2012-09

XPLANE: Real--Time Awareness of Tactical Networks

Clement, Michael R.

Monterey, California: Naval Postgraduate School

NPS-CS-12-003

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

XPLANE: REAL-TIME AWARENESS

OF TACTICAL NETWORKS

by

Michael R. Clement and Dennis Volpano

September 2012

Approved for public release; distribution unlimited.

Prepared for: Office of Naval Research, Code 30
One Liberty Center
875 N. Randolph Street, Suite 1425
Arlington, VA 22203-1995

National Reconnaissance Office
Information Assurance Office
14675 Lee Road
Chantilly, VA 20151-1715

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From-To)*
01-09-2012	Technical Report	01-10-2011 – 01-09-2012

4. TITLE AND SUBTITLE
XPLANE: Real-Time Awareness of Tactical Networks

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Michael R. Clement and Dennis Volpano

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) AND ADDRESS(ES)
Naval Postgraduate School
1411 Cunningham Rd, Bldg. 305, GE-313
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER
NPS-CS-12-003

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research, Code 30
One Liberty Center
875 N. Randolph Street, Suite 1425
Arlington, VA 22203-1995

National Reconnaissance Office
Information Assurance Office
14675 Lee Road
Chantilly, VA 20151-1715

10. SPONSOR/MONITOR'S ACRONYM(S)
ONR, NRO

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
2012-00572

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES
The views expressed in this report are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

14. ABSTRACT

Emerging ad hoc networking environments, such as those currently being adopted by the defense and first response communities, call for a new generation of network monitoring capability. Current monitoring tools must either rely upon measurement protocols designed for a previous generation of systems or leverage only a subset of network devices that support some custom protocol. For certain kinds of networks, we make the case for shifting from a protocol to a language-based approach to measurement and for allowing a para-network facility which we call the XPLANE to reside on every network device, enabling system designers and administrators to craft tailored, localized measurements. The language we describe provides a higher-level abstraction for synchronous measurement while alleviating both the programmer and the interpreter from maintaining synchronization state for the computation. This has significant consequences for the complexity and resiliency of measurements. Our approach also separates localization and measurement from the logical network configuration, enabling diagnosis in the face of device misconfiguration. In this technical report we present the design and implementation of the XPLANE and provide several example applications to illustrate its use.

15. SUBJECT TERMS
Network Operations, Language Constructs and Features, Network Measurement, Network Diagnosis, Distributed Evaluation

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Michael Clement
Unclassified	Unclassified	Unclassified	UU	43	**19b. TELEPHONE NUMBER** *(include area code)* 831-656-1804

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

THIS PAGE INTENTIONALLY LEFT BLANK

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

Daniel T. Oliver
President

Leonard A. Ferrari
Executive Vice President and
Provost

The report entitled *"XPLANE: Real-Time Awareness for Tactical Networks"* was prepared for and funded by the Office of Naval Research and the National Reconnaissance Office.

Further distribution of all or part of this report is authorized.

This report was prepared by:

Michael R. Clement
Research Associate

Dennis Volpano
Associate Professor

Reviewed by:

Peter Denning
Chairman
Department of Computer Science

Released by:

Jeffrey D. Paduan
Vice President and
Dean of Research

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Emerging ad hoc networking environments, such as those currently being adopted by the defense and first response communities, call for a new generation of network monitoring capability. Current monitoring tools must either rely upon measurement protocols designed for a previous generation of systems or leverage only a subset of network devices that support some custom protocol. For certain kinds of networks, we make the case for shifting from a protocol to a language-based approach to measurement and for allowing a para-network facility which we call the XPLANE to reside on every network device, enabling system designers and administrators to craft tailored, localized measurements. The language we describe provides a higher-level abstraction for synchronous measurement while alleviating both the programmer and the interpreter from maintaining synchronization state for the computation. This has significant consequences for the complexity and resiliency of measurements. Our approach also separates localization and measurement from the logical network configuration, enabling diagnosis in the face of device misconfiguration. In this technical report we present the design and implementation of the XPLANE and provide several example applications to illustrate its use.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

The past decade has seen an unending progression of new ad hoc networking technology, each generation surpassing the last in capability and complexity. At the same time, networks are supplanting previous modes of communication in more and more settings such as tactical defense and emergency response. The result is an increasing population of novice users needing to diagnose network problems in the midst of real-world operations. In the middle of putting out a fire, first responders do not have time to run Traceroute to discover a misconfigured router. They need to understand why their networked application is not working and how to make it work. Hence there is a need for a new measurement facility that provides actionable information.

Given the plethora of complex and heterogeneous technologies, measurement facilities must be flexible enough to support diverse networks and applications. Expecting a measurement protocol to be designed once and suffice in all circumstances is unrealistic, while extending protocols to support additional features makes them fragile. One way to ensure a suitable measurement solution is to provide a language in which network designers can express custom measurements tailored to the operational domain and a platform for executing these measurements on the network at hand.

Such a facility must be able to make these measurements efficiently and precisely. Presumably these networks are used for mission-critical tasks; measurement execution should impose only a minimal burden on the network. The facility should also support measurements that can pinpoint the exact nature of a network problem. Localization of the measurement is key to both. By moving the computation to the source of measurement, voluminous raw data need not be transferred to distant parts of the network. Instead, only the code and computation state must be transmitted; we show that for non-trivial examples this can be done within a single Ethernet MTU. Placing a facility on every network device enables diagnostic applications to examine the network from any vantage point, thus supporting more granular analysis and precise results. Granted, not all types of networks are suited for this, but in the case of organizations that already mandate a standard software or firmware load it is not unreasonable to include an extra

piece of software for network diagnosis. Likewise, large organizations that work with device vendors might stipulate support for certain capabilities.

So what is the appropriate computational model for making localized measurements in an ad hoc network? Remote procedure calling (RPC) is a standard technique for localizing procedure calls. RPC requires an implementation to marshal remote procedure calls and returns. This can potentially place state at every network device to keep track of pending calls since a remote call can generate yet another remote procedure call. So RPC is not well suited for low-power, battery-operated devices. The initial procedure call will fail if there is loss of RPC state at any other device. Alternatively, one could adopt a thread semantics whereby a remote procedure call spawns a thread for asynchronous execution at a different device in the network. The thread would include code to handle returning the result of the call to its origin. This places an additional burden on the programmer and is a source of error. Instead, we provide higher-level abstractions for synchronous measurement that are compiled into a tail form prior to execution. Synchronous operations simplify expression of measurements as they eliminate the need for explicit marshaling of results of function calls to the origins of those calls. Further, with our approach, all state remains with a computation and no residual state is left behind once the computation moves away from a device. An intermediate device that aided in the computation at some point can momentarily disappear, losing its state, and still perform subsequent operations for the same computation should it later revisit that device. For networks composed of devices operating in dynamic or harsh conditions, this improves the chances that an application will terminate successfully at its origin.

A consequence of having a facility on every device is being able to coexist on the physical network but live entirely beside the logical network configuration. Moving the computation does not hinge on routing, for example. We accomplish this by using only link-layer broadcasts and placing control of how the computation navigates through the network in the hands of the programmer. Thus no topological state, even about neighboring devices, need be maintained independent of the computation. This allows measurements to succeed in the face of network misconfiguration, and in fact even allows one to make diagnoses about those misconfigurations! Nor do we burden devices with

maintaining a collection of attributes a la SNMP, and instead aim to observe all measurements directly from packet capture and, where necessary, packet injection. Hence all measurements reflect observed behavior "on the wire," not the idiosyncrasies of the implemented network stack.

Toward bringing this concept into reality, we propose a new language (XPL) and platform called the XPLANE with the following novel design aspects:

- All computation is localized to the source of the measurement.
- Code is written synchronously while a novel compilation process transforms it into code that maintains all state at the same location as the computation throughout execution.
- Localization and measurement are performed beside the logical configuration of the network.

The remainder of the paper is organized as follows. In Section II, we describe XPL the language in which measurements are expressed. There a transformation called CPS for Continuation-Passing Style is described. It is what frees network devices from having to manage synchronization state for applications during their lifetimes. Then we illustrate the use of the XPLANE for various measurement and diagnosis tasks in Section III. These tasks are only examples of what can be programmed in the XPLANE, but give a sense of the generality of the language. The last example, a non-trivial data rate measurement, is discussed in greater detail in Section III.B. It measures data rates for all physical paths from a resource to a client and performs measurements even when there are no routes along that path. Our current implementation of XPL is presented in Section IV, followed by discussion of future and related work.

II. THE XPL LANGUAGE

We give an overview of XPL here. To understand why XPL looks the way it does, it is helpful to see the design rationale behind the main language features. In the end, our goal is to express measurement and diagnostic applications that can run successfully within the network with as little impact on the network as possible. We introduce the following language elements to address each aspect of our goal:

- Measurement applications may need to conduct custom experiments across links or observe link behavior first hand. Hence, XPL provides primitives for packet injection and capture.
- Computations should be localized near the source of a measurement to reduce the transfer of measurement data over the network. Thus XPL provides a facility for transferring execution of a computation to another device.
- Measurements should be able to be made even in the face of an unknown or dynamic topology. Therefore, XPL also provides a flood primitive for transferring execution to all immediate neighbors. While flooding can lead to redundant computation at a single device when there are multiple distinct paths, it allows exploration of alternative paths when the expected path fails for some reason.
- Applications should not rely on devices to maintain control state when localization occurs since node memory may be volatile. Thus XPL programs are automatically transformed using continuations to ensure that localization never leaves any control state behind. Further, XPL does not allow distinct threads to communicate via shared device state or otherwise.

One can imagine a different set of operating assumptions about the network. These might change XPL design rationale. For instance, memory loss at devices might not be an issue if routers are not battery operated. Then it might make sense to maintain control state a la traditional remote evaluation and offering an alternative to flooding to

allow programmers the ability to avoid redundant computations. However, we feel our assumptions provide a reasonable starting point for discussion as they represent the more extreme end of the operating spectrum for ad hoc networks.

The XPLANE comprises a collection of XPL interpreters distributed across network devices, one per device (also called a node). An interpreter exposes certain device attributes. Every device currently has the following attributes: identifier (*node*), type (*node.type*), device time (*node.time*) and a list of device interfaces (*node.ifaces*). For now we constrain nodes to bear a single type such as Router, Server or Client. If $k \in$ *node.ifaces* then *node.k.ethaddr* is its hardware address and *node.k.ip* its IP address. An interpreter also exposes code attributes. There are currently three code attributes: size (*sz*) of the Ethernet frame (including the preamble, header and checksum) used to transmit the entire state of a computation, arrival interface (*ai*) and arrival time (*at*). The arrival time should be as close to wire arrival time as possible. Finally, an interpreter provides facilities for injecting and capturing arbitrary packets at any device interface; these are *send* and *pcap*, respectively, which are described in the next section. The language syntax is summarized in Table 1.

Arithmetic	+, −, ∗, /
Logical/Relational	=, <>, >, >=, <, <=, and, or, not
List	$expr :: list$, $list +\!+ list$, null, hd, tl, member, reverse, max, min, average
Variable binding	let $id = expr$ in $expr$
Functions	fun $id(arg,...) = expr$ in $expr$ (lambda arg $expr$)
Application	id arg ...
Conditional	if $expr$ then $expr$ if $expr$ then $expr$ else $expr$
Transfer	On { $expr$ } $expr$ OnFlood { $expr$ }
Send/Capture	send $expr$ $expr$; $expr$ pcap $expr$

Table 1. XPL Language Syntax

There are two operations for transferring execution of an application to a remote network node: *On {e} e′* and *OnFlood {e}* where *e* and *e′* are expressions. Each causes *e*

to be evaluated by one or more remote nodes. The nodes must be immediate neighbors. In the case of *On*, it is the neighbor to which *e'* evaluates and in the case of *OnFlood*, all immediate neighbors. Any free occurrence of a device or code attribute in *e* is bound in *On {e} e'* and *OnFlood {e}*. Evaluation by a remote node implies executing *e* in the context of device attributes there and code attributes associated with communicating *e*. Thus *OnFlood* mirrors SIMD parallelism.

```
fun f(path) =
   if node.type = Server then path++[node]
   else let n = node in
      if not member n path then
         OnFlood {f path++[n]}
in f [ ]
```

Figure 1. Server discovery in XPL

Primitives *On* and *OnFlood* are synchronous operations. Consequently, programmers do not have to write code to explicitly return their values. This leads to less code and potential for coding errors. To illustrate, consider the XPL code in Figure 1. It discovers from a given client, all physical paths to every server; some of these may be logical paths depending on configured routes. Here "++" denotes list append; "::" denotes list construction. The code floods away from a client with nodes added to a path by *f* if they have not already been visited. If a node has been visited by that execution thread, the thread terminates immediately with no value. If a node of type Server is reached then the reverse of the path to it is returned to the client. Thus the algorithm builds all symmetric paths to a server from the client in that the path from the server to the client is always the reverse of the path from the client to the server. Note that no code is written to send a discovered route back to the client.

Contrast this code with the code in Figure 2, which shows how server discovery would have to be written if *OnFlood* had asynchronous thread semantics. Additional code would be needed to return a route to the origin. Function *goback* does this with some nontrivial logic that reverses the path from the client to the server [7]. Handling its boundary cases also requires care. It is just another potential source of error and added complexity that is not necessary.

```
fun goback(path, route) =
   if null (tl path) then node::route
   else On {goback (tl path)
              (hd path)::route
            } hd (tl path)
in fun f(path) =
   if node.type = Server then
      let m = node in
         if null path then [m]
         else On {goback path [m]} (hd path)
   else let n = node in
      if not member n path then
         OnFlood {f n::path}
in f [ ]
```

Figure 2. Server discovery with asynchronous *OnFlood*

Despite their synchronous semantics, the XPLANE does not wait on the evaluation of an instance of *On* or *OnFlood* nor does an XPL interpreter maintain any control state in order to re-synchronize with their values. That is because every XPL program is converted into tail form using a CPS transformation prior to being executed [4, 16], so that computation remaining at that node is packaged with the transferred code. That way their execution in the XPLANE is fire and forget yet programmers can treat them as synchronous operations in their code. See Section IV for details.

III. APPLICATIONS OF XPL

We now present example applications of XPL to illustrate its use. Each application is tailored to a networking environment operating under certain assumptions. Because we anticipate the XPLANE being deployed in a variety of networks, our aim is to allow programmers to craft measurements suited to their environment.

Our first example illustrates how the XPLANE can produce measurements without relying on the native network stacks of devices. Since XPL relies only upon link-layer connectivity, it can be used to get approximations to some measurements when more accurate tools that depend on routes fail completely. For example, the code in Figure 3 approximates data rate between a host and server S one hop away.

```
fun datarate(s, c) =
   if c = 0 then [ ]
   else let h = node in
      let t = node.time in
      let r = On {let nbytes = sz in
         On {(sz + nbytes)/(at − t)} h
      } s in
      r::(datarate s (c − 1))
in average(datarate S 10)
```

Figure 3. Data rate approximation

Execution begins at the host. The data rate is calculated as an average of 10 samples. Function *datarate* recursively builds a list of *c* samples made between the host and node *s*, which is assumed to be a neighbor. For each sample, *h* is bound to the host identifier and *t* to a timestamp at the host. Then the body of the outer *On* is sent to *s* for execution. The size of the transfer is made available there in *sz*, which is stored in *nbytes*. Free variable *t* in the body remains bound to the timestamp from *h*. The body of the inner *On* is then sent back to *h*. There the sum of the size of both transfers is divided by the difference between the arrival time at of the inner *On* and the transfer time *t* of the outer *On*, producing a data rate sample *r* which is added to the list. If *node.time* is close to wire time of the transfer from *h* to *s* and the link is symmetric then the code provides a reasonable estimate of data rate, assuming the execution time at *s* is insignificant. Though

perhaps a crude approach, its real value lies in the fact that it can approximate data rate when there may be no logical path between the host and server due to the absence of a route or incompatible addressing.

A. DUPLICATE IP ADDRESS DETECTION

The next example is an application that checks for the presence of any non-router device (e.g., client or server) whose IP address duplicates the address of a router interface on its local subnet. One could envision a hastily-formed network in which all routers are fixed and have known good configurations, but non-routers are allowed to attach to the network with arbitrary configurations. Were such a device to duplicate the address of a gateway on a subnet where a server resides, the server might ARP for the gateway and instead receive the Ethernet address of this device. The symptom observed at devices outside that subnet would only be an inability to receive data from the server. Presuming the offending device has support for the XPLANE, our application allows a node to detect this condition remotely.

This application uses *OnFlood* to propagate throughout the network, similar to the server discovery example in Figure 1. For every non-router the code reaches, the node identifier and IP address of the arrival interface are recorded. Then that IP address is compared to the address of the router interface facing that device; if they match, a tuple is produced at the originating node.

The code for the application is given in Figure 4. It begins at some node, which is assumed not to be a router itself. The code first floods to all neighbors and proceeds only at those that are routers. At each router, the code first checks that the node has not already been visited. It then floods to all neighbors of the node. If the neighboring node is a router, the code recursively evaluates *dupecheck*. If it is a non-router, the code stores the node identifier in *ndid* and the IP address of the arrival interface in *ndip*, then transfers execution back to the last router. At the router, *ndip* is compared to the IP address of the router arrival interface, which is necessarily the interface facing the non-router. If *ndip* is equal to *node.ai.ip* then the value of the code is a tuple containing the node identifiers of the router and the offender, and the duplicated IP address. The implicit returns for each

On and *OnFlood* produce this value at the originating node. If *ndip* does not equal *node.ai.ip* the code terminates at the router with no value.

```
fun dupecheck(path) =
   if not member node path then
      let path = node::path in
         OnFlood {
            if node.type = Router then
               dupecheck path
            else let ndid = node in
               let ndip = node.ai.ip in
               On {
                  if ndip = node.ai.ip then
                     (node, ndid, ndip)
               } hd path
         }
in OnFlood {if node.type = Router then
               dupecheck [ ]}
```

Figure 4. Duplicate IP check

This algorithm detects all instances of duplicate addresses per the above definition in a single execution. Since the code visits all reachable nodes in the network via every distinct routed path, every non-router is checked against every neighboring router. Through the use of *OnFlood* each thread of computation executes independently of every other thread; the value of each thread that detects a duplicate address is a tuple that is produced at the originating node. Hence the value of the overall computation will be zero or more tuples, one per duplicate address detected.

Now suppose a network in which all routers have support for the XPLANE but all non-routers do not. Rather than visiting each non-router to gather information, we instead use the packet capture facility to examine packets received at each router interface for possible duplicate addresses. Captured packets are retrieved using *pcap* e_1 where e_1 is expected to be an interface at the current node. It produces a set of records corresponding to packets that were captured at that interface. Records contain both the contents of each packet and metadata such as time received or sent. Packet capture is managed at each node and the details of storing records are specific to the implementation. It is possible

that no packets were captured or that when *pcap* is evaluated the records of interest have been overwritten. More details on our current and future packet capture design are provided in Sections IV and V, respectively.

```
fun dupecheck(path) =
   fun ckpcap(ifc, plist) =
      if null plist then [ ]
      else let pkt = hd plist in
         if (pkt.srcip = node.ifc.ip) and
            (pkt.srceth <> node.ifc.ethaddr)
            then [node.ifc.ip]
         else ckpcap ifc (tl plist)
   in fun ckifaces(ilist) =
      if null ilist then [ ]
      else let iface = hd ilist in
         (ckpcap iface (pcap iface))
           ++(ckifaces (tl ilist))
   in let n = node in
      if not member n path then
         let d = ckifaces node.ifaces
            in if null d then
               OnFlood {dupecheck n::path}
            else (n, d)
in dupecheck [ ]
```

Figure 5. Duplicate IP check using *pcap*

The new version of the application is given in Figure 5. It is presumed to begin at any router. If the current node has not been visited already, the code proceeds to check each interface using function *ckifaces*. This function evaluates *pcap* at the interface, producing the resulting packet records which are given as list argument *plist* to function *ckpcap*. Each packet is checked for a source IP address (*pkt.srcip*) matching that of the interface and a source Ethernet address (*pkt.srceth*) different than that of the interface. Upon encountering any packet matching both criteria, *ckpcap* immediately evaluates to a list containing the IP address of the interface. If no such packets are found, the function produces the empty list. Function *ckifaces* concatenates these lists ultimately producing a list of IP addresses, one for each interface where a duplicate IP is detected. If this list is nonempty, *dupecheck* evaluates to a tuple containing the node identifier and the list,

which is produced at the originating node by executing the continuations. Otherwise, the code floods to all neighboring routers and repeats the process.

Unlike the code in Figure 4, this version may not detect all duplicate addresses in a single execution. A thread that reaches a router reporting a duplicate will not continue to flood to neighbors. This version demonstrates how, while the XPLANE affords the most capability when all devices participate, useful observations can still be made when only deployed in the infrastructure, say all routers. But what if the XPLANE is not situated on all routers? Then the XPLANE would require an external mechanism to support discovery of the plane and an alternative means for localization. It would also be limited in its ability to diagnose problems within that mechanism, just as operating within the logical network would limit its ability to reason about aspects of the network. Yet there would remain useful applications even in such a limited deployment.

B. RESOURCE DELIVERY RATE

Now we turn our attention to a more complex application utilizing both *send* and *pcap*. The idea is to write an XPL application that provides unique content perspectives for clients on a network. Content is stored as files on servers and clients wish to download them using a simple UDP protocol. Clients have different means of connecting to the network with different media and bandwidths. They may also have different points of attachment. So each client's path to a file may be different. The application must produce for a file, the expected data rate at which it can be delivered along all paths to a given client. The rate must be a function of path bandwidths, file size and packet size.

The inspiration for our approach comes from [13, 14], in which the total delay for a transfer is divided into the delay for the first packet and the delay for all remaining packets, assuming every packet follows the last across each link, as in a pipeline. Recognizing the cumulative nature of network delays, we illustrate how to compute total delay incrementally using XPL by accruing link latency and router forwarding delay along a path.

To measure link latency without requiring links be symmetric or clocks be synchronized, we measure inter-packet delay (IPD). IPD is defined to be the time from one packet to the next packet (first bit to first bit, or last bit to last bit for same-sized

packets) as observed at the receiving interface [17]. See Figure 6. Assuming little or no sending delay between successive packets, IPD represents precisely the time it takes to move one packet across a link, subsuming delays caused by serialization, link latency and media contention (overhead). It is straightforward to determine empirically by capturing packets at the receiver interface and calculating the difference between timestamps of successive packets. We are interested in the average IPD over a sample stream of packets of the same size sent from a node *u* to a node *v* in one hop which we denote $ipd_{u,v}$.

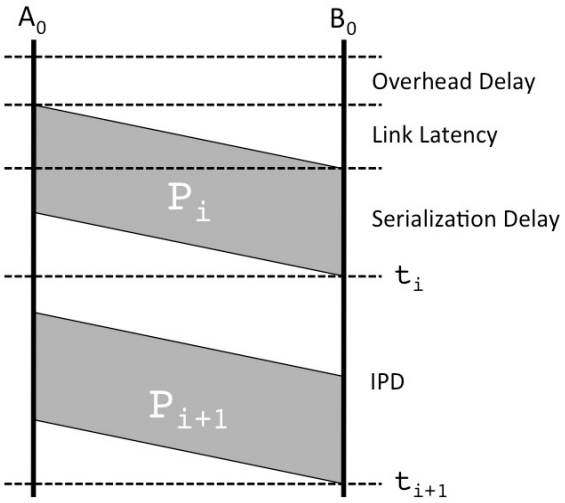

Figure 6. Measuring IPD

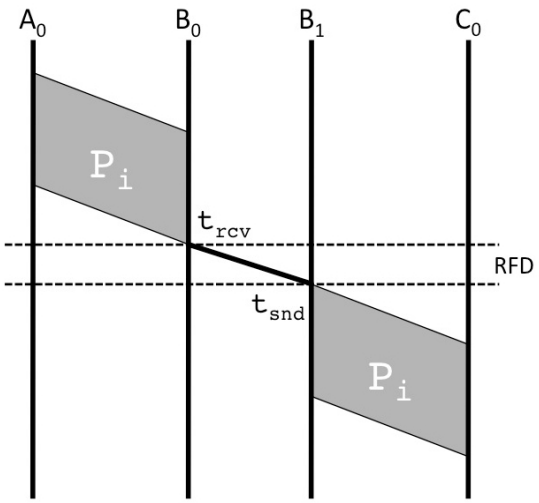

Figure 7. Measuring RFD

Router forwarding delay (RFD) is measured by capturing packets and subtracting the inbound timestamp from the outbound timestamp. See Figure 7. Packet capture must be possible at both the receiving and sending interfaces, and packets must be uniquely identifiable so their timestamps can be correlated. Ideally, packet timestamps must be made when the last bit is received and when the first bit is sent. We are interested in the average RFD over a sample stream of packets of the same size that transit a router *u* which we denote rfd_u. We define first-packet delay d_f and remainder delay d_r in terms of *ipd* and *rfd*. First-packet delay for a path of length *k* with $k - 1$ routers becomes

$$d_f = \sum_{i=1}^{k} ipd_{i,i+1} + \sum_{i=2}^{k} rfd_i$$

Since each link in the path exhibits a different IPD, the total delay for the remainder of the packets must use the maximum delay across the path. This is analogous

to calculating path capacity using the minimum link capacity. So for file size *b* in bytes and packet size *ps*, the remainder delay d_r for the transfer is

$$d_r = (\frac{b}{ps} - 1) \cdot max_{j \in 1...k} \;\; ipd_{j,j+1}$$

Figure 8 depicts d_f and d_r for three packets sent two hops from A to B to C. The average IPD from B to C dominates the transfer and therefore defines d_r. Total delay for the transfer becomes $d_f + d_r$ [14]. So the overall file transfer rate *r* in bits per second is

$$r = \frac{8 \cdot b}{d_f + d_r}$$

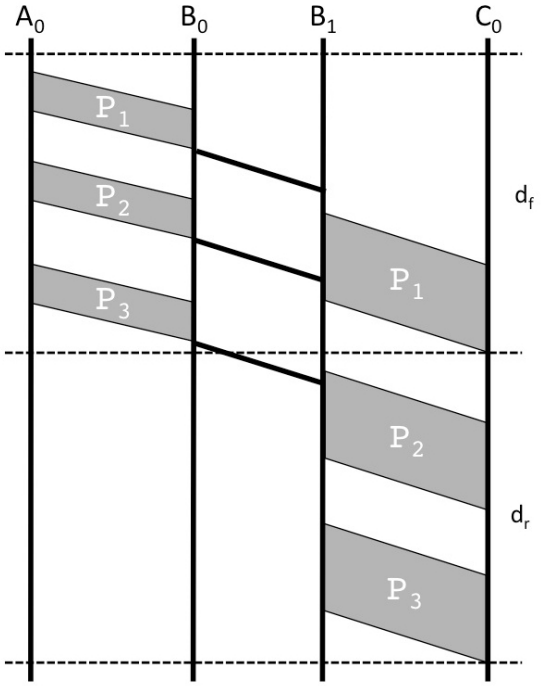

Figure 8. Total transfer delay for 3 packets across 2 hops

1. A localized path data rate algorithm in XPL

The algorithm is designed to compute a data rate for each physical path from a single server to every client on the network for a given resource size. Paths are uniquely identified by a sequence of router interface IP addresses ordered from client gateway to server. The server initiates a data rate measurement for each path in parallel that terminates at a client. Rates for a given resource are then returned to the server where

they are stored and may be subsequently discovered by clients using another XPL application that queries all servers for a specific resource identified by a uniform resource name. The discovery application constructs a path to the server which the server uses to look up the expected data rate for the given resource name. This application is similar to server discovery given in Figure 1.

```
fun prate(pa, ip, rai, df, mi, rps, rpc) =
   if not member node pa then
      let pa = node::pa in
      let aip = node.ai.ip in
      let ipdip = On {
         let iprb =
            [EchoReply, rps, node.ai.ip, aip]
         in fun s(c) =
            if c > 0 then send node.ai iprb;
               s (c − 1)
            else let sip = node.ai.ip in
               (On { P (pcap ai) (sip, aip)
                   } hd pa, sip)
         in s 100 } hd (tl pa) in
      let ip = (hd (tl ipdip))::ip in
      let rfd = if null (tl (tl pa)) then 0
         else let rprb = [EchoReply, rps,
         aip, hd ip, hd (tl ip)]
         in fun t(c) =
            if c > 0 then send node.ai rprb;
               t (c − 1)
            else On {
               R (pcap node.ai) (pcap rai)
                  (aip, hd (tl ip))
               } hd (tl pa) in t 100 in
      let df = df + (hd ipdip) + rfd in
      let mi = max(i, mi) in
      if node.type = Router then
         let rai = node.ai in
            OnFlood {prate pa ip rai df mi
               rps rpc}
      else let rate = (rps ∗ rpc ∗ 8)
         / (df + mi ∗ (rpc − 1))
         in [pa, aip::ip, rate]
in let pa = [node] in
   OnFlood {prate pa [ ] NIL 0 0 1450 724}
```

Figure 9. All-paths data rate algorithm in XPL

Figure 9 shows the code for path rate (*prate*), which takes as arguments the resource packet size (*rps*) and packet count (*rpc*) for the file, the latter being derived from the size of the file and *rps*. One can envision a version of the code that determines *rps* from the path MTU along the way and calculates *rpc* at the client. These arguments are used not only for the final data rate calculation but also for local measurements; the rate for each path is indeed customized to both the path and the file.

Any algorithm that involves sending probes should limit probing to immediate neighbors if possible to minimize network traffic. *Prate* is designed never to need to send probes to nodes more than two hops away. It achieves this by moving the computation close to where measurements are made. We refer to the location of the *prate* computation along the path as the "current" node; "previous" means closer to the server and "next" means closer to a client. Upon visiting a node, *prate* first checks whether it has visited this node already. If not, the node becomes the current node and *prate* prepends the node's identifier to list *pa* and IP address of the interface on which it arrived in *aip*. Then all code within the first *On* is executed on the previous node in the path. What actually gets transmitted over the link to the previous node is more than just the code within *On*. It includes a continuation for *prate* that when executed by the previous node, sends *prate* and the state of its execution back to the current node telling it how to resume execution of *prate* there.

While at the previous node, code within *On* performs an IPD measurement across the link by continuously sending 100 ICMP Echo Reply packets to the current node, since Echo Reply packets have the same header overhead as UDP packets and they do not elicit a response from the receiver. Each packet is sent using the *send* primitive. The primitive is an XPL construct of general form *send* e_1 e_2; e_3 where e_1 is expected to evaluate to an interface, e_2 to a packet descriptor, and e_3 to an arbitrary value. The value of *send* e_1 e_2; e_3 is the value of e_3. A packet descriptor is a list containing a packet type, payload size, source IP address, optional source-routed intermediate IP addresses, and the final destination IP address. For descriptor *iprb* the packet type is set to EchoReply and the payload size is set to *rps* so that measurements reflect actual resource packet sizes. The source address is that of the arrival interface *node.ai.ip*, and the final destination addressis *aip*. The full sequence of packets is sent by evaluating expression

```
send node.ai iprb; s (c − 1)
```

Its value is the value of recursive call *s (c − 1)*, which is ultimately the value of the surrounding else clause.

Upon sending all packets, the code records the IP address of the interface of the previous node that faces the current node. Next, the code constructs a list containing this IP address along with the computed IPD. Since IPD is computed from the receiving end, the code transfers again to the current node and looks for captured IPD packets. Captured packets are retrieved using *pcap* as described in Section III.A. Since *pcap* does not filter packet records we introduce function *P* to filter records of interest and extract necessary information. For brevity, we do not provide an XPL definition of *P* here; it filters records by source and destination address and calculates the average IPD from packet timestamps. Once computed, the code returns to the previous node via the continuation, assembles the list containing IPD and interface IP address, and returns to the current node to continue evaluation.

The observant reader may wonder why IPD computation cannot simply occur outside the *On* that localized the code at the previous node. In order to correlate sent and captured packets, the capture must occur within the scope of the *send*; that is, within the expression following the semicolon. We also note that once code is transformed into tail form certain transfers can be optimized away during evaluation. The transfer back to the previous node after computing the IPD is one instance.

Next, the average RFD is computed if the path constructed thus far has at least two nodes, that is, there is at least one intermediate router in the path. This entails a second transfer of *prate's* execution state to the previous node to collect RFD probe results. Because we may want to measure RFD for a router other than the one specified in the routing table, packet descriptor *rprb* includes an intermediate address for the router, which comes from the first element of list *ip*. Once the probing duration elapses, a new packet record processing function *R* is evaluated at the router. Like *P* discussed above, *R* filters on source and destination addresses. However, it does so on packets captured at both the receiving interface and on the forwarding interface (*rai*). It then calculates the average difference between send and receive timestamps. While IPD is measured in the forward direction or toward the client, RFD is measured in the reverse direction.

Assuming that RFD is independent of input and output interfaces, this results in fewer transfers of execution and thus is both more efficient in terms of communication and more reliable in the face of transient node failures.

Once the state of execution returns to the current node, cumulative measurement variables are updated; *mi* is the maximum path IPD in d_r. Finally, if the current node is a router then a new instance of *prate* is flooded from its interfaces to all adjacent nodes; otherwise, the final path data rate calculation is made and returned in a list along with node identifiers and IP addresses for the path taken.

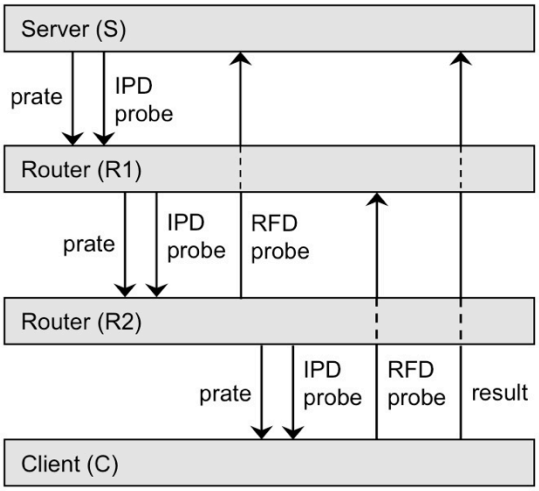

Figure 10. Flow and probes of *prate* across 2 routers

Figure 10 shows a summary of the flow of *prate* and its probes over a path with two intermediate routers, omitting transfers back and forth between nodes during measurement. Notice that each IPD and RFD measurement is localized in the network, involving at most three adjacent nodes. The data rate of a path is calculated incrementally, requiring a small, constant set of values to be maintained. Moreover, common path prefixes can be measured once and used in multiple path calculations. Hence XPL is a natural fit for expressing this measurement from a single server to multiple clients. The *OnFlood* primitive enables us to take advantage of common prefixes. It also allows us to explore alternative physical paths from server to client. Since *prate* relies only on adjacent nodes for IPD measurement and uses source routing for

RFD measurement, the routing configuration on nodes is unimportant. All routes to remote subnets can be omitted and prate still succeeds!

C. TRAFFIC IMPACT OF PRATE ON NETWORK

As with any active measurement approach, there is a cost to probing and querying a network. On a network such as the one depicted in Figure 10, *prate* causes the XPLANE to send a total of 27 packets containing computation (considering that some transfers are optimized away). Of these, 9 are considered overhead or "extra" since all XPLANE messages are broadcast, meaning an *On* message is still transmitted out all device interfaces. The XPLANE filters these messages by destination ID before they are interpreted. Another 5 of these are "return" messages resulting from the evaluation of continuations built up during execution. Table 2 breaks down the number of packets by type and link.

	On	Flood	Return	Extra	IPD	RFD
S-R1	2	2	1	2	100	100
R1-R2	3	2	2	4	100	200
R2-C	3	1	2	3	100	100
Total	8	5	5	9	300	400

Table 2. XPLANE messages and probe packets by link

Messages vary in size depending upon the execution state carried inside, but for our implementation and this particular application, the average size Ethernet frame is roughly 1330 bytes. Thus the total traffic generated for XPLANE computation is about 36 KB on this network. This assumes we pre-seed all nodes with functions *P* and *R*; under our current implementation of packet processing their expression requires an additional 650+ bytes of marshaled code. However, we anticipate achieving a dramatic reduction in marshaled code size; see Section V for further discussion.

In addition, prate causes a total of 500 ICMP Echo Reply messages for IDP and RFD probes, resulting in 700 frames being emitted since RFD probes traverse two links. Assuming probes are sized to fit exactly within a single MTU, that constitutes just over 1.5 MB of probes emitted to execute *prate*. This is a small fraction of the communications cost of Iperf with default settings, which consumed between 12 MB and

112 MB per run, depending upon the rate parameter given. Part of this savings is provided by using an incremental measurement approach. Observe that this algorithm reduces necessary probe traffic by being incremental. Wherever multiple paths spur off from an intermediate node, IPD measurements for the common path prefix are reused for each path. The choice to implement RFD in reverse comes with an added traffic cost; a measurement designer might choose either to implement RFD in the forward direction (requiring additional XPLANE messaging) or to omit RFD entirely if forwarding delays are considered inconsequential since they only apply to the first packet sent.

D. TESTING ENVIRONMENT AND RESULTS

Details of our prototype implementation are provided in Section IV; here we discuss in some detail the execution of the code on a small bench-top test network and provide performance results comparing it to an off-the-shelf measurement tool.

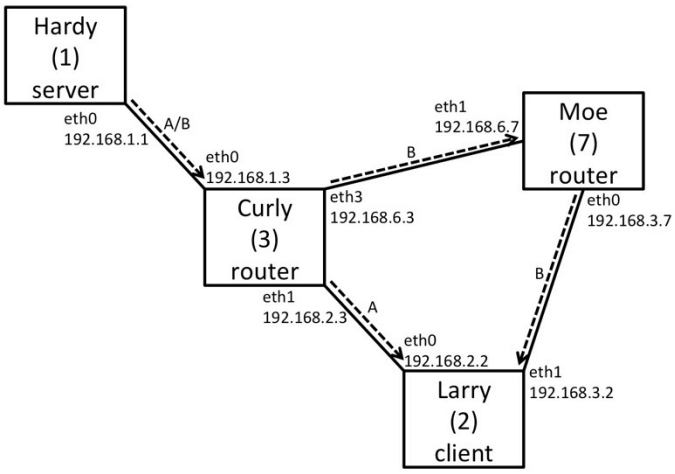

Figure 11. Test network

The test network is depicted in Figure 11. It is comprised of four Linux machines connected directly to one another via wired Ethernet. All links are operating in full-duplex; the link from Curly to Larry is operating at 10 Mbps while all others are operating at 100 Mbps. Machine processors range from 3 GHz in the server and client down to 433 MHz in the routers, though the prototype implementation has been run on 233 MHz processors with acceptable performance. For all tests, *prate* was initiated from

Hardy and traversed two paths to Larry, one directly via Curly (path A) and the other via Curly and then Moe (path B).

Our first test to validate the utility and performance of both the XPLANE and our algorithm was discovering all paths from server to client, approximating the expected data rate for each path. We chose Iperf as a benchmark for path data rate, since it determines data rate through actual data transfer vice incremental calculations. Because *prate* does not account for packet acknowledgement, we chose to operate in UDP mode. The sending rate was set to match the speed of the slowest link on each path (10 Mbps on path A and 100 Mbps on path B). As a side-note, when running Iperf with sending rates exceeding the speed of the path, we observed a decrease in reported data rates ranging between 0.1% and 2.3%; we suspect this is due to queuing at the sender.

We first ran *prate* on the network, originating from Hardy. Then we ran Iperf on each path separately, adjust routes as necessary. Since *prate* relies only on adjacent nodes for IPD measurement and uses source routing for RFD measurement, the routing configuration on nodes was unimportant; in fact, all routes could be omitted and the algorithm would still succeed!

	Path A	Path B
Iperf	9.544	94.55
XPL prate	9.522	93.72

Table 3. Iperf versus XPL *prate*, values in Mbps

Table 3 shows the path data rate estimates for both Iperf and *prate*. We see that *prate* provides estimates within 1% of Iperf for both paths, and unlike Iperf there was no need to tune sending rate parameters for the *prate* code; the IPD and RFD measurements "sense" the rate of each link by virtue of sending at the links maximum rate.

As pointed out above, *prate* enables customization of two resource parameters, packet size and number of packets. Due to the effects of pipelining across a path, data transfers consisting of many packets amortize the delay of transferring the first packet; likewise, the choice of packet size affects the efficiency of the pipeline and hence overall delay. An interesting property of this algorithm is that all measurements are made independently of the number of resource packets; hence with a single set of

measurements we can calculate the expected average data rate for a resource of any size, as long as it uses constant-sized packets matching those used for probing.

Bytes	Packets	Path A	Path B
4096	3	9.07	50.16
16384	12	9.40	77.21
65536	45	9.48	89.23
262144	179	9.50	92.85
1048576	713	9.51	93.80

Table 4. Transfer size versus expected data rate, values in Mbps

Given the amortizing behavior of larger transfers, resources with a small packet count should appear to have lower data rates than those with higher packet counts. Therefore, we expect to see the data rate asymptotically approach its maximum as *rpc* increases. Table 4 shows this behavior for total transfer sizes ranging from 4 KB up to 1 MB. All calculations are for the maximum size payload that kept packets to within a single MTU; in this case, 1472 payload bytes per simulated UDP packet (the payload size was decreased in RFD measurement to allow room for the IP source routing option). Note that while the algorithm assumes the last packet is full-sized, the presented calculations allow for a partial-sized final packet.

IV. XPLANE IMPLEMENTATION

We have developed a prototype implementation of the XPL interpreter by extending the TinyScheme interpreter [12]. This has also entailed building a custom link-layer communications library on top of the Linux packet socket interface, and developing software on top of libpcap [19] to handle packet capture. The entire interpreter runs inside userspace on an unmodified Linux 2.6 series kernel; the compiled code is under 300 KB.

All XPL primitives are implemented as C functions inside the interpreter, except *On* and *OnFlood* which required the addition of new language syntax to control evaluation of the body. Implementing *On* and *OnFlood* involves marshaling code for transmission by the link-layer communications library. The first step in marshaling is to create a code closure. A snapshot of the running code is extracted from the interpreter including all bound variables. TinyScheme provides a mechanism for producing code closures, which uses the same internal representation as Scheme lists. The contents of the closure are then written in Scheme syntax to a C string. Though inefficient, every closure computed for prate fit inside one Ethernet frame! The *pcap* function added to TinyScheme performs a query against a separate process that runs continuously in the background, capturing packets into a database. Rather than storing a full capture of all packets, the process uses a BPF [9] filter to limit which packets it admits and selectively stores packet attributes according to the algorithms we intend to run on a network. The implementation of a general *pcap* facility is a subject of ongoing work.

A. LINK-LAYER COMMUNICATION

The XPLANE uses a custom link-layer communications library. This allows the XPLANE to operate outside of logical paths determined by higher-layer addressing, routing, packet filters, address translation, and so on. XPLANE packets are implemented as Ethernet frames with their own IEEE-assigned Ethertype. The library is built using the Linux packet sockets API so no kernel modification is needed.

The XPLANE packet format is shown in Figure 12. Currently, only the Marker, Version, Packet Length, Receiver ID, and Checksum fields in the header are utilized.

Notice that no additional node attributes are included in the header; this reinforces the design decision that the code itself is responsible for accessing and carrying all network information, while the XPLANE only provides the means to move code from a node to its neighbors. Sender ID, Sequence Number, and Fragmentation are provided to support larger code sizes if necessary. A 160-bit message authentication field is also provided as a placeholder for future security extensions.

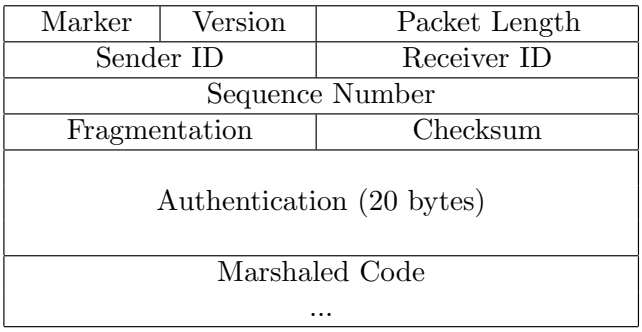

Figure 12. XPLANE packet format

All XPLANE packets are sent as Ethernet broadcasts out all device interfaces when *On* or *OnFlood* is executed. This has the advantage of not requiring an ARP-like facility but it can lead to unnecessary XPLANE packet handling at nodes that would not occur if packets were unicast. This cost should not be ignored as the interpreter is running in userspace. An alternative would be to implement an ARP-like service in the XPLANE mapping neighbor node identifiers to (interface, MAC address) pairs. Then *On* would produce unicast frames except in very fluid topologies where broadcast would be used instead.

B. CPS TRANSFORMATION

Before an application can be interpreted, it is transformed into tail form using a continuation-passing style (CPS) transformation [4, 16]. Traditionally tail form has been exploited by functional programmers to exploit properly tail-recursive implementations. It allows a constant-space, tail-recursive function to in fact execute in constant space. The transformation provides the same property here; however, it provides yet another

valuable property. It allows us to preserve the synchronous semantics of *On* and *OnFlood* without requiring an interpreter to wait for them.

For instance, consider XPL code for a variant of ARP:

```
fun whohas(a) = OnFlood {
   if node.ai.ip = a then node.ai.ethaddr
   }
in print whohas 10.0.0.1
```

It is not in tail form.

The contents of the *OnFlood* sub-expression execute at neighboring nodes; the final value of *whohas 10.0.0.1* is the hardware address for the neighboring node with IP address 10.0.0.1. Although this value must ultimately reside at the requesting node where print is evaluated, there is no code given to send the address back to the requester. We could write code by hand to send it back as in

```
fun whohas(a) =
   let m = node in OnFlood {
      if node.ai.ip = a then
         let ea = node.ai.ethaddr in
            On {print ea} m
   }
in whohas 10.0.0.1
```

The instance of *On* shifts printing of the discovered hardware address back to the requester. In this case the programmer must write code to explicitly send the address back to the requesting node, as well as to compute the requesting node and bind it to *m*. Taking this approach leads to tedious and error-prone code as shown in Figure 2. The other way to proceed is to transform the original ARP code into tail form automatically using continuations. This yields

```
fun whohas(a, k) =
   let m = node in OnFlood {
      if node.ai.ip = a then
         (lambda v On {k v} m)
            node.ai.ethaddr
      }
in whohas 10.0.0.1 (lambda u print u)
```

where *k* is a continuation parameter and *lambda* denotes an anonymous function. The hardware address is sent to the requester by applying the continuation *(lambda v On {k v} m)* to *node.ai.ethaddr*. This code is synthesized automatically from the original

version by the CPS transformation, allowing algorithms to be expressed more concisely by the programmer while maintaining the advantages of tail form code in execution.

As another example, applying the CPS transformation to the server discovery code in Figure 1 yields the code in tail form in Figure 13.

```
fun f(path, k) =
   if node.type = Server then
      k (path++[node])
   else let n = node in
      if not member n path then
         OnFlood {
            f (path++[n])
               (lambda v On {k v} n)
            }
in f [ ] (lambda u u)
```

Figure 13. Server discovery in tail form

To see how it works, suppose the code starts on node A and gets flooded to node C via another flood from intermediate router B. Then the sequence of continuations formed on the way to C becomes

```
(lambda u u)
(lambda v On {
   (lambda u u) v} A)
(lambda w On {
   (lambda v On {
      (lambda u u) v} A) w} B)
```

with the last continuation, which β reduces to

```
(lambda w On {On {w} A} B)
```

reaching node C where it's applied to path *[A, B, C]*.

V. FUTURE WORK

A more efficient packet capture and processing capability needs to be designed and implemented for XPL. Supplying a filter to *pcap* would help reduce the amount of packets that must be captured. Filter specification could leverage an existing language such as BPF [9] or NetPFL [2]. However, the amount of captured data could still be overwhelming. Hence we are exploring partial evaluation techniques that can minimize or avoid altogether the need to send packet captures between nodes. Ideally, only conclusions drawn from captures or some distillation of them would ever be transmitted.

The XPL interpreter is currently single threaded. Incoming XPLANE packets are queued and processed sequentially. This implies any waiting the interpreter does on behalf of an application prevents it from executing other XPL code during this time. Although our experience so far has not revealed this to be an issue, more testing is needed. Multi-threading may be necessary.

More work is also needed to improve communication and interpreter performance. Currently the entire interpreter lives in userspace, so the timing of certain operations is subject to the multiprocessing behavior of the underlying operating system. This can adversely impact the quality of measurements that rely on attribute *node.time* or successive *send's*. Communication cost can be reduced by shrinking marshaled code size, which can be done by partially evaluating continuations.

An area that has received little attention in XPL so far is security. Security was a major consideration in the active networks research. There is the threat of runaway code wreaking havoc on network performance and code that alters the behavior of devices in some malicious way. Even though a router may not forward a broadcast, the fact that XPL provides a flooding primitive in the context of unbounded recursion still seems like an invitation for trouble. Yet there are applications where it is useful, especially discovery. While we recognize that these security concerns constitute a technical challenge on their own, any solution to them will be relative to a particular threat model. So we have chosen instead to focus primarily on the functionality of XPL that if not done properly would limit its utility long before security concerns would. Transmitted code frames do have a header that currently includes room for a 160-bit authentication code if

desired. We imagine using it and perhaps the resource-bound technique of PLAN [5] to address security concerns when the need arises.

VI. RELATED WORK

There has been extensive research in the design of information planes for networks. See [8, 18, 20] for examples. While this work shares some of our goals, it is focused on aggregating data about network state from a predefined set of measurements already situated at observer nodes, limiting users to observe only what information the devices expose. These approaches also presume a reliable and known network upon which their information bus operates. Although they may tolerate temporary network faults and infer network properties from measurements made at reachable portions of the network, none can execute measurements at remote nodes in the face of misconfiguration or operate without a priori knowledge of the network topology.

Significant work has also been done in programming open networks from higher-level specifications describing desired behavior or properties [3, 6, 10, 11] and encoding protocols as programs to be executed within the network [1, 15]. This work is primarily aimed at defining network behavior top-down rather than observing it in order to enhance network awareness.

The closest work to ours is PLAN (Programming Language for Active Networks), which provides a programmatic interface that can be used both to construct new measurements as well as to recreate protocols such as datagram delivery [5, 7]. XPL's support for localization was inspired by PLAN. PLAN however lacks the semantics and features that we believe are essential in a language targeted for network measurement. It does not provide a flood primitive, relies on service routines at devices to provide data for applications rather than an intrinsic observation capability, and imposes an asynchronous distributed computing model on programmers that makes coding measurement and diagnostic applications more burdensome than it needs to be in many practical cases.

THIS PAGE INTENTIONALLY LEFT BLANK

VII. CONCLUSION

The XPLANE is a new platform where one can execute tailored measurements on ad hoc networks, with minimal impact on, and support from, the network. We present the design of both the platform and a new language, called XPL, in which one can express a wide variety of measurement and diagnostic applications. We also discuss our prototype implementation and provide a number of example applications. Performance analysis of the implementation is actively underway.

Our design reflects a specific set of operating assumptions about the network. The incorporation of a flood primitive reflects the assumption that the network topology may not be known at a node at the time when code is executed, for example. There are tradeoffs between efficiency in communication and the level of support required from the network. We are continuing to evolve the language to provide support for efficiently and successfully executing applications in different kinds of ad hoc networks.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] Andrew Campbell, Herman G. De Meer, Michael E. Kounavis, Kazuho Miki, John B. Vicente, and Daniel Villela. A survey of programmable networks. ACM SIGCOMM Computer Communication Review, 29(2), 1999.

[2] Luigi Cimimiera, Marco Leogrande, Ju Liu, Fulvio Risso, and Olivier Morandi. A tunnel-aware language for network packet filtering. In Proceedings IEEE Global Telecommunications Conference, pages 1–6. IEEE, 2010.

[3] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In Proceedings 16th ACM SIGPLAN International Conference on Functional Programming. ACM, 2011.

[4] Daniel P. Friedman and Mitchell Wand. Essentials of Programming Languages, pages 203–224. MIT Press, Cambridge, MA, 3rd edition, 2008.

[5] Michael Hicks, Pankaj Kakkar, Jonanthan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A packet language for active networks. In Proceedings 3rd ACM SIGPLAN International Conference on Functional Programming. ACM, 1998.

[6] Timothy Hinrichs, Natasha S. Gude, Martin Casado, John C. Mitchell, and Scott Shenker. Practical declarative network management. In Proceedings 1st ACM Workshop on Research on Enterprise Networking, pages 1–10. ACM, 2009.

[7] Pankaj Kakkar, Michael Hicks, Jon Moore, and Carl A. Gunter. Specifying the PLAN network programming language. Electronic Notes in Theoretical Computer Science, 26, 1999.

[8] Harsha V. Madhyastha, Tomas Isdal, Michael Piatek, Colin Dixon, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. iPlane: An information plane for distributed services. In Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI ’06, pages 367–380, Berkeley, CA, USA, 2006. USENIX Association.

[9] Stephen McCanne and Van Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In Proceedings USENIX Winter 1993 Conference. ACM, 1993.

[10] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In Proceedings 39th ACM Symposium on Principles of Programming Languages. ACM, 2012.

[11] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. Declarative infrastructure configuration synthesis and debugging. Journal of Network and Systems Management, 16(3):235–258, 2008.

[12] Dimitrios Souflis and Jonathan S. Shapiro. TinyScheme. http://tinyscheme.sourceforge.net/home.html, 2012.

[13] W. Richard Stevens. TCP/IP Illustrated, Vol 1: The Protocols, pages 503–506. Addison-Wesley, Reading, MA, 1st edition, 1994.

[14] W. Richard Stevens. TCP/IP Illustrated, Vol 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols, pages 300–302. Addison-Wesley, Reading, MA, 1996.

[15] David Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary Minden. A survey of active network research. IEEE Communications Magazine, pages 80–86, 1997.

[16] Mitchell Wand and Daniel P. Friedman. Compiling lambda-expressions using continuations and factorizations. Computer Languages, 3(4): 241–263, 1978.

[17] Xinyuan Wang, Douglas Reeves, and S. Wu. Inter-packet delay based correlation for tracing encrypted connections through stepping stones. In Computer Security - ESORICS 2002, volume 2502 of Lecture Notes in Computer Science, pages 244–263. Springer Berlin / Heidelberg, 2002.

[18] Mike Wawrzoniak, Larry Peterson, and Timothy Roscoe. Sophia: An information plane for networked systems. Technical Report IRB-TR-03-048, Intel Research, Berkeley, 2003.

[19] http://www.tcpdump.org/pcap3_man.html. PCAP Manual, 2003.

[20] Praveen Yalagandula, Puneet Sharma, Sujata Banerjee, Sujoy Basu, and Sung-Ju Lee. S3: A scalable sensing service for monitoring large networked systems. In Proceedings of the 2006 SIGCOMM workshop on Internet network management, INM '06, pages 71–76, New York, NY, USA, 2006. ACM.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Research Sponsored Programs Office, Code 41
   Naval Postgraduate School
   Monterey, CA 93943

4. Mr. Michael R. Clement
   Naval Postgraduate School
   Monterey, CA 93943

5. Dr. Dennis Volpano
   Naval Postgraduate School
   Monterey, CA 93943

6. Mr. John Moniz
   Office of Naval Research, Code 30
   Arlington, VA 22203

7. Dr. Salim Zafar
   National Reconnaissance Office, IAO
   Chantilly, VA 20151