Experiences with NIMI

Vern Paxson, Andrew Adams and Matt Mathis

Abstract—
NIMI (National Internet Measurement Infrastructure) is a software system for building network measurement infrastructures. Its design emphasizes *(i)* large-scale infrastructures composed from diversely-administered hosts, rather than infrastructures controlled by a single entity, and *(ii)* facilitating diverse types of measurements by diverse parties, some of whom are allowed richer access to certain portions of the infrastructure than others. We discuss our experiences with developing and operating the infrastructure to date: problems we have encountered, both foreseen and unanticipated, mistakes we made, and how we have adapted the design to address these. We also explore two key issues for developing a large-scale, extensible infrastructure: the problem of securely updating software on the measurement platforms, and the problem of constraining the resources consumed by different measurements. We argue that both of these can be unified in terms of controlling the behavior of the measurement software, and that the most promising approach for doing so appears to require writing measurement software in a "safe" language such as Java or Python.

I. Introduction

NIMI (National Internet Measurement Infrastructure) is a software system for building network measurement infrastructures. A NIMI infrastructure consists of a set of measurement servers (termed NIMI "probes" or "platforms") running on a number of hosts in a network, and measurement configuration and control software, which runs on separate hosts. A key NIMI design goal is scalability to potentially thousands of NIMI probes within a single infrastructure. Such scaling has utility both for providing pervasive coverage for fault-diagnosis, and for facilitating research on large-scale cross-sections of the Internet, which, given the network's great diversity, is vital for sound science [PF97].

A number of other measurement infrastructures have been developed, such as Surveyor [Al97], Felix [HGDL97], IPMA [La97], and AMP [WB98]. The principle differences between these and NIMI is that the design of NIMI emphasizes *(i)* infrastructures composed from diversely-administered hosts, rather than an infrastructure controlled by a single entity, and *(ii)* facilitating diverse types of measurements by diverse parties, some of whom are allowed richer access to certain portions of the infrastructure than others.

Regarding the first point above, a fundamental aspect of the NIMI architecture is that each NIMI probe reports to a configuration point of contact (CPOC) designated by the owner of the probe system. There is no requirement that different probes report to the same CPOC, and, indeed, there will generally be one (or more) CPOC per administrative domain participating in the infrastructure. But the NIMI architecture also allows for easy *delegation* of *part* of a probe's measurement services, offering, when necessary, tight control over exactly what services are delegated.

V. Paxson is with the AT&T Center for Internet Research at ICSI, at the International Computer Science Institute in Berkeley, CA; and the Lawrence Berkeley National Laboratory in Berkeley, CA. Email: vern@aciri.org. A. Adams and M. Mathis are with the National Laboratory for Applied Network Research, Pittsburgh Supercomputing Center, Pittsburgh, PA. Email: akadams@wraith.psc.edu, mathis@psc.edu.

Regarding the second point, the NIMI design has been decoupled from any particular types of measurements. The measurement tools available to a platform are whatever binaries and scripts that the administrator of the platform has deemed (via the platform's CPOC) appropriate. In addition, access to particular tools is controlled via cryptographically secure credentials: an administrator's decision regarding to whom they will allow what particular type of access is pinpointed to which credentials they decide to give to which parties, and then which per-credential access capabilities they choose to download into each platform via the CPOC. In summary, NIMI is *not* a measurement tool, but a command and control system for managing measurement tools.

The NIMI architecture and structure of NIMI internals are discussed in [PMAM98], [AMMP98]; we give a brief overview in § II before detailing in § III the sorts of difficulties we have encountered in developing and operating NIMI. Our goal is to scale NIMI by an order of magnitude in the coming 1–2 years, and to achieve this requires surmounting two significant hurdles in terms of securely uploading software to NIMI probes in a trustworthy fashion, and resolving resource conflicts between concurrent measurements. We discuss these challenges in § IV.

II. NIMI overview

We can conceptually divide the NIMI architecture into two components, the structure of the individual NIMI platforms, and the different external components that control the platforms. Each platform is viewed as having a narrowly scoped task: to perform measurements and record the results. It is not a platform's role to then analyze the measurements, much less display them in any fashion. Doing so is instead in the realm of external hosts.

NIMI measurement is built around the notion of *scheduling* a measurement for *some future time*. Measurements are not simply made immediately on a demand basis, because doing so introduces biases into large scale measurement studies, in terms of failures to measure serious connectivity problems due to the connectivity problem preventing access to the measurement device in the first place [Pa99]. Instead, each measurement request includes a time at which the measurement should begin. The time can be "immediately," of course, but by building into NIMI a notion of scheduling, we ensure that we have the necessary mechanisms to orchestrate large-scale measurements.

Accordingly, each NIMI platform runs a measurement server whose job is to: authenticate measurement requests as they arrive; check requests against the platform's policy table; queue them for future execution; execute them at the appropriate time; bundle and ship the results of the requests to whatever destination the request specified (a "DAC"–see below); and delete the results when instructed to. Internally, the NIMI probe is divided into two distinct daemons, *nimid*, which is responsible for communication with the outside world and performing access control checks, and *scheduled*, which does the actual measurement scheduling, execution, and result packaging.

As indicated above, security is a basic part of the NIMI architecture. Authentication and encryption of all communication between NIMI components is done using public key credentials. Furthermore, each NIMI probe is configured by its CPOC (or a delegatee of the CPOC) to authorize particular sets of operations per credential. This allows the owner of the NIMI probe complete control over what actions the possessor of a credential can request.

Moving now from the measurement probes themselves to the external elements of the NIMI architecture, the next major component is the CPOC, which serves to configure and administer a set of NIMI probes within the CPOC's sphere. The CPOC provides the initial policies for each distinct NIMI probe, and, over time, provides updates to these policies. (At some point in the future, the CPOC will also act as a repository for NIMI public keys and measurement modules.)

End users who wish to use the infrastructure do so via the Measurement Client (MC) (this is the only NIMI component that an end user actually operates). The MC is a Unix utility that can run on whatever workstation is convenient for the end user, providing that the workstation has access to the user's NIMI credentials. It communicates directly with the NIMI probe(s) involved in the measurement; the CPOC is not involved in the processing of individual measurement requests.

The final component, the Data Analysis Client (DAC), acts as a repository and post-processor of the data returned by the NIMI probe(s) upon completion of a measurement. When an MC sends a measurement request to a NIMI probe, it includes in the request an URL designating the DAC to which the probe should send the measurement results. The DAC can be run as part of the MC, in order to collect immediate results, or as a daemon, to collect on-going measurement results.

A. Measurement modules

As mentioned earlier, an important facet of the NIMI architecture is that NIMI itself has *no* knowledge of particular measurement tools. The view instead is that the tools used to perform the measurements are effectively stand-alone, third party software modules that "plug in" to NIMI.

The strength of modular design lies in correctly designing the interfaces between the different components. While we do not claim perfection in this regard, we believe NIMI is off to the right start in that the expectation is all measurement tools are "wrapped" with a script designed to fit with a uniform API, and it is this script that NIMI invokes when executing a measurement request, rather than the tool directly.

The wrapper serves to interpret certain standardized arguments in a uniform way (although we have not yet developed this feature in earnest); to bundle up measurement results into a standardized form for easier processing by the DAC; and to provide a non-privileged starting point for measurement tools that might themselves require privileges. We return to this latter point in § IV.

Currently, we have deployed the following measurement modules: `traceroute` (end-to-end Internet route measurement [Ja89]); `mtrace` (end-to-end multicast route measurement); `treno` (bulk transfer capacity measurement [MM96]); `cap/capd` (bulk transfer capacity measurement [Al99]); `zing` (generic packet source/sink for one-way and round-trip loss and delay measurement); `mflect` (multicast inference of path properties [To98], [Fr99]); `traffic/discardd` (TCP throughput measurement [Al99]); and `ftp` (a wrapper around the File Transfer Protocol).

Adding a new measurement tool as a module simply requires generating a wrapper for the tool and propagating both the tool and the wrapper to all NIMI probes. Currently this is done via SSH, as are updates to existing measurement modules. However, performing this sort of uploading in a way that is both scalable and trustworthy is one of the hard problems that must be addressed before NIMI scales to a significantly larger size, and we further discuss this problem in depth in § IV.

B. Policy control

NIMI supports diverse policies through the use of Access Control Lists (ACLs) residing on each NIMI probe. An ACL table is comprised of columns representing actions and rows listing credentials. The intersection of an action and credential is a boolean value, or, eventually, a script that can be applied to the arguments of the request to yield a boolean value. If the value is true, then a request by the given credential for the NIMI probe to perform the given action is authorized; otherwise, the request is discarded as unauthorized.

A NIMI probe receives its initial ACL table on startup from its CPOC. The CPOC, however, can delegate some ACL management to other CPOCs. We view such delegation as a vital part of the NIMI architecture, as it makes it easy for a site to participate in a public measurement project: rather than having to manage the access control details particular to that project, the site can delegate the management to someone else more closely involved in the project, safely giving them the right to set up ACL entries, but only for those actions associated with the project.

C. Local management and control

Along with the policy control discussed in the previous section, NIMI is designed to support easy local management and control of a probe's actions; that is, it is simple for the local system administrator to ascertain what the probe is doing, and to ascertain and override the allowed measurement policies by directly inspecting and editing files in the file system.

First, the ACL table exists on a NIMI probe as a flat text file within the `acl/` subdirectory. If a system administrator wishes to remove a particular policy (ACL entry), they can simply edit the file and delete the corresponding ACL row. In addition, upon receipt of a measurement request *scheduled* moves the request into a text file located in the `pending/` subdirectory. When the time to run the measurement arrives, the measurement request is moved to the `active/` subdirectory. Upon completion of the measurement, the measurement request and results (as a *tar* file) are moved to the `dock/` subdirectory to await shipping to the DAC by the *nimid*. The *nimid*, upon delivering the *tar* file to the appropriate DAC, moves the measurement request and *tar* file to the `completed/` directory, where it resides until explicitly deleted by a user. Hence, the status of any measurement is immediately determinable by NIMI or a human system administrator by examining the file system.

D. Communication and security

NIMI messages are encrypted via RSA private/public key pairs, and passed between NIMI components via TCP/IP. A message consists of a header listing the information necessary to decrypt the message body (key name), and an encrypted message body. The message body, in its unencrypted form, consists of one or more message "blocks." Each block type maps to a request that a NIMI component can service, as well as any data necessary to complete the request.

Block type	Request
`BOOT_ME`	*nimid* asks CPOC for ACL config.
`ACL_ADD`	install a new ACL entry
`ACL_DEL`	remove an ACL entry
`TEST_ADD`	schedule a measurement
`ACK_TEST_ADD`	acknowledge receipt of `TEST_ADD`
`TEST_DEL`	remove a measurement
`RESULT_XFER`	receive a measurement result *tar* file
`ACK_RESULT_XFER`	ack receipt of `RESULT_XFER`
`RESULT_FAIL`	inform that a measurement failed
`FLUSH_DOCK`	process all results awaiting delivery
`DOCK_RMDIR`	release specific result file directory
`QUERY`	get listing of measurements
`QUERY_RESULT`	receive listing of measurements
`ERROR`	report remote error
RESULT_FETCH	retrieve a particular result
RESULT_DEL	remove a particular result
KEY_XFER	receive a public key
TOOL_XFER	receive a measurement tool

TABLE I

DIFFERENT BLOCK TYPES IN NIMI MESSAGES.

Table II-D summarizes the different NIMI block types. The bottom four are pending implementation, the top fourteen are implemented.

III. EXPERIENCES

In this section we discuss our experiences with implementing a measurement infrastructure. We begin in § III-A with an overview of the current status of the infrastructure, and then in the remaining sections discuss different categories of problems we have encountered: those due to architectural decisions we made, or inherent to the problem of building a measurement infrastructure; those due to administrative and system heterogeneity; and those relating to NIMI being a good-sized, distributed system. Our goal is to provide a map of some of the pitfalls and context for some of the future changes to NIMI.

A. Status

The NIMI alpha distribution is currently deployed on 35 hosts at the following institutes and networks: ACIRI (Berkeley), APAN (Seoul), AT&T Research, Boston U., CAIRN, CERN (Geneva), Columbia U., Georgia Tech U., ISI East, MIT, NASA Glenn Research Center, Sandia National Laboratory, Stanford Linear Accelerator Center, Swedish Institute of Computer Science, U. of California (Berkeley, Los Angeles, and Santa Barbara), U. College (London), U. of Lulea, U. of Mannheim, U. of Massachusetts, U. of Michigan, U. of Oregon, U. of Pisa, U. of Southern California, U. of Virginia, U. of Washington.

These sites have been provided by volunteers who have given us SSH access to the platforms and sometimes administrative (root) access. All run versions of the FreeBSD or NetBSD operating systems (see § III-C below).

NIMI has been and continues to be used for a number of measurement studies:

- MINC (multicast inference of network characteristics) is a DARPA-sponsored project to estimate network-internal properties, such as loss rates and delays on individual links, using multicast-based tomography [To98], [ABCD+00].
- Web100 (funded by the National Science Foundation) measures and analyzes the end-to-end performance of popular user-level applications such as FTP and HTTP, with an aim to understanding how to improve their performance.
- A large-scale study to characterize the stationarity of Internet path properties (routing, loss, application-level throughput) [ZPSB00].
- A study comparing different implementations of the IPPM [PAMM98] draft metric for bulk transfer capacity [MA99] with throughput attained by native TCP [Al99].
- Ongoing `traceroute`, `treno` and `zing` measurements across the mesh, the latter to track multicast connectivity.

In addition, NIMI is used to monitor the PSC commodity and vBNS networks.

B. Architectural problems

The first class of problems we discuss are those generated by architectural decisions made when designing NIMI. In general, we can address these problems by modifying the architecture accordingly; they are not showstoppers, but illustrate some of the difficulties in realizing a coherent infrastructure design.

Some of problems are due to architectural omissions:

Remote error handling. A particularly large nuisance was neglecting to include in the architecture rich mechanisms for propagating error information. This made tracing failures due to relatively simple errors (e.g., out of disk space) quite difficult based on reports received by an MC running on a remote machine. We are now in the process of retrofitting robust error propagation into the NIMI design. We anticipate doing so to prove tedious but certainly tractable.

Grouping of associated measurements. NIMI was originally architectured such that each measurement submitted to a NIMI probe was a distinct entity. However, it soon became apparent that, for managerial purposes, it was highly convenient to be able to tag individual measurements to indicate they are elements of a larger experiment or "measurement group." We addressed this need by adding an opaque, textual "handle" associated with each measurement, which can then be structured in whatever fashion the user desires. The handle is visible in the sense that a NIMI platform can be asked to search for any pending, active, or completed measurements with handles matching a given pattern, but it is otherwise uninterpreted by NIMI.

A related problem concerns measurement post-processing by the DAC. It is simple for the DAC to run a particular script whenever it receives some measurement results, but for some measurement groups what is really desired is for the DAC to know when *all* results for the group have arrived, and then to run the post-processing script over the collection. However, this doesn't quite work, because as measurements grow in size, the possibility of some component of the measurement failing rises, so in fact the DAC should not wait for *all* results to arrive (because they might not), but merely *most* of the results, where "most" is a slippery notion meaning "all the ones that are actually going to be successful." We are currently working on accommodating these distinctions by incorporating into the DAC mechanisms for specifying how long to wait for which successful subset of measurements in a group before running a group-specific script over the subset.

Public key distribution. From the start, NIMI was designed knowing that to scale to large sizes, a public key server would be essential. However, we delayed addressing this need in the hope that a freeware public key server would become available that we could then integrate into NIMI. To our knowledge, no such system has materialized, so we are now in the process of integrating key distribution into NIMI, with each infrastructure having at least one well-known CPOC where infrastructure elements can register their public keys.

Measurement across NIMI failures. If the *scheduled* crashes for some reason, it is possible that measurements it initiated will continue to run. The current architecture has no provisions for either check-pointing running jobs, or recognizing that a job has survived a *scheduled* restart and is still running. Yet it is important to detect these measurements, as they consume resources and can potentially prevent other measurements from running, or perturb their results, by doing so. It seems likely that the cleanest way to address this problem will be as part of the resource control mechanism discussed in § IV.

End-to-end measurement integrity. NIMI measurement results are transmitted using TCP, and encrypted with strong cryptography, and it seemed natural to trust that this suffices in terms of ensuring measurement integrity. However, as the End-to-End principle points out [SRC84], true integrity can only be achieved end-to-end, where end-to-end means between the original producer of the data and its ultimate consumer. Indeed, we found that *tar* operations for packing up NIMI data could fail due to disk space limitations or a missing *gzip* utility for performing compression, and that due to what appear to be NFS bugs, *tar* files could be mysteriously truncated at sporadic times. These failures, like many others we discuss here, are really scaling problems, in that they are generally very rare when an infrastructure is small, and can be spotted and resolved easily, while for a large infrastructure, they become commonplace and hard to notice. Accordingly, we need to add an application-level integrity check on NIMI measurements to ensure the soundness of the results ultimately unpacked by the DAC.

Other architectural problems concern elements included in the original design that have proved inconvenient or detrimental:

Use of URLs. Influenced (or perhaps over-influenced) by the rise and then ubiquity of the World Wide Web (NIMI was conceived in 1996), URLs (or, worse, abominations thereof) are used to direct all NIMI messages. Although using URLs is indeed probably the correct design choice, given their continuing widespread use, they are tedious for humans to deal with directly. Accordingly, we have retrofitted shortcuts and nicknames based on predetermined default values for unspecified URL elements (such as port numbers); but, in retrospect, we would have a more coherent user interface to the infrastructure had we begun with notions such as NIMI names separate from NIMI URLs in the first place. Much of the difficulty will disappear with the development of a GUI for user interaction with NIMI, but this will be some time in coming due to other developments having higher priority.

Blocking I/O. NIMI uses TCP connections for transferring NIMI messages between remote NIMI components, and for transferring blocks between the *nimid* and *scheduled*. The original design used these in the default configuration, namely, blocking I/O, in which a process blocks if it tries to read input not yet available, or write output that cannot yet go out. It is much simpler to structure programs in terms of blocking I/O rather than non-blocking, but we found that as we scaled up the infrastructure, the possibility of blocking due to failures or overload of a NIMI component became sufficiently high that use of blocking I/O would impede the scheduling of large-scale measurements. We therefore converted the messaging library used by all NIMI components over to non-blocking I/O, which required altering a number of event loops and maintaining data structures reflecting partially-complete communication.

Multiple TCP connections. When an MC sends a measurement request to a NIMI probe's *nimid*, the *nimid* in turn forwards it to the *scheduled*. After the *scheduled* processes the request, it sends back an ACK, which the *nimid* forwards to the MC. The decision to relay through *nimid* comes from an attempt to reduce the *scheduled*'s load for tasks not directly related to measurement, since its measurement functions are timing-sensitive. While we believe this structure remains sound (due to the importance of assuring that the *scheduled*'s operations are not perturbed by delays such as performing public-key authentication checks), it means that a single measurement request can result in 7 different TCP connections:

1. A `TEST_ADD` message from the MC to the *nimid*;
2. The same `TEST_ADD` message from *nimid* to the *scheduled*, once it has past the authentication and policy checks;
3. An `ACK_TEST_ADD` message from the *scheduled* back to the *nimid*;
4. The same message delivered back to the MC from the *nimid*, which handles application-layer reliability (queueing messages for later delivery when TCP connections fail);
5. A `RESULT_XFER` message from the *scheduled* to the *nimid* when the measurement completes;
6. The same message from the *nimid* to the DAC;
7. An `ACK_RESULT_XFER` message from the DAC to the *nimid*.

Clearly, this is considerable overhead in terms of both TCP connection establishment and consumption of socket descriptors. This latter could lead to failures for large-scale measurements, and required modifying the messaging library to include a notion of rationing the allowed number of open connections, queueing messages for later delivery as existing connections finish. The original design was based on short-lived, unidirectional messages, to minimize communication state and therefore better tolerate failures. While this remains the right structuring from the *application* perspective, clearly the transport layer needs optimizing in terms of the messaging library keeping TCP connections open between the *nimid* and the MC or DAC, and using them bidirectionally, as long as there are sufficient resources to do so (and to close and later reopen them as resources are temporarily exhausted and then recovered).

Retry congestion. As noted above, the message library has a notion of a message "retry" queue, to be used if a component to which it needs to send a message becomes unavailable. At first, the retry queue treated all entries as equal, and that worked fine for small-scale measurements. But for large-scale measurements, we found that a component failure could lead to a large number of messages piling up in the queue. The *nimid* could wind up spending much of its time attempting to deliver messages that in fact still could not be delivered; or, once the component recovered, the *nimid* would so bombard it with queued up messages that it overwhelmed the receiving DAC, such that the delivery of many of the messages would fail, causing them to be queued again for retry. Worse, for measurements made across a large number of platforms, when the component recovered they could *all* wind up attempting to deliver previously-undeliverable messages at the same time, resulting in "implosion" at the DAC, and another round of failures.

Both of these are in essence forms of congestion collapse, which we addressed by adding exponential backoff to the amount of time between retries for messages in the queue. While this helps, messages could still amass to the point where the disk would clog with thousands of undelivered measurement results, greatly slowing down NIMI operations that scan directories looking for work. Finally, we implemented a "stale" directory into which we put measurement results that could not be delivered after a set number of attempts.

We finish with an example of a design element that we removed and then later returned. The NIMI probe was originally designed as three distinct elements, a watchdog, a messenger, and a scheduler. This structure complicated debugging because it meant that we could not easily run the messenger or scheduler directly under a debugger, but had to attach to a running process; and it also had lead to ill-advised sharing of global state between the messenger and the scheduler, as facilitated by the *fork* operation the watchdog used to create both. We then altered the structure so that the messenger (which evolved into *nimid*) and the scheduler (*scheduled*) communicated directly, and with no shared state. At this point, the infrastructure was small enough that there did not appear to be a need for an additional watchdog process. That changed, however, as the infrastructure grew and failures became more common. The watchdog is now implemented as a separate Perl script.

C. Administrative and system heterogeneities

In this section we discuss difficulties that arise from heterogeneities in the infrastructure. One form concerns administrative heterogeneities: the NIMI platforms are hosted by different organizations with varying policies concerning management of the platforms. While NIMI was designed from the beginning to accommodate different *measurement* policies, the problems we describe here instead concern policies regarding system management of the platforms. The other form of heterogeneity concerns operating system variations. These we attempted to minimize by using only the closely-related FreeBSD and NetBSD Unix variants for our initial deployment, but even given that degree of homogeneity, we encountered numerous problems.

While it has been a recognized requirement from the beginning that NIMI must cope with administratively diverse environments, a number of these difficulties caught us by surprise, and bode ill for larger scale, more diverse deployment, unless we can devise general solutions for ameliorating them.

NIMI installation. First, one area that was actually *not* much of a problem was installation of the NIMI software itself. Doing so entails:

1. fetching and unpacking either a binary or source distribution;
2. separately fetching the RSAREF or RSAEURO public-key cryptography libraries (this had to be done separately due to United States laws concerning export of cryptography);
3. if using source code, building the ensemble by issuing "`make`";
4. issuing "`make install`" to install the software;
5. generating a private/public key pair;
6. informing the CPOC administrator of the new NIMI's existence and its public key;
7. modifying the machine's startup file to execute NIMI upon reboot; and,
8. starting NIMI running.

The one significant trick with the above sequence is the third and fourth steps: they only work if the software correctly compiles, or the binaries correctly execute, for the given operating system. Preliminary experiences with porting NIMI to Linux, Solaris, and Tru64 Unix has led to converting the software to use *autoconf* for its configuration. This process is nearly complete, and will then entail an extra step, `./configure`, but with the benefit of much simpler portability.

Tools requiring privileged access. The first problem we found much more of a headache than anticipated was the need to install measurement tools requiring privileged access. Early during the design, we decided that NIMI should be capable of running as any user, rather than requiring super-user access, as the latter could present a formidable barrier to widespread public deployment. However, many measurement tools require access to raw sockets or the packet filter, both of which are privileged operations.[1] Consequently, the system administrator of that platform must explicitly configure the tool's installation to allow the privileged access, and this is required any time we install a new version of the tool.

We knew all this, but what we did not anticipate was: *(i)* large-scale measurement can involve frequent changes to existing tools and installation of new tools, and *(ii)* once the infrastructure is large, it can take a very long time before all of the sites have correctly configured the installation of a tool, due to quite

[1] Access to the FreeBSD/NetBSD packet filter (BPF) is controlled by file system permissions on `/dev/bpf*`, so a system can be configured to allow particular users access without requiring privileges. However, attaining this change is itself an administrative burden, and does not solve the problem for access to raw IP sockets.

large turn-around lags when dealing with site administrators via email.

This problem, while mundane, imposes very real constraints on scalability. We discuss approaches to address it in § IV.

Modifications to system configurations. Even with as homogeneous an environment as using only FreeBSD and NetBSD, we found that many systems were configured differently from one another and from our testing systems. Two particular problems were:

Configuring the Berkeley Packet Filter (BPF). A number of measurement tools use BPF (more generally, the user-level *libpcap* library, which on FreeBSD and NetBSD uses BPF) to capture precise measurements of network traffic. One mundane problem with using BPF is that the number of BPF readers is constrained by the number of `/dev/bpf*` devices available. We found that many machines have only one such device, which prohibited any concurrent access to the packet filter. This, for example, could prevent not only simultaneous measurements by different people (which we might want to avoid anyway; see § IV for further discussion), but even some measurements that were, conceptually, single-user. In particular, one experiment wanted to use two `zing` invocations at the same time, one for sending outbound traffic, and another for receiving inbound traffic. Both would attempt to access the packet filter, leading to one always failing.

Building more `/dev/bpf*` devices requires administrative access to the machine. A more serious version of the same problem concerns the machine's kernel configuration, which controls the maximum such devices that can be created. Again, a number of systems have this value configured to just one device. Unfortunately, changing it requires configuring and booting a new kernel, a step some administrators are understandably reluctant to undertake.

Configuring the TCP stack. Some measurements use the native TCP stack to perform transport-level or application-level measurements. These stacks behave differently depending on kernel parameters such as support for "large windows" [BBJ92] and T/TCP [Br94]. Altering these parameters again entails administrative access (and, of course, altering the TCP in more fundamental ways requires building new kernels).

Clearly, the above problems can be diminished by a site giving NIMI administrators privileged access, for example via *sudo*, and that is how we have dealt with these difficulties in a number of cases. But, also clearly, such administrative delegation will not work for large-scale NIMI deployment, because the corresponding trust model is not scalable. (Indeed, many of the current sites already do not delegate administrative access to us.)

Secure administrative access. While it is a hard requirement that in the long run NIMI must not require any interactive access by NIMI developers to NIMI platforms, clearly for a good while to come such access is required for development, maintenance and debugging purposes. All such access must be done in a secure fashion, with SSH being the clear choice. Unfortunately, SSH has a couple of pitfalls when used in an heterogeneous environment. First, SSH version 2 is not compatible with SSH version 1. The latter of these is freeware, and thus many sites are disinclined to upgrade to version 2, but other sites do run it. In addition, SSH is not trivial to install, and subtle configuration differences or errors can render a box inaccessible.

Crypto libraries. NIMI uses RSAREF for its public key cryptography (encryption and authentication). RSAREF, however, was not exportable under United States law, and hence NIMI European and Asian sites instead used RSAEURO. This difference complicates installation (especially since the RSAEURO Makefile integrates assembly code into the distribution by default!). In addition, the length of the keys produced by RSAEURO are stored in a field two bytes shorter then those produced by RSAREF, making it easy to miscommunicate the key—a simple enough problem, conceptually, but debugging cryptographic software can be quite challenging, if the failure mode is simply that a cryptographic signature is rejected.

Inconsistent measurement tool source code. For sound measurement, we sometimes need to know exactly which version of which measurement tool we use, and in general we would like to minimize surprises by using the same version as widely as we can. This is not always easy, however. For example, the version of `mtrace` that comes with NetBSD is different from the version of the source we have acquired that compiles under NetBSD; the version running under FreeBSD is different again. Some tools only work under certain operating system versions; others are only distributed as binaries.

A related problem is the difficulty of keeping the infrastructure up to date, even when consistent, run-everywhere source is available. The problem arises because once the infrastructure is fairly large, then increasingly during software upgrades some of the platforms are offline and fail to be upgraded.

These problems are again mundane but serious, and must be addressed before an infrastructure can coherently scale to a large size. We would like to address both by making NIMI platforms able to download new measurement tools (and new versions of existing tools) as part of the configuration dialog with their CPOC. But to do so, we must first address significant security issues, which we discuss further in § IV.

Kernel flakiness. Oddly, kernel tweaks installed on some of the NIMI systems actually proved harmful to measurement tools running on other NIMI systems. `treno` measurements to one site in particular (running a modified kernel) would periodically hang the `treno` tool running at the remote site performing the measurement, eventually leading to "wedged" `treno`'s consuming all available resources.

Note that what makes this problem serious is not the individual `treno` measurement failure, but that over time the failures would accumulate and render the NIMI platform inoperable for want of resources (e.g., process table slots). This difficulty is really an instance of the more general problem of controlling measurement resource consumption, which we discuss further in § IV.

Multicast woes. Finally, some of the problems came with the measurement domain itself. One of the large NIMI measurement projects involves using multicast traffic to estimate network link properties. As discussed in [ABCD+00], many sites lack solid multicast support, and those that do still suffer from sporadic multicast connectivity across the Internet core. This situation has improved considerably in the past few months, but is clearly out of the hands of the NIMI system itself (other than the degree to which NIMI measurements can help with diagnosing multicast connectivity problems).

D. Programming distributed systems

The last class of problems are related to software engineering—NIMI is a good-sized (40,000 LOC) distributed system—and, in particular, the sort of problems that arise when using the system on a large-scale. We naturally did much of our NIMI development using tests involving at most a handful of measurements, and when one of the NIMI measurement projects began scheduling measurements on the order of thousands per day, a host of latent difficulties cropped up.

Exhausting system resources. The most common such problem relates to exhausting system resources. When programming large systems in C or C++, it is difficult to avoid memory leaks unless one has a tool for automatically finding them. Another form of resource leak was with file descriptors; in certain rare circumstances, the code would fail to close a descriptor after encountering an error condition on it, and eventually these would consume all of the descriptors available to the program.

Other problems with resource exhaustion were due to simplifying assumptions made as we developed the code. Overgenerous default buffer sizes coupled with overly rich data structures led to much more memory consumption per measurement scheduled than actually necessary. A different problem concerned manipulating measurement results: the *nimid* would read in a *tar* archive holding a newly completed measurement's results, and then keep it in memory until it could deliver it to the measurement's DAC. This worked fine until a NIMI user scheduled a measurement that yielded a result file of scores of MB, at which point *nimid* could not read it all into memory, and thus could never deliver it, though it repeatedly tried. The solution to this particular problem is to keep the results on disk and stream them directly onto the socket connection to the DAC once established; and, on the other end, for the DAC to stream the results from its end of the socket again directly onto disk.

Clocks. Unstable system clocks have been an ongoing problem. We did not want to require that NIMI platforms include highly accurate time sources such as GPS, given expense, antenna location, and system installation difficulties. (We note that more homogeneous infrastructures such as Surveyor have been able to surmount these difficulties [Al97].) From previous experience [Pa98] we realized this would complicate one-way measurement, but expected that the clock synchronization would be good enough for purposes of coordinating measurement. In fact, this is far from the case. While some sites use NTP synchronization, others do not, or only synchronize upon reboot, and we have had to deal with clocks off by hours.

We deal with this problem in several ways. First, for measurements involving coordination, we schedule the traffic receivers to start running five minutes (nominally) before the senders, so the coordination will still work in the presence of up to five minutes disagreement between the clocks. Second, when an MC sends a measurement request to a NIMI, it compares a timestamp sent back by the NIMI with its own clock, and flags discrepancies if they are too large so the user is aware of the problem. Third, we are implementing a notion of "relative time," so that an MC can specify a measurement as taking place ΔT seconds in the future rather than at an absolute time T. The main difficulty with implementing relative time is to ensure that the time offset is as accurate as possible *when received by the NIMI*, which can be considerably later than when the MC begins sending the measurement request to the NIMI, due to delays in TCP connection establishment, packet retransmissions, and the like. There will always be some uncertainty with relative times due to variation in packet propagation times between the MC and the NIMI, but these will usually be on the order of hundreds of msec or less, or perhaps seconds if TCP retransmissions are involved.

DNS flakiness. We have found that during an extensive measurement run we will experience occasional DNS errors, such as hosts being unable to resolve the name of other NIMI platforms, or, in more than one case (different NIMIs), unable to resolve their own name! It was natural when we wrote our code to assume that such lookups would always succeed, and to fail in a hard fashion when they didn't (because that "couldn't" happen). In retrospect, we should have been more paranoid, and are now migrating NIMI over to: *(i)* carefully check all DNS return status codes, *(ii)* never trust a DNS call not to block (*scheduled* already avoids doing so), *(iii)* use IP addresses rather than hostnames when possible, and *(iv)* perhaps maintain a private DNS translation cache for backup use during DNS service outages.

Subtle interfaces. Perhaps the most frustrating problem was in converting the NIMI messaging library to non-blocking I/O. Following all of the documentation for setting up non-blocking `connect()` and `accept()` calls resulted in code that worked sporadically. After much consultation and mailing list browsing, we found a note from someone else who had run into the same problem. They reported that `getsockopt()` must be called with `SO_ERROR` to determine the state of the socket file descriptor *prior* to accessing.

IV. Secure Update and Resource Control

As we have developed above, two significant challenges for scaling NIMI—or any measurement infrastructure that strives for extensibility and multiple use—concern securely updating the measurement tools available on a platform, and controlling the resource consumption of individual measurements.

The need for update arises from the requirement of extensibility, though we have found that it also arises just for software maintenance of existing measurement tools. The need for updates to be *secure* arises immediately because measurement tools often require privileged access that an attacker could exploit. This need is especially pronounced for a shared infrastructure, in which the administrative host of a platform may not want to trust particular users of the platform.

The need for resource control rapidly became apparent to us as soon as NIMI was used for concurrent measurement projects. Indeed, we found it also comes up for solo measurement projects: if the project is sufficiently large, then its own subelements (individual measurements, for example) can conflict and derail the measurement. For example, one project scheduled successive measurements that entailed access to the packet filter. When the first of these wedged, the remainder would fail because the first had not released the (scarce) packet filter resource, so they could not access it.

In summary, the combination of an infrastructure supporting extensible measurement and striving to protect its integrity brings with it two intrinsic dangers: measurement code might either compromise the security model, or consume excessive resources and compromise other measurements. We treat these two together, because they are really two sides of the same problem: controlling what the measurement software can do.

A. Trust models

We begin by discussing different trust models, as these delimit how strenuously we must work to ensure safe operation. Currently there are two forms of trust in NIMI: first, the volunteers hosting platforms trust the NIMI developers enough to either grant us administrative access, or to themselves perform administrative actions such as granting privileges to a measurement tool upon request. This form of trust is based on the fact that the volunteers know the people involved in the project and trust us to take appropriate care to protect their systems. This model often makes sense for a research system in its early stages, but clearly will not scale in the future when there are many more administrative domains and people who want to modify elements of a NIMI configuration.

A related form of trust regards "trustworthiness by eminent authority," by which we mean the fact that if asked to install "the new version of `traceroute` available from its usual location," most administrators are willing to do so because they trust the developers of the tool (who do not necessarily have anything to do with NIMI) to ensure that the software they release is safe. This sounds risky, and likely with time will grow more so,[2] but it is in fact practiced by everyone who uses publicly available software except those who rigorously inspect the source code prior to installation.

In summary, NIMI requires a more flexible trust model than the above to make its extensibility practical. Otherwise, researchers who have good ideas for experiments will not be able to get the necessary tools deployed without either ties to a large number of NIMI operators, or to the developers of the measurement tools or NIMI.

B. Threat models

We now turn to the sorts of threats against which we wish to protect the infrastructure.

Subverting the platform. First, uploaded measurement code might exploit a weakness in NIMI to obtain unauthorized access to the platform. Since with time NIMI platforms may be deployed on many networks across the Internet, a systematic attack on the NIMI system could potentially create a major threat. If NIMI platforms could be exploited to obtain root or other non-authorized general access to the host system, an attacker could then use the compromised platforms to mount additional attacks on other systems. Given that the platforms themselves are likely to be well connected to the network, they would be especially attractive for mounting denial of service (DoS) flooding.

Any non-authorized general access can also be used as a stepping stone to hide attacks on other systems. While not unique to NIMI, this sort of illicit activity could very seriously damage NIMI's reputation and render NIMI deployment unacceptable to many.

Also given that the platforms are likely to be well connected, they may be close to network infrastructure carrying aggregated traffic from many users. If a NIMI platform is directly attached to a broadcast network, it could be used to sniff transit traffic. This sort of attack is so attractive that even with the security measures described in this paper, it is expressly recommended that NIMI platforms be attached to media that does not lend itself to sniffing.[3]

An attacker gaining general access to NIMI platforms could also alter system or measurement software in unknown ways, casting doubt on, or even surreptitiously altering, subsequent measurements. In addition, if a NIMI platform has been tampered with it may be difficult to reinstall the system from trusted media, due to a potentially remote location.

Attacks launched from within NIMI. Even without subverting the NIMI platform, the potential exists for malicious measurement software to use NIMI platforms to attack other systems. An uploaded tool might construct attack packets (e.g., for DoS or scanning) under the guise of a measurement. This form of attack is particularly problematic given the desire to ensure that NIMI platforms can perform a wide variety of measurement. Even the very natural requirement that all traffic sourced by a measurement utility must have the NIMI platform as the source address is in tension with the need to provide access to "raw IP" for some measurement programs. But, more generally, NIMI needs mechanisms to control the form of any packets sent by a tool.

In addition, as discussed above, measurement machinery can also be exploited directly to collect private information. Many of the current NIMI tools rely on BPF to precisely timestamp measurement traffic off of the network. But using BPF introduces the possibility of obtaining copies of messages other than those related to the measurement. This can be mitigated by requiring use of a filter restricted to only capture traffic sent to or from the local host, but this does not protect against reading packets meant for some other measurement.

Perturbing other activity. Malicious or defective measurement software can also interfere with other measurements, by consuming excessive NIMI resources or generating bogus traffic that the other measurement will mistake for its own. For example, a measurement that keeps large buffers in memory might cause other measurements to experience page-fault thrashing. Various resource management issues are discussed in § IV-D.

There is also a potential for interaction between NIMI and other activities on the platform. The NIMI architecture does not require that platforms be dedicated to NIMI alone, and a number of sites have deployed NIMI on general-use machines.

C. Update models

We can picture implementing software updates to NIMI platforms in a number of different ways. A simple approach would be to bundle the tools into the NIMI code distribution itself, such

[2] In a recent incident, a public distribution of the popular "TCP wrappers" security software was modified by attackers to include a backdoor, a vulnerability subsequently exploited to gain illicit access to numerous machines.

[3] Note that the NIMI architecture can be used for legitimate passive measurements—there is nothing in the architecture that restricts its use to active measurements. But we omit passive measurements from our general discussions because they are much more problematic for a public infrastructure.

that the tools are updated when NIMI is updated (and perhaps the same trust model is applied to the tools as to the NIMI code). Doing so would almost totally undermine NIMI's extensibility, however. In fact, from our experience to date it is already clear that methods that rely on "out of band" mechanisms to update measurement tools (such as email to an administrator asking them to install the new software) scale very poorly, since some administrators may take an exceedingly long time to attend to such requests.

We instead would like to use the already-existing NIMI machinery for moving data to and from the platforms. Furthermore, the existing measurement scheduling machinery provides an opportunity to verify that the proper tools are present on the platform. If an MC requests a tool that is either not present or out of date (wrong version), then the platform could request the tool from either its CPOC (which could redirect it to a delegated repository) or, perhaps, the MC itself. (The first of these clearly represents a stronger trust model than the second.) Furthermore, if the CPOC does not already have the tool, it could in principle request it from the MC.

Each of these tool delivery routes has advantages and disadvantages. If platforms obtain software via their CPOC, then the distribution machinery will scale better, because all of the CPOCs can concurrently distribute tools to their associated NIMI platforms. The CPOC also has the opportunity to perform an extra inspection (or compilation) step that might be used to provide additional validation of the code. If, on the other hand, a small pool of NIMI platforms obtains the software directly from the MC, then new tools can be deployed more quickly. This approach would better facilitate tool testing.

We believe that both of these mechanisms are needed. If an MC requests a measurement that requires a missing tool, then the platform's *nimid* should first determine if the MC has permission to offer that tool directly. If so, it contacts the MC for the tool. If not, or if the MC replies "use the CPOC," the *nimid* would then determine if the MC has permission to request that the NIMI platform upload software from the platform's CPOC.

If a NIMI platform requests a tool that the CPOC does not have, the CPOC could use a similar mechanism to obtain it. However, with the CPOC there is an additional opportunity to check the appropriateness of the request. This check could be fully automatic (e.g., a code scanner; see below) or partially manual, requiring explicit administrative approval.

The design should permit the administrator responsible for the CPOC to select semi-automatic updating, where new tools are automatically transported to the CPOC and inspected, but held for the administrator's approval before disseminating them to the NIMI platforms. This approach provides a safety net for sites wishing to be more conservative about installing tools. The goals of the other security mechanisms we discuss below should then be to make the entire system sufficiently robust and secure that nearly all domains will elect to rely on fully automatic mechanisms, and that the ones that don't will be sufficiently attentive such that there are not long deployment delays.

D. Resource management

The availability and accuracy of NIMI measurement may be adversely affected by resource contention or starvation on the platforms. As mentioned above, in many ways resource protection parallels security issues, in that both concern controlling the possible behavior of measurement programs.

Many of the resources we need to manage are common to other multi-use systems: CPU, memory, disk space, and I/O activity, especially network activity. Other resources—such as access to the packet filter, specific TCP/UDP ports, or receiving particular forms of ICMP responses—are scarce, and typically bound to specific measurements. Uncontrolled access to these resources will likely cause measurements to fail.

We therefore need mechanisms to ensure that measurements do not suffer for resource starvation, or, conversely, that measurements do not consume more resources than policy allows.

Finally, as noted above, some NIMI platforms are shared-use. The other legitimate users of the systems may need to receive priority access to the resources, perhaps varying with time-of-day, or on an ad hoc basis. These situations again require an ability to limit NIMI resource consumption as policy dictates.

E. Code validation vs. sandboxing

There are two basic approaches we could use to facilitate secure software update and resource control: validating that code behaves correctly, or *sandboxing* code so that it's impossible for it to behave incorrectly in a significant way.

Code validation. The first of these can be done statically, by scanning imported code to analyze its behavior, or dynamically, by monitoring a running program. For general programming languages, the Halting Problem dictates that we cannot realize completely accurate static analysis, but instead must limit the semantic model available to the programs, possibly rejecting some safe programs because we cannot soundly prove their safety. Some languages are much more amenable to static analysis than others; C is particularly bad in this regard, given its highly flexible pointer semantics. Unfortunately, most measurement software is written in C, and it is not clear that rewriting them (either in C or something else) to conform with a restricted semantic model is any easier than rewriting them in a "safe" language (see below).

We can also consider validating code dynamically, but this immediately raises the question of what to do when at run-time we find a program violates its restrictions. If we at that point enforce the restrictions, what we have really done is sandbox the program, to which we now turn.

Sandboxing. In general, using a *sandbox* means that programs run in a restricted environment with access to only limited, tightly controlled resources. In addition, upon program termination the system may discard the environment. This goal is to prevent the program from contaminating other parts of the system.

NIMI has three natural boundaries we could use for sandboxing. First, NIMI platforms keep virtually no permanent state (just the name of their CPOC and a nascent ACL table allowing the CPOC's credential to modify the ACL table). Therefore, in the extreme case, we can in principle reinstall all of the software on the platform with only a short outage. Second, the platform could limit NIMI access to files by *chroot*'ing prior to execution of *nimid*. Third, we could also *chroot* each measurement into its own subtree within the restricted NIMI tree.

Each of these has its own merits and would be an effective tool against some classes of attacks, although in practice we have found that building a working *chroot* environment takes a surprising amount of work, and frequently reinstalling the NIMI software appears much too expensive.

The more fundamental problem, however, is that Unix does not provide the necessary tools for constructing *network* sandboxes. The mechanisms it provides for controlling what ports processes can bind to, what sort of access a program has to the packet filter, and what sort of packets a program can send and receive, all suffer from providing the wrong *granularity* of partitioning. Unlike the file system, for which the permission model is well developed and fine-grained, for network access the permission model is coarse-grained: a program either can or cannot access the packet filter, with no control over the type of filter it uses; it either can or cannot bind to a user port or a privileged port, with no control over which it picks from the range; it either can or cannot use raw IP to craft arbitrary packets, with no control over what can be placed in the individual header fields.

There are several classes of mechanisms we could employ to attempt to construct network sandboxes: introducing kernel modifications to restrict a program's possible behavior; linking measurement programs against a run-time library that monitors and enforces the program's activity; using a separate trusted daemon to monitor the program's activity and proxy sensitive activities such as packet sourcing and recording; or requiring programs to be written in a "safe" language amenable to secure run-time control (see below).

Requiring custom kernels seriously conflicts with ease of NIMI deployment and maintainability, and we must dismiss it out of hand. Adding instrumentation and controls via a run-time library seems at first like an attractive approach. But a determined attacker whose program executes dynamically constructed code can likely evade any run-time library; and even if we disable execution of code fragments in the stack and the heap, a malicious program could still rummage through memory until it finds the data structures used by the run-time library, and modify them to its advantage. It is difficult to see how run-time monitoring can work without resorting to techniques like those discussed in [WLAG93], which appear daunting for non-experts to implement (at all, much less correctly).

A monitor daemon could track system, network and disk accesses. In principle this would be more robust than a run-time wrapper, because the daemon is isolated from the measurement software. However, monitoring daemons are fundamentally limited because they only see events after the fact—a poor fit for preventing malicious activity, though these could work well for some forms of benign activity, such as suspending a program that consumes too much CPU.

The daemon gains much more control if it also *proxies* for the measurement software. For example, the software could be required to only send and receive packets by making requests through the proxy. This approach appears promising for one form of activity, namely using a packet filter to record high-precision timestamps of traffic for later analysis, and we plan to develop a packet-filter server to address the pragmatic packet filter problems we discussed above in § III-C.

But in more general terms, a proxy daemon highlights a basic tension between security and measurement: we need to ensure that security mechanisms do not distort our measurements to such an extent as to render the measurements insufficiently precise. For example, sending packets only via a proxy will incur 10's or 100's of msec latency (due to context switch delays and the like) between the measurement software and the network, making it impossible to precisely control when packets are actually transmitted. This latency is sometimes less of a problem for received packets, because they are timestamped in the driver before being processed by the proxy; but can also present difficulties if the measurement software needs to promptly respond to incoming packets, such as to generate echoes for measuring round-trip times.

Furthermore, since the Unix kernel has few mechanisms for network access control (other than per-process yes/no access to the packet filter), the use of proxies suffers from the same weaknesses as the run-time library approach. We cannot, in fact, force a program to use the proxy for its network activity; if there is any way for a program to contain surreptitious code, it can avoid the proxy. (This same lack of access control mechanisms also undermines another approach similar to proxies, namely using a wrapper that pre-opens allowed network sockets for the program to inherit.)

Given the above difficulties, it appears that the approach with the most promise is to migrate NIMI to using measurement tools written in a safe language. Here by "safe" we mean a language that both does not allow data structures to become corrupted (written by modules other than those explicitly allowed to do so), and that includes mechanisms for confining access to particular interfaces (such as network I/O) in ways that cannot be circumvented. Such languages have been used to implement secure, extensible systems by limiting the semantics available to the programmer, such that unsafe actions either cannot be requested, or can be securely denied by the run-time system.

Given such a language, we would then proceed as follows:

- We use the language to implement NIMI measurement tools. We will constrict the functionality provided by the language to only those operations required for measurement.
- The language needs to support restricted semantics for creating and collecting packets. These semantics must be extensible, but through a separate mechanism than that used for software updates.
- The language environment must include monitoring of resource consumption and mechanisms to enforce resource policy limits.
- A C language API for both the resource control and the packet processing routines should be available so that we can easily update existing tools to use the same resource and packet management resources as tools written in the safe language. These hybrid tools still require "trustworthiness by eminent authority," but will fit better into overall NIMI resource management.

There are a number of languages designed to be safe, such as Java, Perl's "taint" mode, and Python [Py00]. A number of issues arise when assessing which of these might work best: access to the necessary networking and timing primitives; ease of expressing fine-grained control; sufficient efficiency to not impede measurement precision; likelihood of administrators accepting their installation; debugging and maintenance properties of the resulting programs; and ease of porting existing tools.

V. Summary

There is great utility in being able to construct large-scale measurement infrastructures composed from diversely-administered hosts, supporting a wide and extensible range of measurements and access control policies. Doing so also presents major challenges and opportunities for mistakes, though we believe the NIMI design has proved flexible enough (with sufficient effort!) to overcome these.

Two key issues that must still be addressed for NIMI (or any large-scale infrastructure, for that matter) to prove sufficiently scalable for widespread public deployment are the problem of securely updating software on the measurement platforms, and the problem of constraining the resources consumed by different measurements. We argue that both of these can be unified in terms of controlling the behavior of the measurement software, and that the most promising technique for doing so appears to be requiring measurement software to be written in a "safe" language such as Java or Python. We are now beginning to investigate such languages to determine how best to address these problems in NIMI.

VI. Acknowledgments

Jamshid Mahdavi contributed a great deal to the initial NIMI design. We are also very grateful to the numerous volunteers hosting NIMI at their sites and to our MINC collaborators for their on-going feedback; and would like to thank Jeremy Hylton and Steve Bellovin for thought-provoking discussions on the problems of secure upload and resource control.

This work has been supported through funding by the Defense Advanced Research Projects Agency award #AOG205 and by the National Science Foundation under Cooperative Agreement No. ANI-9720674.

References

[AMMP98] A. Adams, J. Mahdavi, M. Mathis, and V. Paxson, "Creating a Scalable Architecture for Internet Measurement," *Proc. INET '98*, Geneva, July 1998.

[ABCD+00] A. Adams et al, "The Use of End-to-end Multicast Measurements for Characterizing Internal Network Behavior," *IEEE Communications*, to appear.

[Al99] M. Allman, private communication, 1999.

[Al97] G. Almes et al, "Surveyor Home Page, Tools, and Infrastructure," *http://io.advanced.org/surveyor/*, 1997.

[BBJ92] D. Borman, R. Braden and V. Jacobson, "TCP Extensions for High Performance," RFC 1323, Network Information Center, SRI International, Menlo Park, CA, May 1992.

[Br94] R. Braden, "T/TCP — TCP Extensions for Transactions: Functional Specification," RFC 1644, DDN Network Information Center, July 1994.

[Fr99] T. Friedman, private communication, 1999.

[HGDL97] C. Huitema, M. Garrett, J. DesMarais, and W. Leland, "Project Felix: Independent Monitoring for Network Survivability," *ftp://ftp.telcordia.com/pub/mwg/felix/index.html*, 1997.

[Ja89] V. Jacobson, *traceroute*, *ftp://ftp.ee.lbl.gov/traceroute.tar.Z*, 1989.

[La97] C. Labovitz et al, "The Internet Performance and Analysis Project," *http://www.merit.edu/ipma*, 1997.

[MA99] M. Mathis and M. Allman, "Empirical Bulk Transfer Capacity," Internet Draft, draft-ietf-ippm-btc-framework-02.txt, Oct. 1999.

[MM96] M. Mathis and J. Mahdavi, "Diagnosing Internet Congestion with a Transport Layer Performance Tool," *Proc. INET '96*, Montreal, June 1996.

[Pa98] V. Paxson, "On Calibrating Measurements of Packet Transit Times," *Proc. SIGMETRICS '98*, June 1998.

[Pa99] V. Paxson, "End-to-End Internet Packet Dynamics," *IEEE/ACM Transactions on Networking*, 7(3), pp. 277–292, June 1999.

[PAMM98] V. Paxson, G. Almes, J. Mahdavi and M. Mathis, "Framework for IP Performance Metrics," RFC 2330, Internet Society, May 1998.

[PF97] V. Paxson and S. Floyd, "Why We Don't Know How To Simulate The Internet," *Proc. 1997 Winter Simulation Conference*, December 1997.

[PMAM98] V. Paxson, J. Mahdavi, A. Adams, and M. Mathis, "An Architecture for Large-Scale Internet Measurement," *IEEE Communications*, 36(8), pp. 48–54, August 1998.

[Py00] "Python Language Website," http://www.python.org/, 2000.

[SRC84] J. Saltzer, D. Reed and D. Clark, "End-To-End Arguments in System Design," *ACM Transactions on Computer Systems*, 2(4), pp. 277–288, November 1984.

[To98] D. Towsley et al, "MINC: Multicast-based Inference of Network-internal Characteristics," http://gaia.cs.umass.edu/minc, 1998.

[WLAG93] R. Wahbe, S. Lucco, T. Anderson and S. Graham, "Efficient Software-Based Fault Isolation," *Proc. Fourteenth ACM Symposium on Operating System Principles*, Dec. 1993.

[WB98] H. Werner-Braun et al, "Active Measurements, Tools, and Infrastructure," *http://amp.nlanr.net/*, 1998.

[ZPSB00] Y. Zhang, V. Paxson, S. Shenker, and L. Breslau, *The Stationarity of Internet Path Properties: Routing, Loss, and Throughput*, in submission, Feb. 2000.