulator with minimal changes to adapt to the simulated network layer.

NS-2 and NS-3 [27, 28] could support ported protocols, however porting the Kad node to an NS-2 custom agent would have required a large porting effort, along with restrictions on the size of the network. Using PDNS - Parallel/Distributed NS to address the scalability issue would have been possible, but it was not clear that the performance would have been acceptable for very large simulations in distributed mode. DVN starts with supporting large numbers of Kad nodes per simulator instance and allows distribution or those instances. NS-3 pays close attention to realism and simulates the undelying layers to a fine detail. Each node is a computer's outer shell and hosts a complete communication stack. In our experiments, we only needed the top layers of the stack, including the application layer where the overlay routing takes place. Therefore, to optimize the simulator we had to trim out the underlying layers from the IP network layer and down.

ModelNet [42] could support ported protocols such as the kad protocol, but it would have performance implications limiting us to simulations size a couple of orders of magnitude smaller than what we have achieved.

SSFNet [8] is an infrastructure built in Java that can support nodes written in Java and C++ as long as they are compliant to the SSFNet API bindings. The simulator has been used to run up to 384,000 nodes in the Network Worm simulation published by Liljenstam et al. [18] In that publication, the authors mention that their model used approximations of the worm infection patterns to generalize the simulation at a coarse level, while only simlating parts of the network in detail. In our publication, we wanted to run a full protocol implementation for the entire network with hundreds of thousands of nodes, which would have been problematic with SSFNet unless we used Liljenstam's method, thereby potentially affecting the accuracy of the simulations.

2.3 Scalability and fidelity of existing simulators

Haeberlen et al. [14] suggests that current simulation and experiments are considering single points in the entire possibility space. They are critical of the trend to accept ns-2 and planetLab [6] results as the general case when in fact generalization might not be appropriate. We believe that the scalability and fidelity of the DVN would allow researchers to have more meaningful simulation results close to actual implementation.

Naicken et al. [23] analyze several simulators along some of the criteria mentioned in the introduction. The simulations surveyed include the following: P2PSim [30], Peer-

Sim [32], Query-Cycle Simulator [38], Narses [25], Neurogrid [26], GPS [43], Overlay Weaver [29], DHTSim [9], and PlanetSim [33]. Only one simulator [32] was reported to support 10^6 nodes, and those were virtual nodes purposely built in Java for the simulator. With DVN we wanted to port an existing protocol implementation in C to the simulator environment, along with the heavier resources it might consume due to the original target being a single machine. Naicken et. al. also report that the majority of papers in the Peer-to-Peer litterature that use simulations tend to use their own custom simulator, built specifically for their purposes. Although DVN was built with Kad simulations in mind, it was an attempt to be more general so as to benefit other projects.

ModelNet [42] allows researchers to run unmodified software prototypes and supports a finer granularity of the network topology than the description supported by the dsim language of DVN, but DVN can simulate much larger networks using ported virtual nodes.

Other simulators such as OPNet, GlomoSim(wireless), NetSim and OMNeT++ have the same tradeoffs of running the real code against scale. In DVN we trimmed down anything that was not needed, from the network layer down, giving us an efficient, scalable and high-fidelity simulator.

Of the simulators mentioned above, only PeerSim can scale to millions of nodes, but it requires code in java which might limit the ability to port the code to a real implementation, or vice versa. The emulator approach by Kato and Kamiya [15] uses java as well, but is at the other end of the spectrum offering high fidelity with real code, but is severely limited in scalability (number of active nodes up to 1000).

MACE [16] is a compiler that outputs C++ source code from protocol specifications. The output generated could be compiled into a DVN module once the event callbacks and datagram network interface are put in place. In DVN, we have a single API to the simulation stack or the real network stack, which can be viewed as a simple MACEDON [37] API. While DVN does not compile protocol code, it can import the code from another implementation for simulation.

2.4 Emulators and virtual machines

VCSTC [39]: The Virtual Cyber-Security Testing Capability (VCSTC) is intended to test the operational functions and security impacts of a given network device. VCSTC then simulates the network environment in which the device will be deployed, which could include large scale networks. While reusing the VCSTC infrastructure would be possible, it would imply major retooling of the architecture to move the focus from a device under test to the behavior of the network itself. VCSTC uses nodes emulated inside of virtual machines to generate the test traffic. In our case, such an architecture would not scale up to the level that we needed.

DETER Test Bed [2]: The Cyber DEfense Technology Experimental Research (DETER) Network is based on a large number of physical and virtual nodes, built on the emulab platform. It allows the execution of actual malware on nodes with a very small likelyhood that the malware would detect that it is not running in the virtual world. The experiments allows researchers to gather empirical evidence on how interconnected malware would interact. The test bed also allows arbitrary topologies and could assist in network and security experiments not necessarily involving active malware. This configuration is very resource heavy and suffers from scalability issues. Accurate results from running protocol implementation can be obtained, but only on 1:10 or smaller models of the network.

3. ARCHITECTURE

3.1 Overview

The DVN simulator is composed of a central engine implementing conservative discrete event scheduling with non-blocking events. Simulations are directed from a distributor that can dispatch tasks to separate processes. The distributor computes a safe windows that allows each process to run a group of nodes in parallel. The scheduler supports pluggable run-time library modules that implement the protocol stacks. Those modules, written in C/C++ allows porting of communication protocols directly from the released implementation enabling a high fidelity representation of the real network. Such a tool could be important to the security analysis on current distributed systems where subtle protocol behaviors can be important. The network topology is composed of networks with predefined statistical delays and packet losses. Those networks are interconnected with to each other with links having their own delay and packet loss characteristics. The simulation scripts loaded by the simulator allows dynamic creation of nodes and networks. The logging mechanism is centralized and allows the protocol implementation to output their own logging metrics for maximum flexibility. A side effect of the fidelity requirement is that the nodes are running in the context of the simulator in the user space process and they have access to the process I/O as well. Thus, they are allowed to direct their output to files of their choosing in addition to using the SVN logging interface.

3.2 Event Scheduler

DVN's scheduler is a variant of a conservative calendar queue[5]. The master instance of DVN spawns one or more worker instances as separate processes. The master instance only coordinates events by using its master event scheduler. Each worker instance gets a copy of the scheduler, and resynchronizes the differences after a run for a safe period of time. The scheduler has a tunable time slot granularity, typically set to one millisecond thus all events are approximated to the nearest millisecond. Inserting an event is done by first hashing the current time slot to a key in our hash table. If there are multiple time slots in a single hash table bucket, a simple minimum heap is used to identify the correct one in $O(\log(1+\alpha))$, where α is the ratio of keys to the size of the hash table. Finally, a simple dynamic array is used to keep track of all concurrent events in a single time slot. DVN uses another minimum heap to keep track of all the time slots, allowing for retrieval of the next time slot in $O(\log n)$ for n occupied time slots on the scheduler. The overall structure for event insertion and lookup is shown in figure 1.

3.3 Parallel and Distributed Simulation

For large simulations, DVN can distribute the work load between multiple processors running parallel instances of DVN. The master DVN instance divides the current simulation into sections containing an equal number of nodes which are then sent to each DVN worker through the Simulation Distributor as illustrated in figure 2. Each DVN worker instance loads the module libraries that contain the imple-

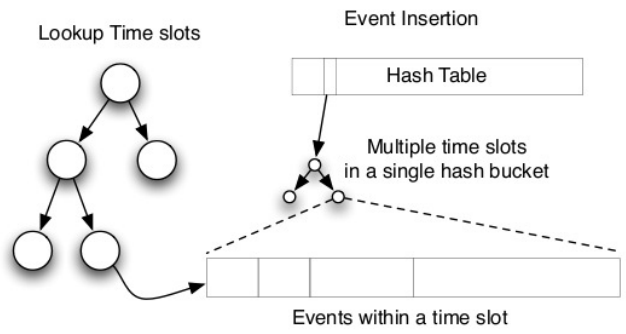


Figure 1: DVN's event scheduler

mentation of the protocol. The Simple Network Routing Interface (SNRI) layer resides between the protocol implementation and the network stack, providing event callbacks and datagram messaging services. DVN supports two models of data gathering; logs can be made from the central dispatcher or individual modules can log events based on their own internal states.

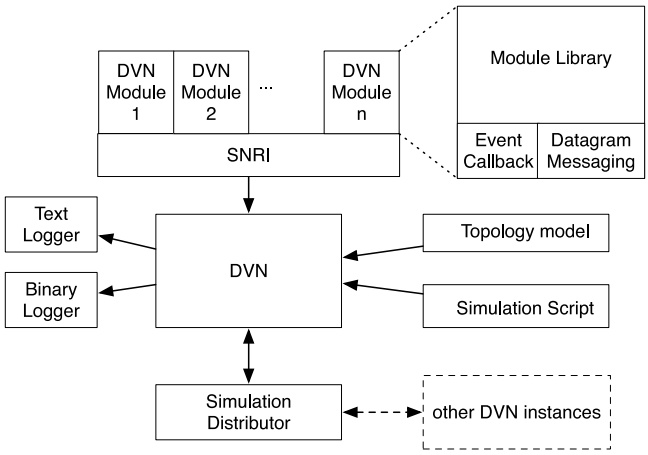


Figure 2: DVN's Architecture

3.4 Simple Network Routing Interface (SNRI).

SNRI defines a simple set of inbound/outbound operations on a DVN module. It allows the protocol implementation to set callback events or send datagrams to a remote host. When an event is triggered or a message received, SNRI makes the appropriate calls into the module library. SNRI was designed to be simple to enable rapid development and testing of a protocol implementation. Using a lightweight wrapper, the protocol code can then be deployed on a real network stack. This allows for the identical code to run both on DVN and on a real network.

3.5 DVN Modules

Virtual nodes running in the simulator communicate asynhronuously with the underlying simulated network layer. This avoids the potential use of blocking calls that could cause a DVN worker to halt. An incoming datagram for a node instance will trigger a callback function to allow the node to process it. Global variables within a module can be safely used as long as they are registered via SNRI, which will then allow DVN to track those variables as part of the current node's state and swap those in prior to running that node's logic. Because DVN can load more than one module in a given simulation, it allows a user to create a heterogeneous network with multiple variants of a protocol, or even completely different protocols. This feature is an important tool to study application layer protocols such as overlay network protocols.

The porting process from a real world implementation to a dvn module requires some effort, but SNRI was designed to streamline the process. The original protocol source code should be in C/C++ to be able to link against SNRI. The isolation of the communication protocol is done in four main steps. First, the algorithm is extracted by removing the upper application layer, including the Graphical User Interface if any. Next, the lower network layer is substituted with the SNRI layer. Since SNRI provides a datagram interface, the porting effort from UDP to SNRI should be minimized. Next, the program flow might need to be adapted to run on top of the callback system. SNRI does not provide blocking calls in order to ensure that a virtual node safely coexists in a process with other node instances, so the protocol algorithms have to use an event callback approach. Finally, the global variables have to be registered with SNRI to allow DVN to track an individual node's state.

3.6 Logging.

The ability to gather meaningful statistics for the simulation results is one of the key components of a simulator [24]. DVN provides a central logging mechanism that can save events in ASCII text format or binary format. Individual modules can log events based on their internal states, or DVN can log events from the simulation itself. The statistics collection methods are left to the module developers in order to ensure maximum flexibility in terms of the type, frequency, and amount of data meaningful to the analysis of the simulated network. Future plans include a direct connection to a database for centralized logging to help in data analysis without compromising on flexibility.

3.7 Network Model.

DVN is intended to support simulations for application layer overlay protocols and thus only models the underlying network with delays and reliability. It allows for a coarse granularity topology definition of small networks interconnected into a larger network with multihop paths being approximated with a single delay and reliability model. The model includes a base delay, a width and a long tail allowing a small number of packets to be delivered with very long delays to match the reported end to end delay measurements [4][3][31]. Using the DSIM language, separate networks can be created as an event during the simulation and those networks can be connected at anytime using events as well. This model was chosen to provide an adequate abstraction to avoid burdening the user with specifying all possible links in the system.

3.8 Simulation Description Language (DSIM).

To aid in the simulation setup, DVN provides a mini scripting language called DSIM, which is built using Flex [10].

DSIM is a simple language used to model simulations within DVN by defining events. It describes all components involved in a DVN simulation: network topology using network creation events, node instantiation events, network events, and simulation timing with start and stop events. The language allows creations of small interconnected networks, and the specification of the delay models withing those networks. The network definition allows hierarchical networks analogous to the AS-level topology of today's Internet. Scripting commands are available to allocate and introduce nodes into the network by dynamically loading modules described in the porting process above. All events have a time stamp allowing the simulation to alter the network topology at any time to simulate large-scale network events. The language also supports simultaneously loading multiple modules to allow for heterogeneous systems that have multiple types of nodes.

3.9 DVN in parallel and distributed mode

Basic Layout.

DVN is designed to support parallel and distributed simulations by using a simple master-worker model between a single master synchronization process and worker processes running subsets of the simulation. The master of a simulation is responsible for assigning virtual nodes to actual DVN worker instances running the simulation. It then distributes the events for relevant virtual nodes to the assigned DVN workers. Each worker keeps track of local events and communicate events that affect nodes on remote workers by talking directly to those parallel instances. The master is responsible for synchronizing all the workers by computing and sending a safe simulation window to all workers and waiting for them to complete before starting the next time slot.

Paralell and Distribution System.

Each worker system talks to the master for synchronization, and talks to every other worker directly to communicate network events. The master dispatches events that indicate simulation events such as nodes join and the workers each manage the raw handles to the node instances. As the simulated node process their individual events,they create packets and timed event, those are tracked and processed by the worker. Packets sent to nodes that resides on remote processes will be sent directly to those processes after the appropriate process has been resolved from the destination address of the packet. In this way, DVN limits potential simulation overhead by allowing the workers to directly communicate only when they need to. Once all events for a given simulation time have been completed, the master engages all the workers to begin processing the next safe time window.

Event cone worker node decoupling.

In order to determine events that are safe to process in parallel, in DVN we introduce a model where the master computes a safe execution window for all worker instances, based on the network delay. The core idea is that events generated on a node will take a minimum delay to get to the destination. Thus, in the meantime a remote worker process can safely run in parallel until the time when it will be affected by the generated event. This model is analogous to the special relativity light cone as defined in Lorenz-Minkowski geometry [22], where a subject will not experience an event from a remote source sooner than the time taken for light to travel from the source to the subject.

Consider G to be the set of all networks in our simulation, with each network having a network model specifying a mean delay, d and a width δd. We define $\tau = d - \delta d$ to be the fastest that a packet can move from a node to another node within the same network. In other words, packets will arrive no sooner than $t + \tau$ at the destination, for a packet sent at time t. Thus,

$$\forall i \in \{1, ..., |G|\}, \tau_i = d_i - \delta d_i$$

Our safe window is defined by w,

$$w = t + \min(\tau_i), \text{ where } t \text{ is the current virtual time.}$$

The events in the scheduler are sorted by time and the master scheduler checks the time window of events that are safe to process before dispatching those windows to parallel workers.

DVN Worker Instance.

To manage its internal events, each worker maintains a scheduler for packets, a scheduler for timers, and a scheduler for DSIM events that include a portion of nodes that will be created on a particular worker following the simulation script. At each synchronization step, the worker is given a time segment to process from the master and notifies each scheduler to process all events within that window.

DVN executes directly the module code as a library and thus allows calls to the system I/O. Using this property, a bridge module can be created with a dual stack; one talking on DVN using SNRI, and the other talking over a real IP stack. That bridge module can act as a NAT router and forward requests from the real world into the simulator after stripping the appropriate headers off. This method allows the validation of the protocol implementations running in the simulator, allowing a greater fidelity to the real world. In this configuration, the bridge module will have to compensate for the difference between the wall time and the simulation time prior to injecting messsages between the two networks.

4. SIMULATING THE KAD NETWORK ON DVN

The Kad network is a large deployed peer-to-peer distributed hash table [40]. The network hosts over a million nodes on average and can be accessed various clients, one of them being aMule [1]. To the best of our knowledge this network has never been simulated on a very large scale. There has been reported performance issues with this network [41]. Some of the proposed fixes would require varying levels of client modification and could potentially have serious side effects if released alongside "broken" clients. This makes it a very interesting study subject for large scale simulations on DVN. We used the original aMule code base to extract the Kad Distributed Hash Table engine as a stand alone executable first. Then we ported it to a DVN module. To ensure correctness we verified that the behavior of the Kad module on DVN matched the one of the real Kad nodes on a physical network.

4.1 Porting Kad

We used aMule v2.1.3 and isolated the Kad protocol implementation by removing the graphical user interface, resulting in a stand alone executable that can be used as the reference implementation when testing the network behavior. We then used the stand alone protocol implementation and replaced system calls made to the operating system with DVN functions including those to send and receive packets. Since both of those network functions operate on basic data structures, the porting process was relatively simple. Such a process is outlined in figure 3. All references to the system time were changed to references to the virtual DVN time. The memory management of Kad was done using static and global variables. This was changed to reside within a single global structure that is allocated at initialization time and passed to the simulator as part of the node initialization procedure. DVN then tracks the global variables for each node automatically, ensuring that each node's state is available when the code for that node is running.

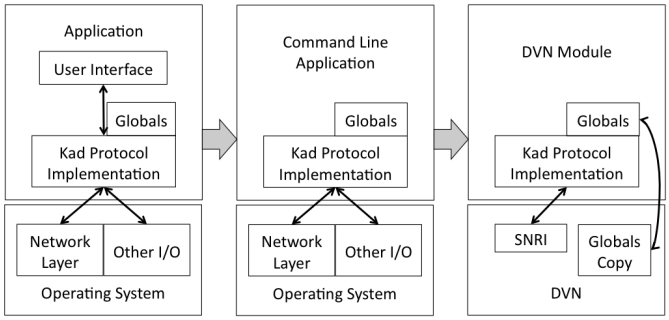


Figure 3: Summary of the porting process of the Kad protocol implementation to run on DVN

After the Kad protocol implementation was ported and running, we found the simulation runs to be slow. This wasn't an issue with DVN, but rather the Kad module itself. The original implementors had only intended one copy of this software to run per machine. After the porting process, a virtual Kad node was still consuming a lot more CPU and memory resources than a module developed from the ground up for DVN. When simulating around 10^5 nodes, bottlenecks started to become evident. The Kad implementation had timers that were checked very frequently for expiration. We changed the Kad node to use callback mechanisms instead of a frequent polling. Additionally, Kad was allocating three very large static buffers to hold the textual results of searches, which were not needed in our simulations and were therefore removed. After those fixes we found that the memory consumption went down and the simulated message rates improved.

4.2 Evaluation

We evaluted the correctness and performance of DVN by comparing the behavior of the virtual network to experiments using real kad nodes in a controlled environment.

4.2.1 Simulation and real word

To verify the correctness of DVN, we compared the network behavior of virtual Kad nodes running on DVN with real nodes running in a controlled network. In this experiment, we ran 3,000 virtual nodes for two virtual hours on DVN and we ran 3,000 real Kad nodes on a separate testbed (call it Itlabs), consisting of 14 actual machines running several hundred instances of real nodes. Each real node had a unique ID, and were using different UDP ports. Figure 4 shows the number of lookup messages [1] sent per 60-sec window. The virtual DVN Kad nodes and real Itlabs Kad nodes exhibit the same behavior. An average latency of 250ms was used in the DVN simulations. Measurements carried out by Leonard and Loguinov [17] mentions latencies around 200ms between DNS servers. Thus for end users we chose a 250ms delay and verified that our simulations agreed with the real world experiments.

4.2.2 Symmetric Links

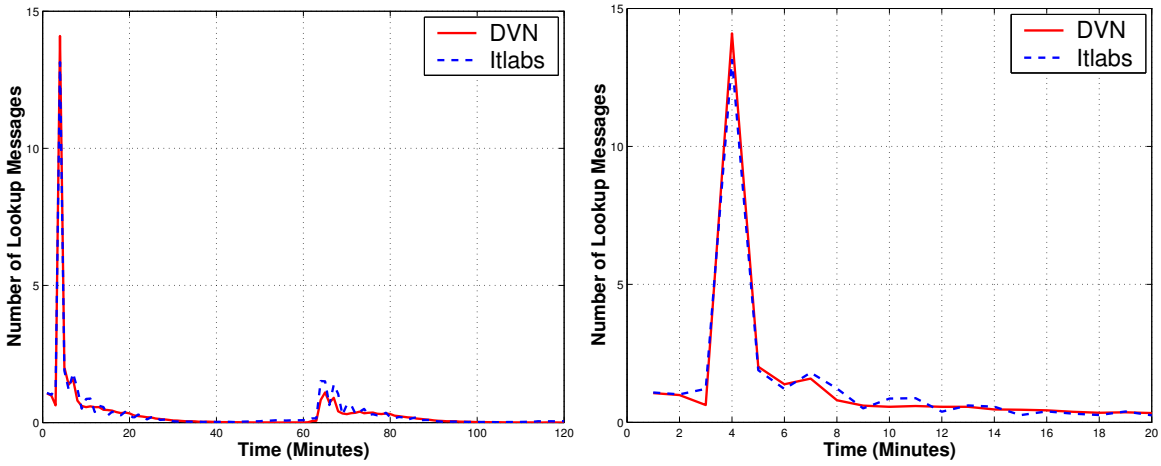
We compared the memory footprint between running a Kad node on a normal machine and running it on DVN. 1000 nodes are created in DVN and run for 30 virtual DVN minutes. The peak traffic happens 200 seconds in the simulation. The virtual memory usage for DVN at that point was 43.1 MB. The same experiment was performed, deploying 1000 Kad nodes on a single machine and letting them run for 30 minutes. During the peak traffic time, each Kad node was using about 1.5 MB for a total of 1.5 GB of virtual memory for 1000 Kad nodes deployed on a single machine.

We looked at the global behavior of a small virtual Kad network over time. Figure 5 shows multiple snapshot of the global connection map. The nodes' routing tables are displayed as rows with "hotter" colors indicating a higher density of nodes appearing in routing tables around that region. The nodes were sorted by hash IDs to reveal the XOR-metric connections of the Kad network. The bright vertical streaks on the plot indicates initial bootstrapping nodes that are very popular since they were in the network first and were used to bootstrap new nodes in the netwok. The diagonal blocks indicates nodes keeping other nodes with a close hash ID in buckets within their routing tables.

A symmetric link is where node A has node B in its routing table and node B also has A in its routing table. We want to know the percentage of symmetric links in a Kad node's routing table. This allows us to determine whether the Kad network is symmetric or asymmetric. We expect that the Kad network, due to its design, would be mostly symmetric, because each node would be added in the same bucket at the appropriate level of the routing table.

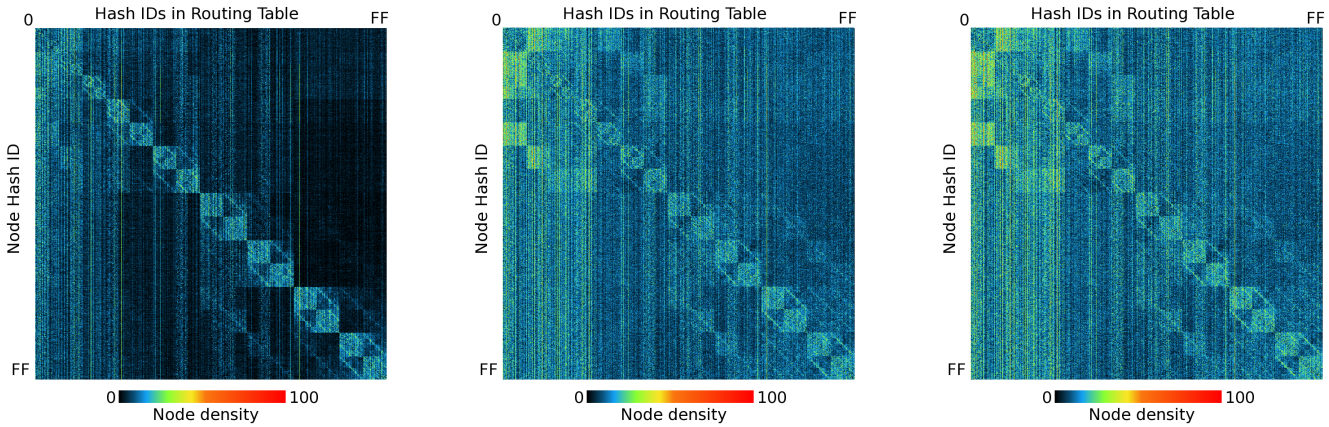
Finding the number of symmetric links in a routing table involved having access to both a node, its routing table, and the nodes in the routing table. This would be hard to evaluate on the real network as we do not control all the nodes; our real deployment is limited to 16,000 nodes. Crawling the whole Kad network and polling every node's routing table is possible but would involve a substantial bandwidth cost and would not result in an instantaneous snapshot of the network, since it would take on the order of one hour at 100Mbps to poll every node's routing table. By that time, the earlier nodes that were polled would have had their routing tables changed due to churn. Moreover, deriving the percentage of symmetric links analytically is difficult due to its dependence on network dynamics, mostly node churn again. Thus, simulation was the most feasible way of quantifying

[1]These lookup messages are used to maintain nodes routing tables.



(a) 2 hours (b) 20 minutes

Figure 4: Kad node network behavior measurement



(a) initial (b) 1 hour (c) 3 days

Connection map of a 50,000 node simulation sorted by node hash IDs. At time t=0, t=1 hour and t=3 days. The warmer colors indicate a higher density of nodes present on routing tables. The checkered pattern along the diagonal is caused by the Kad buckets where nodes keep track of other nodes close to their hashes. The popular nodes appear as vertical streaks.

Figure 5: Global view of a sample Kad network experiment sorted by hash ID

how symmetric the Kad network actually is. We found that the percentage of symmetric links in our simulated environment is 53%. Thus, for a particular node A, about half of the nodes in its routing table also have point to A in their routing table.

4.3 Heterogeneous Kad Simulation

To ensure greater flexibility within simulations, DVN supports heterogeneous nodes within a single simulation. In this experiment we ran 200 Kad nodes over 30,000 seconds. Half of these nodes ran our standard Kademlia module, while the other half ran a modified version. This modified version was programmed to stop responding to network activity at 15,000 seconds, emulating a packet dropping attack by multiple nodes. Figure 6 presents measured the routing table changes of a single node, while Figure 7 shows the number of different messages sent and received. Interestingly, routing table size does not change for about 5,000 seconds after the modified nodes died, meaning that it takes more than 5,000 seconds to stabilize the routing table. The latter figure shows that the number of outgoing messages increases, while the number of incoming messages decreases showing that unresponsive nodes had a noticeable impact during the time interval.

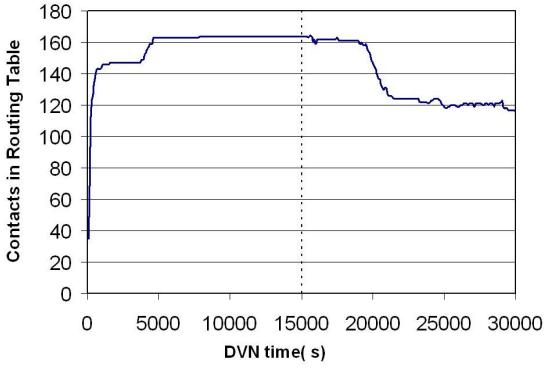


Figure 6: Kad node network behavior measurement

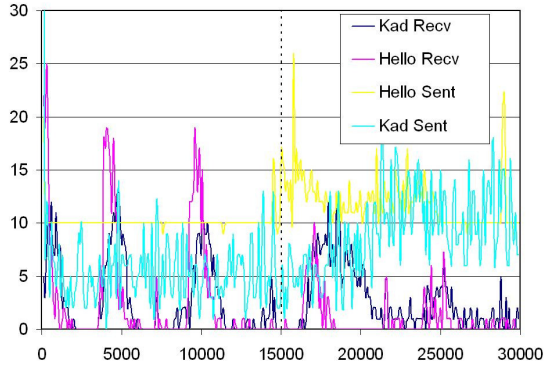


Figure 7: Kad node network behavior measurement

5. PERFORMANCE

5.1 Scalability with Kad nodes

Our test platform included a DELL PowerEgde 6950 with 4 dual-core AMD 8216 (2.4GHz) CPUs and 16GB of RAM. We ran DVN with the Kad module, logging output serially to a file. We generated DVN simulation (DSIM) files for 1,000, 5,000, 10,000 and 20,000 nodes. Each of the simulations were on a single simulated network with 250ms latency for message delivery and no dropped messages. The bootstrap set consisted of 10 interconnected nodes. Then the rest of the nodes were added in batches of 300 nodes at a time, using any of the nodes in the aforementioned set as a bootstrap node. We were able to successfully perform simulations of 200,000 Kad nodes on this machine. Maximum memory usage during this simulation was 10 GB or 50 KB per node. We were able to simulate 14 DVN hours in 77.4 hours – a slowdown of 5.5 times. While this is slower than realtime, it is expected for such a large simulation.

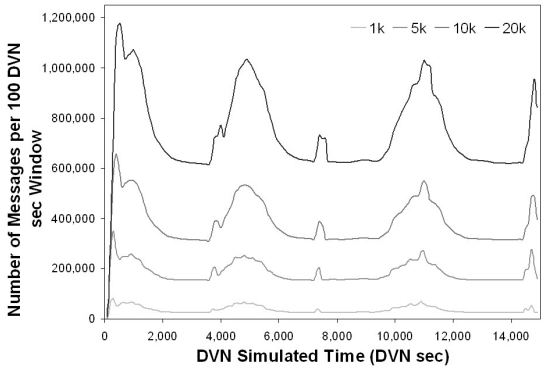


Figure 8: DVN Performance (messages) for 1, 5, 10 and 20 thousand Kad nodes

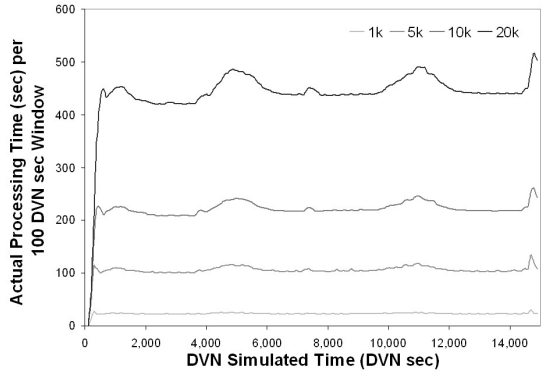
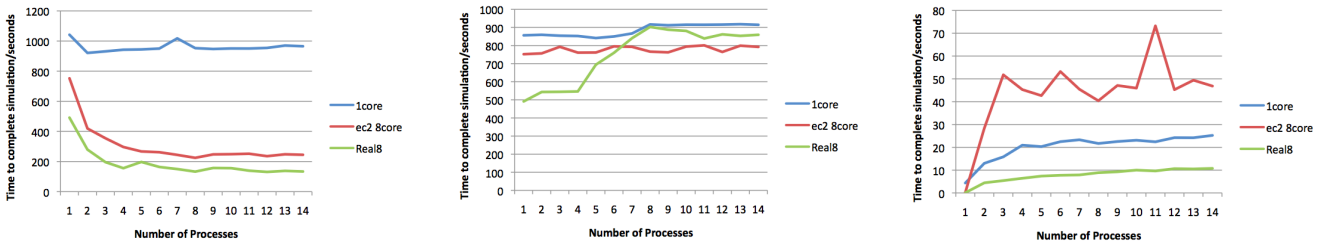


Figure 9: DVN Performance (virtual time speedup) for 1, 5, 10 and 20 thousand Kad nodes

The output logs were divided into windows of 100 DVN seconds. The number of messages sent in each 100-seconds window was recorded and plotted as the simulation progressed, as shown in Figure 8. These messages include "Hello", "Kademlia" and "Bootstrap" messages and are used for discovering the network. The initial spike is due to the bootstrapping nodes discovering the network. The periodic waves



(a) Actual completion time (b) Time spent in user space (c) Time spent in system calls

Figure 10: Time to complete a simulation and number of parallel processes used

are due to various routing table maintenance messages. We also measured the amount of time required to process each of those 100 DVN second time slots and plotted them as the simulation progresses as shown in Figure 9. It is clear that the processing time is directly proportional to the number of messages being transmitted within the system. The number of messages is directly proportional to the number of nodes being simulated. Message transmission events are visible only when large traffic spikes are observed and those become the dominant factor affecting network performance.

The overhead incurred by the DVN simulator is relatively small compared to the resource consumption of the actual Kad nodes. The actual DVN architecture requires less than 30 MB. The rest of DVN's memory consumption depends on the nodes' state and the network traffic, since the scheduler will need to hold on to the packets until delivery. Thus, the memory requirement is dependent on the number of nodes, the number of packets and the size of those packets. In our overhead estimation experiments, we ran between 1000 and 1500 nodes at 100 nodes increase per experiment. The memory consumption on the heap was noted for each experiment. We were able to estimate that the memory consumption for each Kad node on a small simulation was about 165 KB on average. This number goes up with larger networks since each node has a larger routing table that occupies more space. The complete resulting overhead of DVN including memory used to hold all messages during transit was found to be less than 30%.

5.2 Scalability using multiple processes

DVN leverages the common multicore architecture of modern processors by distributing the work between multiple parallel instances. On a single machine, it forks multiple worker processes and manages the Inter-Process Communication (IPC) with a section of shared memory. We ran simulations on multiple platforms using between 1 and 14 parallel intances of DVN workers and measured the time taken to complete those simulations. We ran our tests using a simulation generating a large number of events per simulated millisecond. This would create the same conditions for the scheduler as a large simulation would. The module that generates those events was kept as simple as possible so that the computation of the scheduler would dominate, allowing us to benchmark it appropriately. We ran the same simulation on three platforms while being careful not to exceed the amount of physical memory on those platforms to avoid memory paging costs. The platforms were as follows:

- Single processor, single core AMD Athlon 64 3200+ at 2.0 GHz with 4GB of memory, named "1core"
- 8 virtual core instance on Amazon Elastic Compute Cloud (20 EC2 compute units), with 7GB of memory, named "ec2 8core"
- 8 core (dual processor, quad-core) Intel i7-920 at 2.6 GHz with 6GB of memory, named "real8"

The results are shown in Figure 10. As expected, in Figure 10 (a), the "1core" machine performs the worse. It showed no significant gains from using more than one worker instance. The multi-processor machines saw significant gains until 5 parallel instances were used. At which point, the master scheduler was spending more time to distribute events to instances, and many instances had to wait for others to complete their current time slice, even with decoupling enabled. This behavior is evident in Figure 10 (b) where there is a jump time spent by the user processes on a simulation with 5 or more parallel worker instances and some of them have to wait for other instances to complete. The simulation on EC2 behaved in a manner very similar to a single core machine, even though it had 8 virtual cores. That behavior might be due to the way the hypervisor allocates computing resources. Figure 10 (c) shows the amount of time spent in system calls. With increasing parallel instances, there are more accesses to the shared memory segment, along with locking and unlocking of the relevant portions for a write event. The simulation on EC2 again shows a peculiar shape that could be attributed to the hypervisor's behavior.

Figure 11 shows the time to complete a simulation with 50,000 Kad nodes on an 8 core machine against the number of parallel worker instances. A noticeable bump can be see when we use more than 8 instances on an 8 core machine as we run out of parallel processors to support the parallel worker instances.

6. DVN V/S WIDS

A simulator with similar requirements as DVN is WiDS[20], a toolkit aimed at facilitating the development and debugging of distributed system. It can operate in a distributed fashion and has been reported to run very large simulations on the order of 10^6 virtual nodes built for WiDS, using hundreds of machines in a distributed simulation. DVN and WiDS share very similar design characteristics and WiDS has been known to be very scalable, given enough processing power. In that respect, it is the current state of the art, making WiDS a very good comparison point for DVN. There

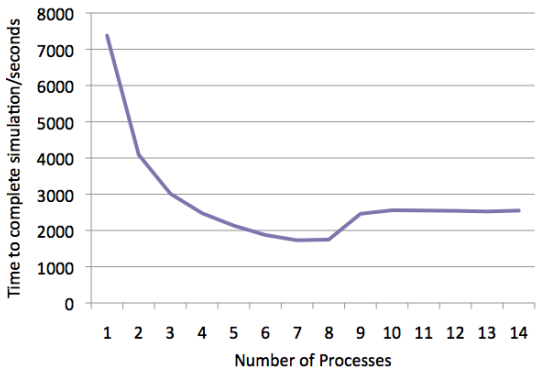


Figure 11: 50,000 node virtual Kad network with multiple worker simulator processes

are some differences that make DVN more flexible and more efficient than WiDS, making DVN better suited for large scale modeling of networks with heterogeneous nodes.

6.1 General Architecture Comparison

Event Scheduler DVN and WiDS use a similar construction for the event scheduler. WiDS uses a C++ map for all the time slots it is tracking, and DVN uses a combination of a hash table and a binary tree to resolve collisions in individual entries of the hash table. WiDS then uses a linked list to track individual events within a time slot. DVN uses an array that grows dynamically. When a particular time slot is processed, DVN can free a large array very effectively. Because the hash table starts with a large number of entries, the constant time lookup dominates for most simulations.

Although WiDS has a conservative scheduler, it introduces the "Slow Message Relaxation" (SMR) which is an optimistic strategy. With SMR, the distributed simulation running on a parallel instance will try to go ahead for a window of virtual time, even if that window has not been identified as safe. If that parallel instance is given a message with a timestamp for which the local simulation has exceeded, WiDS identifies the message as a "Slow Message" and simply overwrites the message's timestamp to the current time. While this avoids state conflicts that would cause a rollback, a large number of Slow Messages could affect the statistical properties of the network model. DVN ensures that windows provided to its parallel instances is safe by using message delays in the Event Cone Decoupling mechanism.

Modules DVN uses dynamic libraries for its node modules. With the dsim language to run the simulations. Dsim allows a user to run instances of multiple modules implementing multiple protocols in a single simulation. Those modules are loaded dynamically as instances of those nodes are created following the script inside the dsim file. WiDS runs the node itself, with a modified lower layer allowing for multiple instances of the same node, but precluding node diversity.

Events Events on both DVN and WiDS can be divided into two categories: network and callbacks events. Callback events are inserted directly at the target time on the scheduler and network events are inserted with an added delay calculated using the topology of the network between the sender and receiver nodes. The events are dequeued at the proper time and dispatched to the destination virtual node for processing. With both types of events going through the scheduler, the efficiency of the scheduler is of great importance.

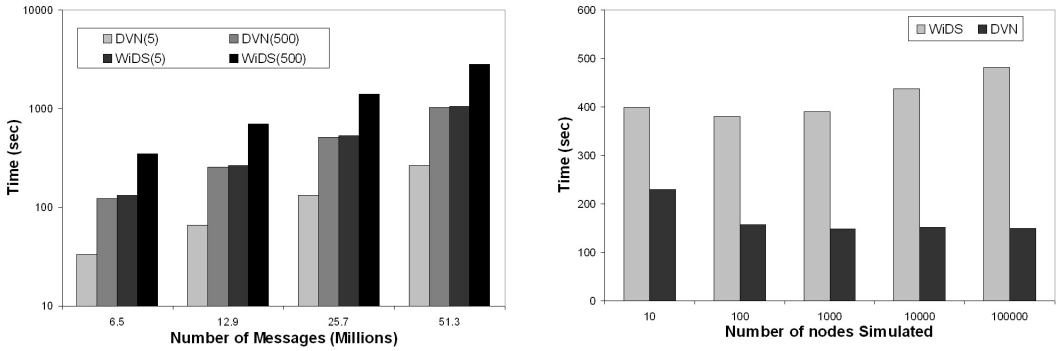
Topology Because of DVN emphasis on performance and scalability network events are simulated at the application layer above the OSI layer 4 (Transport). DVN assigns all nodes a virtual IP address. Each node in a DVN simulation is placed into a network. Each of these networks has a user specified delay and packet success rate. networks can be connected to one another by specifying the interconnect delay and inter network packet success rate. The resulting topology is similar to the AS level graph, with DVN abstracting intra-AS communication with a single delay/loss model. WiDS models the network topology by specifying every single necessat edge between communicating virtual nodes in the network. This requires n^2 connections for an n node simulation, or an a priori knowledge of the nodes that will talk to each other. Additionally, as the simulation size grows, the description of the network becomes a challenge and the parsing time for the simulation file increases, affecting the start time.

Parallel Simulation To achieve greater scalability DVN can distribute parts of the simulation to parallel instances with state synchronization over a shared memory link. A master instance reads the simulation script and allocates an equal number of nodes for each dvn worker connected to it and forwards the network topology along with the nodes. The master node then tracks the time slots and orchestrates the workers who track their own event lists. During the simulation of a given time slot, if a node on a dvn worker needs to send a message to a node residing on another dvn worker, the communication happens over the physical network connecting both workers.

6.2 Empirical comparison

We evaluate the performance of DVN and WiDS based on the number of nodes, the number of messages, individual message sizes and general memory overhead. For these tests we wanted to mimic the behavior of a simple peer-to-peer network. Both modules would start by sending a message to a single bootstrap node. After receiving a response from the bootstrap, nodes send messages to a randomly chosen node within the simulated network. Nodes send messages at specified intervals using the delayed callback mechanism on both simulators. The contents of the data packets was generated randomly. The size of each packet was configurable within the test module. The tests were ran on a single machine. Our testbed consisted of a 2.0GHz Pentium 4 desktop with 1 GB of ram running Windows XP Professional SP2. All tests were run on the Windows operating system. DVN tests were compiled using cygwin version 2.573.2.2. WiDS was compiled using Visual Studio 2008. In both instances, the compiler optimizations were turned on and debugging and profiling turned off.

Figure 12a shows the result of running experiments with varying message lengths and varying number of messages per simulation. For this experiment we ran 10,000 nodes under a varying message loads. It is apparent that the length of the messages affects the performance of the simulation due to memory allocation, and populating the messages with data. In the plot, we show that DVN can process a system



(a) 10,000 nodes for 2 hours

(b) 25.7 million messages sent by a varying amount of nodes

Figure 12: WiDS and DVN Comparison

sending 500 bytes messages at the same speed WiDS can process a system with 5 bytes messages. Furthermore the DVN system sending 5 byte messages runs about four times faster than WiDS under the same conditions.

The second experiment in Figure 12b compares the impact of the number of nodes on both WiDS and DVN. For this test we fixed the number of messages at 25.7 million. During our test simulations DVN ran in constant time regardless of the amount of nodes simulated. Since the computation time on DVN is dominated by the scheduler, i.e. assuming a relatively small processing time on our simple test module, this is expected behavior. Both tests degrade linearly as the number of messages increase. Although the number of nodes do not affect performance significantly, WiDS has some extra degradation as the number of nodes increases. An interesting note on the plot is the higher simulation time for a small network of 10 nodes. At this scale, the node's processing code dominates over the scheduler code and causes the spike.

6.3 Discussion

While DVN and WiDS achieve somewhat similar goals, programming for and working with them has some pragmatic differences. To send datagram messages within WiDS one allocates a buffer of type **WIDSBuffer**. The actual size of this buffer is set within a configuration file. This is the same configuration file that WiDS reads node information from when starting a simulation. Within DVN a buffer of the standard C type **char** is used to send messages. The effect of this that DVN can choose its buffer size dynamically, during runtime. A WiDS programmer must input the size of the largest buffer that will be needed into the configuration file. This can lead to a large performance hit in certain circumstances. In our original experiments, WiDS was running an order of magnitude slower than DVN until we noticed this configuration option.

WiDS stores configuration files in xml format. While this is it easy to read, it causes a one time performance hit at the start of the simulation. To ensure a fair comparison, all times reported in Figure 12 exclude this load time. Each node and each network link needs to be described individually, making it inconvenient to run large simulations of over 10,000 nodes.

The dsim language created for DVN allows the description of individual nodes, but allows groups of nodes to be created as well. To avoid abnormal spikes due to the instantaneous creation of a large number of nodes, DVN will spread the node creation based on parameters specified in the dsim input. As a result, the configuration times for DVN were very small, even for large simulation of 10^6 nodes.

7. CONCLUSION AND FUTURE WORK

DVN is a very scalable high fidelity network simulator supporting real world implementation of application layer protocols, inluding peer to peer networks. It provides a flexible platform that can be used to quantify network performance and degradation due to attacks, as well as the effectiveness of countermeasures in a reproducible manner. Dsim files and modules can be easily distributed to the community for simulations in a reproducable manner. Distribution of the module code to developers can ensure a cross compilation to their platform of choice for a real world implementation of algorithms and countermeasures.

There are some optimization still possible with DVN. Operating with parallel instances using shared memory currently works well, but our implementation that uses the same algorithms over a network for distributed simulations has shown to be dificult. While distributed simulations are currently possible, we still need to make the scheduling and synchronization system more robust. Simulating churn with nodes entering and leaving the network is an important part of the simulation. Currently, the dsim language does not include nodes leaving the network. While that behavior can be embedded in the module code, it would be cleaner if handled by the master script. Finally, the topology modeling is a simplified version of today's network. Different link characteristics can happen even with a single autonomous system. A finer granularity of the network model would allow researchers an even higher fidelity simulation than is now provided.

8. REFERENCES

[1] aMule network. `http://www.amule.org`.

[2] T. Benzel, R. Braden, D. Kim, C. Neuman, A. Joseph, K. Sklower, R. Ostrenga, and S. Schwab. Experience with DETER: A testbed for security research. In *2nd International Conference on Testbeds and Research Infrastructures for the Development of Networks and*

Communities, 2006. TRIDENTCOM 2006, page 10, 2006.

[3] J. Bolot. End-to-end packet delay and loss behavior in the Internet. In *Conference proceedings on Communications architectures, protocols and applications*, page 298. ACM, 1993.

[4] C. Bovy, H. Mertodimedjo, G. Hooghiemstra, H. Uijterwaal, and P. Van Mieghem. Analysis of end-to-end delay measurements in Internet. In *Proc. of the Passive and Active Measurement Workshop-PAMâĂŹ2002*, 2002.

[5] R. Brown. Calendar queues: a fast 0(1) priority queue implementation for the simulation event set problem. *Commun. ACM*, 31(10):1220–1227, 1988.

[6] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *ACM SIGCOMM Computer Communication Review*, 33(3):00–00, July 2003.

[7] J. Cowie, H. Liu, J. Liu, D. Nicol, and A. Ogielski. Towards realistic million-node internet simulations. In *International Conference on Parallel and Distributed Processing Tech- niques and Applications (PDPTA)*, 1999.

[8] J. Cowie, A. Ogielski, and D. Nicol. The SSFNet network simulator. *Software on-line: http://www. ssfnet. org/homePage. html*, 2002.

[9] DHTSim. `http://www.informatics.sussex.ac.uk/users/ianw/teach/dist-sys`.

[10] Flex. `http://flex.sourceforge.net/`.

[11] R. Fujimoto. Parallel discrete event simulation. In *Proceedings of the 21st conference on Winter simulation*, page 28. ACM, 1989.

[12] R. M. Fujimoto. *Network Simulation (Synthesis Lectures on Communication Networks)*. Morgan and Claypool Publishers, 2006.

[13] R. M. Fujimoto, K. Perumalla, A. Park, H. Wu, M. H. Ammar, and G. F. Riley. Large-scale network simulation: How big? how fast? In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2003.

[14] A. Haeberlen, A. Mislove, A. Post, and P. Druschel. Fallacies in evaluating decentralized systems. In *International Workshop on Peer-to-Peer Computing*, 2006.

[15] D. Kato and T. Kamiya. Evaluating DHT Implementations in Complex Environments. In *Proceedings of the International Workshop on Peer-to-Peer Computing*, 2007.

[16] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. In *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, 2007.

[17] D. Leonard and D. Loguinov. Turbo king: Framework for large-scale internet delay measurements. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pages 31 –35, apr. 2008.

[18] M. Liljenstam, D. M. Nicol, V. H. Berk, and R. S. Gray. Simulating realistic network worm traffic for worm warning system design and testing. In *WORM '03: Proceedings of the 2003 ACM workshop on Rapid malcode*, pages 24–33, New York, NY, USA, 2003. ACM.

[19] S. Lin, A. Pan, R. Guo, and Z. Zhang. Simulating large-scale p2p systems with the wids toolkit. In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2005.

[20] S. Lin, A. Pan, Z. Zhang, R. Guo, and Z. Guo. WiDS: an integrated toolkit for distributed system development. In *Hot Topics in Operating Systems*, 2005.

[21] P. Maymounkov and D. Mazíeres. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *International Workshop on Peer-to-Peer Systems*, 2001.

[22] C. Misner, K. Thorne, and J. Wheeler. *Gravitation*. WH Freeman & co, 1973.

[23] S. Naicken, A. Basu, B. Livingston, S. Rodhetbhai, and I. Wakeman. Towards yet another peer-to-peer simulator. In *International Working Conference on Performance Modelling and Evaluation of Heterogeneous Networks*, 2006.

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

[25] Narses Network Simulator. `http://sourceforge.net/projects/narses`.

[26] NeuroGrid. `http://www.neurogrid.net`.

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

[28] The ns-3 Network Simulator. `http://www.nsnam.org/`.

[29] OverlayWeaver. `http://overlayweaver.sourceforge.net`.

[30] P2PSim. `http://pdos.csail.mit.edu/p2psim`.

[31] V. Paxson. End-to-end Internet packet dynamics. In *Proceedings of the ACM SIGCOMM'97 conference on Applications, technologies, architectures, and protocols for computer communication*, page 152. ACM, 1997.

[32] PeerSim P2P Simulator. `http://peersim.sourceforge.net`.

[33] PlanetSim: An Overlay Network Simulation Framework. `http://planet.urv.es/planetsim`.

[34] G. Riley and M. Ammar. Simulating large networks: How big is big enough? In *Conference on Grand Challenges for Modeling and Simulation (ICGCMS)*, 2002.

[35] G. Riley, R. M. Fujimoto, and M. Ammar. A generic framework for parallelization of network simulations. In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 1999.

[36] G. F. Riley, R. M. Fujimoto, and M. H. Ammar. Network aware time management and event distribution. In *Workshop on Parallel and Distributed Simulation (PADS)*, 2000.

[37] A. Rodriguez, C. Killian, S. Bhat, D. Kostić, and A. Vahdat. Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. In *NSDI'04: Proceedings of the 1st conference on*

Symposium on Networked Systems Design and Implementation, 2004.

[38] M. T. Schlosser and S. D. Kamvar. Simulating a file sharing p2p network. Technical report, Stanford Univ., 2002.

[39] G. Shu, D. Chen, Z. Liu, N. Li, L. Sang, and D. Lee. VCSTC: Virtual Cyber Security Testing Capability–An Application Oriented Paradigm for Network Infrastructure Protection. *Testing of Software and Communicating Systems*, pages 119–134, 2008.

[40] M. Steiner, T. En-Najjary, and E. W. Biersack. A global view of kad. In *ACM SIGCOMM conference on Internet measurement*, pages 117–122, New York, NY, USA, 2007. ACM.

[41] D. Stutzbach and R. Rejaie. Improving lookup performance over a widely-deployed DHT. In *INFOCOM*, 2006.

[42] 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 Oper. Syst. Rev.*, 36(SI), 2002.

[43] W. Yang and N. Abu-Ghazaleh. GPS: A General Peer-to-Peer Simulator and its Use for Modeling BitTorrent. In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2005.