# Towards a Declarative Language and System for Secure Networking

Martín Abadi*† and Boon Thau Loo‡*
*Microsoft Research †UC Santa Cruz ‡University of Pennsylvania

***Abstract***

*In this paper, we present a declarative language and system for describing and implementing secure networks. Our proposed language, SeNDlog, is an attempt at unifying Binder, a logic-based language for access control in distributed systems, and Network Datalog (NDlog), a database query language for declarative networks. The contributions of this paper are as follows. First, we highlight the similarities and differences between Binder and NDlog with regards to their notion of location, trust model, and evaluation strategies. Second, we motivate and propose the SeNDlog language that combines features from Binder and NDlog. Third, we demonstrate the use of SeNDlog for specifying secure networks and present directions for future work.*

## 1 Introduction

Designing secure network protocols is a difficult process. We believe that this process has been made more tedious and error-prone by the use of conventional, imperative programming languages. In this paper, we present a declarative language and system for secure networking. Our work has largely been inspired by recent efforts at using declarative languages that are aimed at simplifying the process of specifying and implementing security policies and networks. Our proposed language, *SeNDlog*, builds upon and unifies two languages: (1) *Binder* [8], a logic-based language for expressing access control decisions in distributed systems, and (2) *Network Datalog* (*NDlog*), a database query language for declarative networks [17, 16, 15].

Access control is central to security and it is pervasive in computer system. Over the years, logical ideas and tools have been used to explain and improve access control. Several logic-based languages [2] such as Binder, SD3 [11], D1LP [14] and SecPAL [18] have been proposed to ease the process of expressing and encoding access control policies. In this paper, we focus on Binder since it has a simple design, and is most similar to *NDlog*.

Like Binder, the *NDlog* language also has its roots in Datalog and logic programming. Because of its support for recursive queries over network graphs [22], *NDlog* allows compact, clear formulations of a variety of routing protocols and overlay networks which themselves exhibit recursive properties.

Despite being developed by two different communities and used for different purposes, Binder and *NDlog* are both based on logic, and extend traditional Datalog in surprisingly similar ways: by supporting the notion of context (location) to identify components (nodes) in distributed systems. This suggests the possibility of unifying these languages to create an integrated system, exploiting good language features, execution engine, and optimizations. From a practical standpoint, this integration has several benefits, ranging from ease of management, one fewer language to learn, one fewer set of optimizations, finer-grain control over the interaction between security and network protocol, and the possibility of doing analysis and optimizations across levels.

This coincidence further suggests that we may be able to dispense with much of the special machinery proposed for access control, and instead rely on distributed database engines to process these policies, leveraging well-studied query processing and optimization techniques. Interestingly, it has been shown previously [3] that Binder is similar to data integration languages such as Tsimmis [7] proposed by the database community, further indicating that ideas and methods from the database community are directly applicable to secure networking.

The contribution and organization of the paper are as follows. In Section 2, we present a background review of the Datalog language. Based on Datalog, we introduce in Section 3 the *NDlog* and Binder languages. We highlight the similarities between the Binder and *NDlog* languages in terms of their use of context and location, and differences in terms of their trust model and evaluation strategies. In Section 4, we motivate and present the *SeNDlog* language. In order to demonstrate the flexibil-

ity of *SeNDlog*, we present two motivating examples in Section 5: (1) an authenticated version of the path vector routing protocol as presented in the declarative routing paper [17], and (2) generating certified node identifiers [6] in Distributed Hash Tables (DHTs) [4]. We further discuss other possible use cases, such as implementing a secure version of DNS, and supporting programmable network environments [9, 23]. Last, we conclude in Section 6 with a discussion of future research directions.

## 2 Review of Datalog

Datalog is a recursive query language primarily used in the database community for querying graph structures. We provide a short review of Datalog, following the conventions in Ramakrishnan and Ullman's survey [22]. A Datalog program consists of a set of declarative *rules* and a query. Since these programs are commonly called *"recursive queries"* in the database literature, we will use the term "query" and "program" interchangeably when we refer to a Datalog program.

A Datalog *rule* has the form `p :- q1, q2, ..., qn.`, which can be read informally as `q1` and `q2` and `...` and `qn` implies `p`". Here, `p` is the *head* of the rule, and `q1`, `q2`,...,`qn` is a list of *literals* that constitutes the *body* of the rule. Literals are either *predicates* applied to *fields* (variables and constants), or boolean expressions that involve function symbols (including arithmetic) applied to fields. We sometimes refer to these fields as *attributes*. The rules can refer to each other in a cyclic fashion to express recursion. The order in which the rules are presented in a program is semantically immaterial. The commas that separate the predicates in a rule are logical conjunctions (*AND*); the order in which predicates appear in a rule body also has no semantic significance (though implementations typically employ a left-to-right execution strategy). The query specifies the output of interest.

The predicates in the body and head of traditional Datalog rules are relations, and we will refer to them interchangeably as predicates, relations, or tables. Each relation has a *primary key*, which consists of a set of fields that uniquely identifies each tuple within the relation. In the absence of other information, the primary key is the full set of fields in the relation.

The names of predicates, function symbols, and constants begin with a lower-case letter, while variable names begin with an upper-case letter. Most implementations of Datalog enhance it with a limited set of function calls (which start with "`f_`" in our syntax), including boolean predicates, arithmetic computations, and simple list operations. Aggregate constructs are represented as functions with field variables within angle brackets (`<>`). We will not consider negated predicates since they are not supported by Binder or *NDlog*.

## 3 Binder and *NDlog*

In this section, we introduce and compare the Binder and *NDlog* languages.

### 3.1 Binder

Binder is a language for expressing the logics of access control. Basically, a Binder program is a set of Datalog-style logical rules. Unlike Datalog, Binder has a notion of *context* that represents a component in a distributed environment and a distinguished operator `says`. For instance, in Binder we can write:

```
b1 may-access(P,O,read) :- good(P).
b2 may-access(P,O,read) :-
   bob says may-access(P,O,read).
```

The `says` operator implements one of the common logical constructs in authentication [13], where we assert `p says s` if the `principal p` supports the statement `s`. The above rules `b1` and `b2` can be read as "any principal `P` may access any object `O` in `read` mode if `P` is good or if `bob` says that `P` may do so".

A principal in Binder refers to a component in a distributed environment. Each principal has its own local *context* where its rules resides. Binder assumes an *untrusted* network, where different components can serve different roles running distinct sets of rules. Because of the lack of trust among nodes, a component does not have control over rule execution and message generation at other nodes. Instead, Binder allows separate programs to interoperate correctly and securely via the export and import of rules and derived tuples across contexts. For example, rule `b2` can be a local rule that is executing in the context of principal `alice`, which imports derived `may-access` tuples from the principal `bob` into its local context via `bob says may-access(p,o,read)` in its rule body.

The `says` operator abstracts from the details of authentication. In one specific implementation, communication happens via signed certificates, where derived tuples and rules signed using the private key of the exporting context can be imported into another context and checked using the corresponding public key.

### 3.2 *NDlog*

*NDlog* is a database query language for expressing declarative networks. We illustrate *NDlog* using a simple example of two rules that computes all-pairs of reachable nodes:

```
r1 reachable(@S,D) :- link(@S,D).
r2 reachable(@S,D) :- link(@S,Z), reachable(@Z,D).
```

The rules r1 and r2 specify distributed transitive closure computation, where rule r1 computes all pairs of nodes reachable within a single hop from all input links, and rule r2 expresses that "if there is a link from S to Z, and Z can reach D, then S can reach D." By modifying this simple example, we can construct more complex routing protocols, such as the distance vector and path vector routing protocols. See [17] for more details.

*NDlog* has a notion of location that is similar to Binder's context, through the use of a *location specifier* attribute in each predicate, prepended with an @ symbol. This attribute is used to denote the location of each corresponding tuple. For example, all reachable and link tuples are stored based on the @S address field.

### 3.3 Comparing Binder and *NDlog*

Having introduced Binder and *NDlog*, we now elaborate on the differences between these two languages:

**Trusted vs Untrusted Networks:** One of the important requirements of both Binder and *NDlog* is the ability to support rules that express distributed computations, where nodes can communicate with each other. Hence, *NDlog* supports the notion of location that is similar to Binder's notion of context. NDlog is designed for a fully trusted environment, where the location relates to data placement. Each *NDlog* rule takes as input predicates with different location specifiers, and derived tuples that when sent to another node are blindly accepted by the recipient. On the other hand, Binder assumes an untrusted network, where rules are executed with their own context, and communication happens via the use of "says"; unlike in *NDlog*, reliable authentication is required.

**Export of Derived Tuples:** In Binder, there is no integration of the security policy with the policy for exporting data. To illustrate, we consider the rule b2 from Section 3.1. The principal alice that runs these rules may wish only to export may-access(P,O,read) to the principal P, and not all nodes. It is not possible to express this restriction in Binder. Hence, any principal can import the may-access(P,O,read) tuple derived by alice. Being able to restrict the sending of messages to selected recipients is an important requirement in secure network protocols, both from performance and secrecy standpoints. *NDlog* achieves that with the use of location specifiers at the rule head.

**Bottom-up vs Top-down Evaluation:** Most access control languages including a practical implementation of Binder [19] and SD3 utilize a goal-oriented top-down evaluation (backward-chaining from head to body) strategy. Specific requests are made as goals, which are then resolved against the security policies. On the other hand, network protocols are long-running processes, and incrementally recompute and repair routes based on changes to the underlying network. Hence, *NDlog* programs are executed in a bottom-up (or forward-chaining) evaluation [21] where the bodies of the rules are evaluated to derive the heads. This has the advantage of permitting set-oriented optimizations while avoiding infinite recursive loops, and at the same time, is a better fit for the incremental continuous execution model of network protocols.

## 4 *SeNDlog* Language

In this section, we present the *SeNDlog* language, which unifies Binder and *NDlog* with the following goals. First, we require that *SeNDlog* be as expressive as Binder and *NDlog* in order to support the security policies and network protocols that have been previously supported by both languages. Second, the language constructs of *SeNDlog* should support authenticated communication and also enable the differentiation of nodes according to their roles. Third, *SeNDlog* should flexibly support both trusted and untrusted environments. Fourth, to leverage existing execution engines and fit the incremental continuous execution model of network protocols, *SeNDlog* must be amenable to efficient execution and optimizations by a distributed query engine using a bottom-up evaluation strategy.

### 4.1 Rules within a Context

In the *SeNDlog* language, we allow a set of rules (including tuples) to reside at a particular node. We do this at the top level for each rule (or set of rules), for example by specifying:

```
At N,
  r1 p :- p1,p2,...,pn.
  r2 p1 :- p2,p3,...,pn.
```

In the example above, the rules r1 and r2 are in the context of N, where N is either a variable or a constant representing the principal where the rules reside. If N is a variable, it will be instantiated with local information upon rule installation. In a trusted world, N can simply be the address of a node. In general, N might be the address/public-key pair of a node. In a multi-user environment, N can further include the user name.

We can attach additional conditions c1,c2,...,cn that are used to determine at runtime whether a node serves a certain role:

```
At N, c1,c2,...,cn
  r1 p :- p1,p2,...,pn.
  r2 p1 :- p2,p3,...,pn.
```

In the above example, a principal N can execute the rules r1 and r2 only if all the conditions c1,c2,...,cn are satisfied at runtime. This allows the role of a principal to be defined based on runtime conditions.

### 4.2 Communicating Contexts

Much as in Binder, the *SeNDlog* language allow different principals or contexts to communicate via import and export of tuples. The movement of tuples serves two purposes: (1) maintenance messages as part of a network protocol's updates on routing tables, and (2) distributed derivation of security decisions. Imported tuples from a principal N are automatically quoted using "N says", to differentiate them from local tuples. During the evaluation of *SeNDlog* rules, we allow derived tuples to be communicated among contexts via the use of *import predicates* and *export predicates*:

**Definition 1** An *import predicate* is of the form "N says p" in a rule body, where N is the principal that is asserting the predicate p.

**Definition 2** An *export predicate* is of the form "N says p@X" in a rule head, where principal N exports the predicate p to the context of principal X. Here, X can be a constant or a variable. If X is a variable, in order to make bottom-up evaluation efficient, we further require that the variable X occur in the rule body. As a shorthand, we can omit "N says" if N is the principal where the rule resides.

With the definitions above, a *SeNDlog* rule is a Datalog rule where the rule body can include import predicates, and the rule head can be an export predicate. We provide a concrete example with the following four *SeNDlog* rules e1-e4:

```
At N,
 e1 p(X,Y) :- p1(X), p2(Y).
 e2 p(X,Y,W) :- Y says p1(X), Z says p2(W),
                Z!=N.
 e3 p(Y,Z)@X :- p1(X), Y says p2(Z).
 e4 Z says p(Y)@X :- Z says p(Y), p1(X).
```

Rule e1 is a traditional Datalog rule. Rule e2 contains two predicates p1 and p2 imported from Y and Z respectively. Rules e3 and e4 contain an import predicate each, and export their derived heads to X.

Note that, in rule e4, the export principal Z differs from the principal N. To ensure that p is indeed asserted by Z, we introduce the *honesty constraint* in all *SeNDlog* rules:

**Definition 3** A *SeNDlog* rule in the context of principal N is *honest* if the following condition is satisfied: if the rule head is "X says p", where X is a constant or a variable, either X is N, or "X says p" occurs in the body of the rule.

The honesty constraint enables a simple, secure implementation. Specifically, for security, whenever a principal other than N exports N says p, it should provide a proof that this is the case; the proof is a signature by N. With the honesty constraint, the principal may simply forward the signature that corresponds to the occurrence of N says p in the rule body.

Like *NDlog*, *SeNDlog* allows derived tuples to be exported to *specific* nodes via the export predicates. This is done as a way of enforcing secrecy and also performance (avoiding broadcast of tuples).

### 4.3 Different Levels of "says"

Like Binder, *SeNDlog* utilizes "says" as an abstraction for the details of authentication. In a statement Z says advertise(X,Y)@W, both Z and W can be treated as extra arguments of advertise and the usual rules of logic applies locally.

Note that the implementation of "says" may depend on the system and its context. In a hostile world, "says" may require digital signatures. For example, in rule e3 from the previous section, N should check that p2 indeed came from Y by checking the signature of the imported tuple against Y's public key. In a more benign world, "says" may simply append a cleartext principal header to a message—and this will of course be cheaper. Somewhere in between, the use of digital signatures may be applied only to certain important messages: there is a trade-off between security and efficiency, and the language does not provide any leverage in deciding how that trade-off should be made. Note however that the policy writer could easily provide hints along with rules, indicating that some "says" are more important than others. Going further, one could have multiple operators with different security levels.

## 5 *SeNDlog* Examples

Having presented the *SeNDlog* language, we provide two examples of secure network programming written using *SeNDlog*: an authenticated version of the basic path vector routing protocol and secure assignment of DHT node identifiers. Through these examples, we demonstrate the flexibility of *NDlog* and present several of its language features.

In addition to these examples, we have also explored using *SeNDlog* to specify DNSSec, which is a secure version of DNS as presented in SD3 [11]. In an extensible shared testbed environment [23, 20], *SeNDlog* can be used concurrently as a language for implementing declarative networks and trust management [5], for example, ensuring that the users

who are executing these networks have sufficient authorization for code loading, communication, and the utilization of shared resources.

### 5.1 Authenticated Path Vector Protocol

```
At Z,
 z1 route(Z,X,P) :- neighbor(Z,X),
                    P=f_initPath(Z,X).
 z2 route(Z,Y,P) :- X says advertise(Y,P),
                    acceptRoute(Z,X,Y).
 z3 advertise(Y,P1)@X :- neighbor(Z,X),
                         route(Z,Y,P),
                         carryTraffic(Z,X,Y),
                         P1=f_concat(X,P).
```

Our first example shows the basic path vector protocol as presented in the declarative routing[17] paper, with the additional use of the "says" operator. At every node Z, this program takes as input neighbor(Z,X) tuples that contain all neighbors X for Z. The input carryTraffic and acceptRoute tables are used to represent the export and import policies of node Z respectively. Each carryTraffic(Z,X,Y) tuple represents the fact that node Z is willing to serve all network traffic on behalf of node X to node Y, and each acceptRoute(Z,Y,X) tuple represents the fact that node Z will accept a route from node X to node Y.

At every node Z that runs the above program, the Z says advertise(Y,P) tuples containing the path to destination node Y is communicated among neighboring nodes. As noted in Section 4.2, we omit "Z says" for brevity in rule z3. The use of "says" ensures that all advertise tuples are verified by the recipient for authenticity. The eventual outcome of executing the program is the generation of route(Z,X,P) tuples, each of which stores the path P from source Z to destination X

Rule z1 takes as input neighbor(Z,X) tuples, and computes all the single hop route(Z,X,P) containing the path [Z,X] from node Z to X. Rules z2-z3 are used to compute routes of increasing hop count. Upon receiving an advertise(Y,P) tuple from X, Z uses rule z2 to decide whether to accept the route advertisement based on its local acceptRoute table. If the route is accepted, a route tuple is derived locally, and this results in the generation of an advertise tuple which is further exported by node Z via rule z3 to selected neighbors X based on the policies of the local carryTraffic table. Each exported advertise tuple has a new path P1 which is computed by prepending neighbor X to the input path P using the f_concat function. A more complex version of this protocol will have additional rules that derive carryTraffic and acceptRoute, avoid path cycles and also derive shortest paths with the fewest hop count.

### 5.2 Secure DHT Node Identifiers

In our second example, we use *SeNDlog* to specify the assignment of node identifiers in DHTs [6]. Our version of the code avoids a security weakness in a DHT where malicious nodes can occupy a large range of the key space. This example also demonstrates the use of *SeNDlog* to specifying the different roles for nodes. We describe this example using Chord as our DHT. There are three sets of rules for three types of nodes: (1) a new node NI joining the chord ring, (2) the certificate authority CA, and (3) the landmark node LI. Each node runs its respective set of rules as follows:

```
At NI,
 ni1 requestCert(NI,K)@CA :- startNetwork(NI),
                             publicKey(NI,K),
                             MyCA(NI,CA).
 ni2 nodeID(NI,N) :- CA says nodeIDCert(NI,N,K)
 ni3 CA says nodeIDCert(NI,N,K)@LI :-
        CA says nodeIDCert(NI,N,K),
        landmark(NI,LI).

At CA,
 ca1 nodeIDCert(NI,N,K)@NI :-
        NI says requestCert(NI,K),
        S=secret(CA,NI),
        N=f_generateID(K,S).

At LI,
 li1 acceptJoinRequest(NI) :-
        CA says nodeIDCert(NI,N,K).
```

In rule ni1, a node NI that wishes to join the Chord ring first exports a requestCert tuple to its CA (as indicated in the entry in its MyCA table) to request *nodeID certificates*. Upon receiving the request, the CA generates a nodeIDCert(NI,N,K) tuple containing the nodeID certificate, which is then exported back to node NI. The nodeIDCert(NI,N,K) tuple contains the IP address of node NI, the corresponding public key K, and a generated identifier N randomly chosen from the keyspace using the function f_generateID(K,S) that takes as input the public key of K and a previously exchanged secret S known only to the CA and NI.

Upon importing the nodeIDCert tuple from the CA, using rule ni2, node NI initializes its local node identifier which stored as a nodeID(NI,N) tuple. It also forwards the nodeIDCert to its landmark node LI in order to join the chord ring.

At the landmark node LI, nodeIDCert is imported and checked for authenticity. If nodeIDCert is accepted, the landmark node derives an acceptJoinRequest(NI) tuple that can further be used to generate a lookup request to locate the successor node on behalf of node NI. The rest of the

Chord rules as presented [16] can then be used by node `NI` to implement the rest of the Chord protocol.

## 6 Related Work and Conclusion

In terms of related work, through our use of Binder, our work is related to a large literature on access control in distributed systems (*e.g.*, [11], [14] and [18]). In addition, through *NDlog*, our work is related to a large literature of network specification languages (*e.g.*, [12], [10]). Occasionally, these two bodies of work have intersected, for instance, when Jim et al. described DNSSec in SD3, and presented the *D3log* language [12] which has similarities to *NDlog*. However, this intersection is remarkably small. We are not aware of any previous efforts with the goals and scope of ours.

In summary, we have proposed *SeNDlog* as a declarative language for secure networking. Our initial language design is based on unifying the Binder and *NDlog* languages, and is intended for execution using a distributed query processor.

Our research is proceeding in several directions. First, while we have focused on Binder, we plan to consider language features from other access control languages. Second, we are planning a full-fledged system implementation via enhancements to the *P2* declarative networking system [1]. This implementation will require adding the support of principals serving different roles, introducing the "`says`" operator, and supporting communication in untrusted networks, where tuples are communicated via signed certificates. We intend to explore the limitations of our initial language design by implementing a variety of secure networks. Third, we intend to investigate analysis and cross-layer optimization opportunities that arise as a result of having a single integrated system that unifies network and security specifications.

## References


[1] P2: Declarative Networking. http://p2.cs.berkeley.edu.

[2] Abadi, M. Logic in Access Control. In *Symposium on Logic in Computer Science* (2003).

[3] Abadi, M. On Access Control, Data Integration and Their Languages. *Computer Systems: Theory, Technology and Applications, A Tribute to Roger Needham Springer-Verlag* (2004), 9–14.

[4] Balakrishnan, H., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. Looking Up Data in P2P Systems. *Communications of the ACM, Vol. 46, No. 2* (Feb. 2003).

[5] Blaze, M., Feigenbaum, J., and Keromytis, A. D. The role of trust management in distributed systems security. In *Secure Internet Programming* (1999), pp. 185–210.

[6] Castro, M., Drushel, P., Ganesh, A., Rowstron, A., and Wallach, D. Secure Routing for Structured Peer-to-peer Overlay Networks. In *Proceedings of Usenix Symposium on Operating Systems Design and Implementation* (2002).

[7] Chawathe, S., Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J. D., and Widom, J. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *16th Meeting of the Information Processing Society of Japan* (Tokyo, Japan, 1994).

[8] DeTreville, J. Binder: A logic-based security language. In *IEEE Symposium on Security and Privacy* (2002).

[9] GENI. Global Environment for Network Innovations. http://www.geni.net/.

[10] Griffin, T. G., and Sobrinho, J. L. Metarouting. In *ACM SIGCOMM* (2005).

[11] Jim, T. SD3: A Trust Management System With Certified Evaluation. In *IEEE Symposium on Security and Privacy* (May 2001).

[12] Jim, T., and Suciu, D. Dynamically Distributed Query Evaluation. In *ACM Symposium on Principles of Database Systems* (2001).

[13] Lampson, B., Abadi, M., Burrows, M., and Wobber, E. Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems 10*, 4 (1992), 265–310.

[14] Li, N., Grosof, B. N., and Feigenbaum, J. Delegation Logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and System Security (TISSEC)* (Feb. 2003).

[15] Loo, B. T., Condie, T., Garofalakis, M., Gay, D. E., Hellerstein, J. M., Maniatis, P., Ramakrishnan, R., Roscoe, T., and Stoica, I. Declarative Networking: Language, Execution and Optimization. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (June 2006).

[16] Loo, B. T., Condie, T., Hellerstein, J. M., Maniatis, P., Roscoe, T., and Stoica, I. Implementing Declarative Overlays. In *ACM Symposium on Operating Systems Principles* (2005).

[17] Loo, B. T., Hellerstein, J. M., Stoica, I., and Ramakrishnan, R. Declarative Routing: Extensible Routing with Declarative Queries. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (2005).

[18] Moritz Y. Becker and Cedric Fournet and Andrew D. Gordon. SecPAL: Design and Semantics of a Decentralized Authorization Language. Tech. Rep. MSR-TR-2006-120, Microsoft Research, 2006.

[19] Pimlott, A., and Kiselyov, O. A Logic-based Trust-management System. In *International Symposium on Functional and Logic Programming* (Apr. 2006).

[20] PlanetLab. Global testbed. http://www.planet-lab.org/.

[21] Raghu Ramakrishnan and S. Sudarshan. Bottom-Up vs Top-Down Revisited. In *Proceedings of the International Logic Programming Symposium* (1999).

[22] Ramakrishnan, R., and Ullman, J. D. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming 23*, 2 (1993), 125–149.

[23] Tennenhouse, D., Smith, J., Sincoskie, W., Wetherall, D., and Minden, G. A Survey of Active Network Research. In *IEEE Communications Magazine* (1997).