FIRE: Flexible Intra-AS Routing Environment

Craig Partridge, *Fellow, IEEE*, Alex C. Snoeren, *Student Member, IEEE*,
W. Timothy Strayer, *Senior Member, IEEE*, Beverly Schwartz, Matthew Condell, and Isidro Castiñeyra

Abstract—**Current routing protocols are monolithic, specifying the algorithm used to construct forwarding tables, the metric used by the algorithm (generally some form of hop count), and the protocol used to distribute these metrics as an integrated package. The Flexible Intra-AS Routing Environment (FIRE) is a link-state, intra-domain routing protocol that decouples these components. FIRE supports run-time-programmable algorithms and metrics over a secure link-state distribution protocol. By allowing the network operator to dynamically reprogram both the properties being advertised and the routing algorithms used to construct forwarding tables, FIRE enables the development and deployment of novel routing algorithms without the need for a new protocol to distribute state. FIRE supports multiple concurrent routing algorithms and metrics, each constructing separate forwarding tables. By using operator-specified packet filters, separate classes of traffic may be routed using completely different routing algorithms, all supported by a single routing protocol.**

This paper presents an overview of FIRE, focusing particularly on FIRE's novel aspects with respect to traditional routing protocols. We consider deploying several current unicast and multicast routing algorithms in FIRE, and briefly describe our Java-based implementation.

Keywords—**IP Routing, Active Networks, Class-Based Forwarding, Differentiated Services, Virtual Private Networks**

I. INTRODUCTION

A ROUTING protocol has three constituent functions: it defines a set of metrics upon which routing decisions are made; it distributes this information throughout the network; and it defines the algorithms that decide the paths packets use to traverse the network. Furthermore, a well-designed protocol contains security mechanisms to protect the routing infrastructure from attack as well as from mischance or misconfiguration. In today's routing protocols, these functions are tightly integrated and cannot be unbundled. When a network operator chooses to use IS-IS [10] or OSPF [30], for instance, the information that is distributed about each link and the algorithm that is used to select paths are fixed; the operator is largely unable to tune the system to use a new algorithm or different metrics. The operator may select a more sophisticated set of metrics and routing algorithm by moving to Cisco's Enhanced Interior Gateway Routing Protocol (EIGRP) [48], but the operator is then forced also to accept EIGRP's state distribution and security mechanisms.

Recent work has attempted to harness the power of Active Networking [43] to provide extensible routing functionality in network nodes. Many techniques provide such flexibility by allowing individual data packets to explicitly participate in routing decisions [21], [22], [34], [43], creating a large range of security and stability concerns. Others limit such functionality to specially authorized and authenticated control traffic, but either severely restrict functionality [45], require per-packet processing on the forwarding path [49], or both [39].

FIRE, the Flexible Intra-AS Routing Environment, is an attempt to provide a more flexible routing system without sacrificing forwarding performance. Operators control a variety of key routing functions, including choosing which algorithms are used to select paths, choosing what information is used by the algorithms, and identifying traffic classes to be forwarded according to the specified algorithms. Expressing these ideas a bit more formally, FIRE splits the standard routing protocol into its constituent parts: secure state distribution, computation of forwarding table(s), and the generation of state information (i.e., determining what values to distribute). FIRE then exposes its state distribution functionality, making computation of forwarding tables and the generation of state information programmable at run-time.

The motivation for the novel aspects of FIRE springs directly from three simple observations:

- Routing algorithms continue to evolve.
- Today's simple link metrics are often insufficient to support new routing algorithms.
- Network providers are increasingly interested in providing specialized routing for different classes of traffic.

Over the past several years there have been considerable ferment and change in the design of multicast routing protocols [8], [18], [23]. Potential improvements for unicast routing have also been developed [6], [44]. Getting any of these algorithms actually deployed is difficult. Because current routing protocols have intertwined their algorithm with their state distribution mechanism, deploying new algorithms has meant, in most cases, implementing entirely new protocols. In short, the barrier to deploying new routing algorithms is very high. Indeed, part of the motivation for FIRE came from our experience trying to deploy a new unicast routing algorithm that finds approximately optimal paths based on multiple, orthogonal link metrics [11].

Most protocols use hop counts to approximate the cost of a link. More sophisticated protocols like EIGRP compute the metric from a mix of several link properties, but the EIGRP equation for combining metrics is fixed and represents a balance between possibly conflicting values. FIRE distributes a rich set of properties for use by routing algorithms. Properties can be configured values, extracted from router MIBs, or even dynamically generated by operator-provided applets. FIRE provides for programmable property generation, and properties can be regenerated when conditions suggest that forwarding tables need updating.

This work was supported by the Defense Advanced Research Projects Agency (DARPA).

C. Partridge is with BBN Technologies, Cambridge, MA 02138 USA (e-mail: craig@bbn.com).

A. C. Snoeren is with the MIT Lab for Computer Science, Cambridge, MA 02139 USA and with BBN Technologies, Cambridge, MA 02138 USA (e-mail: snoeren@lcs.mit.edu).

W. T. Strayer, B. Schwartz, and M. Condell are with BBN Technologies, Cambridge, MA 02138 USA (e-mail: strayer@bbn.com; bschwart@bbn.com; mcondell@bbn.com).

I. Castiñeyra was with BBN Technologies, Cambridge, MA 02138 USA. He is now with Pluris, Cupertino, CA 95014 USA (e-mail: isidro@pluris.com).

A preliminary version of this paper was presented at ACM SIGCOMM '00 in Stockholm, Sweden, August 2000.

The increasing heterogeneity of Internet connectivity strongly suggests that there is a growing diversity in path choices. Different paths between two points will be better suited for different applications. An obvious example is satellite links, which are being used more frequently in the Internet and which offer high bandwidth but also high delay. Most routing protocols in use today forward all traffic based upon the same forwarding table. Traffic classes may be differentiated with respect to resource reservation [9] and queuing priority [33], but packets are generally routed identically. In FIRE, each packet is routed based upon a forwarding table constructed to best suit its particular traffic class, as determined by the contents of the packet's header. By assigning different classes of traffic to separate forwarding tables, FIRE allows network operators to optimize for each class. Each forwarding table is constructed with a different algorithm, which may use its own set of metrics.

This paper is organized as follows. Section II provides a brief overview of FIRE, while sections III, IV, V, and VII present novel aspects of FIRE that are particularly interesting when compared to existing protocols, including configuration and management, programmable routing algorithms, dynamic properties, and our security model. Section VI describes FIRE's state distribution mechanism in more detail. Section VIII provides a quick tour of the programming interface to FIRE, while section IX demonstrates its flexibility by sketching FIRE implementations of several popular routing algorithms. Section X very briefly discusses our FIRE implementation, and section XI examines the strengths and shortcomings of FIRE's design. We survey related work in section XII before concluding in section XIII.

II. FIRE OVERVIEW

A traditional routing protocol generates a single forwarding table at each router, which the router then uses to determine where to forward incoming traffic. FIRE extends that notion by generating a set of forwarding tables, each uniquely defined by three pieces of information: the algorithm used to compute the table, the properties used by the algorithm in its computations, and a packet filter that determines which classes of traffic use the forwarding table. In FIRE, all three of these variables may be configured by the network operator at run time.

A. Algorithms

Routing algorithms in FIRE are downloaded Java programs. The algorithms are designed to use distributed network properties to generate a local forwarding table. Each instance of an algorithm is run in a separate Java Virtual Machine (JVM) on the router itself. This sandboxing prevents a buggy or malicious routing algorithm from disabling the entire router.

B. Properties

A FIRE system is composed of two classes of entities: nodes and links. Nodes consist of both routers and broadcast networks, while links are uni-directional adjacencies between nodes. Each entity in the FIRE system has a unique ID and a set of properties associated with its ID. The ID encoding specifies whether the entity is a link, subnet, or router.

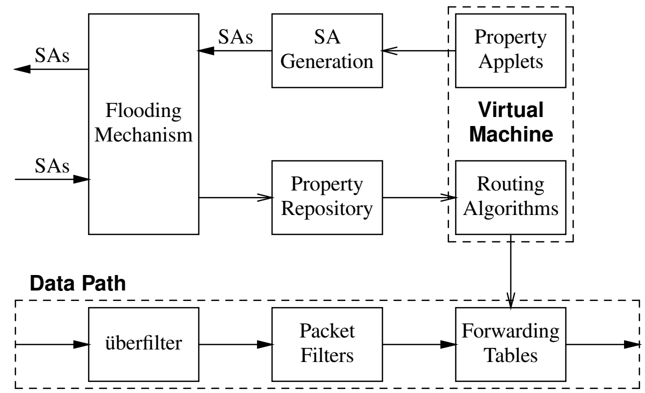

Fig. 1. Architecture of a FIRE Router. Each property applet and routing algorithm instance is run in a separate virtual machine.

A node is responsible for generating values for each of the properties being advertised in the network. Property values for links are generated by adjacent nodes. Network properties are the responsibility of nodes called Designated Routers, which are selected using a distributed election process similar to OSPF [41]. Some property values may be configured into the nodes (e.g., multicast support or policy-based cost values). Others may be readily available from the router's MIB (e.g., average queue length, CPU utilization, etc.). FIRE also allows operators to write their own property applets. Like algorithms, property applets are written in Java and each is executed in its own JVM instance.

All property values are distributed to every node in the network using reliable flooding. Each node stores these values in a *property repository* in order to build a complete and consistent map of the network. Updates of the values of these properties are periodically flooded throughout the network through the use of State Advertisements (SAs), refreshing the repositories at each router.

C. Filtering

A single instance of FIRE may manage several forwarding tables, and thus serve several classes of traffic concurrently. Traffic is classified using operator-specified filters and forwarded according to the generated forwarding tables. The tables themselves are generated by the routing algorithms which are run on the property repository, an internal link-state database containing property values for each entity in the network.

FIRE permits multiple instances of the FIRE protocol to be running concurrently in a system. Each FIRE instance is wholly self-contained, propagates its own state, and maintains a separate set of forwarding tables. This feature is designed to support Virtual Private Networks (VPNs), where overlays are used to make a single network look like several independent networks.

The internal architecture of a FIRE router is shown in Fig. 1. In the router's data path, all incoming traffic on the router first passes through an *überfilter* that assigns the traffic to a particular instance of FIRE. A packet can belong to only one instance, hence the filters must be disjoint. (In a router with only one instance of FIRE, the überfilter would contain a single entry that matches all packets.) Within a FIRE instance, packet filters determine which forwarding table the packet is to use. The packet's destination address is looked up in the indicated table, and the packet is forwarded accordingly.

D. Design philosophy

The most basic contribution of FIRE is the ability to modify *at runtime* the routing algorithms and property metrics used to generate forwarding tables. This dynamism requires great care to ensure robust, reliable behavior; the sheer scope of configuration and, even more importantly, programming options made available by FIRE's model tremendously increases the chance of misconfiguration or buggy implementations. This vulnerability has guided our design of the FIRE protocol. We have sought to make FIRE as stable a platform as possible.

The desire for robustness manifests itself in FIRE's security model, which is based on the philosophy of containment. SAs are signed by their creator to prevent modification in flight; advertisements can be suppressed or damaged in flight but cannot be surreptitiously modified or spoofed. Furthermore, each FIRE entity has an associated authorization certificate specifying what information it is allowed to advertise. These certificates are used, for example, to prevent a malicious or misconfigured router from advertising a route to a distant portion of the network and "black-holing" traffic.

III. Configuration & management

Central to the FIRE model is the notion of an *Operator*. FIRE works within a particular Autonomous System (AS)—an area completely controlled by one administrative entity. That entity appoints one or more people (e.g., a network operations center) as the Operator. The Operator is authorized to configure the network through two mechanisms: the Operator Configuration Message (OCM) and the Operator Configuration File (OCF).

A. Configuration messages and files

The OCM is a special State Advertisement that contains the configuration rules for the FIRE system, the set of OCFs that are to be loaded, and names one of them as the *running* OCF. An OCF lists the routing algorithms to use, the properties to advertise, and the filters to map traffic classes to forwarding tables. Both the OCM and OCFs are cryptographically signed with the Operator's secret key. There can be only one OCM valid at any point in time. The OCM is injected into the network by an Operator at any FIRE node, and is distributed along with normal routing updates throughout the network by the standard flooding mechanism.

Upon receipt of the OCM, a node retrieves the listed OCFs from one of the file repositories specified in the OCM. File retrieval is facilitated by a special, simple file transfer protocol called the Large Data Transfer Protocol (LDTP) [35]. This protocol is based on the Trivial File Transfer Protocol (TFTP) [42], enhanced to protect against insertion attacks and reduce its vulnerability to Denial of Service attacks. When an OCF is retrieved from a file repository, the OCF is parsed and any additional support files downloaded. In particular, the OCF contains the list of routing algorithms and property applets, along with a list of file repositories where these files can be obtained. LDTP is again used to retrieve these files. All files to be downloaded are cryptographically signed to ensure integrity.

The OCM tags each OCF with one of three directives: *load, advertise*, or *run*. An OCF load directive simply causes the router to retrieve the files from the file repository. An OCF advertise directive additionally forces the generation of property values and their issuance as SAs once the OCF has finished loading. In addition to the steps required by the loading and advertising states, an OCF number tagged with the run directive causes routers to run the associated routing algorithms to generate forwarding tables. The resulting forwarding tables, along with the specified packet filters, are installed into the router.

The list of OCFs can change from one OCM to the next. In fact, this is precisely how the Operator introduces new OCFs into the system, and prepares the network to run them. A careful Operator will iterate a particular OCF through the loading and advertising stages, ensuring the expected operation before switching it to the run state. Old OCFs that are removed in succeeding OCMs are purged from the system, along with any algorithm files they use.

B. OCF 0

There is one OCF that is considered immutable—OCF 0. All entities must advertise OCF 0 properties regardless of what other OCFs are loaded or which one is running. This OCF contains the minimum amount of information necessary to maintain routing functionality, and is always operational.

An OCF 0 forwarding table is always built using SPF [28] (Dijkstra's algorithm [16]) with hop count as its metric. FIRE control traffic is always sent using this forwarding table. By ensuring control traffic is forwarded using a straightforward, reliable routing table, OCFs and their associated files can be downloaded regardless of the state of (in)operation of the currently running OCF.

OCF 0 contains exactly four properties:

FIRE Metric: The hop count for this entity used to build the SPF routing table for FIRE management traffic.

IP Addresses: The set of IP addresses associated with this entity. For links, this is the set of IP addresses associated with the destination interface. For routers, this includes any stub hosts that are reachable through this node. Networks do not participate in this property.

FIRE Up: This boolean value states that FIRE is currently running for this node or link. If set to false, no traffic is routed through this entity.

OCFs Loaded: A list of the OCF numbers (other than zero) for which this entity is flooding SAs. Only router nodes participate in this property.

In addition to being used by the OCF 0 routing algorithm, these properties are also available to routing algorithms running at any other OCF number.

IV. Algorithms

After an algorithm's support files are downloaded, the code is loaded into an execution environment on the router. Our FIRE implementation invokes algorithms in a Java Virtual Machine (JVM). If the algorithm is being asked to generate multiple forwarding tables based on different properties, a separate JVM (each with the same algorithm code) is created for each instance. Because we expect SPF to be a frequently used algorithm, in addition to being an integral part of OCF 0 routing for FIRE packets, SPF is implemented as a built-in function rather than a downloadable applet.

A. *Programming interface*

The FIRE algorithm programming interface is intentionally very simple. Whenever new information is inserted into the property repository by the reception of an SA, a snapshot of the repository is made and the routing algorithm invoked. The algorithm's job is to generate a forwarding table from the snapshot of the repository.

The programming interface does not explicitly support incremental updates. Our reasoning is that requiring algorithms to support incremental updates is both unreasonable (some algorithms may not have a straightforward way to do incremental updates) and, as an added complexity, subjects them to additional bugs. However, FIRE does permit the JVM to preserve state across algorithm invocations, so programmers are free to perform incremental updates if they wish. The programming interface is discussed in more detail in section VIII-A.

B. *Algorithm frequency*

When determining how frequently to run an algorithm, the key issue is correctness of routing. The fundamental idea behind link-state routing is that if everyone has the same information (the flooding protocol ensures information at all nodes will converge) and runs the same algorithms, they will get the same results and routing tables will be consistent. Obviously the sooner one runs algorithms in response to updates, the faster the convergence and the less likely that routing loops or black holes will occur. In our view, loop freedom and black hole avoidance is vital to proper operation, so FIRE runs algorithms whenever new information arrives.

C. *Thrashing*

Given FIRE's predilection for invoking routing algorithms, it becomes important to protect against thrashing, where any new piece of information could cause an algorithm to run. FIRE tries to avoid thrashing in three ways:

First, algorithms cannot be stopped in mid-run. They run to completion with the property snapshot they have, and if an update is received while running, the algorithms are simply invoked again as soon as they complete. So FIRE algorithms are guaranteed to generate forwarding tables, regardless of the rate of incoming property updates.

Second, FIRE dallies slightly before invoking an algorithm. Rather than starting up each algorithm as soon as one new or updated SA arrives, FIRE waits a brief, configurable period (usually a few seconds) to allow additional new information to arrive, since routing updates tend to come in bursts. Indeed, conventional wisdom holds that routing protocol traffic tends to be either very heavy (lots of new SAs) or very light (very few SAs) at any given moment.[1] Dallying tries to ensure that during periods of heavy traffic, algorithm runs balance responsiveness with efficiency.

The final protection is not on algorithms themselves, but instead results from FIRE's limits on SA frequency. Since SAs can only be issued at a specified maximum rate (which is enforced by neighboring routers as part of the flooding protocol), a particular node cannot trigger system-wide algorithm runs too frequently. This mechanism is discussed in more detail in section VI-B.

V. PROPERTIES

Any information needed by routing algorithms to construct forwarding tables must be distributed throughout the network. FIRE packages this information into typed values called *properties*. Properties differ from metrics used in traditional routing algorithms. Metrics are weights, assigned to a link, that influence the link's likelihood of inclusion in forwarding paths. Properties, on the other hand, are applicable not only to links, but to routers and networks as well.

The OCF defines the set of properties that a node or link must advertise. Some of these properties will make sense for both nodes and links, some will make sense only for nodes or links, and some will make sense for only some nodes or some links. The OCF's grammar allows the Operator to specify the class of entity that should participate in advertising a particular property. If a participating entity is unable to generate a value for a particular property (perhaps it does not have the required hardware to support routing based on that property), the property's value can be set to *unsupported*.

Links are abstractions and cannot themselves issue SAs. The same is true for network nodes. The node that is the originating endpoint of a link is responsible for advertising the link. For network nodes, the Designated Router undertakes the responsibility of advertising for the network node and, in addition, for the links going from the network node to adjacent nodes as well. This implies that all network properties must either be statically configured or able to be measured by an adjacent router.

A property can be generated by (a) using a configured value, (b) obtaining information from the router's MIBs, or (c) running a property applet. Configured values and MIBs are assumed to be in place prior to the circulation of an OCF containing algorithms that rely on these values.

A. *Property applets*

The ability to generate dynamic properties is one of the most powerful aspects of FIRE, as well as its most dangerous. FIRE borrows from the Active Networks [43] philosophy, allowing downloaded code to be executed on the router itself. Unlike algorithms, however, which simply compute a function over the provided property repository, property applets[2] need access to a far larger set of capabilities, possibly including file and network access. Clearly this represents a potential security risk. In addition to requiring all downloaded code to be cryptographically signed by a software authority, our FIRE implementation uses Java's security infrastructure [20] to provide a balance between code security and applet functionality. The FIRE specification, however, allows for implementations to provide support for additional execution environments. The large body of work on

[1] We have not found a careful study that discusses this behavior within an autonomous system. Chinoy's study [12] of backbone advertisements supports the idea that updates are bursty.

[2] Note that neither routing algorithms nor property applets are *applets* in the strict Java sense of inheriting from the *java.applet.Applet* class.

Proof-Carrying Code [32] could be leveraged for installations with particularly tight security constraints.

Regardless of language or execution environment, property applets are provided sufficient security permissions to interact with the router and any directly connected links. Since routers need only advertise properties related to themselves, adjacent links, or directly-connected networks (in the case of Designated Routers), applets have no need for multi-hop communication. Similarly, they are not provided with any network-level services that would require multi-hop communication by the router, such as name resolution. Other than these basic restrictions, applets are allowed to execute arbitrary Java instructions. It is left up to the software signing authority to ensure that approved property applets function appropriately for use in a production environment.

B. *Property updates*

Scheduling property applets is another difficult task. The operator schedules applets to be run at specified intervals in the OCF. Applets must be run at discrete times, however, and the processes the applets are trying to capture may not be discrete in nature. Furthermore, even if applets were run continuously, some control must be placed on the advertisement of new values. FIRE cannot, in general, determine when properties have changed materially, since property values and types are arbitrary. Therefore property applets themselves are charged with notifying FIRE when their generated values have changed, and to what degree. For instance, noisy properties such as CPU utilization or queue length may be too variable to advertise at each update; instead the applet may chose to advertise only significant deviations from recent history.

FIRE uses the applet notification mechanism to determine when to issue new SAs. In the absence of explicit notification, FIRE issues new SAs periodically, at some configurable interval, usually on the order of tens of minutes. If, however, one or more property values have changed recently, FIRE will schedule new SAs to be issued at a rate commensurate with the configured maximum SA rate. If an applet has indicated that a property value has changed dramatically enough to warrant immediate notification, an SA is issued immediately, subject to the flapping rules discussed in section VI-B.3.

VI. Peering & state distribution

Every routing protocol needs a mechanism to discover adjacent routers, termed *neighbors* or *peers*, and to reliably distribute state information to all other routers in the system. Developing secure, robust mechanisms to support these functions can be quite difficult. Many previous routing protocols have been plagued by limited functionality, such as neighbor discovery algorithms that do not support uni-directional links [30], or buggy implementations [38].

Because of the difficulties in implementing state distribution and peering protocols, we decided not to make these functions programmable. Rather, FIRE fixes these essential mechanisms as built-in (non-programmable) infrastructure. Routing algorithms running on top of FIRE need not concern themselves with the subtle details involved in convergent, soft-state distribution.

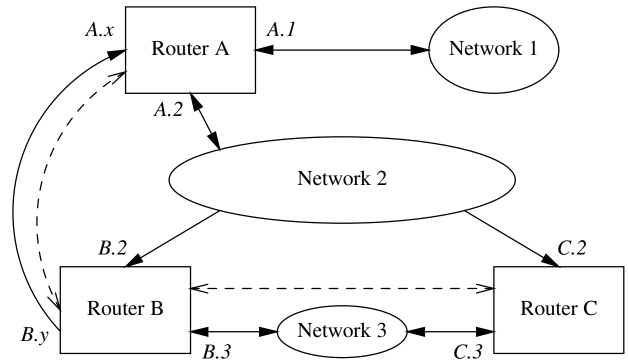

Fig. 2. An example FIRE network model. Solid arrows represent directed adjacencies; dashed arrows indicate router peering relationships.

FIRE builds on a considerable body of prior work [25], [30] to provide mechanisms that are secure, efficient, and robust.

A. *Peering*

Topologically, FIRE models a network as a mesh of *nodes* connected by *links*. Besides all participating routers in the network, broadcast subnets are also treated as nodes, as in OSPF [30], thereby reducing the number of links between routers on a broadcast subnet from $O(n^2)$ to $O(n)$.

Unlike OSPF, however, FIRE explicitly supports uni-directional links. Links are defined as uni-directional, so a bi-directionally connected pair of nodes has two links between them, one in each direction. Two nodes are considered neighbors if some combination of adjacencies (that does not pass through another router node) supports bi-directional communication between the two nodes—a two-way link is not required.

A.1 Neighbor discovery

Neighbor discovery in FIRE is handled by the Peering Protocol [41], which is based largely upon OSPF's Hello Protocol [30]. Fig. 2 depicts the peering relationships formed in a sample internetwork containing six nodes: three routers and three broadcast networks. The solid arrows represent directed adjacencies. Since FIRE models links as uni-directional, there are two links between Router A and Network 1, as opposed to the single, directed link from Router B to Router A. The links between Router A and Network 2, Router B and Network 3, and Network 3 and Router C are similarly represented as two separate links. Suppose Networks 1 and 3 were Ethernets, and Network 2 was a satellite network. In this example Router A has the only uplink to Network 2, while all routers have downlinks. The dashed lines depict the peering relationships that would be established in this topology. Routers B and C peer in the standard fashion across Network 3. Routers A and B also form a peering relationship, even though they must use two separate physical links in order to communicate. In contrast, Routers A and C do not peer, as no single-hop path exists from C to A.

A.2 Designated Routers

In addition to identifying neighbors, the Peering Protocol also selects a *Designated Router* (DR) and *Backup Designated Router* (BDR) for each broadcast subnet. Designated Routers serve two purposes. First, they are responsible for issuing property updates for the subnet and its associated links. Second, they help to limit the number of peering relationships established over a particular subnet. Routers on a broadcast subnet peer only with the DR for the network, rather than having to peer with all routers on the subnet.

Routers functioning as DRs in FIRE must have a bidirectional connection (possibly using two separate interfaces) to the network they are representing. In the case of wireless broadcast networks, there may be no guarantee that a single node can communicate with all other FIRE nodes on the wireless subnet. If there exists no node that can reliably broadcast to all the others, the wireless network must be modeled (for the purposes of FIRE peering) as a mesh of point-to-point links, rather than as a broadcast medium. The same is true for non-broadcast multi-access networks (NBMA) networks.

To prevent flapping of the DR (and hence repeated re-issuance of the network properties for which the DR is responsible), the Peering Protocol also elects a Backup Designated Router. If the DR ever fails, the BDR assumes the role of the DR, and a new BDR is elected. If two DRs (or BDRs) are ever present on a single subnet due to network healing, one of the candidates is selected based upon election rules very similar to those of OSPF.

B. State distribution

Once a peering relationship has been established between two or more neighboring routers, they begin exchanging routing information through the use of SAs. These SAs are reliably flooded throughout the network, thereby providing a robust, convergent method of distributing shared state across the network.

B.1 State Advertisements

SAs contain information about the AS in which FIRE is running. There are four types of SAs: configuration, certificate, external route, and property. *Configuration SAs*, also known as Operator Configuration Messages (OCMs), are generated by the operator and are detailed in section III. *Certificate SAs*, as discussed in section VII, are used to distribute public keys and authority certifications. Routes managed by protocols other than FIRE (such as exterior routes) are advertised through the use of *External Route SAs*.

Property SAs advertise an entity's metrics for a particular OCF. Each node and link has an associated property SA for each OCF being advertised. The particular properties listed are determined by the OCF and the type of the entity being advertised. The payload of a property SA is a self-parsing S-expression containing values for each property. Special flag values are available to define a property as being unsupported, or that a particular entity is a non-participant.

B.2 SA refreshment

An SA is uniquely identified by its type, the entity it describes, and its OCF number. Every SA is also timestamped and has a sequence number, so SAs with the same type, entity identifier, and OCF values can be ordered. In addition to being timestamped, each SA is also given an expiration time, after which it is considered invalid, and is no longer flooded by FIRE. To prevent instability caused by expiring SAs, routers periodically renew SAs they have generated before the previous version expires. To reissue an SA, the router first replaces the superseded SA in its own SA cache with the newly generated version, and then floods the new SA to each of its neighbors.

B.3 Damping

Whenever an attached interface comes up, the router must advertise its existence by issuing SAs. In order to prevent *flapping*, the rapid re-issuance of SAs for the same entity, we utilize the Skeptic model from Autonet [37]. When a property update warrants a new SA, the Skeptic delays slightly before issuing the SA. It not only limits the rate of SA issuance to a fixed maximum rate, but penalizes rapidly changing SAs by exponentially increasing the delay with each new request. As request frequency decreases, the Skeptic reduces the delay penalty accordingly.

However, if a router determines an associated link has gone down and was previously advertised as being up, it immediately generates a new SA for that link indicating the link has gone down. This insures inoperable links are always eliminated from the topology.

C. Reliable flooding

Each FIRE router maintains a cache of all current SAs. The purpose of the state distribution mechanism is to maintain a consistent shared view of the set of current SAs across all routing nodes. Reliable flooding is employed to make sure that an SA generated by one node is eventually received by all nodes in the AS. Stated simply, each router is responsible for forwarding any new SA to all of its neighbors. To prevent SAs from being unnecessarily flooded to neighbors that have already indicated they have a copy, either by preemptively acknowledging it, or by actually sending the SA itself, routers maintain state for each neighbor associated with every SA in its cache.

C.1 State message transmission

Information is exchanged between FIRE nodes using State Messages [36]. A State Message contains either an SA or an acknowledgment of the receipt of one or more SAs. State Messages are flow controlled using a simple windowing protocol, where the transmission window is specified in terms of the number of SAs that may be transmitted without acknowledgment.

C.2 State message processing

Upon receipt of a State Message, a router first looks into its cache to see if it has already been received. If so, it simply acknowledges the message and completes processing. If, on the other hand, the message contains an SA the router has not seen before, it firsts validates that the SA header is properly formed. If the header is invalid for whatever reason, the router immediately sends an acknowledgment to the sender, echoing back the header information.

Routers receiving acknowledgments compare the enclosed header with outstanding transmissions. If the SA was damaged in transmission, the headers will not match, and the SA will be retransmitted as if unacknowledged. If, however, the corruption occurred in the sender itself (hence any retransmissions would be equally useless), the acknowledgment will cause the offending router to cease its transmissions of the corrupted SA.

Assuming a received SA has a well-formed header, a router then verifies the signature before sending an acknowledgement, caching the SA, and flooding the SA to its neighbors. In the case of a non-DR router on a broadcast subnet, it has only one neighbor on that subnet: the DR. Only the DR floods SAs to every member of the subnet.

D. Synchronization

Reliable Flooding distributes SAs to all routers currently connected to the network. When new routers come up, however, or disjoint portions of the network are reconnected, the SA repositories must be resynchronized. This synchronization is done through the *State Dump* process.

The State Dump process is initiated whenever a new adjacency is formed. Since the adjacency may form asynchronously due to the beaconing process of the Peering Protocol, the start of the dump is delayed for some period or until the adjacent router initiates it. The State Dump procedure seeks to swiftly synchronize two neighbors while generating the near minimum amount of traffic. No special message types are used by the process; it sends only SAs and acknowledgments. This allows the standard reliable flooding mechanisms to ensure any new SAs discovered during the State Dump process are flooded appropriately.

VII. Security

Implementing a security infrastructure for a routing protocol presents an interesting problem. To provide most security services, one needs a key infrastructure. But generic key infrastructures generally require pre-existing packet routing functionality. Our solution is to implement the security infrastructure that FIRE requires within FIRE itself.

A. Attacks

A routing protocol, FIRE in particular, is subject to attacks of two basic types:

Wiretapping: Attackers are assumed to have access to the communications links in such a way that they may modify, suppress, insert, or replay FIRE messages or fragments. FIRE must therefore protect against such attacks. Insertion of bogus fragments could prevent a re-assembled message from being accepted at the destination; replaying old messages might disrupt current activities (such as electing a DR); flooding a router with bogus FIRE messages or tampering with data in FIRE messages could result in a denial of service.

Subversion: The routers and other FIRE nodes may be subverted, either physically such that an attacker completely controls a router's behavior or by use of compromised keying material such that an attacker may originate messages that appear to come from the router. A subverted node could send out inaccurate data, possibly affecting a much larger portion of the network.

Many of the harmful behaviors described could also occur due to bugs in software or hardware. Thus, FIRE's design expects that a certain amount of misbehavior (intentional or not) will occur. FIRE's security model is built on a philosophy of containment: our goal is to bound the effects of misbehavior and detect the misbehavior whenever possible.

FIRE meets this goal using three sets of mechanisms. First, FIRE employs a certificate infrastructure with a tiered authority structure. These certificates advertise the public keys of nodes, links and entities such as the Operator. All SAs are digitally signed to provide end-to-end authentication and integrity. Second, FIRE makes use of IPsec [24] to protect against certain hop-by-hop attacks that end-to-end security measures cannot prevent. IPsec's authentication, data integrity, and anti-replay services make it very difficult for a non-participating entity to inject FIRE traffic. Third, and most importantly, the FIRE protocols are designed to be robust against the failure or subversion of an individual router or set of routers.

B. Certificates and digital signatures

FIRE's basic security mechanisms are based on public key cryptography, using X.509 [1] certificates, and patterned after the existing work on Secure OSPF [31] and Secure BGP [25]. Each FIRE node participates in a certificate hierarchy that is managed by the FIRE system. At the top of the hierarchy is the Root. Each separate FIRE instance has its own certificate hierarchy, therefore different ASs will have unique Roots. Routing information shared at border routers must be secured by the exterior gateway protocol.

The Root creates certificates for a set of principals that are entitled to perform various actions. These principals include the Operator and a Software Master. The Operator is the logical entity that runs the network. The Software Master is the logical entity that approves algorithm and property applets (the choice of which approved programs are used is left to the Operator). Below the Operator sit the FIRE nodes themselves. Each node has its own public-private key pair and certificate. FIRE certificates are circulated as part of the normal FIRE flooding protocols in special Certificate SAs. Each node is responsible for flooding its own certificates.

Each SA is signed by its creator. Thus, even though almost all FIRE messages are relayed through other nodes, messages are protected from tampering in-flight. Because each message is unambiguously linked to its signer, advertising of false information is limited and can be traced. A node can only lie about the information it is entitled to advertise.

C. Protocol robustness

Some security features are implemented by the FIRE protocols themselves. Most notably, FIRE implements reliable flooding. Reliable flooding ensures that if one uncompromised path exists between a creator of a message and a consumer, the consumer will eventually see the message. Reliable flooding achieves this guarantee by (a) allocating buffering in each node on a peer-by-peer basis, so no one peer can consume all the buffering in a node and cause a Denial of Service attack that prevents a message from being relayed; and (b) flooding every message over all links.

Unlike many previous routing protocols, FIRE's flooding mechanism does not utilize any explicit request messages. The avoidance of such *pull* mechanisms prevents Denial of Service attacks launched by malevolent routers that continue to request already-received information from adjacent routers.

FIRE also limits the spread of disinformation. The information that a node may advertise is limited by a signed set of permissions contained in an X.509 attribute certificate. Thus, particular routers can be restricted to advertising links only to adjacent networks, preventing a subverted router from being able to black hole arbitrary traffic.

Other FIRE protocols also have protective mechanisms. LDTP has mechanisms to protect against attacks similar to SYN-flooding, where an attacker creates multiple partial sessions that tie up resources at the LDTP server. Similarly, the Peering Protocol contains features that make it difficult for routers to cause the DR to change unless the existing DR goes down.

VIII. Programming interfaces

One of the key features of FIRE is the ability to dynamically load new routing algorithms and to define new properties using applets. Ideally, FIRE implementations would use a language designed to support self-proving applets (programs that could be verified to meet certain constraints). By making it possible to verify (within limits) how a program behaved, the risk of software bugs could be dramatically reduced. Our reference implementation, however, uses Java, both because the self-proving languages were not mature enough (not one had a stable virtual machine that could be embedded into the FIRE implementation) and because Java's popularity reduces the learning curve for FIRE algorithm and applet developers.

The choice of Java, however, forced us to pay more attention to the design of FIRE's programming interfaces. Writing graph algorithms is a notoriously difficult problem—bugs are common. This difficulty is compounded by Java's support for a range of complex language features such as concurrency and event handling, which also tend to produce programmer bugs. FIRE's programming interface is deliberately simple, in an effort to encourage a straightforward programming style.

A. Routing algorithm interfaces

A routing algorithm is provided with two separate programming interfaces: the algorithm interface and the forwarding table API. Every algorithm must implement a specific Java public interface containing exactly three functions: *init()*, *run()*, and *cleanup()*. The algorithm is invoked by FIRE through calls to this interface.

After an algorithm's code files are loaded, a separate instance of the JVM is created for each invocation of the algorithm and the *init()* function is called with an array of strings specified in the OCF. The argument strings may be used to pass arbitrary operator-specified values into the algorithm (similar to *C*'s `argv`), such as threshold values. The role of *init()* is to do any initialization required and prepare the environment for forwarding table generation.

FIRE periodically invokes the algorithm's *run()* method on a snapshot of the repository for the relevant OCF. Table I shows a sample property repository for the network in fig. 2. The repository contains two properties the operator has selected for OCF 1, in addition to the OCF 0 properties. Adjacencies can be deduced from FIRE IDs (FIDs), as can the type of entity. Entities not participating in a particular property are denoted as *NP*.

TABLE I
A Property Repository

	OCF 0				OCF 1	
FID	FIRE Metric	IP Addresses	FIRE Up	OCFs Loaded	Delay (ms)	Drop %
A	1	A.1, A.2, A.x	*true*	1	*NP*	2%
A → 1	0	A.1	*true*	*NP*	2	0%
1	0	*NP*	*true*	*NP*	*NP*	*NP*
1 → A	0		*true*	*NP*	2	0%
A → 2	0	A.2	*true*	*NP*	250	*unsup*
2	0	*NP*	*true*	*NP*	*NP*	*NP*
2 → A	0		*true*	*NP*	250	3%
2 → B	0		*true*	*NP*	250	3%
2 → C	0		*true*	*NP*	250	3%
B	1	B.2, B.3, B.y	*true*	1	*NP*	0.5%
B → A	0	B.y	*true*	*NP*	40	1%
B → 3	0	B.3	*true*	*NP*	1	0%
3	0	*NP*	*true*	*NP*	*NP*	*NP*
3 → B	0		*true*	*NP*	1	0%
3 → C	0		*true*	*NP*	5	0%
C	1	C.2, C.3	*true*	1	*NP*	0%
C → 3	0	C.3	*true*	*NP*	5	0%

An algorithm's implementation has built into it property names to use as input. The OCF specifies a mapping from currently advertised property names to the names required by the routing algorithm. OCF 0, for example, maps *FIRE Metric* to the name expected by the built-in SPF algorithm in order to compute routing tables for FLINT traffic. If the operator desires to run the algorithm on a different property in another OCF, a new name mapping would be used. If, on the other hand, more sophisticated preprocessing is desired, such as EIGRP's cost function, a wrapper function can be employed to read the existing repository and rewrite the desired metrics into the appropriate property, before passing the repository on to the standard routing algorithm.

At the end of a run, a routing algorithm must generate a forwarding table using the provided forwarding table API. Again the interface is deliberately simple, providing only four functions: *update_entry()*, *delete_entry()*, *purge_table()*, and *do_updates()*. The forwarding table is created before the JVM is invoked, so the table is always present (no initialization is required). Furthermore, the interface does not distinguish between modifying and adding an entry; the algorithm simply states that it needs the specified entry by calling *update_entry()*, and the API will either modify the existing entry or add a new one as required. All updates are batched for efficiency, and changes to the forwarding table are only made when the *do_updates()* method is called.

B. Property applet interface

The interface for property applets is similar to that for algorithms. FIRE calls *init()* when the property is first specified by the current OCF and passes in the argument string from the OCF. As with routing algorithms, when the property applet is no longer in use, FIRE terminates it by calling *cleanup()* before shutting down the enclosing JVM.

Unlike algorithms, which respond to changes, the goal of

```
public class SA_update
{
  // report data back to FIRE
  public static native void report_data (
                                Object value);

  // tell FIRE it should send an SA in the next cycle
  public static native void value_changed ();

  // tell FIRE to send an SA ASAP
  public static native void force_SA ();
}
```

Fig. 3. Java SA API

property applets is to detect changes. One might imagine this difference would result in very different ways of invoking them. While it would be desirable to allow applets to specify a complex, event-driven mechanism to trigger themselves, such an implementation proved far too complex and mistake-prone. Instead, the applet invocation interface is essentially identical to that of algorithms; an applet is simply invoked periodically by calling *run()*. The timing and frequency of FIRE's calls to *run()* are specified in the OCF.

When an applet is run, it may choose to record an updated value for the property or leave the existing value alone. For especially noisy properties, it may be desirable to squelch the values within the applet itself, rather than continually issuing SAs with different values. Furthermore, some changes may be significant while others are not (e.g., a change in measured queueing delay from 50 ms to 49 ms probably isn't very important, but a change from 50 ms to 10 ms likely is). FIRE itself has no idea what changes in values are significant, so the API shown in fig. 3 allows the property applet to provide a notion of significance. The applet can simply record a new value, in which case the new value will be advertised only when the next SA is periodically generated (assuming the value is not updated again before the SA is generated). If the change is significant, however, the applet can call *value_changed()*, indicating that it would be desirable to send an SA with the new property value. Or the applet can indicate with *force_SA()* that the property's value has changed so dramatically that an SA should be sent immediately (subject to SA damping rules).

IX. Applications

In order to validate our claim that FIRE provides a robust and easy-to-use platform for rapid routing algorithm deployment, we have implemented several different routing protocols. First, we present our implementation of SPF, showing that basic routing protocols can be implemented in a straightforward manner using the API provided by FIRE.

We then present a wrapper function that shows how additional functionality can be added to previously defined routing algorithms in a straightforward manner. Finally, to demonstrate that FIRE provides sufficient capabilities to implement complex routing algorithms, we consider three of the most popular multicast routing algorithms, based on the intuition that multicast algorithms are likely to be more complicated than typical unicast algorithms.

A. SPF

Fig. 4 presents an abridged version of our (unoptimized) implementation of an SPF routing class in FIRE; it dutifully mimics Dijkstra's algorithm as presented in [13]. In the interest of brevity, some details of the algorithm (including most error handling) have been omitted in favor of code that interacts with the FIRE classes. For clarity, this version only builds routes to networks, ignoring routeable addresses on the router interfaces. As discussed in section VIII-A, the class implements the algorithm interface, although in this case the only interesting function is *run()*.

We do not describe the operation of Dijkstra's algorithm here; interested readers can find a careful treatment in [13]. We note only that the straightforward implementation of Dijkstra's algorithm can be considered in two steps: It first initializes an adjacency matrix data structure given a particular source (in this case, the router for whom the table is being constructed) and then conducts a series of relaxation steps to compute a predecessor matrix for the shortest paths from the source.

The *initializeSingleSource()* function performs the matrix construction here. It uses the repository's pair-wise *adjacent()* predicate to construct an initial adjacency matrix.[3] More interesting in this case, however, is the algorithm's interaction with the repository class in order to retrieve the appropriate metrics. The *name2index()* function is used to map the predefined string to the index of the appropriate property as defined by the operator in the OCF. Each entity is then extracted from the repository and stored locally for efficient access, taking note of the network entities (the destinations of interest).

The relaxation step is not presented here, although there is a slight caveat in its implementation. Since entities may by distributing State Advertisements but not (yet) be willing to route traffic (as indicated by the *FIRE Up* property), the algorithm must make sure it does not relax through entities that are unusable. Furthermore, it may be asked to build an SPF table on a metric that not all entities participate in (latency, for example), in which case it should consider relaxing through only those entities participating in that particular property.

Finally, after constructing a predecessor matrix for the shortest paths, the routing algorithm uses the forwarding table API to insert routes for each reachable network entity in the repository. As can be inferred from the call to *purgeTable()*, this SPF implementation does not support dynamic updates—it rebuilds the entire routing table each time. In general, however, an implementation could consult state from previous iterations to optimize the table updates.

A.1 Augmenting existing algorithms

Perhaps more interesting is fig. 5, which shows all the code necessary to implement a policy-based derivative of SPF. By preprocessing the property repository before invoking Dijkstra's algorithm, we are able to remove all adjacencies that do not meet some requirement; in this instance, we do not allow traffic to be routed over entities that cannot meet a minimum bandwidth. We

[3] While such a predicate allows for random access, it imposes an $O(n^2)$ overhead on any algorithm (such as Dijkstra) that expects an adjacency matrix representation, although other, more optimal representations may be available in general.

```
public class SPF implements Algorithm
{
  private FID       _me;
  private int[]     distance;
  private Entity[] node;
  private Vector[] adj;
  private Vector    networks;
  private int       numNodes, metric, addr, fireUp;

  public void init (FID me, String[] args) {
    networks = new Vector(); _me = me; }

  public void run (Repostiory repo) {

    dijkstra(repo, _me);

    // For each network in domain, find its next-
    // hop and add it to forwarding table
    ForwardingTable.purgeTable();
    for (int i = 0; i < networks.size(); i++) {
      Entity n = (Entity) networks.elemAt(i);
      Entity nHop = nextHop(node.fid);
      if(nHop != null)
        ForwardingTable.addEntry(n.fid.get_locaddr(),
          n.fid.locaddr_cidr_masklen(), nHop.fid);
    }
    ForwardingTable.doUpdates();
  }

  public void cleanup () { }

  // Dijkstra's algorithm from CLR [13, p. 527]
  public void dijkstra (Repostiory repo,
                        FID source) {

    initializeSingleSource(repo, source);
    for (int i = 0; i < numNodes; i++) {
      int u = extractMin(); // closest node
      for (int v = 0; v < adj[u].size(); v++)
        relax(u,((Integer)adj[u].elemAt(v)).intV());
    }
  }

  // Initialize the working state for Dijkstra
  public void initializeSingleSource (
              Repostiory repo, FID source) {

    metric = repo.name2index("metric");
    addr   = repo.name2index("address");
    fireUp = repo.name2index("up");

    numNodes = repo.entities.length;
    networks.removeAllElements();

    distance = new int [ numNodes ];
    node     = new Entity [ numNodes ];

    for (int v = 0; v < numNodes; v++) {
      node[v] = repo.entities [v];
      if(node [v].fid.equals(source)) {
        distance [v] = 0;
      } else distance [v] = Integer.MAX_VALUE;

      if (repo.entities [v].fid.network())
        networks.addElement(repo.entities [v]);

      ...Build adjacency matrix adj[0...numNodes] [...] using the
        Entity.adjacent(FID) predicate...
    }

  // Edge relaxation over participating, up vertices
  private void relax (int src, int dst) {...}
}
```

Fig. 4. Abridged FIRE SPF Implementation. Function declarations and FIRE-provided classes are shown in **bold**, elided code is denoted by "..."

```
public class BaudRateGuarantee extends SPF
                                implements Algorithm
{
  private int min_baud;

  public void init (FID me, String[] args) {
    super.init(me, args);
    min_baud = Integer.valueOf(args[0]).intValue();
  }

  public void run (Repository repo) {

    int baud_index, up_index;
    baud_index = repo.name2index("baudrate");
    up_index   = repo.name2index("up");

    // down entities whose baud rate is too low
    for (int i = 0; i < repo.entities.length; i++) {
      Entity e   = repo.entities[i];
      Value baud = e.values[baud_index];
      Value up   = e.values[up_index];
      if(((Integer)baud.value).intValue() < min_baud)
        up.value = new Boolean(false);
    }

    // run SPF over the modified repository
    super.run(repo);
  }
}
```

Fig. 5. A FIRE routing class that provides a baud rate guarantee. Error checking has been removed due to space constraints.

note, however, that this simplistic high-pass filtering mechanism does not imply that the remaining entities could actually provide such a guarantee; doing so would require interfacing with a resource reservation mechanism [9] on the routers, which is outside the scope of this work. FIRE could, however, help enumerate the candidate paths that a resource reservation mechanism should explore while attempting to secure resources between a particular source/destination pair.

We find this filtering paradigm particularly applicable to policy-based routing constraints. For instance, similar extensions can be written to ensure packets intended for a private audience are never transmitted over wireless or bridged networks or shared links by exporting the appropriate properties and preprocessing the repository in a manner similar to that shown above.

B. Multicast

Multicast routing protocols contain some of the most complicated routing algorithms currently in use today. We examine how three different classes of algorithms—dense mode, sparse mode, and EXPRESS [23]—could be implemented in FIRE.

We note that unlike unicast operation, when all hosts implicitly request service by attaching to the network, the goal of multicast (as opposed to broadcast) protocols is to deliver content only to those end hosts explicitly requesting service. This requires a signalling mechanism between end hosts and routers. Most multicast protocols in use today use the Internet Group Membership Protocol (IGMP) [19] for this purpose, although some have developed their own extended protocols as well [23]. These host/router signalling mechanisms are orthogonal to the operation of the routing algorithm itself, hence they are not discussed here. Clearly for a FIRE router to support any type of multicast routing, it would have to implement the appropriate membership protocol as well. For purposes of discussion, we assume such an implementation could store state accessible to a property applet (perhaps through a MIB).

B.1 Dense Mode

Most of the original multicast routing algorithms, Reverse Path Forwarding (RPF) and its variants, Truncated Reverse Path Forwarding (TRPF) and Reverse Path Multicasting (RPM), are best suited for domains with a large number of members in each multicast group. These algorithms are implemented in both distance-vector (Distance Vector Multicast Routing Protocol (DVMRP) [46] and Protocol Independent Multicast-Sparse Mode (PIM-SM) [18]) and link-state (Multicast-OSPF (MOSPF) [29]) intra-AS protocols.

The particular link-state variant of RPM found in MOSPF, which builds individual Shortest Path Trees (SPTs) for each group, can be implemented in FIRE directly by building shortest path trees precisely as described in [29]. Similar to OSPF, FIRE already maintains a notion of a Designated Router, hence no additional machinery is necessary to ensure only one router multicasts to each subnet. The information contained in a Group Membership LSA (the groups to which a router is interested in subscribing) could be implemented as an additional *group* property that each router would participate in. Upon receipt of an IGMP join message from an end host, a FIRE router would update its group property to contain the new multicast address. This processing could be done through a property applet that monitored some list of addresses maintained by the IGMP implementation.

One drawback of this implementation is that the rate of subscription updates is governed by the maximum rate of SA issuance within the network. If an end host issues a join message immediately after a router has reached its SA limit for the current time period, it must wait until the damping algorithm allows the router to emit a new SA before it will be grafted to the distribution tree. Furthermore, if the IGMP processing simply queues requests to be serviced by a property applet run by FIRE, an additional lag is imposed until FIRE invokes the multicast property applet. This could be ameliorated by allowing the IGMP implementation to directly invoke specific property applets.

Further, MOSPF builds the shortest path trees "on demand," when the first multicast packet for a group is received. FIRE currently only invokes routing algorithms (tree construction) on SA arrival, not arbitrary packet arrival. Hence for proper operation FIRE should pre-construct shortest path trees for any group a host has registered interest in, regardless of whether traffic is actually flowing or not. In some instances, this may waste router resources on unused multicast addresses. We examine the issues involved in event-driven algorithm and property invocation in detail in section XI-C.

Similar interactions occur with prune messages, although this has less impact on end hosts, as they do not notice the lag as a disruption in service, only as wasted bandwidth. We note, however, that FIRE's link-state implementation obviates the need for explicit pruning employed by distance-vector protocols such as DVMRP's poison reverse messages.

B.2 Sparse Mode

In networks where group membership is small with respect to the number of entities, many implementations of reverse path multicast algorithms become increasingly wasteful, either because they initially flood traffic to new groups throughout the entire network until prune messages are received from uninterested routers [46], or because they must store state for currently unused groups as discussed above. This observation lead to the development of sparse mode algorithms that provide better scalability properties in the wide area. While the link-state nature of FIRE limits the scalability of any algorithm it implements, we examine the viability of a property-based paradigm here, and return to explore extending FIRE to distance vector in section XI-D.

Core-Based Trees (CBT) [8] are one such algorithm. The sparse mode of the Protocol Independent Multicast protocol (PIM-SM) uses Rendezvous Points as cores for its distribution trees. Selecting optimal cores in a metric-based routing protocol is an open problem; current implementations use either a single Bootstrap router (BSR) to select cores for the entire domain (as found in PIM-SM), or manual placement. In a property-based system, however, routers can dynamically advertise the property that they are willing to be cores. Each router might support a *multicast-core* property whose value is a list of multicast groups for which the router is currently a core. The routing algorithm would then perform a distributed computation, evaluating the potential cores for one that was well-placed for the intended set of multicast recipients (since both the potential core information and membership information is flooded throughout the network). While the optimal core location remains difficult to determine, and likely changes with group membership, the availability of global topology and membership information enables a reasonable heuristic, which would be an improvement over current protocols. This computation could be enhanced by having routers advertise how heavily they are loaded (e.g., the number of groups for which they are already serving as cores) and factoring load into the core selection algorithm.

A further optimization implemented by PIM-SM allows routers to switch over to a shortest path tree rooted at the source for higher performance if traffic reaches a certain level. Since FIRE distributes membership information to each router in the network, this decision could be made in a more intelligent fashion, utilizing global information about the topologies of both trees. Note, however, that the rate of SA flooding is not uniform throughout the network, so a race condition could exist if the switch was not implemented properly. If a router suddenly issued an SA joining the shortest path tree and removing itself from the core-based tree, the SA may be propagated to the CBT first, interrupting the delivery of traffic from the initial distribution tree before it resumes from the SPT, causing a disruption of service. Hence a robust implementation would likely belong to both trees for a short while before issuing a new SA resigning from the initial CBT.

B.3 EXPRESS

Recently, Holbrook and Cheriton extended the multicast model to support the notion of channels, which allow for explicit subscription control and provides membership information to the source [23]. Their EXPlicitly REquested Single Source (EXPRESS) protocol implements a reverse path forwarding algorithm to build distribution trees, but extends it to allow multicast sources to specify an authentication key required for group subscription, and to request timely estimates of channel membership. Both of these classes of information can be expressed quite naturally in the FIRE model.

In order to provide channel authentication, each router could participate in a *key* property, where the property was a set of keys associated with the channels originating at that source. Since each key is only distributed by its source, but flooded throughout the network, this presents no additional scaling complications. To communicate subscription information, the group property discussed previously could be extended from a simple boolean value to a counter, which would indicate the number of end hosts subscribed directly to that particular router. By summing over the appropriate property entry for all routers in the network, a source router could compute a reasonable estimate of the current channel membership, although the timeliness of this value would be governed by the rate of SA advertisement in the network.

X. Implementation

We have implemented FIRE as a user-level daemon with supporting kernel modifications for FreeBSD. Multiple FIRE daemons may coexist to support separate FIRE instances on the same router . Each FIRE daemon, however, must be responsible for a disjoint set of traffic. This section discusses some of the more interesting features of the implementation.

A. Forwarding

FIRE places two new requirements on the IP forwarding mechanism. First, it requires every packet to be filtered. Second, it requires support for multiple forwarding tables. Both changes were novel for a BSD kernel.[4]

A.1 Packet filtering

FIRE packet filters are specified using the Berkeley Packet Filter (BPF) [27] syntax, which provides a well-known and highly portable specification language. For performance reasons, we restrict FIRE BPF filters to the IP header. Even with this limitation, however, the BPF kernel implementation is not adequate for filtering at line speed. We chose to implement Dynamic Packet Filtering (DPF) [17] in the FreeBSD kernel with a device driver interface similar to BPF. Our kernel DPF implementation provides multiple filter sets, each of which is individually accessible using standard UNIX file system semantics (and permissions).

One particular filter set is known as the *überfilter*. All IP packets forwarded by the router (by *ip_forward()*) are passed through the überfilter. If no match is found, the packet is forwarded according to the default kernel forwarding table (the default kernel table exits to support legacy applications that need access to the old kernel routing table). If, however, the packet

[4]We note that in-kernel packet filtering itself is not new, and is used for efficient protocol demultiplexing in the Nemesis [26] and Scout [7] operating systems, among others.

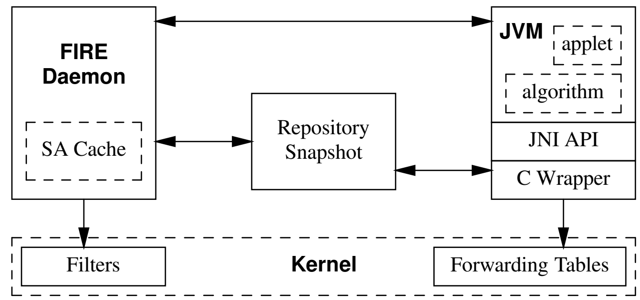

Fig. 6. FIRE Daemon Implementation

matches a filter installed in the überfilter, it is passed through the specified secondary filter set. A matching filter in the secondary set specifies the appropriate forwarding table to use. If no match is found, the packet is dropped. Each FIRE daemon requests its own filter set, and then installs a filter in the überfilter corresponding to the set of traffic it is responsible for, causing DPF to route any matching packets through the daemon's secondary filter set.

A.2 Forwarding tables

Once the correct forwarding table has been selected, FIRE follows the standard IP forwarding mechanism and finds a best matching prefix for the destination address. Due to the additional performance cost of filtering, we replaced the standard BSD radix-trie lookup tables [40] ($O(W)$ performance, where W is the length of an address) with the ETH-WASHU lookup algorithm [47]. This algorithm both reduces the worst-case lookup to $O(\log W)$ and is more space efficient than the BSD algorithm.

B. Sandboxing

Each algorithm or applet is run in its own Java Virtual Machine (JVM), interfacing with FIRE through the use of the provided Java Native Interface (JNI) API discussed in section VIII. The JNI functions are supported by a thin C wrapper that interfaces with both the FIRE daemon and the kernel. Forwarding table updates are communicated directly to the kernel by the wrapper. In the case of algorithms associated with OCFs that are only *advertised*, and not yet *running*, the forwarding table functions are stubbed, and requested updates are logged, but not yet installed into the kernel forwarding tables.

Communicating property values is a bit trickier, however. Since the Property Repository is constantly in a state of flux, a static snapshot is provided to routing algorithms when they are invoked. This snapshot is passed to the JVM through the file system. The C wrapper reads the repository snapshot files and marshals them into appropriate Java structures as required by the algorithm. The FIRE daemon passes control messages to the JVM through a UNIX pipe. Results of property applets are communicated back to the FIRE daemon in a similar fashion. Fig. 6 depicts the interaction of the various parts of our FIRE implementation.

C. Performance

We have performed a preliminary performance analysis on our prototype implementation. Because FIRE is implemented in a FreeBSD kernel rather than a commercial router platform, the forwarding performance results are not particularly interesting. Our implementation forwards packets about 33% slower than stock FreeBSD, mostly due to deficiencies in our prototype code. We believe this performance gap would be almost entirely eliminated in an optimized implementation.

Of more import is the performance of the routing protocol itself, measured in packet exchanges. Our FIRE implementation generates at most 2 packets (ignoring retransmissions due to packet loss) on a subnet for each entity advertised (one packet from the creating entity to the DR, and one multicast packet from the DR to the other nodes). ACKs are bundled together, hence the exact ACK count varies dramatically based on the precise timing of events. At OCF 0, which provides SPF functionality, our message count is a small multiple of OSPF (approximately a factor of 3, since, unlike OSPF, we count links and networks as separate entities). Each additional advertised OCF generates a similar number of packets, resulting in a linear growth in traffic. Note, however, that one OCF may contain properties for generating multiple forwarding tables, hence the incremental cost of additional forwarding tables is only in the size of the messages, not the packet count.

XI. Discussion

FIRE allows for the rapid deployment of novel routing algorithms, and the dynamic reconfiguration of operational networks. In order to provide such a high level of flexibility while maintaining reasonable levels of security, performance, and robustness, several architectural aspects of FIRE impact its applicability to certain classes of routing. We re-examine these design decisions here, and comment on the plausibility of alternative schemes.

A. Event-driven invocation

While FIRE allows property applets great freedom in the type of computation they can perform, the multicast algorithms discussed in section IX-B highlight the additional flexibility that could be gained by allowing applets to be invoked based upon an event-driven model. As discussed in section VIII-B, however, such a mechanism proved difficult to implement in a secure and robust fashion. In general, consuming resources based upon an external event presents an opportunity for Denial of Service attacks. In the particular instance of invoking property applets upon the receipt of a certain type of packet, attackers spoofing the appropriate packet could cause FIRE routers to invoke property applets arbitrarily.

While invocation messages could be authenticated using FIRE's certificate hierarchy, such a mechanism would violate FIRE's security model, which is based on containment. A subverted router could issue properly authenticated messages which would cause adjacent routers to begin invoking property applets at an uncontrolled rate, possibly issuing similar invocation messages to next hop routers, allowing one subverted router to consume resources throughout the entire network.

Related issues arise when a router issues SAs too frequently. FIRE implements a damping mechanism (described in section VI-B.3) that limits the rate of SA generation at any one router. While the Skeptic model could be extended to handle the property invocation messages described above, it has similar pitfalls. The inability to issue SAs at a rapid rate is one of the prime motivations for additional inter-router communication like that described above, hence limiting its rate in a similar fashion is counter productive.

B. Synchronization

In a similar vein, properties are bundled together in a single advertisement that is issued sporadically. An alternative approach, the logical consequence of the previous discussion, is to allow properties to be advertised individually, thereby enabling the update rate of certain volatile properties such as multicast group membership to be decoupled from properties involved in unicast routing.

In the general case, however, it becomes difficult to reason about the freshness of property data for any particular neighbor when each property is updated at its own rate. Associating different rates with individual properties would require multiple, separate damping policies. It seems dubious to assume that the Operator has a sufficient understanding of the various requirements of each routing algorithm to appropriately set the damping threshold for every property individually. Additionally, it is not clear how FIRE could determine when it has a sufficiently fresh version of the link-state database to invoke particular routing algorithms. If routing algorithms depend on multiple properties, invoking an update on each new individual property arrival would result in a rapid increase in computational expense.

Fundamentally, each of these limitations stems from the fact that FIRE specifies a rigid flooding mechanism. As the multicast examples demonstrate, hardwiring state distribution can contribute to less-efficient implementations of certain algorithms. We believe this is only one symptom of a basic tradeoff between security and efficiency, and that the robustness guarantees FIRE's flooding mechanism provides, both in terms of security and protection against misconfiguration, outweigh the performance impact in most cases.

C. Property fidelity

The granularity of FIRE's properties allows not only for routers to advertise their state, but for links to advertise different properties in different directions, and properties of networks distinct from links. As discussed in section X-C, making nodes, both link directions, and networks all first-class entities in the routing system imposes a substantial overhead on the number of routing messages flooded throughout the network. The examples in section IX clearly demonstrate how router properties can be leveraged to ease the implementation of sophisticated algorithms, but the utility of property advertisements for the remaining entities bears closer examination.

C.1 Links

It is clear that links are asymmetric in reality, and it is important to advertise them separately. The upstream and downstream traffic for many stub networks' border routers vary dramatically, hence any properties that attempt to capture attributes of the link that are dependent on traffic need to be handled independently. Even properties that are not traffic related may benefit from the separation. For instance, an operator could annotate links with a conduit property, exposing the physical co-location of various network links.

C.2 Networks

The evidence in support of network properties is somewhat less conclusive, although there are specific instances for which network properties are distinct from adjacent links. For example, each router connected to a heavily-loaded gigabit Ethernet may advertise a link bandwidth of 1Gbps. A routing algorithm searching for a high-bandwidth path might select such a router, only to find that only a fraction of the bandwidth was actually available across the Ethernet. If, on the other hand, the network was advertising a *media* property, the algorithm could recognize that despite the high link speed, the network media is shared, hence the available bandwidth across the network may be substantially less.

D. Distance vector

FIRE is a link-state protocol; an interesting question is whether one could develop a FIRE-like distance-vector protocol. The major difference between distance-vector and link-state protocols is that distance-vector protocols summarize the state information at each hop in a path. So rather than receiving property advertisements of each network entity, a router simply learns the properties of a path to each destination, as seen by its neighbor. In bandwidth-constrained and highly-connected networks (e.g., some wireless networks) distance vector's ability to summarize the data at each hop is an advantage because it limits the spread of topology updates, reducing bandwidth usage.

One approach to distance vector would retain multiple properties and property applets while replacing routing algorithms with *summarization algorithms*, whose job it would be to receive and process each received routing advertisement and generate the new advertisement to transmit. (Presumably different portions of an advertisement would be summarized by different summary applets in order to support multiple properties.)

Unfortunately, we are aware of at least two serious problems with the summarization approach. First, robustness is sacrificed; the failure of a summarization applet means a router either stops advertising information or advertises incorrect information about its neighbors. Second, there is a conflict with FIRE's security model—in FIRE, a node's advertisements are digitally signed by the node. In a distance-vector system, a node's information is only transmitted to its neighbors, which then summarize the information in their advertisements. As a result, in distance vector, a router must trust its neighbor to correctly summarize. Adding a complete verification trail of all changes would effectively undo the summarization process; rather, some short verification must be achieved, which remains an open problem in general.

E. Inter-AS operation

The task facing an inter-AS protocol is even more daunting. Not only does it need to summarize the routes internal to the AS, but an inter-AS variant of FIRE would need to somehow equate its internal, class-based routes to routes advertised by other ASs. Unfortunately, as routing information crosses AS boundaries, it is likely that the traffic classes and advertised properties within each AS will differ, based upon the diverse interests of adjacent ASs. Hence the summarization process would be further complicated by the need to translate or map the traffic classes and advertised properties in each AS to the adjacent ones.

TABLE II
ACTIVE NETWORKING COMPARISON

System	Technique	Layer	Domain	Language
Active Bridging [3]	Device		Bridging	Caml
Router Plugins [15]	Device		Services	C
ANTS [43]	Capsule		General	Java
PLANet [22]	Capsule		General	PLAN
PAN [34]	Capsule		General	Java[5]
Joust [21]	Capsule		General	Java
Smart Packets [39]	Capsule		Management	Sprocket
NetScript [49]	Control	NVN	Management	NetScript
Tempest [45]	Control	ATM	Management	Ariel
FIRE	Control	IP	Routing	Java

XII. RELATED WORK

There has been considerable work over the past decade on supporting different types of service. The focus of the vast majority of this work has been on queueing disciplines such as Fair Queueing [14], and techniques such as Guaranteed Service and Differentiated Services [33] that use these queueing schemes to support different service levels. This work, however, has not examined routing system support. FIRE is, therefore, complementary to these schemes, in that FIRE provides support for routing according to different requirements, but does not specify queueing regimes.

Other routing protocols have sought to be extensible. OSPF [30] allows the definition of new state advertisement messages (this feature was used for both Secure and Multicast OSPF). IS-IS [10] has similarly flexible features. The major distinction between these protocols and FIRE is that both protocols require recoding of existing implementations to generate and use the new information being advertised, while FIRE provides run-time support. FIRE's dynamism also contrasts with Multi-Protocol Label Switching (MPLS) [4], where an operator can manually configure paths through the network for specific classes of traffic. The MPLS solution is static, while FIRE, like most routing protocols, offers a completely dynamic solution.

While FIRE provides the ability to download and execute arbitrary code on production routers, FIRE is not a traditional Active Network—unlike capsule-based techniques [22], [34], [43] user generated traffic cannot affect routing behavior in FIRE. The separation of operator-initiated control flow and forwarded data traffic allows FIRE to sidestep many of the difficult performance and security issues associated with Active Networking, although not all. In particular, FIRE's authentication and authorization scheme addresses similar problems as those dealt with in SANE [2]. In addition, resource isolation between routing algorithms, property applets, and individual FIRE instances is crucial for deployment in production environments. In the particular instance of JVMs, these issues have been addressed recently in the KaffeOS [5].

[5] Native (i386) object code was also explored, but is not platform-independent.

Many aspects of FIRE grow out of its particular choice of domain. Table II presents a taxonomy of Active Networking systems, categorizing each according to its general technique and application domain. For informational purposes we also include the language used to encode the "active" components. We divide the systems into three general classes: those that function via direct interaction with the components (device), those that distribute code in data traffic (capsule), and those that use special control traffic to alter node behavior (control). Note that most systems focusing on network management, configuration, or routing, share FIRE's separation of control and data traffic. By limiting the domain scope, FIRE can provide greater levels of assurance and performance than more general-purpose Active Networking techniques.

For those systems that expose networking control interfaces, we classify them in terms of their layer of operation. NetScript provides a scripting language for constructing an overlay network, called a NetScript Virtual Network (NVN), for handling specified classes of packets. While sufficiently general to perform routing functionality, the design of NetScript requires active processing of packets by NetScript agents to perform specific forwarding tasks. FIRE's avoidance of such per-packet computation provides a fundamental performance advantage.

Indeed, the previous work most closely aligned with FIRE's goals is Tempest [45], which, like FIRE, focuses on so-called "out-of-band" control functions, such as forwarding policies and resource reservation. As an implementation, however, Tempest partitions an ATM switch into multiple virtual *switchlets*, each of which can be configured by different administrators. In many ways, FIRE is a generalization of the Tempest approach, providing enhanced forwarding capabilities at the IP level, where they are most commonly implemented in today's networks.

XIII. Conclusion

FIRE significantly increases the control network operators have over how their network routes traffic. Operators can change routing algorithms and metrics at run time and dynamically configure which traffic classes are forwarded by the various algorithms. By enabling network operators to change the routing algorithms employed by the protocol, FIRE significantly lowers the barrier to deploying new algorithms in an AS.

The fine granularity of FIRE properties allows routing algorithms to make highly-informed decisions. We have demonstrated that FIRE supports the implementation of existing routing algorithms, and provides a straightforward mechanism for extending them to support novel policy-based routing constraints. It remains to be seen, however, whether routing algorithms can be developed to take full advantage of FIRE's property-based routing paradigm.

As opposed to capsule-based Active Networking techniques, which provide general execution environments to mobile code, or restrictive remote router configuration mechanisms, FIRE attempts to strike a balance of flexibility, performance, and security that is appropriate for a dynamic routing infrastructure. By focusing on the particular domain of IP routing, FIRE provides much of the convenience and expressiveness of Active Networking while ensuring the level of robustness and forwarding performance required of a production routing protocol.

Acknowledgements

We are indebted to David Andersen, Robert Morris, Ram Ramanathan, Robert Shirey and the anonymous reviewers for their invaluable comments on previous versions of this paper. Steve Kent provided expert guidance on several security issues. Benjie Chen assisted with the performance measurement of our prototype. Ayan Banerjee, Rahul Biswas, Matt Fredette, Fabrice Tchakountio, Laurie Thompson, and Greg Troxel contributed to the reference FIRE implementation.

References

[1] C. Adams and S. Farrell. Internet x.509 public key infrastructure certificate management protocols. RFC 2510, IETF, Mar. 1999.
[2] D. S. Alexander, W. Arbaugh, A. Keromytis, and J. M. Smith. A secure active network environment architecture. *IEEE Network*, 12(3):37–45, May 1998.
[3] D. S. Alexander, M. Shaw, S. M. Nettles, and J. M. Smith. Active bridging. In *Proc. ACM SIGCOMM '97*, pages 101–111, Sept. 1997.
[4] D. O. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, and J. McManus. Requirements for traffic engineering over MPLS. RFC 2702, IETF, Sept. 1999.
[5] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proc. USENIX OSDI '00*, pages 333–346, Oct. 2000.
[6] S. Bahk and M. E. Zarki. Dynamic multi-path routing and how it compares with other dynamic routing algorithms for high speed wide area network. In *Proc. ACM SIGCOMM '92*, pages 53–64, Aug. 1992.
[7] M. Bailey, B. Gopal, M. Pagels, L. Peterson, and P. Sarkar. Pathfinder: A pattern-based packet classifier. In *Proc. USENIX OSDI '94*, pages 115–123, Nov. 1994.
[8] A. Ballardie, P. Francis, and J. Crowcroft. Core based trees (CBT): An architecture for scalable multicast routing. In *Proc. ACM SIGCOMM '93*, pages 85–95, Sept. 1993.
[9] B. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol (RSVP) — version 1 functional specification. RFC 2205, IETF, Sept. 1997.
[10] R. Callon. Use of OSI IS-IS for routing TCP/IP and dual environments. RFC 1195, IETF, Dec. 1990.
[11] I. Castineyra. Hop-spec: specifying the capabilities of the hop within an internetwork. Technical report BBN-TR-7813, BBN Technologies, 1992.
[12] B. Chinoy. Dynamics of Internet routing information. In *Proc. ACM SIGCOMM '93*, pages 45–52, Sept. 1993.
[13] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.
[14] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *Internetwork: Research and Experience*, 1(1):3–26, Jan. 1990.
[15] D. Descaper, Z. Dittia, G. Parulkar, and B. Plattner. Router plugins: A software architecture for next generation routers. In *Proc. ACM SIGCOMM '98*, pages 229–240, Aug. 1998.
[16] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
[17] D. Engler and M. F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proc. ACM SIGCOMM '96*, pages 53–59, Aug. 1996.
[18] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. Protocol independent multicast-sparse mode (PIM-SM): Protocol specification. RFC 2362, IETF, June 1998.
[19] W. Fenner. Internet group membership protocol, version 2. RFC 2236, IETF, Nov. 1997.
[20] J. Gosling, B. Joy, and G. Steele. *The Java language specification*. Addison Wesley, Reading, Massachusetts, 1996.
[21] J. Hartman, L. Peterson, A. Bavier, P. Bigot, P. Bridges, B. Montz, R. Piltz, T. Proebsting, and O. Spatscheck. Joust: A platform for liquid software. *IEEE Networks*, 12(4):50–56, July 1998.
[22] M. Hicks, J. Moore, D. S. Alexander, C. Gunter, and S. Nettles. PLANet: An active internetwork. In *Proc. IEEE Infocom '99*, pages 1124–1133, Mar. 1999.
[23] H. Holbrook and D. Cheriton. IP multicast channels: EXPRESS support for large-scale single source applications. In *Proc. ACM SIGCOMM '99*, pages 65–79, Sept. 1999.
[24] S. Kent and R. Atkinson. Security architecture for the Internet protocol. RFC 2401, IETF, Nov. 1998.

[25] S. Kent, C. Lynn, and K. Seo. Secure border gateway protocol (S-BGP). *IEEE J. Select. Areas Commun.*, 18(4), Apr. 2000.

[26] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE J. Select. Areas Commun.*, 14(7):1280–1297, Sept. 1996.

[27] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. USENIX Technical Conference*, pages 259–269, Jan. 1993.

[28] J. McQuillan, I. Richer, and E. Rosen. The new routing algorithm for the arpanet. *IEEE Trans. Commun.*, 28(5):711–719, May 1980.

[29] J. Moy. Multicast extensions to OSPF. RFC 1584, IETF, Mar. 1994.

[30] J. Moy. OSPF version 2. RFC 2328, IETF, Apr. 1998.

[31] S. Murphy, M. Badger, and B. Wellington. OSPF with digital signatures. RFC 2154, IETF, June 1997.

[32] G. Necula. Proof-carrying code. In *Proc. ACM POPL '97*, pages 106–119, Jan. 1997.

[33] K. Nichols, V. Jacobson, and L. Zhang. A two-bit differentiated services architecture for the Internet. RFC 2638, IETF, July 1999.

[34] E. Nygren, S. J. Garland, and M. F. Kaashoek. PAN: A high-performance active network node supporting multiple code systems. In *Proc. IEEE OPENARCH '99*, pages 78–89, Mar. 1999.

[35] C. Partridge, I. Castineyra, B. Schwartz, and F. Tchakountio. Large data transfer protocol. Technical memo BBN-TM-1265, BBN Technologies, Nov. 2000.

[36] C. Partridge, I. Castineyra, W. T. Strayer, A. C. Snoeren, and B. Schwartz. FIRE state message protocol specification. Technical memo BBN-TM-1245, BBN Technologies, July 2000.

[37] T. Rodeheffer and M. Schroeder. Automatic reconfiguration in Autonet. In *Proc. ACM SOSP '91*, pages 183–197, Oct. 1991.

[38] E. Rosen. Vulnerabilities of network control protocols: An example. *ACM CCR*, 11(3):10–16, July 1981.

[39] B. Schwartz, A. W. Jackson, W. T. Strayer, W. Zhou, R. D. Rockwell, and C. Partridge. Smart packets for active networks. *IEEE Trans. Comp. Sys.*, 18(1):67–88, Feb. 2000.

[40] K. Sklower. A tree-based routing table for Berkeley Unix. Technical report, University of California, Berkeley, 1993.

[41] A. C. Snoeren, C. Partridge, W. T. Strayer, and I. Castineyra. FIRE peering protocol specification. Technical memo BBN-TM-1244, BBN Technologies, July 2000.

[42] K. Sollins. The TFTP protocol (revision 2). RFC 1350, IETF, July 1992.

[43] D. Tennenhouse and D. Wetherall. Towards an active network architecture. *ACM CCR*, 26(2), Apr. 1996.

[44] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, May 1999.

[45] J. E. van der Merwe, S. Rooney, I. M. Leslie, and S. A. Crosby. The Tempest — a practical framework for network programmability. *IEEE Network*, 12(3):20–28, May 1998.

[46] D. Waitzman, C. Partridge, and S. Deering. Distance vector multicast routing protocol. RFC 1075, IETF, Nov. 1988.

[47] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed IP routing lookups. In *Proc. ACM SIGCOMM '97*, pages 25–36, Sept. 1997.

[48] R. White, A. Retana, and D. Slice. *EIGRP for IP*. Addison Wesley, Reading, Massachusetts.

[49] Y. Yemini and S. da Silva. Towards programmable networks. In *Proc. IFIP/IEEE Intl. Work. on Dist. Systems: Operations and Management*, Oct. 1996.

Craig Partridge (Fellow) is a Chief Scientist with BBN Technologies, Cambridge, MA, where he does research on various aspects of internetworking. He is chair of ACM SIGCOMM and the former editor in chief of IEEE Network Magazine and ACM Computer Communication Review. He is co-consulting editor of the Addison Wesley Professional Computing Series. He received the A.B., M.S., and Ph.D. degrees from Harvard University, Cambridge, MA.

Alex C. Snoeren (Student Member) received the B.S. degrees in computer science and applied mathematics from the Georgia Institute of Technology, Atlanta, GA, in 1996 and 1997, respectively, and the M.S. degree in computer science in 1997. He is currently a Ph.D. candidate in the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology, Cambridge, MA. He is also with BBN Technologies, Cambridge, MA, as a Scientist in the Internetworking Research Department. His research interests include systems, networking, and mobile computing.

W. Timothy Strayer (Senior Member) received the B.S. degree in mathematics and computer science from Emory University, Atlanta, GA, in 1985 and the M.S. and Ph.D. degrees in computer science from the University of Virginia, Charlottesville, VA, in 1988 and 1992, respectively. He joined BBN Technologies, Cambridge, MA, in 1997, where he is a Senior Scientist in the Internetworking Research Department. His research interests include transport protocols, active networks, satellite packet switching, virtual private networks, and routing systems.

Beverly Schwartz is a Scientist at BBN Technologies, Cambridge, MA, in the Internetworking Research Department where she works on applying Active Networking technology to network management and routing protocols. She received the B.S. degree in electrical engineering from Tufts University, Somerville, MA, in 1985, and the M.S. degree in computer science from Harvard University, Cambridge, MA, in 1989. She designed and implemented much of the state distribution mechanism in FIRE.

Matthew Condell received the B.S. and M.Eng. degrees in computer science from the Massachusetts Institute of Technology, Cambridge, MA in 1996. He joined BBN Technologies, Cambridge, MA in 1996 where he is a Scientist in the Internetworking Research Department. His interests include network security, active networking, and policy.

Isidro Castiñeyra received the M.Sc. and Ph.D. degrees in electrical engineering from the Massachusetts Institute of Technology, Cambridge, MA. The work described in this paper was conducted while he was Division Scientist in the Internetwork Research department of BBN Technologies, Cambridge, MA. He is presently with Pluris, Cupertino, CA. His research interests center around system resource management and routing. He has conducted research on routing for Quality of Service (QoS) applications; on scalable, secure, mobile communication; and on scalable routing architectures for QoS (NIMROD).