A Virtual Platform for Network Experimentation

Olaf Landsiedel, Georg Kunz, Stefan Götz, Klaus Wehrle
Distributed Systems Group
RWTH Aachen University, Germany
{landsiedel,kunz,goetz,wehrle}@cs.rwth-aachen.de

ABSTRACT

Although the diversity of platforms for network experimentation is a boon to the development of protocols and distributed systems, it is challenging to exploit its benefits. Implementing or adapting the systems under test for such heterogeneous environments as network simulators, network emulators, testbeds, and end systems is immensely time and work intensive.

In this paper, we present VIPE, a unified virtual platform for network experimentation, that slashes the porting effort. It allows to smoothly evolve a single implementation of a distributed system or protocol from its design up into its deployment by leveraging any form of network experimentation tool available.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network Communications

General Terms

Design, Experimentation

Keywords

Resource Virtualization, Network Experimentation, Simulation, Deployment

1. INTRODUCTION

Although the primary interest of researchers and developers of communication protocols lies in the functionality and performance of their protocols, they are faced with a tremendous engineering overhead when it comes to implementing them. This problematic fact is quickly illustrated by a simple example: the life cycle of any new protocol.

The realization of the new protocol typically begins with implementing the design ideas in a network simulator to test and evaluate system behavior and scalability. While working towards an algorithmically complete system, it may be desirable to move to a different simulator platform, e.g. to exploit different models and richer visualization features. However, the engineering effort required for such a move (different APIs, abstraction levels, models, etc.) is likely to be prohibitively expensive for such a goal despite the fundamental similarities among network simulators.

With a promising protocol at hand, scientific publication demands to evaluate it in more detail and more realistic settings than a simulator. In the simplest case, a user-space implementation is required to determine real-world client performance. This transition results in an even larger development effort since the existing algorithms now need to be adapted to a completely different programming model (synchronous instead of asynchronous event handling) and yet again a different programming API. Even so, such an effort results in support for a single platform because the overhead of covering multiple operating systems is significant. Consequently, pushing the new protocol into testbeds or deployments of a different architecture, building a user or developer community etc. often remains elusive. All this applies even more so to low-level protocols typically implemented in OS kernels where development is inherently more complex and difficult.

Although this observation may seem trite, it reflects in fact a counter-intuitive problem: On the one hand, protocols cover a narrow problem domain and their implementations depend only on a small and clearly identifiable set of functionality. On the other hand, a massive engineering effort is necessary to run the same protocol algorithms and functions on different platforms and environments. Thus, today's protocol development remains restricted to a small set of evaluation tools in practice.

This paper introduces VIPE, a VIrtual Platform for network Experimentation, which bridges heterogeneous evaluation tools and platforms for communication protocols. In its "write once, run almost everywhere" environment, protocols can seamlessly move back and forth between the stages of simulation, emulation, evaluation on testbeds, and deployment. In VIPE, this is a matter of installing the platform-specific development tools and typing *make windows-kernel* instead of *make omnet* so the development cycle is governed by refinement instead of re-implementation. Overall, VIPE aims to bring protocol development closer to its ideal of a stage-wise refinement instead of re-implementation and it allows for a tight feedback loop between the different development stages.

The contribution of this paper is three-fold: 1) we identify the minimal core functionality required by communication protocols, 2) we show that this core functionality can

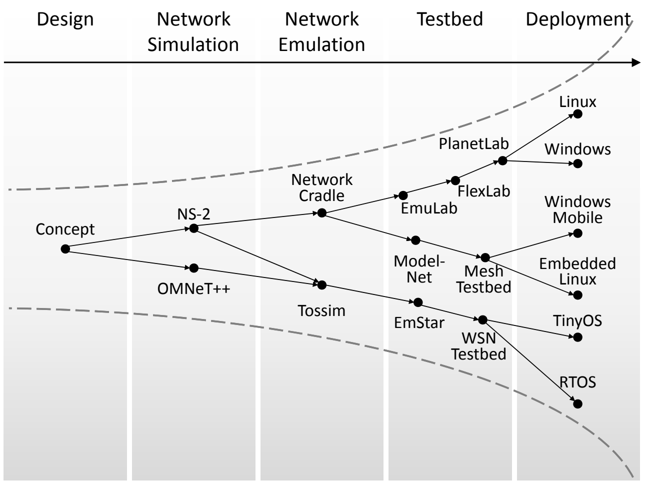

Figure 1: During their evolution, protocols pass through a heterogeneous set of evaluation tools, requiring multiple re-implementations.

be provided uniformly on a wide range of platforms at a negligible performance impact, and 3) we demonstrate that this approach effectively eliminates cross-platform porting efforts which benefits network experimentation in particular. We have implemented widespread communication protocols such as TCP, IP, and Chord in VIPE showing that protocols require only a small, well-defined platform API. This leads to very light-weight abstraction layers with a non-intrusive execution model contained in about 1000 lines of code per platform. With VIPE, network researchers can readily test and deploy protocols in the kernel and user space of Linux, Windows XP, and Windows CE / Mobile, on network simulators (ns-3, OMNeT++), testbeds (EmuLab, PlanetLab), and sensor nodes (TinyOS). Since VIPE-protocols integrate with the protocol collection of a target platform, VIPE leverages the protocol stacks of operating systems and the extensive model collections of today's network simulators.

Section 2 generalizes and analyses the problems outlined above with regard to network experimentation. Section 3 presents the design of VIPE. Its practicability, effectiveness, and performance impact on many platforms is evaluated in Section 4. Section 5 discusses related work and Section 6 presents future goals. We conclude in Section 7.

2. ANALYSIS

Ever since network simulation was established, the community has been holding a never ending discussion on its credibility and degree of realism [1,13–15,24,33]. Hence, network researchers and developers feel an increasing pressure to deliver experimentation results for simulation, testbed, and real-world settings for a reliable and realistic evaluation of protocols and distributed systems (see Figure 1).

Tool Chain Explosion

To increase the credibility of protocol and system evaluation, the community proposed new tools to bridge between simulation and the real world. For example, network emulation [7,12,21,41], virtualization [3,4,18,26], and testbeds [2,37,44] received new attention.

Large-scale research projects such as ns-3 [19] and GENI [34] as well as the rapid growth of PlanetLab further underline the importance of new substrates for network experimentation and evaluation.

The result is a large number of tools, each focusing on a distinct aspect of the problem space. Each of them provides its own benefit for an individual point in the protocol development process.

Platform Diversity

The variety of evaluation tools is further aggravated by the increasing diversity and number of platforms that communication protocols and distributed systems aim to support. Communication systems and particularly the Internet reaches into new domains such as mobile, ad-hoc, mesh, and wireless sensor networks. Each domain requires distinct tool chains for evaluation and deployment.

Isolation of Evaluation Tools

The heterogeneity in the individual steps of the evaluation process requires implementations to be duplicated for nearly all platforms and tools in the evaluation process, resulting in a painstaking and time-consuming process. For example, moving a protocol from a network simulator to PlanetLab or from OS user to kernel space without re-implementation can be considered impractical, if not utopian. Consequently, it is prohibitively complex to employ more than a small number of the evaluation tools available and to achieve a tight feedback loop with the design and evaluation phases. This feedback is even more severely limited between different networking domains such as infrastructure-based, sensor, or mesh networks.

3. VIRTUAL PLATFORM ARCHITECTURE

The virtual platform aims to ease the development and evaluation of new distributed systems and protocols. Typically, these are implemented from scratch and can thus be easily tailored to the virtual platform. Platform abstraction in VIPE is not bound to any specific protocol layer. However, protocols that undergo the final transition into the kernel-domain benefit the most from VIPE, as kernel and user space run-time environments differ the most.

It is not our intent to make VIPE a standard run-time for protocols deployed in operating systems. Operating system and application vendors typically rely on their own implementations of protocols, optimized for their target platforms and use cases. Instead, VIPE aims to ease the work of researchers to reach a version deployable in the wild for field tests, to enable early adoption and to reduce dependencies from operating system vendors. We believe that clean-slate approaches to the Internet architecture can benefit from a virtual platform in particular, as they are expected to pass the development cycle a number of times before reaching the required maturity. Additionally, for protocols that have left or are leaving the research domains and are awaiting roll-out in major operation systems such as SCTP [39], DCCP [22] or HIP [32] an environment such as VIPE can speed up an initial deployment on a wide range of platforms and reduce dependencies from major operating system providers.

3.1 Architecture

To achieve cross-platform portability, one fundamentally needs to find a stable common ground on which software can be built or run on independently of the underlying architecture. Today, this problem may appear to be solved, since we can rely on such techniques as interpreted languages or hardware and system virtualization. However, their genericness comes at the expense of a significant engineering effort per platform for porting and maintenance. More importantly, these techniques cannot be practically applied in such heterogeneous platforms relevant to network experimentation as OS kernels or simulators.

In contrast, VIPE builds on two simple observations: a) across platforms, protocols form a bounded application domain that depends on only a small number of core run-time primitives and b) due to best practices in system design, the APIs to these run-time primitives closely resemble each other. It should thus be possible to create a uniform development and runtime environment for cross-platform protocols. On the one hand, the development environment comprises a programming paradigm, a standard library of core functionality, and a run-time. On the other hand, the runtime environment maps a virtual platform (i.e., its standard library API) to the native platforms that protocols execute on. Although this may appear as a seemingly trivial task, the challenge and the basic premise of our work is to devise a) a development environment that is complete but lightweight for a low engineering effort and b) a runtime environment that is complete but lightweight for a low execution overhead.

The remainder of this section discusses how the virtual platform addresses these challenges, detailing on language aspects, the standard library, the packet and protocol models, and the runtime environment.

3.2 Programming Language and Paradigms

The virtual platform relies on event-based activation, i.e. asynchronous event handling, as the unifying programming paradigm because it maps naturally to the protocol domain and integrates well with all target platforms. It is the native execution model in most simulators and OS protocol code in the kernel domain typically centers around event handler functions. This asynchronous programming paradigm also allows transparently exploiting multiple processor cores for improved performance by parallelizing event handling.

VIPE intentionally does neither allow synchronous (i.e. blocking) event handling nor threads. Although this deviates from typical user-space paradigms such as socket programming, we believe that developers experienced either in network simulation, OS kernel development or asynchronous socket programming adopt naturally to this model. Moreover, a narrow API and a fixed execution paradigm are fundamental requirements for achieving VIPE's lightweight architecture.

In terms of programming languages, VIPE does not bind itself to a single programming language. The API of its standard library and run-time is accessible by any modern language. Hence, the choice of language to implement a distributed systems or communication protocol on top of VIPE's API merely depends on the targeted evaluation platforms. For example, Python or Java are acceptable only if the implemented protocol is not targeted for OS kernels or sensor nodes where these languages are not supported.

3.3 The Standard Library

With our protocol standard library, the virtual platform provides the core functionality necessary to implement protocols. It provides unified interfaces to resources in five areas: network packets, memory management, timing, synchronization, and network devices (see Figure 2). For programming convenience, it additionally contains a small number of frequently used data structures. Section 4.1 shows how this narrow scope proves sufficient even for complex protocols.

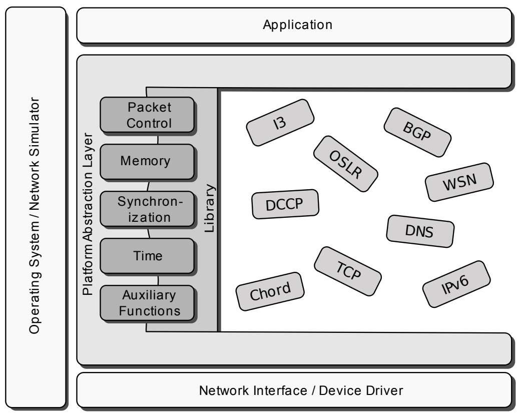

Figure 2: The virtual platform provides unified interfaces to the resources of target platforms.

Our design exploits the observation that on all target platforms the native APIs for these areas are very similar as they stem from or resemble the C standard library. We interpret this effect as a stable trend across platforms towards best practices in API design, which also fosters VIPE's adoption of new platforms. Consequently, it provides a well-established and familiar API to a wide range of users from OS kernel to network simulation developers. Furthermore, these similarities enable a lightweight implementation, mainly consisting of pre-processor aliases or slim wrappers, resulting in an efficient code base and low implementation effort.

3.3.1 Generic Network Data Handling

The virtual platform facilitates a generic, layer independent representation of network packets and a corresponding API for packet manipulation. This API provides unified primitives for tasks like de-/allocating packets, adding and removing headers, etc.

The generic packet representation resembles a simplified version of the *socket_buffer* data structure of the Linux kernel. It easily maps to the native data structures of Linux and Windows and reduces the implementation complexity. We explicitly trade advanced features such as packet chaining, i.e. the representation of packet payload through a list of disjoint memory blocks, for small complexity. Based on this generic packet representation, we furthermore integrated access to network devices in the kernel domain, network sockets in the user-land and for both corresponding counterparts in network simulation.

Targeting a lightweight abstraction layer, VIPE aims to leverage the native functionality of the underlying platforms by mapping its generic primitives to the native counterparts. Kernels and network simulators proof particularly valuable in this regard since VIPE can utilize a rich infrastructure for handling network I/O and packets. In contrast, the user space typically does not provide such native functionality, thus requiring its own yet simple implementations in VIPE.

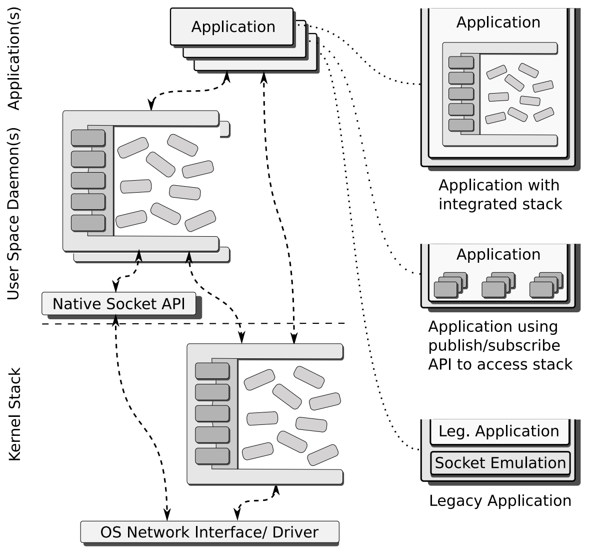

Figure 3: The virtual platform enables the execution of protocols and distributed systems in different OS protection domains.

3.3.2 Memory, Time, and Synchronization

VIPE's standard library provides primitives for memory allocation and manipulation such as *malloc*, *free*, and *memcpy*. Additionally, it offers synchronization primitives in the form of locks to protect shared data from concurrent access by event handlers. All those primitives map closely to functionality available on the target platforms.

The virtual platform integrates one-shot and recurring timers with the event system, for example to let protocols trigger packet retransmissions. System time (e.g., for time stamps) and timers in VIPE are at the granularity of milliseconds, which is natively available on all supported platforms. While this resolution satisfies most protocols, VIPE additionally provides high resolution timers in the order of nano seconds on platforms where supported as its only feature which is not fully portable.

As a convenience feature, the standard library defines an API for commonly used data structures, such as lists, queues, and hash tables. Depending on the availability of native counterparts, the protocol standard library falls back on those or provides a default implementation itself.

Overall, we observed that best practices in system design resulted in high similarities between the APIs of typical network simulators and operating systems in kernel and user space, limiting run-time overhead and reducing implementation complexity.

3.4 The Runtime

The virtual platform does not aim at replacing existing protocol stacks. Instead it provides a container for new protocols to be tested in different environments.

VIPE encapsulates single protocols as modular building blocks. These protocols constitute instantiable entities from which virtualized protocol stacks can be composed. Furthermore, the protocols may be placed in separate protection domains to achieve fault isolation which is desirable for evaluating unstable or untrusted protocols.

In an operating system, the virtual platform facilitates the execution of protocols in three different environments: (1) kernel space, (2) user space, and (3) inside applications, depending on the developer's demands, such as low latency, isolated execution, or application integration (see Figure 3). Generic interfaces for packet transition to and from the virtual platform allow VIPE to integrate deeply with target systems and connect its stack to existing protocols, network devices, and sockets. This ensures that protocol development can focus on the protocol or distributed system of interest. At the same time, it can rely on existing protocols on any network layers below, on, and above the new protocol.

Network simulators and the user space in general are much more amendable to debugging and testing than OS kernels, embedded systems, or the distributed nature of PlanetLab. Hence, with debuggers, memory-leak analysis, or unit testing these provide the required tool chain for a deep analysis and testing of protocols. Additionally, VIPE naturally taps into the deployment infrastructure available for PlanetLab, Emulab and network simulators.

Platform	Lines of Code
Linux Userspace	778
Linux Kernel	815
Windows XP Userspace	739
Windows XP Kernel	1546
Windows CE Userspace	1189
Windows CE Kernel	(5864) 1161[1]
TinyOS (1.x)	423
ns-2	474
OMNeT++	818

Table 1: Lines of code per virtual platform implementation

4. EVALUATION AND PRACTICAL EXPERIENCE

In this section, we evaluate VIPE's implementation complexity and its performance. Furthermore, we discuss our practical experiences and outline the limitations of the chosen architecture.

4.1 Implementation Complexity

We implemented VIPE on a broad selection of target platforms, ranging from network simulators (ns-2, OMNeT++), user and kernel space of common operating systems (Linux, Windows XP) to embedded systems (TinyOS, Windows CE). Aiming for both a seamless integration into OS kernels and a reuse of code across platforms, our implementations of the virtual platform base on C. Additionally, VIPE incorporates higher level languages such as Java and Python by providing access to its API and callback hooks via language bindings where applicable. In this regard, VIPE benefits from native language interfaces such as Java's JNI or SWIG for Python [5].

VIPE implements the virtual platform with about 1000 lines of code for each target platform (see Table 1). The Windows kernel-based implementation is more complex because it needs to cover the broad NDIS API and maintains an additional management data structure per packet.

[1]Modified lines of code (total in parentheses) of the NDIS base driver.

	Linux 2.6.22		Windows XP	
	Kernel	User	Kernel	User
Memory Management	0.0%	0.0%	0.0%	0.0%
Synchronization	0.0%	0.0%	0.0%	0.0%
Timer Management	0.0%	0.0%	0.0%	0.0%
Device `send()`	0.1%	0.1%	14.5%	0.2%
Device `receive()`	0.1%	0.2%	11.5%	0.1%
Packet `create()`	0.1%	0.0%	34.2%	0.0%
Packet `delete()`	0.1%	0.0%	9.7%	0.0%
IPv4 Router	0.0%	-	3.0%	-

Table 2: CPU performance overhead of the virtual platform.

TinyOS and ns-2 on the contrary have a very narrow native API and thus their VIPE implementation is very lightweight. The implementation for OMNeT++ in turn also utilizes OMNeT++'s visualization capabilities, thus increasing the code base slightly. On average, the virtual platform requires about one order of magnitude less code than related approaches such as the Network Cradle [21] or OppBSD [7].

Hence, VIPE already covers a wide range of target platforms with a small code base and low engineering effort. Nevertheless, it is sufficient to implement an Internet protocol stack including Ethernet, ARP, IP{v4,v6,X}, TCP, and Chord (see Section 4.3). This underlines that its narrow waist design is sound and can be expected to apply to further platforms and protocols.

Concluding, our evaluation shows that the concept of a lightweight virtual platform can successfully be applied in practice. However, we do not necessarily consider the proposed interfaces final or complete for next generation protocols. Instead, we expect them to evolve over time and hope to spark a discussion on a possible design of a narrow waist for protocol evaluation and the interfaces it provides.

4.2 Performance Evaluation

Our test setup consists of three off-the-shelf hyperthreaded Pentium IVs at 3.0 GHz with 1 GB RAM running Linux Debian Etch and a 2.6.22 kernel. Performance measurements are conducted using the time stamp counter of the CPU that provides near cycle accuracy.

Due to the design decisions in favor of a narrow API, VIPE imposes a remarkably small overhead on system performance. For selected resources, Table 2 illustrates a summery of the overhead of the virtual platform compared to the corresponding native platforms. The functions for memory management, timing, and synchronization exhibit a negligible performance overhead on both Linux and Windows. VIPE typically implements them as very thin wrappers of the native API and optimizing compilers can hence eliminate the overhead of function calls via code inlining.

For packet oriented operations, the results are similar except for the Windows kernel space. It suffers from an execution overhead of about 11%–34% due to VIPE's generic packet representation. For each packet, the virtual platform allocates, maintains, and deallocates this managemanet data structure. On Linux in contrast, all management functions operate on the native packet representation via wrappers.

We determine the overall system performance by comparing an IPv4 router implemented on top of VIPE with its native counterparts in the Linux and Windows kernels.

Resource	Design Trade-off
Execution model	no threads
Packet representation	no packet chaining
Network devices	no hw specific interfaces
Timer	milli-second granularity (default)
Memory	non-paged memory in kernel
Synchronization	single lock type (no r/w)

Table 3: VIPE's narrow API trade-offs platform specific functionality for lightweight abstraction.

Our benchmark saturates all routers with packets of varying size and measures the number of dropped packets. Overall, the Linux based implementation performs equally well as its native counterpart whereas VIPE in the Windows kernel is about 3% slower than its native counterpart.

Since VIPE is still work in progress, our implementations of more complex protocols such as TCP and Chord are not yet mature enough for conducting a sound performance analysis.

4.3 Practical Experiences

We argue that VIPE's primary benefit is to significantly reduce the development effort of protocol and network experimentation. However, this claim is not as easily quantifiable in numbers as performance results or lines of code. In order to corroborate our claim, we implemented a complete TCP/IP stack on top of VIPE. Besides relatively simple protocols such as Ethernet, ARP, and IPv{4,6,X}, our stack furthermore includes TCP and the DHTs Chord and Pastry. The following summarizes our experiences.

Debugging is a tedious process in operating system kernels, embedded devices, and distributed systems in general due to a lack of insight and controllability. In contrast, user space environments provide the required debugging tools and network simulators enable a global view on a distributed communication system, supporting the functional evaluation of protocols. VIPE's seamless transition between different platforms enables an evaluation of communication protocols not only in terms of performance, but also in terms of functionality before being transferred to the operating system kernel or an embedded system. Thus, the virtual platform ensures that the code is already extensively tested before being used in an OS kernel. It proved particularly helpful for dynamic memory issues such as memory leaks which tend to be challenging to track down in operating system kernels.

Overall, after debugging protocols in a network simulator, no functionality-related bugs were discovered on other platforms later in the development process.

4.4 Limitations

Our design decisions in favor of a lightweight abstraction and narrow interfaces result in a set of limitations in VIPE's architecture. Table 3 shows an overview of the most important design trade-offs in this regard. While some of these features may be desirable, VIPE's architectural premise is to design its API as narrow as possible to increase maintainability and to simplify the integration of new platforms. Thus, we believe that it is essential for VIPE's architecture to rely on a lightweight packet representation and asynchronous event handling.

5. RELATED WORK

Although platform abstraction has been widely used to ensure portability, we argue that previous work focuses only on a fraction of the protocol development process. This section discusses related work beyond the research addressed in our problem analysis (see Section 2).

For performance reasons, network and transport layer protocols are typically implemented in the operating system kernel. This limits experimentation because kernels are not very amenable to debugging and testing. This is one of the motivating factors for research projects to move kernel network stacks to user space. For example, Daytona [35] and Arsenic [36] port Linux stacks while Alpine [11] is based on the FreeBSD network stack. The Network Simulation Cradle [21] and OppBSD [7] port OS network stacks into network simulators such as ns-2 [31] and OMNeT++ [42]. Due to the tight integration of network stacks into the operating system kernel, such ports are challenging and maintenance is non-trivial. Additionally, their monolithic design stifles extensions and modifications. However, the fact that these demanding ports are actively maintained nevertheless shows the demand for fully featured protocol stacks for experimentation and evaluation. Furthermore, these stacks merely bridge two domains, such as network simulator and operating system kernel.

Similarly, Click [23] allows to compose and execute modular software routers in the Linux kernel and user space, the FreeBSD kernel, and the ns-2 network simulator. While providing resource abstraction for these specific systems, Click does not aim to achieve a lightweight abstraction layer across all stages of development and evaluation. In contrast, basing on the identification of similarities between the targeted systems, VIPE provides a shim layer of abstraction of the resources required by communication systems that has on average 1000 lines of code per platform – about one order of magnitude less than the Network Simulation Cradle or OppBSD.

The need for protocol experimentation sparked research on virtualization in network experimentation, allowing the execution of multiple network stacks on a single machine [6, 16,20,43,45]. To provide further insight others provide user-space implementations of wide-spread protocols [8–10,17,29, 40]. Our virtual platform architecture enables the seamless integration of these approaches into the development cycle.

Restricted to their own and very specific environment, some domains provide simulation and emulation tools which facilitate a limited transition of network protocols. For example, in the domain of wireless sensor networks TOSSIM [27] allows the seamless transition of TinyOS [28] protocols between network emulation and deployment.

6. FUTURE WORK

Based on the fact that the network stacks for simulation and deployment facilitate the same implementation, VIPE naturally lays the ground for simulation calibration. As future work, we envision the automatic collection of fine-grained timing traces from any physical platform or a corresponding system model and feeding this data back into network simulation models to increase their accuracy in terms of timing and energy usage [25,30,38].

For the same reason, VIPE's architecture inherently supports co-simulation: the integration of network simulators into a live network. The shared implementation guarantees seamless interoperability among simulators and operating systems and avoids artefacts due to abstraction in simulation models. Hence, VIPE enables large scale evaluation in heterogeneous environments composed of network simulators and different types of operating systems.

7. CONCLUSION

Observing the isolation in the individual approaches for network experimentation and evaluation, we present VIPE as a virtual platform to cross the barriers between network simulators, testbeds, and deployment platforms.

Based on the virtual platform, protocols can be readily deployed and evaluated on a large set of platforms: by integrating network simulators, testbeds, and production systems, the virtual platform enables a tight feedback loop in the protocol development cycle and eases protocol experimentation. Overall, the transition between the individual stages in the development and evaluation process of protocols in VIPE is a stepwise refinement of a implementation instead of a repeated re-implementation. Furthermore, it makes a protocol implementation available for deployment on a large number of platforms allowing to catalyze a large user community without going through the lengthy standardization process and waiting for its integration into major operating systems.

VIPE forms an integral part of our daily protocol experimentation process. The resulting large base of practical experiences greatly substantiates the viability of our approach. Overall, we believe that the virtual platform improves the quality of protocol evaluation and experimentation at a significantly lower engineering effort than achievable so far.

8. REFERENCES

[1] M. Ammar. Why we still don‘t know how to simulate networks. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2005.

[2] R. Bajcsy, T. Benzel, M. Bishop, et al. Cyber defense technology networking and evaluation. *Commun. ACM*, 47(3), 2004.

[3] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating System Support for Planetary-Scale Network Services. In *Proc. of the 1st Symposium on Network System Design and Implementation (NSDI)*, March 2004.

[4] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In vini veritas: realistic and controlled network experimentation. In *Proc. of the conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM)*, 2006.

[5] D. M. Beazley. SWIG: an easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk Workshop*. USENIX Association, 1996.

[6] C. Bergstrom, S. Varadarajan, and G. Back. The distributed open network emulator: Using relativistic time for distributed scalable simulation. In *PADS '06: Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*, 2006.

[7] R. Bless and M. Doll. Integration of the FreeBSD TCP/IP-stack into the discrete event simulator OMNet++. In *Proc. of the 36th conference on Winter simulation (WSC)*, 2004.

[8] T. Braun, C. Diot, A. Hoglander, and V. Roca. An experimental user level implementation of TCP. Technical report, INRIA Sophia Antipolis, France, 1995.

[9] P. A. Dinda. The minet tcp/ip stack. Technical report, Northwestern University, 2002.

[10] A. Edwards and S. Muir. Experiences implementing a high performance tcp in user-space. In *Proc. of the conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM)*, 1995.

[11] D. Ely, S. Savage, and D. Wetherall. Alpine: a user-level infrastructure for network protocol development. In *Proc. of the 3rd conference on USENIX Symposium on Internet Technologies and Systems (USITS)*, 2001.

[12] K. Fall. Network emulation in the vint/ns simulator. In *ISCC '99: Proceedings of the The Fourth IEEE Symposium on Computers and Communications*, 1999.

[13] S. Floyd. Maintaining a critical attitude towards simulation results (invited talk). In *WNS2 '06: Proceeding from the 2006 workshop on ns-2: the IP network simulator*, Oct. 2006.

[14] S. Floyd and E. Kohler. Internet research needs better models. *Computer Communication Review*, 33(1):29–34, 2003.

[15] S. Floyd and V. Paxson. Difficulties in simulating the internet. *IEEE/ACM Trans. Netw.*, 9(4):392–403, 2001.

[16] A. Grau, S. Maier, K. Herrmann, and K. Rothermel. Time jails: A hybrid approach to scalable network emulation. In *PADS '08: Proceedings of the 2008 22nd Workshop on Principles of Advanced and Distributed Simulation*, 2008.

[17] H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Deploying safe user-level network services with ictcp. In *Proc. of the 6th conference on Symposium on Opearting Systems Design & Implementation (OSDI)*, 2004.

[18] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, and G. M. Voelker. To infinity and beyond: time-warped network emulation. In *NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation*, 2006.

[19] T. R. Henderson, S. Roy, S. Floyd, and G. F. Riley. ns-3 project goals. In *WNS2 '06: Proceeding from the 2006 workshop on ns-2: the IP network simulator*, page 13, 2006.

[20] X. W. Huang, R. Sharma, and S. Keshav. The ENTRAPID protocol development environment. In *IEEE INFOCOM (3)*, 1999.

[21] S. Jansen and A. McGregor. Simulation with real world network stacks. In *Proc. of the 37th conference on Winter simulation (WSC)*, 2005.

[22] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), mar 2006.

[23] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[24] S. Kurkowski, T. Camp, and M. Colagrosso. Manet simulation studies: the incredibles. *Mobile Computing and Communications Review*, 9(4):50–61, 2005.

[25] O. Landsiedel, H. Alizai, and K. Wehrle. When timing matters: Enabling time accurate and scalable simulation of sensor network applications. In *Proc. of the 2008 International Conference on Information Processing in Sensor Networks (IPSN 2008)*, pages 344–355, Washington, DC, USA, 2008. IEEE Computer Society.

[26] J. Lepreau et al. Protogeni. In *1st GENI Engineering Conference*, Oct. 2007.

[27] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, 2003.

[28] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The emergence of networking abstractions and techniques in tinyos. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, 2004.

[29] C. Maeda and B. N. Bershad. Protocol service decomposition for high-performance networking. In *Proc. of the fourteenth ACM symposium on Operating systems principles (SOSP)*, 1993.

[30] C. J. Mauer, M. D. Hill, and D. A. Wood. Full-system timing-first simulation. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 108–116, New York, NY, USA, 2002. ACM.

[31] S. McCanne and S. Floyd. *UCB/LBNL/VINT Network Simulator - ns (version 2)*, April 1999.

[32] R. Moskowitz and P. Nikander. Host Identity Protocol (HIP) Architecture. RFC 4423 (Informational), May 2006.

[33] V. Paxson and S. Floyd. Why we don't know how to simulate the internet. In *WSC '97: Proceedings of the 29th conference on Winter simulation*, 1997.

[34] L. Peterson et al. Geni design principles. *Computer*, 39(9):102–105, 2006.

[35] P. Pradhan, S. Kandula, W. Xu, A. Shaikh, and E. Nahum. Daytona : A user-level tcp stack. Technical report, MIT, 2002.

[36] I. Pratt and K. Fraser. Arsenic: a user-accessible gigabit ethernet interface. In *Proc. of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2001.

[37] R. Ricci, J. Duerig, P. Sanaga, D. Gebhardt, M. Hibler, K. Atkinson, J. Zhang, S. Kasera, and J. Lepreau. The Flexlab approach to realistic evaluation of networked systems. In *Proc. of the Fourth Symposium on Networked Systems Design and Implementation (NSDI 2007)*, Cambridge, MA, Apr. 2007.

[38] V. Shnayder, M. Hempstead, B. rong Chen, G. W.

Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proc. of the 2nd international conference on Embedded networked sensor systems (SenSys)*, 2004.

[39] R. Stewart, M. Ramalho, Q. Xie, M. Tuexen, and P. Conrad. Stream Control Transmission Protocol (SCTP) Partial Reliability Extension. RFC 3758 (Proposed Standard), may 2004.

[40] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing network protocols at user level. In *Proc. of the conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM)*, 1993.

[41] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. S. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. *SIGOPS Oper. Syst. Rev.*, 36(SI):271–284, 2002.

[42] A. Varga. The OMNeT++ Discrete Event Simulation System. In *Proc. of the European Simulation Multiconference (ESM)*, June 2001.

[43] E. Weingärtner, F. Schmidt, T. Heer, and K. Wehrle. Synchronized network emulation: Matching prototypes with complex simulations. In *Proceedings of the First Workshop on Hot Topics in Measurement & Modeling of Computer Systems (HotMetrics'08)*, 2008.

[44] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[45] M. Zec. Implementing a clonable network stack in the freebsd kernel. In *USENIX Annual Technical Conference, FREENIX Track*, June 2003.