Speccast

Leonid Poutievski, Kenneth L. Calvert, and James N. Griffioen
Laboratory for Advanced Networking
University of Kentucky
Lexington, KY 40506-0046

***Abstract*— In this paper we describe a new network service called Speccast. Speccast offers a generalized addressing and routing abstraction on which a rich variety of semantic services can be built, and, as such, provides a vehicle for studying the relationships among routing, addressing and topology. Unlike overlay-based systems, we study a more basic problem, in which the topology of the network is given, and there is not necessarily any pre-existing underlying network service. In the speccast model, each packet carries a destination predicate and the network's job is to deliver the packet to all nodes satisfying that predicate.**

After showing how this generalized routing service subsumes other services both traditional (unicast, multicast) and emerging (publish-subscribe), we present a layered solution for a specific class of simple predicates. We examine the tradeoffs in various forms of our approach, and compare it to existing solutions for unicast and multicast. Studies using transit-stub graphs show that our generic service performs comparably to existing solutions for traditional services, while also effectively supporting new and emerging services.

***Index Terms*— Routing, system design, addressing and location management, network applications and services, simulations.**

I. Introduction

Systems that offer network-layer-like services—i.e. systems that forward packets through a series of relay nodes until they reach one or more destinations—have received a good deal of attention from the research community in recent years. Examples include computed-routing systems such as Chord [1], CAN [2], Tapestry [3] and many others; rendezvous/indirection systems such as i3 [4]; publish-subscribe systems such as Siena [5] and intentional multicast [6]. All of these systems assume the existence of an efficient underlying unicast addressing and routing system (viz., the Internet) on which they introduce their own higher-level address space and topology (i.e. neighbor relationships among participating nodes). This has the advantage that address spaces and topologies can be tailored to specific applications or higher-level services (e.g. distributed hash tables). It has the disadvantage that efficiency in the overlay routing does not, in general, correspond to efficiency in the underlying unicast system. Although steps can be taken to increase the correlation between the two, they may reduce the advantages of independence and the overall flexibility of the system. Moreover, the network as a whole must bear the overhead of implementing both the underlying system and the overlay system.

In this paper we introduce a new network service we call *speccast*. The idea of speccast is that each packet carries a predicate specifying its destination, and the network's job is to deliver the packet to all nodes (and only those nodes) that satisfy the predicate. Speccast is interesting as a service because it is very general. Depending on the nature of the predicates that can be specified, speccast subsumes traditional network-layer services (unicast and multicast), but can also support higher-level services like publish-subscribe systems. A network-layer implementation of speccast—i.e., one that does *not* rely on the existence of any underlying network service and builds upon a given topology—is of interest because it can offer both traditional and higher-level services using the same mechanism.

Clearly the utility of speccast depends on what predicates can be specified as destinations, and how they relate to the characteristics of nodes that are of interest to applications. In this paper we describe an approach to implementing speccast for a simple predicate language (positive combinations of atomic propositions in disjunctive normal form). Our implementation is stand-alone, and does not assume the existence of any underlying unicast addressing or forwarding mechanisms. An obvious question is whether such a service can be implemented in a manner that scales to support dynamic networks of millions of nodes. Although we do not claim to answer that question definitively in this paper, our results highlight tradeoffs and show that when the predicate set satisfies certain conditions, our implementation can provide traditional unicast and multicast services with reasonable additional overhead compared to native implementations, and in many cases with *lower* overhead than overlay services such as Tapestry. Of course, our implementation also offers the generality of speccast, supporting other services such as subcasting, directed multicast, and publish-subscribe.

In this paper we restrict our attention to the steady state: all results assume that the topology is fixed, and necessary state information is precomputed. While the assumption of a static network is strong, we believe it is necessary to first assess efficiency and performance in that scenario, before progressing to the harder problem of dynamic networks.

The main contributions of this paper are: (i) introduction of the speccast service and discussion of its relationship to other network-layer-type services; (ii) description of a network-layer implementation for a simple predicate language based on atomic propositions; (iii) a simulation-based investigation of the efficiency of our implementation approach, comparing it to alternative network and application-layer approaches including an idealized version of the Internet's network layer, Tapestry, and a publish-subscribe architecture. Our results highlight the relationships among topology, address space structure, and forwarding efficiency.

In the next section we define the speccast service, describe its relationship to other routing-type services, and discuss the metrics we use for comparing and evaluating network-layer solutions. In Section III, we describe a speccast implementation for a simple predicate language. The implementation is structured as two semi-independent layers. The lower layer solves the problem of delivering a packet to every node that satisfies a particular atomic proposition. The upper layer delivers packets to nodes that satisfy conjunctions and disjunctions of atomic propositions, using the lower layer. In Section V we present the results of simulation studies of the overhead of our solution in terms of delay, network load, and per-node state for traditional unicast and multicast services. Finally, Section VII offers some observations and the conclusion that our layered solution offers a reasonable tradeoff among per-node state information, forwarding delay, and excess packet transmissions, compared to other approaches.

II. Generalized Routing

In this section we describe the general speccast problem as well as the particular instance for which we propose a solution. We also discuss the criteria for comparing network-level solutions.

A. The Speccast Problem

The speccast service is defined as follows. Let **N** denote a set of nodes, and let **P** be a set of predicates on nodes, i.e. functions from **N** to $\{true,false\}$. Each packet m carries a predicate $m.dest \in \mathbf{P}$ called its *destination predicate*. The predicate specifies the set of nodes to which m is to be delivered. That is, the nodes of the network conspire to deliver m to all nodes n such that $m.dest(n) = true$.

The speccast service is *best-effort*; it will deliver the message m to nodes satisfying $m.dest$, provided certain conditions are met (e.g., the network is not overloaded). Moreover, each packet is forwarded and delivered independently of all others, i.e. speccast is a *datagram* service. The speccast *problem* is to implement this service abstraction, on a given network represented by an undirected graph.

This problem statement is very general. It says nothing about the nature of the predicates used to specify destinations, nor does it specify anything about which nodes satisfy what properties. In particular, no relationship is assumed between the predicates satisfied by a node and that node's location in the topology, and no underlying routing or addressing mechanism is assumed.

Any solution to this problem must describe (i) the information carried in each packet (in addition to the destination predicate); (ii) the information stored at each node, i.e. the "forwarding data structure"; (iii) the algorithm used by nodes to originate/forward packets, which takes as inputs the forwarding data structure and the information from the packet, and returns the set of channels on which the packet should be transmitted; and (iv) distributed algorithms for initializing the forwarding data structures at each node and updating them across changes in network topology (and possibly in predicate set, i.e. which nodes satsify which predicates).

Although any practical speccast service must deal with dynamic topologies, an efficient solution for the static case is a necessary condition for one that can adapt to dynamically-changing topologies. We therefore focus in this paper on the static version of the problem (i.e., the topology and the set of predicates are both constant and unchanging). Although we describe algorithms for initializing forwarding data structures in Section III, we do not consider dynamic topologies here.

B. The Predicate Language

Any solution to the speccast problem must be designed for a specific **P**—i.e. with full knowledge of what predicates can be defined, their semantics, and how they are encoded for transmission with the message. Depending upon the properties of **P** and which nodes satisfy what predicates, the speccast service abstraction can subsume many traditional and emerging network-layer addressing/routing services. For example, *unicast* service is supported if **P** contains a *point predicate* for each node to which messages need to be delivered. (A point predicate is satisfied by exactly one node and thus can be used as an identifier.) Classical (any-source) *multicast* service can be supported by having a predicate in **P** for each multicast group address, which is satisfied by exactly the nodes belonging to that multicast group. A *publish-subscribe* service can be provided if there is a predicate in **P** for each published element, which is satisfied by all nodes subscribing to receive information corresponding to that element. (See Section VI for a discussion of the relationship between existing pub-sub solutions and our speccast approach.) In short, the flexibility of predicates allows the speccast service to be used in a wide variety of ways.

To take advantage of this flexibility, we would like to have a system with a powerfully expressive predicate language, in which a large number of distinct predicates—certainly many per node—can be defined. However, it is also the case that the *efficiency* of any solution will depend strongly on the language. In general, the more powerful the language for specifying predicates, the more costly it is to answer questions about them. Thus, in general there will be a tradeoff between the expressive power of the predicate language and the efficiency of a solution using it. This brings us to the question of metrics used to compare network-layer service solutions.

C. Evaluating and Comparing Solutions

Because speccast subsumes other network-layer services, it provides an excellent basis for measuring the benefits that can be achieved when using a specialized routing service, optimized for a specific environment—for example, the service provided by the current Internet, which uses hierarchical addresses in combination with address locality to improve scalability. To compare approaches, we first need metrics to measure the costs associated with each approach. The most obvious metric is *correctness*: Does each message reach every

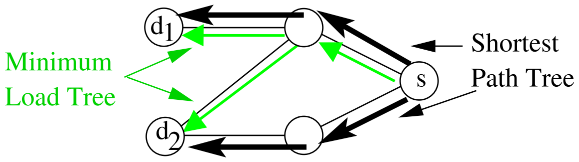

Fig. 1. The shortest path tree to d_1 and d_2 does not produce the tree with minimum load.

destination node(s)? To compare correct solutions, we define several measures of the cost of delivering a packet carrying a given destination predicate p from a given source node n to all nodes that satisfy p.

- **Network load**. The number of links over which a message is forwarded en route to its destinations. Note this includes all links over which the message is forwarded, even if the link does not lead to any node that satisfies p. We define the *network load ratio* to be the ratio of the total number of edges crossed by a message versus the number of links in the shortest-path tree from the source n to all nodes satisfying p. (Note that the shortest path tree does not necessarily imply the smallest network load, as shown in Figure 1. It is therefore possible for a solution to achieve network load ratios less than one. This effect has been observed before in "shared tree" multicast protocols such as CBT and PIM-SM [7].)
- **Delay**. We define the *delay cost* of a routing solution to be the sum, over all nodes d satisfying p, of the number of edges traversed on the path from n to d, when that solution is used. We define delay *stretch* to be the ratio of the algorithm's measured delay cost versus the minimum possible delay cost (i.e., the shortest paths).
- **Network State**. We measure the *state cost* of a solution as the amount of information stored at all nodes combined. This is a function of the size of **N** and **P**.
- **Forwarding-time Computation.** The complexity of the forwarding computation performed by each node on each packet gives an indication of the processing load imposed by the service on routers. This cost depends on the properties of **P**.

To illustrate the various costs, consider some simple, but non-scalable, solutions. First, consider a *broadcast* approach in which all packets are simply *flooded* throughout the network. Each node forwards every incoming message on all incident channels over which the message has not yet been either transmitted or received. To keep messages from looping, each message is randomly assigned a unique (with high probability) identifier, and each node keeps track of the message IDs that have been transmitted on each outgoing channel in the recent past, using (for example) a small amount of ephemeral state [8] or a Bloom filter. This approach is effective, requires fixed amounts of state, minimizes delay, and requires very little forwarding-time computation. However, its network-load cost is large (unless almost all predicates are satisfied by almost all nodes) because each packet is transmitted over every link.

As another example, consider a *source-routed* approach. If the source node n has complete knowledge of the network graph, as well as knowledge of which nodes satisfy the destination predicate p, it can determine the optimal path(s) the packet should follow to the desired destinations. By encoding this set of paths (i.e., a tree) in the packet header, optimal network load and delay can be achieved with modest forwarding-time cost. However, the requirement that each node know the entire network topology *and* the predicates satisfied by every node means the per-node state costs grow at least linearly in the size of the network.

These examples illustrate the tradeoffs as well as the flexibility allowed by the problem statement. It may be acceptable to deliver a packet to some nodes that do *not* satisfy the destination predicate (thereby increasing network load), if doing so results in some systematic savings in state, delay, or forwarding-time computation. In the next section we present a speccast solution for a simple predicate language based on atomic propositions.

III. A Layered Solution

This section describes an implementation of speccast for a simple predicate language: positive boolean combinations of atomic propositions. Given is a finite set S of atomic propositions; in what follows, a, b and c denote arbitrary members of S. The predicates in **P** are disjunctions of conjunctions of members of S. Thus b, $a \wedge b \wedge c$, $a \vee d$, and $(a \wedge b) \vee c$ all denote members of **P**.[1] Initially we assume nothing about relationships among the atomic propositions—that is, knowing that a node satisfies a tells us nothing about whether it satisfies b or any other atomic proposition. Later we will show how knowledge of a certain type of structure in S can be exploited to improve performance.

Our approach decomposes the problem into two semi-independent subproblems. The first is to deliver a packet to all nodes satisfying any given atomic proposition b. We refer to the solution to this subproblem as the "base layer". The second subproblem is to use the base layer to deliver a packet to the set of nodes satisfying any given disjunction of conjunctions (i.e. boolean expression in disjunctive normal form) while minimizing delivery to nodes that do *not* satisfy the predicate. We refer to the solution to the second subproblem as the "composition layer". Because we do not assume the existence of any underlying network service, our solution requires nodes to exchange information with their neighbors in order to initialize and maintain their forwarding data structures.

A. Base Layer: Deliver to Atomic Propositions

The problem of "delivery to all nodes satisfying an atomic proposition" could be solved using an any-source multicast service (e.g., classical IP multicast), simply by using a multicast group address per atomic proposition and having each node that satisfies an atomic proposition join the corresponding

[1] Negation is not included, but could be added in a straightforward manner at the cost of approximately doubling the amount of state.

multicast group. Unfortunately, existing multicast services cannot be used, because they are unaware of our composition layer which needs to be invoked at intermediate routers every time a multicast packet is forwarded. Because existing multicast services cannot be used, our base layer solution constructs its own spanning tree for each atomic proposition b, which connects all the nodes that satisfy b. In general this tree will include nodes that do not satisfy b, i.e. it is a Steiner tree.

Every node in the network maintains a base layer forwarding table containing one entry per atomic proposition. In all cases, the entry for attribute b indicates a set of interfaces. If node n is part of the tree for attribute b, it indicates all interfaces that connect to n's neighbors in the tree. If n is *not* part of the tree for b, its entry indicates a single interface that leads toward the tree for b.

When a packet arrives that needs to be forwarded to nodes satisfying a, the node looks up the entry for a and forwards the packet over all indicated interfaces (except the one on which the packet arrived, if any). Thus once a packet reaches the tree for atomic proposition a, it is forwarded to all nodes satisfying a. As we will see later, this propagation can be short-circuited or pruned by the upper layer. The use of a shared tree minimizes overall network load and network state is reduced, possibly at the cost of increased delay.

The forwarding table information for the base layer can be set up using a simple distance-vector-type protocol in which each node announces to its neighbors the information it has about every attribute tree. Nodes exchange lists of tuples of the form $\langle$atomic proposition, tree ID, distance, tree size$\rangle$, where the tree ID is a random identifier used in merging unconnected subtrees, the distance is the number of hops from the node to the closest tree it knows about (0 if the node is on the tree, infinity if it does not know of any tree for the atomic proposition), and the tree size is an estimate of the number of nodes that satisfy the atomic proposition in the tree. The latter two fields are used in the composition layer in a manner described later.

Initially each node considers itself the only member of the tree for exactly those atomic propositions that it satisfies. When a node learns of another tree for the same atomic proposition with a lower ID, it "joins" that tree; the result is that eventually the two trees are merged into one (and nodes along the path between the two trees are added to the tree even if they do not satisfy the atomic proposition). This process does not produce a minimal tree; however, the problem of constructing a minimum Steiner tree is computationally hard. As we shall see later, the additional overhead resulting from the use of a (possibly) non-optimal tree is modest.

B. Composition Layer

As described above, destination predicates must be given in disjunctive normal form.[2] To reach the set of nodes satisfying such a composite boolean expression, the composition layer uses the base layer to forward a packet to all nodes satisfying certain selected atomic propositions in the expression. This is carried out via a two-step process: first the originating node selects a single atomic proposition from each conjunction. Then a copy of the packet is forwarded to all nodes satisfying each selected atomic proposition, using the base layer. This weakening process results in some nodes receiving the packet that do not satisfy its destination predicate, but every node that satisfies the original predicate satisfies the weaker version and will receive the packet. Thus we are trading some additional network load for a reduction in forwarding state and delay.

For example, the predicate $D = (a \wedge b) \vee c$ might be weakened to $a \vee c$ or to $D' = b \vee c$. The services of the base layer are then used to send the packet to nodes satisfying the weaker predicate, i.e. all nodes satisfying b and all nodes satisfying c. Forwarding by the base layer is based on D', although the original predicate D remains in the packet. At each node m, the predicate D is checked to see whether it is satisfied by m; if so, the packet is delivered to the applications layers of the node. In addition, each node forwards the packet according to the algorithm described below.

The obvious question is which atomic proposition to select from each conjunction. Various heuristics, of varying degrees of sophistication, are possible. In our implementation, we use a combination of tree size and distance to the tree (from the originating node); both are approximate indicators of the network load (number of links traversed) that will result from delivering the packet via that tree. Thus we choose the atomic proposition from each conjunction that minimizes the sum of these values.

Once the destination predicate is weakened, the list of selected atomic propositions is placed in the packet along with the original destination predicate, and the packet is forwarded to each, using the base layer. (Recall that each node has a forwarding table entry for each attribute, regardless of whether it satisfies that attribute or not.)

As an example, consider the predicate $D = (a \wedge b) \vee (c \wedge d)$. Suppose that a and c are the atomic propositions with the smallest sum of distance and tree size. The packet is forwarded with a header containing the information below:

$a \wedge b$	$a(1)$	$c \wedge d$	$c(1)$

The header contains a list of disjuncts, each consisting of a conjunction and an indication of the atomic proposition selected from that conjunction, plus a *pending bit* for that atomic proposition (indicated in the figure by the number in parentheses). The pending bit for a indicates whether the packet still needs reach the tree for a.

When forwarding a packet, the composition layer at node n carries out the following steps.

1) For each selected atomic proposition a such that n is part of the tree for a: forward the packet out all interfaces belonging to that tree (except the one on which the packet arrived), clearing the pending bit for a in the process.
2) For each selected atomic proposition b such that n is *not* part of the tree for b *and* the pending bit is set: add the

[2] We could as easily have chosen CNF; the approach is practically identical.

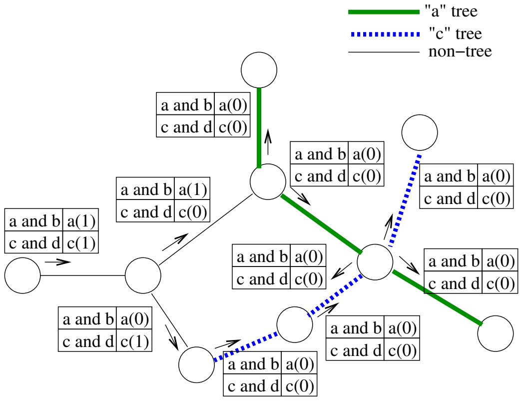

Fig. 2. Packet forwarding example

interface toward b's tree to the list of outgoing interfaces for the packet.

3) For each interface in the list created above, forward a copy of the packet on that interface, clearing the pending bits for each atomic proposition whose tree does *not* lie in that direction.

These steps ensure that the packet follows the shortest path to the tree for each selected atomic proposition. They also allow a single copy of a packet to follow the paths toward multiple trees until the paths diverge.

To reduce delay, the first step above results in packets traversing the *union* of trees that intersect. That is, if a packet is traversing one spanning tree and reaches a node that also belongs to the tree of one of the other selected atomic propositions, the composition layer duplicates the packet and forwards the duplicate(s) along the latter tree as well. This lets packets reach the nodes satisfying a faster by taking multiple routes: by following the "gradient" of nodes that are not on the a tree, and via the trees of other selected atomic propositions that intersect the a tree. (It is this optimization that requires that all selected atomic propositions be carried in all copies of the packet.) Figure 2 illustrates the use of this optimization, as well as the use of the pending bits.

C. Duplicate Suppression

Because packets can encounter a selected atomic proposition's tree multiple times, care must be taken to avoid excessive duplication and waste of bandwidth. (For example, trees that intersect in two places would result in packets looping forever.) For this purpose our approach requires a *duplicate suppression* mechanism that keeps packets from being forwarded over the same link more than once in the same direction. This mechanism can be implemented in several ways. One possibility is to use *ephemeral state processing* (ESP) [8]: When a packet is forwarded, it leaves a bit of ephemeral state on each interface it traverses; packets carry an ESP instruction which causes subsequent instances of the same packet to be dropped for a short time. Another possibility is to record each interface in the packet as it passes through that interface, using a Bloom filter as in Icarus [9]. These two approaches ensure that, with high probability, packets encounter an interface at most twice. A final possibility, which can catch any cases missed by the first two, is to have packets carry a hop-limit field (with initial value large enough to ensure they can reach all nodes), and be discarded when it reaches zero.

D. Optimization: Filtering

A drawback to the proposed implementation is that in general, packets will reach some number of nodes that do not satisfy their destination predicates. We would like to reduce the amount of "overdelivered" packets without increasing delay, perhaps by maintaining additional state.

Filters provide a means of *shared learning* among packets. The idea is to record additional "learned" state at key nodes in the network to prevent packets from flowing to nodes that do not satisfy the predicate. The additional state records information about the existence (or non-existence) of nodes matching certain atomic propositions. For example, if node n on the spanning tree for x learns that there are no nodes that satisfy atomic proposition y along one of its branches, it can filter all messages containing the predicate $x \wedge y$, preventing them from unnecessarily traversing that branch of the spanning tree.

Positive filters record the existence of nodes matching a certain atomic proposition, and can be propagated along with the standard routing messages used to establish the spanning trees (and the gradients to the spanning trees). Negative filters can also be used, but depend on traffic flowing through the tree to prune branches that do not have nodes matching a certain atomic proposition.

The use of filters comes at a cost in additional state. If the total number of atomic propositions is $|S|$, in the worst case the use of filters can increase the total network state by $2|S|^2$ entries. Some of the experimental results presented later reflect the use of positive filters (results are labeled as such).

E. Optimization: Default Routes

In the Internet, a substantial reduction in total state is achieved through the use of "default routes" in treelike parts of the network. In particular, routers that have only a single interface (i.e. those that connect to only one other router and possibly many hosts) only need a small number of forwarding table entries, to cover their local networks; everything else is forwarded to the (single) neighbor.

A similar optimization is possible in our speccast solution, for nodes that have only one or two interfaces. A node with a single interface has no choice about where/how to forward a packet. A node with two interfaces simply forwards each packet received on one out the other and vice versa. This increases network load slightly, but reduces the amount of state required.

IV. Structured Predicate Sets

The solution presented in the previous section makes no assumptions about the relationships among predicates in $\mathbf{P}$.

As a result, every node in the network has a forwarding table entry for every atomic proposition. Still, we would like to be able to reduce the amount of network state, for example by replacing a *group* of forwarding table entries that contain similar information with a *single* entry. This is of course exactly the abstraction that occurs in Internet routing: groups of destinations are represented in forwarding tables by a single entry, identified by the address prefix common to all the destinations.

What is needed is a way to represent a set of atomic propositions with a single atomic proposition. For example the single atomic proposition "network element" might represent the set of atomic propositions "router", "hub", "bridge", and "NATbox". For this to work, it is necessary to be able to recognize the relationship among attributes. In particular, some means is required to determine that "network element" is the appropriate forwarding table entry to use for the "hub" atomic proposition. Of course, such relationships can be iterative, so that "router" itself represents both "CiscoRouter" and "JuniperRouter".

At forwarding time, packets are forwarded toward the tree corresponding to the (top-level) atomic proposition corresponding to each selected atomic proposition. Nodes in the top-level atomic proposition's tree must also have the necessary state to reach lower-level atomic propositions. Once a packet reaches a node on the top-level tree, it will be forwarded to an atomic proposition tree at the next level (down) in the hierarchy, and so on.

Although this approach increases the total number of atomic propositions in the system, the hope is that most nodes do not lie on many lower-level trees, and thus will only need to store forwarding state for the topmost atomic propositions(plus forwarding state for that node's specific branch of the hierarchy). In the worst case every node lies on all spanning trees and must maintain state for every atomic proposition in the system—including the (extra) atomic propositions added to create the hierarchy. In the best case, the forwarding state is only needed for a node's local branch in the hierarchy. In the Section V we investigate the trends in forwarding state on common network topologies.

As noted above, taking advantage of the existence of these "abstraction atomic propositions" requires that the forwarding framework be able to determine the relationships among atomic propositions. However, it is undesirable to embed information about specific relationships among atomic propositions in the forwarding framework; far better to encode such relationships generically, say in the way atomic propositions are identified. In particular, the existence of an abstraction relationship between two atomic propositions can be indicated by a prefix relationship between their respective identifiers. Thus for example a would be easily recognized as an abstraction of ax and ay.

Note that supporting hierarchical atomic propositions also allows the system to reduce the number of filters it has to maintain. For example, if a is an abstraction of ax, then we do not need to maintain (positive) filters for a on the ax tree since a is implied by ax.

V. EVALUATION

To better understand the performance and overhead costs of a general purpose routing protocol like speccast, we simulated two versions of the speccast service. The first is the basic two-layer service with the *filters* optimization; we call this version *non-hierarchical speccast*. The second version did not support filters, but did incorporate support for *hierarchical atomic propositions*. We call this version *hierarchical speccast*. Both implementations follow the descriptions presented in Section III.

A. Alternative Routing Approaches

To compare our speccast service against existing routing approaches, we simulated conventional network-level routing methods as well as emerging overlay routing approaches. First, we simulated a (standard) shortest path first (SPF) routing service, which forwards each packet along the shortest path to its destination. Second, in view of the popularity of overlay networks, we simulated two different classes of overlay networks.

One, the **Broker Overlay**, consists of a set of servers or "brokers" that assist in the routing of packets from sources to destinations. It is similar to content distribution and publish/subscribe systems in which content is published to the overlay and the data is then retrieved via the overlay (examples include systems like Siena [5], Intentional Naming System [6], and Gryphon [10]). We simulated the broker overlay as a set of servers. Clients that have content to publish inform a nearby broker (server) who advertises the content on behalf of the client. Similarly, a client in search of certain content will contact its nearest broker, asking the broker to retrieve the data on the client's behalf. Note that our brokers are "ideal," in the sense that every broker knows the exact content available at every other broker (i.e. we do not consider the cost of achieving this state of knowledge). When packets are routed from a source to a destination, they first go to the broker nearest the source. The broker, knowing the content available at all other brokers, forwards the packet to all brokers responsible for content "matching" the destination. The latter brokers, in turn, forward the packet to the end systems that match the destination. At each hop, the packet takes the shortest unicast path to its next hop (i.e., the overlay depends on an underlying unicast service for client-broker and broker-broker communication).

To compare to more structured overlay approaches, we also simulated a network in which servers are arranged in a regular logical topology, with well-defined routes between nodes in the overlay. Examples of this type of overlay include Chord [1], CAN [2], Tapestry [3], and Pastry [11]. We use **Tapestry** [3] as the representative structured overlay. In Tapestry each node is assigned a unique ID, which is structured as a number written in some base b. The number of digits needed in the ID is $d = \log_b(N)$, where N is the number of nodes in the graph and b is the base. Each node maintains a routing table of size

$d \times b$. Entry i ($0 \leq i < b$) at level k ($0 < k \leq d$) contains the IP address of the "closest node" whose first k digits in its ID match current node's ID, and whose digit at the $k+1$st position is i. Packets are initially routed using the ith entry in level 1 at sender. The second hop then consults the jth entry at level 2, where j is the second digit in the address. Forwarding continues like this for up to d hops, at which point the packet has reached the destination. If the source shares a k-digit prefix with the destination, the route lookup simply starts with the $k+1$st digit. In our simulations, each node's Tapestry routing tables always point to the *nearest* neighbor with any prefix; in reality this is the best case scenario for a Tapestry network.

B. The Experiments

To measure the performance and cost of the above approaches, we simulated four different network routing services based on speccast: a *unicast* service, a *multicast* service, a *multi-unicast* service, and a *random predicate* service. For each service, we measured the *network delay*, *network load*, and *network state*.

Unicast. To implement a unicast service using speccast, each node in the network was assigned a unique "address" (point predicate); random instances of these point predicates were then used as packet destinations. How the address was structured depended on the routing service being simulated. The following briefly describes the structure used by each routing service.

In our simulations, the Broker and SPF routing services both treat addresses as "flat" (unstructured) bit strings. Unfortunately, this implies that every node must maintain a shortest path route to every other node, causing the total amount of network state to grow as N^2, where N is the number of nodes in the network. In the case of the broker system, state is only saved at the brokers (which for our experiments was about 2% of the nodes in the graph) and thus the overall network state is relatively small despite the flat address space. (We disregard the state requirements for SPF, being mainly interested in its performance as a baseline for delay and network load.)

In both the speccast services and Tapestry, the address is regarded as a sequence of base b digits. Since we have a fixed number of nodes N, the number of digits needed in the address is $d = log_b N$. In the Tapestry simulation, these digits are used directly by the routing algorithm described above. In the non-hierarchical speccast simulation, we associate one atomic proposition with each pair (i, v)—where i is a digit position and v is one of b possible digit values—corresponding to the predicate "the ith digit is equal to v." Each address can thus be uniquely represented as a conjunction of d atomic propositions. Thus the total number of atomic propositions in the system is $b \times d$. This highlights one of the benefits of speccast: a large number of distinct groups of nodes can be identified using a logarithmic number of atomic propositions.

As the base b grows, the number of atomic propositions needed to uniquely identify N nodes also grows until the base reaches a point where N can be represented with smaller number of digits. At that point the amount of state drops and then grows again until d drops again. This effect can be seen in the network state graphs described later (Figure 8). To measure the effect b has on the network state, load, and delay, we varied b from 2 to 30 in our experiments (while keeping N constant and equal to the number of possible destinations in each test topology). In the hierarchical speccast simulations, the set of atomic propositions contains a single atomic proposition for each string of k digits, for k between 1 and d. These atomic propositions are arranged hierarchically according to the prefix-based method described earlier. So the attribute corresponding to the string 1223 has a parent attribute corresponding to the string 122.

Multicast. We considered an *any-source multicast service* implemented with speccast by using individual atomic propositions as the analog of multicast addresses. That is, to send a packet to the group, a sender simply addresses the packet to a particular atomic proposition, which is satisfied by all group members. Note that unlike conventional multicast, the speccast-based service could also be used to send packets to unions or intersections of groups. With attributes structured hierarchically as described above, groups could also be partitioned into subgroups, which could be useful for, say, subcasting. For the multicast service we compared speccast to a service (denoted SPF-T) that routes along a source-specific shortest path tree. Such a tree produces optimal delays (each packet follows the shortest path to each destination) and near-optimal loads.

Multi-Unicast. We simulated a *multi-unicast service*, which (like multicast) supports delivery to a set of nodes. However, to implement multi-unicast with speccast, the sender enumerates the set of destination nodes by formulating a destination predicate that is a disjunction of unicast (point) predicates. Thus, where the plain multicast service is better for large groups, the multi-unicast service is more suitable for groups consisting of just a few nodes (defined by senders on-the-fly). Again we use SPF-T as the basis for comparison, assuming that a single packet was sent and duplicated en route at branch points in the (source-specific) shortest path tree. In addition, for this service we simulated the use of multiple unicast packets, and also modified the other algorithms (Broker and Tapestry), to do multi-unicast. For Tapestry, packets were sent along multiple Tapestry paths; for the Broker system, the Brokers formed the internal nodes of the shortest path tree (optimized for delay rather than load).

Random Predicates. To assess the performance of speccast for arbitrary predicates, we randomly assigned atomic propositions to each node. We then formed destination predicates consisting of conjunctions (and disjunctions) of randomly selected attributes, and measured the cost of sending packets with those destinations from random nodes. For the conjunction tests, nodes were assigned 5 atomic propositions selected from a set of 30 atomic propositions. For the disjunction tests, each node was assigned 1 out of 200 possible atomic propositions, causing most atomic propositions to be satisfied by multiple machines. Again we used the SPF-T as the baseline for evaluation of delay and network load. We compared this service to multiple unicast and to the extended broker-based multicast service described above.

C. Topology and Locality

All experiments were run on three different transit-stub graphs generated using GT-ITM [12]. Each topology consisted of 3080 nodes comprising four transit domains averaging 14 nodes each, with each transit node connecting to 3 stub domains averaging 18 nodes each. (Thus most of the nodes in these graphs were in the stub domains.) In addition, each graph contained 12 extra transit-stub edges and 56 additional stub-stub edges; these extra edges increase the number of paths between stub nodes in the graph, and thus provide more diversity of routing.

Locality is a correlation between a node's location in the topology and the set of predicates satisfied by a node. With locality, knowing that node n satisfies atomic proposition a provides information about whether other nodes near n in the topology will also satisfy a. To evaluate the effect of locality on performance and overhead, we ran two versions of the unicast experiment using transit-stub. In one, we randomly assigned addresses (attributes) to nodes; as a result, the addresses of nodes near each other in the topology were independent (i.e., no locality). In the other version, addresses were assigned according to node numbers in the GT-ITM graph structure. Because of the way transit-stub graphs are constructed, nodes near each other in the topology have similar node numbers, and thus received similar addresses. As a result, nodes in the same network domain usually have a common address prefix. This effect is similar to the locality observed in the current Internet, where machines within a domain tend to share the same network prefix.

D. Experimental Results

We use the metrics described in Section II-C to evaluate each of the unicast service implementations described above. Figures 3 and 4 show the network delay stretch with and without locality, respectively. The curves in the figures show the ratio of the measured delay in hops to the optimal (shortest path) delay. Speccast with hierarchical attributes has the best delay other than the optimal SPF. Because packets are redirected out of their way (i.e., through two brokers), the broker service exhibits a slightly higher delay than hierarchical speccast. This delay, however, is sensitive to the number of brokers; as the number of nodes acting as brokers increases, the length of paths from source to destination decrease. It is also worth noting that for SPF, brokers, and speccast locality has relatively little impact on delay. The same is not true for Tapestry. Tapestry's hierarchical routing improves substantially with locality. Without locality, Tapestry sends packets back and forth across the network because nodes near one another in the hierarchy are not near one another in the topology.

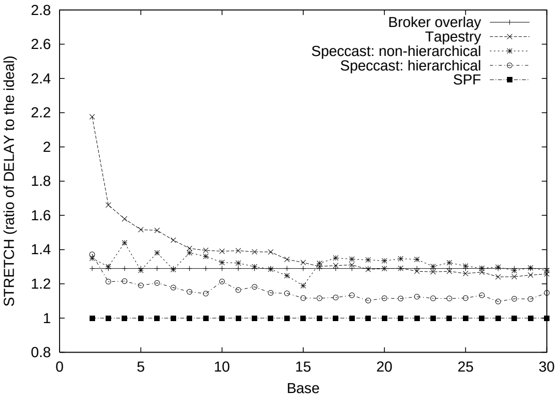

Fig. 3. Unicast: network delay with locality.

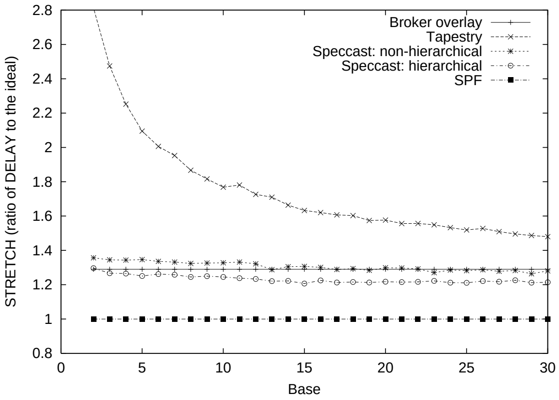

Fig. 4. Unicast: network delay without locality.

Given the similarities between speccast and Tapestry—both route according to the digit hierarchy—one might be surprised that speccast exhibits a generally lower delay than Tapestry. The explanation is found in the fact that speccast delivers the packet to the nearest spanning tree node and then forwards it along that tree. Tapestry, on the other hand, delivers the packet to the next node that matches the next digit in the destination; this node may be further away than the nearest spanning tree node. On average, hitting the tree as quickly as possible and forwarding along it does better.

Figures 6 and 5 show the network load imposed on the system by each algorithm. Unlike delay, the network load

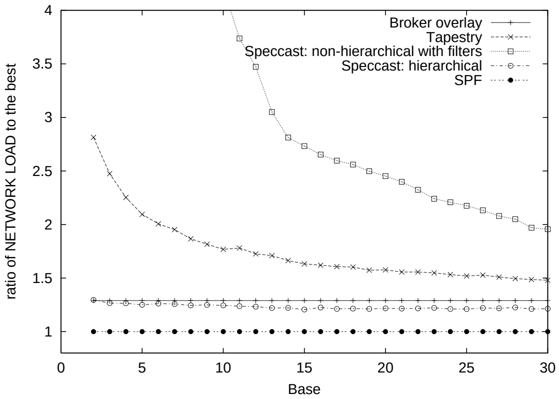

Fig. 5. Unicast: network load without locality.

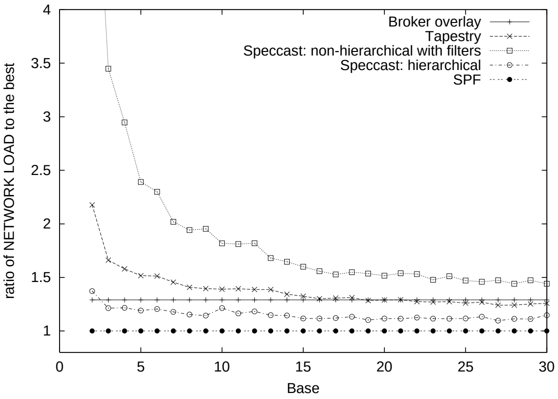

Fig. 6. Unicast: network load with locality.

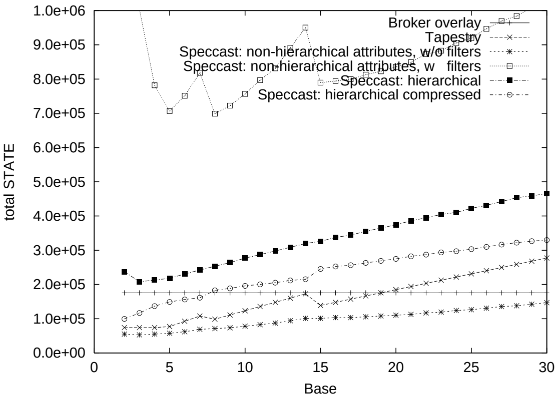

Fig. 7. Unicast: network state without locality.

is noticeably affected by locality. When nodes with similar "addresses" are near one another, the spanning trees used by speccast are smaller and are better approximations of the minimal spanning tree. The same is true of the overlay network created by Tapestry. Note that hierarchical speccast also out-performs Tapestry in network load. As we noted earlier, packets reach their destination more directly with speccast, but this does not necessarily imply lower network load. However, hierarchical atomic propositions *do* prevent the packet from flowing along multiple branches of a tree; instead, when the packet hits the tree that matches some prefix of the destination address, it flows only along the branch that leads to a longer matching prefix. Finally, note that even with filters, non-hierarchical speccast still pays a price in terms of network load. However, the problem is worse for small bases. For larger bases, non-hierarchical speccast comes within a factor of 2 of the optimal load, falling below 1.5 when locality is present.

Figures 7 and 8 show the network state consumed by each algorithm. Again, locality is a significant factor in the cost of speccast (as well as other approaches). Without locality, each atomic proposition tree contains many interior nodes that do not satisfy the atomic proposition, but which must maintain tree state anyway (i.e. at least two tree pointers instead of a single "gradient" pointer). With locality, the number of

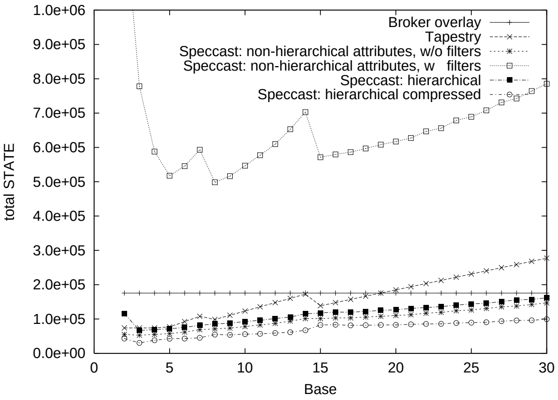

Fig. 8. Unicast: network state with locality.

TABLE I

UNICAST: PER-NODE STATE FOR SPECCAST

Number of Nodes	600	1480	3080
non-hierarchical, no locality	134	186	256
non-hierarchical, with locality	103	133	185
hierarchical, no locality	33	54	79
hierarchical, with locality	8	13	26

such nodes drops significantly, and total network state along with it. Tapestry, on the other hand, requires space $b \times d$ at all times, regardless of locality. Thus it compares quite favorably when there is no locality. The "non-hierarchical without filters" curve is included in these graphs to quantify the cost of the performance improvements afforded by filters in the non-hierarchical case (as plotted in earlier graphs). In the absence of locality, non-hierarchical speccast without filters requires even less state than Tapestry because speccast does not maintain state for unused atomic propositions while Tapestry maintains state for $b \times d$ entries even if some are not in use.

In view of these results, it is worth considering how the average per-node state grows with the number of nodes in the network in our speccast solution. Table I shows the average amount of state (table entries) per node required to provide unicast service (using base 15 addresses) for the four possible combinations of hierarchy and locality. In all cases, per-node state grows sub-linearly. This is not surprising, since the amount of state per node is proportional to the size of S, and it only needs to grow logarithmically to provide unicast service.

Figure 9 shows the network delay and network load when speccast is used for multicast delivery. The results confirm expectations: Because speccast constructs a shared distribution tree, it performs much like a core-based multicast tree [7], reducing the network load (ratio to SPF-T below 1) at the expense of longer delays (stretch > 1).

Figures 10 and 11 show the delay for multi-unicast services with and without locality.[3] As the number of destinations

[3]In these experiments point predicates were assigned to nodes as in the unicast experiments; base-15 addresses were used.

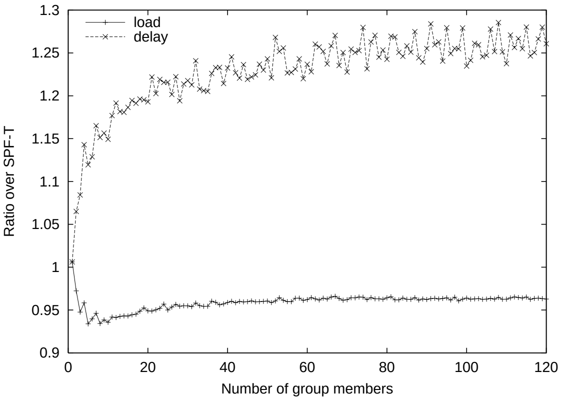

Fig. 9. Multicast: delay and network load.

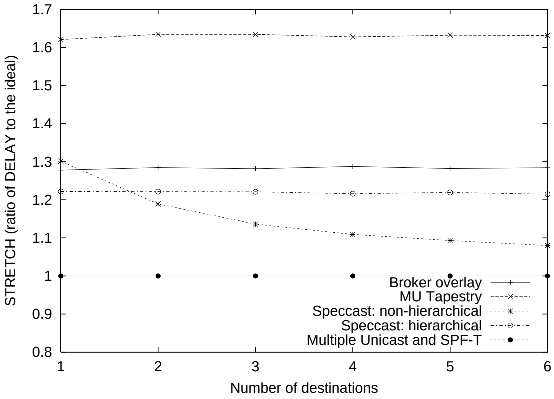

Fig. 11. Multi-unicast: network delay without locality.

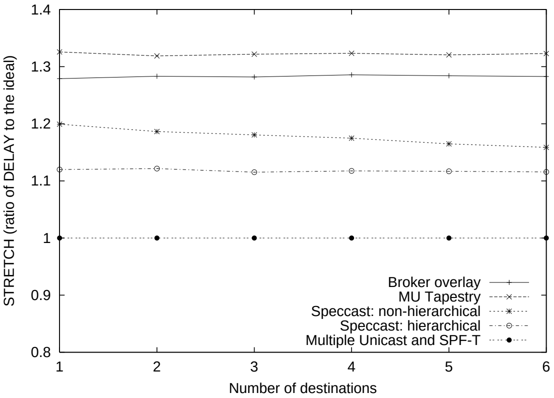

Fig. 10. Multi-unicast: network delay with locality.

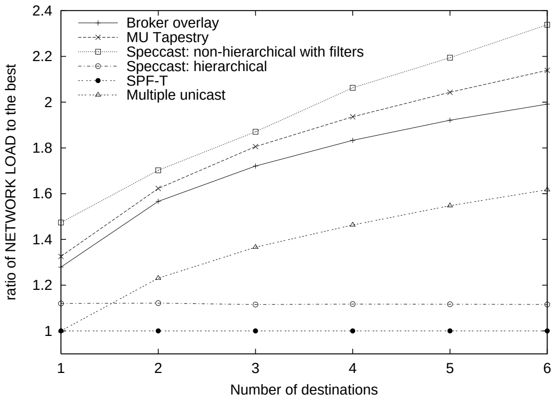

Fig. 12. Multi-unicast: network load with locality.

(disjuncts in the destination predicate) increases, the delay remains roughly constant for all algorithms. This is due, in part, to the small number of destinations with relatively direct (non-shared) routes to each destination The delay with non-hierarchical speccast actually decreases slightly because messages can jump from tree to tree, occasionally reaching the set of destination nodes more directly.

Figures 12 and 13 show the network load for multi-unicast. With locality, SPF-T and hierarchical speccast are able to capitalize on shared paths to the destinations, whereas all the other algorithms send a separate copy to every destination, resulting in a load that grows with the number of destinations. When the destinations are scattered (no locality), all methods basically end up sending independent messages to each destination. Non-hierarchical speccast is the most strongly affected by the presence or absence of locality. In all cases, however, the load grows more slowly as the number of destinations increases.

Figures 14–17 show the delay and network load for the random predicates experiments. For conjunctions, the network load under non-hierarchical speccast increases rapidly with the number of conjuncts; the reason is that filters are less effective for larger conjunctions. This is because a filter for atomic proposition b on the tree for atomic proposition a only tells of the existence (or not) along a branch of nodes satisfying both a and *one* other atomic proposition, say b. A packet destined for $a \wedge b \wedge c$ might therefore be forwarded down a branch of the a that contains $a \wedge b$-nodes and $a \wedge c$-nodes, but no $a \wedge b \wedge c$-nodes. Note that the load for multiple unicast actually decreases. This is due, in part, to the fact that the number of nodes satisfying a conjunction decreases rapidly as the size of the conjunction increases.

Finally, we consider the cost of the forwarding-time computation performed for speccast—*as we have implemented it*. We stress that these are not necessarily lower bounds. Assume the destination predicate D is a disjunction of k conjunctions, each having at most c conjuncts. Assume also that an atomic proposition has already been selected for each conjunction. For each of these, the node must determine on which outgoing interfaces the packet should be transmitted to progress it toward (or along) the tree for that atomic proposition. In the hierarchical case, this computation amounts to a longest-prefix match; thus the computational cost is comparable to that of the Internet Protocol (for each disjunct, i.e. times k). For the non-hierarchical case, the selected attribute for each disjunct must be looked up in a table; this can be a roughly constant-time operation assuming the number of atomic propositions is not too large or that hashing is a reasonable approach. If filters are used, in the worst case a similar constant-time lookup is

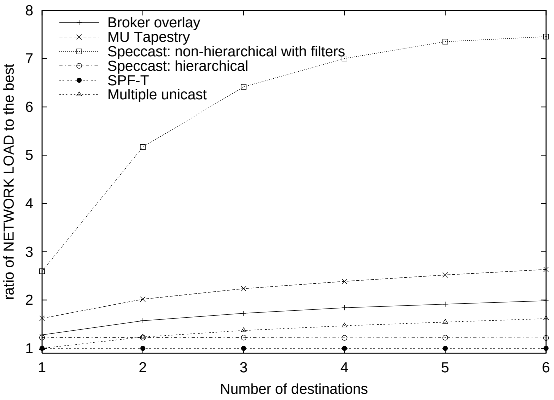

Fig. 13. Multi-unicast: network load without locality.

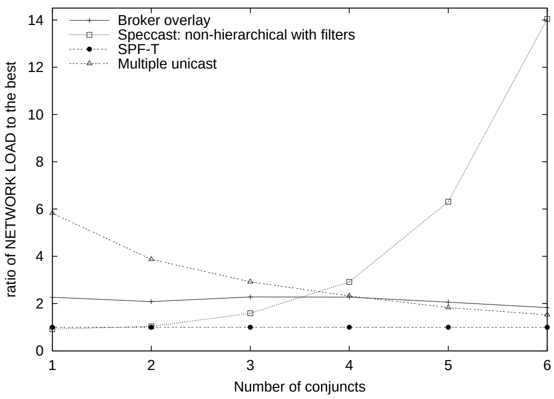

Fig. 15. Random predicates: network load vs. number of conjuncts.

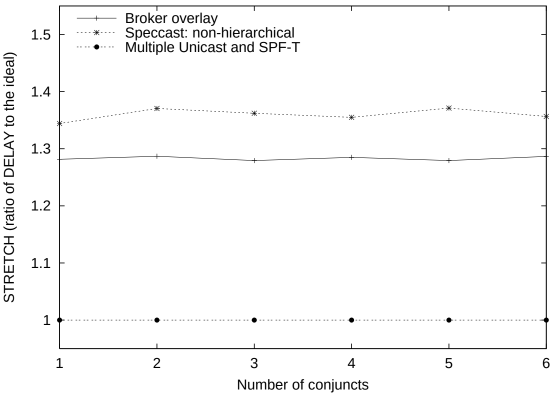

Fig. 14. Random predicates: delay vs. number of conjuncts.

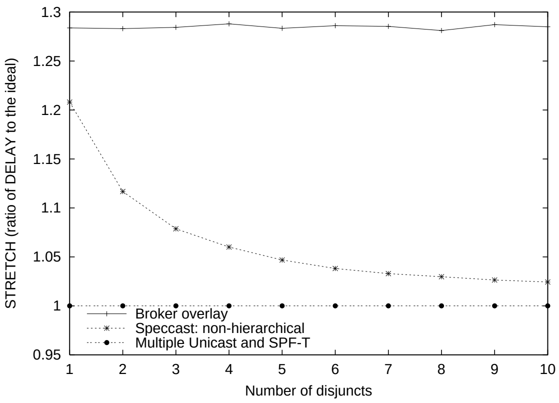

Fig. 16. Random predicates: delay vs. number of disjuncts.

incurred for each of the (up to) $c-1$ other atomic propositions in the conjunction, for each outgoing interfaces.

VI. Related Work

The work most closely related to speccast deals with so-called *publish-subscribe* systems. A number of such systems have been proposed [5], [13], [10]. Speccast resembles such systems in its use of an association between predicates and nodes to define the semantics of message delivery. However, their problem statement (as defined, for example, by Carzaniga and Wolf [14]) is different in that their predicates apply to datagrams rather than nodes. That is, they postulate a set **D** of datagrams, and a set **F** of predicates on datagrams. Associated with each node is a predicate (sometimes defined as a set of *subscriptions*), and the network is responsible for delivering to each node all datagrams that satisfy its predicate.

This problem is a dual of the speccast problem; given sufficiently powerful predicate languages in both systems, any pattern of source-to-destination deliveries that can be achieved using one service can be achieved using the other. Nevertheless, there are differences. One is that essentially all of the published work is overlay-based. Another is that the definition of unicast semantics is less clear in a framework in which point predicates are satisfied by exactly one *datagram*. The pub-sub model gives the receiver ultimate control over comes to it, while the speccast model gives the sender control.

Intentional multicast in INS [6] solves a problem similar to our speccast problem by routing in a shared-tree overlay. INS is a special case of the system that we simulate in the case of idealized broker overlay. Because of the tree connection of brokers, load and delay for such system will be significantly worse than in the idealized simulation.

A good deal of work on attribute-based routing has been done in the context of sensor and ad-hoc networks [15]. Many such systems [16] are designed for special-purpose networks that are designed to solve specific problems. Our goal was to design a generic service and evaluate its costs. We could use ideas from directed diffusion to solve the speccast problem. In [15], the first data packet sent to a destination is exploratory; its purpose is to trigger state setup for a flow of subsequent packets. Our goal was a system in which all packets are forwarded independently.

Internet Indirection Infrastructure (i3) [4] is another generic overlay service that can also subsume a number of existing services. It achieves such power by routing over distributed hash table (such as Chord [1]) using rendezvous points. To get to a destination, a packet may have to travel throughout multiple rendezvous points. Our speccast solution trades band-

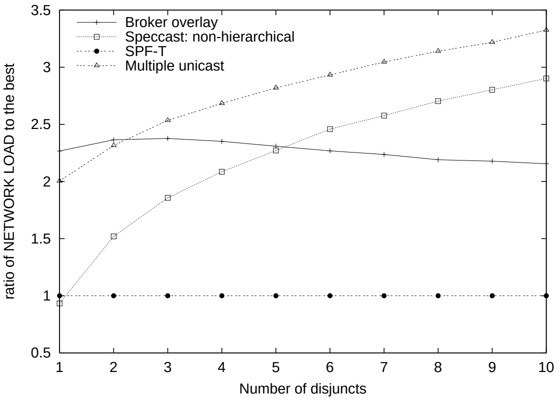

Fig. 17. Random predicates: network load vs. number of disjuncts.

width for reduced delay, sending packets along multiple direct paths to a set of destinations.

A possible avenue for optimization is to exploit information about the "density" of atomic propositions, i.e. the fraction of nodes that satisfy each one. This idea has previously been explored in the context of a still another pub-sub system [17]. The study explores different approaches of using multicast for broker-based publish-subscribe systems, and considers the effect of locality ("regionalism of attributes"). One of the main results is that when the locality is low and the number of nodes that share the same subscription (satisfy the atomic proposition, in our terminology) is not, it is easier and cheaper just to flood packets to all brokers.

VII. Conclusions

We have introduced Speccast, a very general form of routing. Our two-layer implementation, which supports positive disjunctive normal form combinations of atomic propositions, does not assume the existence of any underlying unicast or other service, and can support traditional unicast and multicast services at modest overhead cost in terms of delay, network load, and state. We believe the increase in flexibility obtained through speccast is worth the additional overhead costs. The implementation presented here makes very few assumptions about the structure of the predicate set; we believe overhead can be reduced even further, without loss of flexibility, through judicious structuring of the set of atomic propositions.

For example, one set of atomic propositions could be "assigned" to nodes in a manner that respects locality, while another set is assigned on a strictly semantic basis. As a result it becomes possible to address packets to nodes that fulfill a particular funcition, in a particular part of the network, for example. Some attributes might be assigned with respect to *geographic* locality. The beauty of speccast is that the same framework can be used for any predicate-structuring scheme.

In this paper we have assumed a static network. In addition, we have ignored the question of who or what determines which nodes satisfy which predicates. The next challenge is to adapt the approach presented here to deal with both dynamic networks and dynamic predicate sets. Such a capability will be required to make speccast practically deployable.

References

[1] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *SIGCOMM*, August 2001.

[2] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker, "A scalable content-addressable network," in *SIGCOMM*, August 2001.

[3] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE J. Select. Areas Commun.*, 2003.

[4] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana, "Internet indirection infrastructure," in *SIGCOMM*, August 2002.

[5] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf, "Achieving scalability and expressiveness in an Internet-scale event notification service," *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, vol. 20, 2000.

[6] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley, "The design and implementation of an intentional naming system," *ACM SOSP*, vol. 20, 1999.

[7] Kenneth L. Calvert, Ramesh Madhavan, and Ellen W. Zegura, "A comparison of two practical multicast routing schemes," Tech. Rep. GIT-CC-94/25, College of Computing, Georgia Institute of Technology, February 1994.

[8] Kenneth L. Calvert, James N. Griffioen, and Su Wen, "Lightweight network support for scalable end-to-end services," *ACM SIGCOMM*, 2002.

[9] Andrew Whitaker and David Wetherall, "Forwarding without loops in Icarus," in *Proceedings IEEE OPENARCH 2002*, June 2002, pp. 63–75.

[10] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Robert E. Strom Jay Nagarajarao, and Daniel C. Sturman, "An efficient multicast protocol for content-based publish-subscribe systems," *IEEE International Conference on Distributed Computing Systems*, pp. 262–, 1999.

[11] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[12] Kenneth L. Calvert, Matthew B. Doar, and Ellen W. Zegura, "Modeling Internet Topology," *IEEE Communications Magazine*, June 1997.

[13] G. Robert Malan, Farnam Jahanian, and Sushila Subramanian, "Salamander: A push-based distribution substrate for internet applications," *Usenix*, 1997.

[14] Antonio Carzaniga and Alexander L. Wolf, "Content-based networking: A new communication infrastructure," *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, 2001.

[15] John S. Heidemann, Fabio Silva, Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, and Deepak Ganesan, "Building efficient wireless sensor networks with low-level naming," in *Symposium on Operating Systems Principles*, 2001, pp. 146–159.

[16] Tomasz Imielinski and Samir Goel, "Dataspace - querying and monitoring deeply networked collections in physical space," *IEEE Personal Communications Magazine, Special Issue on Networking the Physical World*, 2000.

[17] Lukasz Opyrchal, Mark Astley, Joshua S. Auerbach, Guruduth Banavar, Robert E. Strom, and Daniel C. Sturman, "Exploiting ip multicast in content-based publish-subscribe systems," *Middleware*, pp. 185–207, 2000.