An Architecture for Secure Wide-Area Service Discovery

Todd D. Hodes, Steven E. Czerwinski, Ben Y. Zhao, Anthony D. Joseph, and Randy H. Katz
Computer Science Division
University of California, Berkeley
{hodes,czerwin,ravenben,adj,randy}@cs.berkeley.edu

The widespread deployment of inexpensive communications technology, computational resources in the networking infrastructure, and network-enabled end devices poses a problem for end users: how to locate a particular network service or device out of those accessible. This paper presents the architecture and implementation of a secure wide-area Service Discovery Service (SDS). Service providers use the SDS to advertise descriptions of available or already running services, while clients use the SDS to compose complex queries for locating these services. Service descriptions and queries use the eXtensible Markup Language (XML) to encode such factors as cost, performance, location, and device- or service-specific capabilities. The SDS provides a fault-tolerant, incrementally scalable service for locating services in the wide-area. Security is a core component of the SDS: communications are both encrypted and authenticated where necessary, and the system uses a hybrid access control list and capability system to control access to service information. Wide-area query routing is also a core component of the SDS: all information in the system is potentially reachable by all clients.

Keywords: service discovery

1. Introduction

The decreasing cost of networking technology and network-enabled devices is enabling the large-scale deployment of both [51]. Simultaneously, significant computational resources are being deployed within the network infrastructure, and this computational infrastructure is being used to offer many new and innovative services to users of these network-enabled devices. We define such "services" as applications with well-known interfaces that perform computation or actions on behalf of users. For example, an application that allows a user to control the lights in a room [23] is a service. Other examples of services are printers, fax machines, music servers, and web services such as the FreeDB.org CD database.

Ultimately, we expect that, just as there are hundreds of thousands of web servers, there will be at least hundreds of thousands of services available to end users. Given this assumption, a key challenge will be *locating* the appropriate service for a given task, where "appropriate" has a user-specific definition (e.g., cost, location, accessibility, etc.). Clients cannot be expected to track which services are running or to know which ones can be trusted. Thus, clients will require a directory service that enables them to locate the services that they are interested in using, and this service will have to address such issues as trustworthiness, secure access, (dis)trust management, endpoint mobility, complex query support, and scaling behavior. We have built such a platform, the Ninja[1] *Service Discovery Service* (SDS). The SDS enables clients to more effectively search for and use the services available via the network.

[1] The Ninja project is developing a scalable, fault-tolerant, distributed, composable services platform [19].

The SDS is a scalable, fault-tolerant, and secure information repository providing clients with directory-style access to all available services. The SDS can store many types of information, including descriptions of services that are available for execution ("unpinned" services), services running at specific hosts ("pinned" services), available service platforms, and passive data. The SDS supports both push-based and pull-based access; the former allows passive discovery, while the latter permits the use of a query model.

Service descriptions and queries are specified in eXtensible Markup Language (XML) [4], leveraging the flexibility and semantic-rich content of this self-describing syntax.

The SDS also plays an important role in helping clients determine the trustworthiness of services, and vice versa. This role is critical in an open environment, where there are many opportunities for misuse, both from fraudulent services and misbehaving clients. To address security concerns, the SDS controls the set of agents that have the ability to discover services, allowing capability-based access control, i.e., to hide the *existence* of services in addition to disallowing access to a located service.

As a globally-distributed, wide-area service, the SDS architecture addresses challenges beyond those that operate solely in the local area: network partitions, component failures, potential bandwidth limitations between entities, workload distribution, and application-level query routing between components.

This paper presents the design of the SDS, focusing on the architecture of the directory service, the security features of the system, and the wide-area query model. Section 2 describes the system design concepts. Section 3 discusses the SDS architecture and its security features. Section 4 discusses wide-area operation. Section 5 presents performance measurements from the SDS prototype implementation. Section 6 situates the work with a discussion of related systems. Finally, we summarize and mention future work in Section 7.

2. Design Concepts

The SDS system is composed of three main components: clients, services, and SDS servers. Clients want to discover the services that are running in the network. SDS servers enable this by soliciting information from the services and then using it to fulfill client queries. In this section, we will discuss some of the major concepts used in the SDS design to meet the needs of service discovery, specifically accounting for our goals of scalability, client and service mobility, support for complex queries, and secure access.

2.1. Announcement-based Information Dissemination

In a system composed of hundreds of thousands of servers and services, the mean time between component failures will be small. Thus, one of the most important functions of the SDS is to quickly react to faults. Additionally, we would like to support at least coarse-grained mobility of clients and services, allowing them to change the point where they connect into the system as they move.

The SDS addresses these issues by using soft state throughout the system [35]. Soft state is maintained through the combination of *periodic multicast announcements* as the primary information propagation technique, and information *caching* rather than reliable state maintenance in system entities. The caches are updated by the periodic announcements or purged based on the lack of them. In this manner, component failures and mobility are tolerated in the normal mode of operation (periodic sending and receiving) rather than addressed through a special recovery procedure [1]. The combination of periodicity and the use of multicast is often called the “announce/listen” model in the literature; it is appropriate where the weaker semantics of “eventual consistency” suffice (versus transactional semantics). The announce/listen model initially appeared in IGMP [9], and was further developed and clarified in protocols such as RTP/RTCP and the MBone Session Announcement Protocol [30]. Refinement of the announce/listen idea to provide for tolerance of host faults (leveraging multicast’s indirection within cluster computing environments [2]) appeared in the context of the AS1 “Active Services” framework [1]. We will describe our specific use of announce/listen in Sections 3.1 and 3.2.

2.2. XML Service Descriptions

Rather than use flat name-value pairs (as in, e.g., the Session Description Protocol [22]), the SDS uses XML [4] to describe both service descriptions (the identifying information submitted by services) and client queries. XML allows the encoding of arbitrary structures of hierarchical named values; this flexibility allows service providers to create descriptions that are tailored to their type of service and that can be extended and “subtyped” through the use of multiple namespaces (schemas).

```
<?xml version="1.0"?>
<printcap>
   <color>yes</color>
   <postscript>yes</postscript>
</printcap>
```

(A)

```
<?xml version="1.0"?>
<!doctype printcap system
  "http://www/˜ravenben/printer.dtd">
<printcap>
   <name>print466; lws466</name>
   <location>466 soda</location>
   <color>yes</color>
   <postscript>yes</postscript>
   <duplex>no</duplex>
   <rmiaddr>http://joker.cs/lws466</rmiaddr>
</printcap>
```

(B)

```
<?xml version="1.0"?>
<!doctype printcap system
  "http://www/˜ravenben/printer.dtd">
<printcap>
   <name>lws720b</name>
   <location>720 soda</location>
   <color>yes</color>
   <postscript>n/a</postscript>
   <duplex>yes</duplex>
   <rmiaddr>http://ant.cs/lws720b</rmiaddr>
</printcap>
```

(C)

Figure 2. (A) an example XML query, (B) a matching service description, and (C) a failed match.

Valid service descriptions have a few required standard parameters, while allowing service providers to add service-specific information – e.g., a printer service might have a color tag that specifies whether or not the printer is capable of printing in color. An important advantage of XML over name-value pairs is the ability to validate service descriptions against a set schema, in the form of Document Type Definitions (DTDs). Unlike a database schema, DTDs provide flexibility by allowing optional validation on a per tag granularity. This allows DTDs to evolve to support new tags while maintaining backwards compatibility with older XML documents.

Services encode their service metadata as XML documents and register them with the SDS. Typical metadata fields include location, required capabilities, timeout period, connection protocol, and contact address/port. Clients specify their queries using an XML template to match against, which can include service-specific tags. A sample query for a color Postscript printer and its matching service description are presented in Figure 2.

2.3. Privacy and Authentication

The SDS assumes that malicious users may attack the system via eavesdropping on network traffic, endpoint spoofing, replaying packets, making changes to in-flight packets (e.g., using a “man-in-the-middle” attack to return fraudulent information in response to requests), and the like. To thwart such attacks, privacy is maintained via encryption of all information sent between system entities (i.e., between clients and SDS servers and between services and SDS servers). To reduce the overhead of the encryption, a traditional hybrid of asymmetric and symmetric-key cryptography is used – an long-lived asymmetric key is used to deliver a per-session symmetric key.

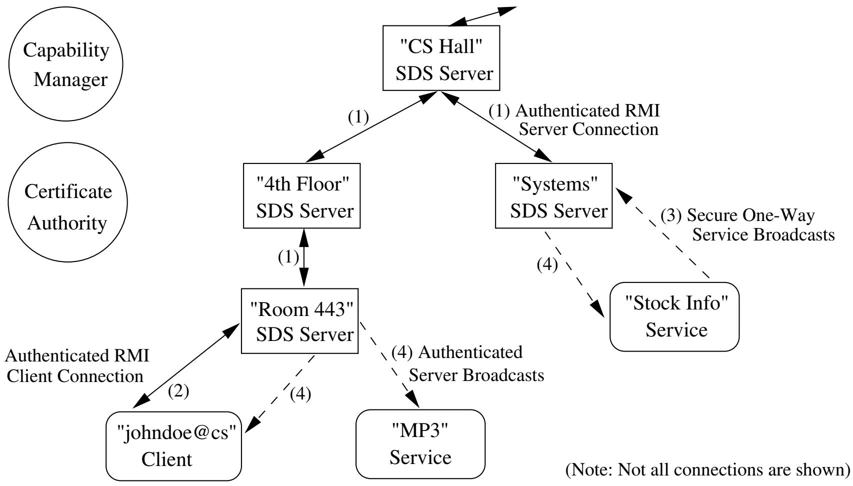

Figure 1. Components of the Service Discovery Service. Dashed lines correspond to periodic multicast communication between components, while solid lines correspond to one-time Java RMI connections.

However, encryption alone is insufficient to prevent fraud. Thus, the SDS uses cryptographic methods to provide strong authentication of endpoints. Associated with every component in the SDS system is a principal name and public-key certificate that can be used to prove the component's identity to all other components (see Section 3.3.1). By making authentication an integral part of the SDS, we can incorporate trust levels into the process used by clients to locate useful services. Clients can specify the principals that they both trust and have access to, and when they pose queries, a SDS server will return only those services that are run by the specified, trusted principals.

For example, the official network support staff in a computer science department could maintain an official CS Division principal. All the services they maintain, from printers to email servers, would be signed using this principal. Whenever clients perform searches, they could specify that they only desire services run by the CS Division principal, ensuring that only the official email servers and printers would be returned. This prevents them from accidentally connecting to services run by individuals in the department. Of course, they can make use of these service by simply including those individual's principals in their searches.

The SDS also supports the advertisement and location of *private* services, by allowing services to specify which "capabilities" are required to learn of a service's existence. Capabilities are signed messages indicating that a particular user has access to a class of services. Whenever a client makes a query, it also supplies the user's capabilities to the SDS server. The SDS server ensures that it will only return the services for which the user has valid capabilities. Section 3.3.2 elaborates on the use of capabilities.

Section 3.3 provides details of our use of authentication and encryption in the architecture, while Section 5.1.1 presents our measurements of the cost of these security components.

2.4. Hierarchical Organization

As a scalability mechanism, SDS servers organize into multiple shared hierarchies. Figure 1 illustrates an example configuration with a single hierarchy. Service announcers and queriers dynamically discover some server in the hierarchy and interact with the entire system through it, similar to DNS [32]. The coverage of a particular SDS server is called a "domain," and it is defined as a list of CIDR address ranges that can change with time.

In addition to providing a structure for the neighbor relationships between running servers, hierarchical organization also provides a mechanism for shedding server load – if a particular SDS server is overloaded, a new SDS server can be spawned on a nearby machine (if available), assigned to be a child of the overloaded server, and allocated a portion of the network extent and, thus, a portion of the load.

Discussion of hierarchical organization is treated in Section 4.3.

3. Architecture

Figure 1 illustrates the architecture of the Service Discovery Service, which consists of five components: SDS servers, services, capability managers, certificate authorities, and clients. In the following sections, we describe the components that compose the SDS, focusing on their roles in the system and how they interact with one another to provide SDS system functionality.

3.1. SDS servers

3.1.1. Basic operation

Each server is responsible for sending authenticated messages containing a list of the domains that it is responsible for on the well-known SDS multicast channel. These domain advertisements contain the multicast address to use for sending service announcements, the desired service announcement rate, and contact information for the Certificate Authority and Capability Manager (described in Sections 3.3.1 and 3.3.2). The messages are sent periodically using announce/listen. The aggregate rate of the channel is set by the server administrator to a fixed fraction of total available bandwidth; the maximum individual announcement rate is determined by listening to the channel, estimating the message population, and from this estimate, determining the per-message repeat rate, ala SAP [30] and RTCP [45]. (SDS servers send this value out as a part of their advertisements so individual services do not have to compute it.) Varying the aggregate announcement rate exhibits a bandwidth/latency trade-off: higher rates reduce SDS server failure discovery latency at a cost of more network traffic. Using a measurement-based periodicity estimation algorithm keeps the traffic from overloading the channel as the number of advertisers grows, allowing local traffic to scale.

3.1.2. Cluster operation and fault tolerance

SDS servers can utilize local computer clusters to address coarse-grained load balancing and add robustness to node failures. In the case of load balancing, when the service load reaches a certain threshold on an SDS server, it can optionally spawn a new child server. The new server is assigned to be a child of the parent in one or more hierarchies, and is allocated a portion of the existing load by accepting a fraction of the parent's network extent. In the case of fault tolerance, nearby servers that share multicast connectivity act as mirrors, sharing local multicast state updates. If a server goes down, a peer will notice and, silent to the clients and services, take over [1].

If a server with no transparent backups goes down, its neighbors will notice the lapse in heartbeats and optionally attempt to restart it (possibly elsewhere if the node itself is no longer available). Restarted servers populate their databases by listening to the existing service announcements, thereby avoiding the need for an explicit recovery mechanism. Additionally, because registered services are still sending to the original multicast address while this transition occurs, the rebuilding is transparent to them. If more than one server goes down, recovery will start from the top of the hierarchy and cascade downwards using the regular protocol operation.

In the case of a network partition, a parent will detect the loss of its child's heartbeats and either start a new child to serve the child's domain or add the child's domain to its own announcements. It will think the child has crashed even though it has not. The disconnected child will attempt to find a new parent. If it finds one, it will graft onto the hierarchy at this new point, and if not, it simply continues operating as before. Clients and services will continue to use the running server on their side of the partition, possibly after a delay of one or more announcement periods for those transitioning to the newly-spawned child or to the parent (i.e., they need to hear either a new or modified announcement). Operation continues as usual until the network partition heals. At this point, there will be two servers advertising overlapping network extent, possibly with different parents. This is detected either when these servers hear each other's announcements on the bootstrap address, or when a child hears two overlapping announcements. (Clients will be the only ones able to detect this when the servers are using directed broadcast rather than multicast to serve multiple subnets, as is done with BOOTP, DHCP, and the like.) At this point, based on their combined load, they either elect one to be a transparent mirror (as described above) or they split the domain into non-overlapping sections to service independently. The children may still not share a parent, but this doesn't affect the correctness of the protocol operation. Advanced hierarchy maintenance protocols can detect this non-optimal operating behavior at a coarse time scale and adapt to it by notifying particular servers to change their network extent; while we have not defined such a process, it can be implemented using the existing protocol mechanisms.

3.1.3. Accepting services and clients

An SDS server's domain is specified as a list of CIDR network address/mask pairs. This syntax allows for complete flexibility in coverage space while providing efficient representation when domains align to the underlying topology. Once an SDS server has established its own domain, it begins caching the service descriptions that are advertised in the domain. The SDS server does this by decrypting all incoming service announcements using the *secure one-way service broadcast* protocol (see Section 3.3.4), a protocol that provides service description privacy and authentication. Once the description is decrypted, the SDS server adds the description to its database and updates the description's timestamp. Periodically, the SDS flushes old service descriptions based on the timestamp of their last announcement. The flush timeout is an absolute threshold which currently defaults to five times the requested announcement period.

The primary function of the SDS is to answer client queries. A client uses Authenticated RMI (Section 3.3.5) to connect to the SDS server providing coverage for its area, and submits a query in the form of an XML template along with the client's capabilities (access rights). The SDS server uses its internal XSet [55] XML search engine to search for service descriptions that both match the query and are accessible to the user (i.e., the user's capability is on the service description's ACL). Depending upon the type of query, the SDS server returns either the best match or a list of possible matches. In those cases where the local server fails to find a match, it forwards the query to other SDS servers based on its wide-area query routing tables as described in Section 4.

Note that SDS servers are a trusted resource in this architecture: services trust SDS servers with descriptions of private services in the domain. Because of this trust, careful security precautions must be taken with computers running SDS servers — such as, e.g., physically securing them in locked rooms. On the other hand, the SDS server does not provide any guarantee that a "matched" service correctly implements the service advertised. It only guarantees that the returned service description is signed by the certificate authority specified in the description. Clients must decide for themselves if they trust a particular service based on the signing certificate authority.

3.2. Services

Services need to perform three tasks in order to participate in the SDS system. The first task is to continuously listen for SDS server announcements on the global multicast channel in order to determine the appropriate SDS server for its service descriptions. Finding the correct SDS server is not a one-time task because SDS servers may crash or new servers may be added to the system, and the service must react to these changes.

After determining the correct SDS server, a service then multicasts its service descriptions to the proper channel, with the proper frequency, as specified in the SDS server's announcement. The service sends the descriptions using authenticated, encrypted one-way service broadcasts. The service can optionally allow other clients to listen to these announcements by distributing the encryption key.

Finally, individual services are responsible for contacting a Capability Manager and properly defining the capabilities for individual users (as will be described in Section 3.3.2, below).

3.3. Security Components

3.3.1. Certificate Authority

The SDS uses certificates to authenticate the bindings between principals and their public keys (i.e., verifying the digital signatures used to establish the identities of SDS components). Certificates are signed by a well-known Certificate Authority (CA), whose public key is assumed to be known by everyone. The CA also distributes *encryption key certificates* that bind a short-lived encryption key (instead of a long-lived authentication key) to a principal. This encryption key is used to securely send information to that principal. These encryption key certificates are signed using the principal's public key.

The operation of the Certificate Authority is fairly straightforward: a client contacts the CA and specifies the principal's certificate that it is interested in, and the CA returns the matching certificate. Since certificates are meant to be public, the CA does not need to authenticate clients to distribute the certificate to them; possessing a certificate does not benefit clients unless they also possesses the private key associated with it. Accepting new certificates is also simple, since the certificates can be verified by examining the signatures that are embedded within the certificates. This also means the administration and protection of the Certificate Authority does not have to be elaborate.

3.3.2. Capability Manager

The SDS uses capabilities as a hybrid access control mechanism to enable services to control the set of users that are allowed to discover their existence. In traditional access control, SDS servers would have to talk to a central server to verify a user's access rights for each search. Capabilities avoid this because they can be verified locally, eliminating the need to contact a central server each time an access control list check is needed.

A capability proves that a particular client is on the access control list for a service by embedding the client's principal name and the service name, signed by some well-known authority. To aid in revocation, capabilities have embedded expiration times.

To avoid burdening each service with the requirement that it generate and distribute capabilities to all its users, we use a Capability Manager (CM) to perform the function. Each service contacts the CM, and after authentication, specifies an access control list (a list of the principal names, as described in Section 2.3, of all clients that are permitted access to the service's description). The CM then generates the appropriate capabilities and saves them for later distribution. Since the signing is done on-line, the host running the CM must be secure. Capability distribution itself can be done without authentication because capabilities, like certificates, are securely associated with a single principal, and only the clients possessing the appropriate private key can use them.

3.3.3. Authenticated Server Announcements

Due to the nature of SDS servers, their announcements must have two properties: they must be readable by all clients and non-forgeable. Given these requirements, SDS servers sign their announcements but do not encrypt them. In addition, they include a timestamp to prevent replay attacks.

3.3.4. Secure One-way Service Description Announcements

Protecting service announcements is more complicated than protecting server announcements: their information must be kept private while allowing the receiver to verify authenticity. A simple solution would be to use asymmetric encryption, but the difficulty with this is that asymmetric cryptography is extremely slow. Efficiency is an issue in this case, because SDS servers might have to handle thousands of these announcements per hour. Using just symmetric key encryption would ensure suitable performance, but is also a poor choice, because it requires both the server and service to share a secret, violating the soft-state model.

Our solution is to use a hybrid public/symmetric key system that allows services to transmit a single packet describing themselves securely while allowing SDS servers to decrypt the payload using a symmetric key. Figure 3 shows the packet format for service announcements. The *ciphered secret* portion of the packet contains a symmetric key (S_K) that is encrypted using the destination server's public encryption key (E_K). This symmetric key (S_K) is then used to encrypt

ID	Ciphered Secret	Payload
Sender Name	$\{$Sender, Destination, Expire, S_K, Sign(C_P)$\}_{E_K}$	$\{$Data, Time, MAC$\}_{S_K}$

Figure 3. Secure One-Way Broadcast Packet format: S_K – shared service-to-server secret key, Sign(C_P) – signature of the ciphered secret using the service private key, E_K – server public key, and MAC – message authentication code.

the rest of the packet (the data payload).

To further improve efficiency, services change their symmetric key infrequently. Thus, SDS servers can cache the symmetric key for a particular service and avoid performing the public key decryption for future messages for the lifetime of the symmetric key. Additionally, if the service desires other clients to be able to decrypt the announcements, the service needs only to distribute S_K.

The design of one-way service description announcements is a good match to the SDS soft-state model: each announcement includes all the information the SDS server needs to decrypt it.

3.3.5. Authenticated RMI

For communication between pairs of SDS servers and between client applications and SDS servers, we use *Authenticated Remote Method Invocation* (ARMI), as implemented by the Ninja project [52]. ARMI allows applications to invoke methods on remote objects in a two-way authenticated and encrypted fashion. The choice of ARMI is a function of our use of Java and orthogonal to the system design; the necessary functionality can be mapped onto other secure invocation protocols.

Authentication consists of a short handshake that establishes a symmetric key used for the rest of the session. As with the other components in the SDS, ARMI uses certificates to authenticate each of the endpoints. The implementation also allows application writers to specify a set of certificates to be accepted for a connection.

The performance of ARMI is discussed in Section 5.

3.4. Bootstrapping

The SDS bootstrapping technique is analogous to "foreign agent solicitation" and "foreign agent advertisement" in Mobile IP [33] extended beyond a single local subnet. Clients discover the SDS server for their domain by listening to a well-known SDS global multicast address. Our assumption is that all participating subnets will be covered by some SDS server that has multicast connectivity to its potential clients; in the case where a server does not have multicast connectivity to some portion of its network extent, it will try directed broadcasts to those subnets. If these are filtered (due to their potential use in denial-of-service attacks), affected clients will only be able to use manually specified or previously-discovered SDS servers. Alternatively or additionally, as an optimization, a client can solicit an asynchronous SDS server announcement by using expanding ring search (ERS) [10]: TTL-limited query messages are sent to the SDS global multicast address, and the TTL is increased until there is a response.

4. Wide-Area Support

The previous section detailed the local interactions of SDS servers, clients, and service advertisers. In this section, we describe our approach to server-to-server interaction. In this regime, the key problem is scaling with respect to the number of service descriptions and queries in the system.

We begin with a discussion of the basic problem posed by distributed multi-criteria search, and use this to motivate our approach to addressing this issue, a hierarchical query filtering infrastructure.

4.1. The Challenge of Multi-criteria Search

One novelty of the SDS is that it attacks a more difficult problem than other lookup infrastructures. This is due to the allowance for multi-criteria selection in queries (i.e., arbitrary sets of attribute-value pairs rather than a single element in a flat or hierarchical namespace), and the fact that these complex queries are allowed to transit the entire global Internet during resolution. Multiple existing systems present solutions for either complex queries or wide-area distribution independently; few address both.

Many popular service location schemes do not attempt to address wide-area distribution – e.g., Jini's Lookup service [50] and the IETF Service Location Protocol (SLP) [21].[2] Location schemes for name lookup that do provide global-scale operation can be dissected into categories based on their approach to query routing and their support (or lack thereof) for multi-criteria selection. These categories are Centralization, Mapping, and Flooding, and we describe the general principles of each in turn.

Centralization: Schemes that use centralization include Napster [16] and Web search engines. The scheme enables multi-criteria search, and can be scaled up through the use of computer clusters connected by fast LAN or SAN networks [17]. Unfortunately, though, this elegant approach suffers known problems: the cluster is a single point of failure, a single point of litigation (i.e., must secure legal rights to the data it is processing), and has an inherent single owner, which forbids sharing between entities that are unwilling to trust one another with their data.

Name-specified mapping to neighbor(s): Given the limits of centralization, schemes such as Globe [49], OceanStore's Tapestry [56], Chord [47], Freenet [6], and Data-

[2] A deprecated SLP extension [40] does attempt to provision for "cross-domain brokering," but does not give any indication of how to scale such an approach.

Space [24] permit data to remain distributed and partitioned, using some scheme to decide where to pass a query given the name to be resolved. A popular scheme for providing these mappings is *hashing,* e.g., Consistent Hashing [26]. These mappings are 1-to-M, where M is small, thereby giving a namespace-determined, deterministic mapping from a name to a set of nodes. This provides a natural partitioning of the system data, and thus query and inter-server message traffic is carefully managed: only a small number of endpoints are given a query, and together they can unequivocally respond with a negative or positive response.

The problem with namespace-based mapping is that it cannot provide multi-criteria selection. The intuition validating this claim is as follows. Assume that each document in the system is assigned to a unique partition based on some name-based mapping scheme. Without loss of generality, assume documents satisfying Criteria_1 maps to Nodes_1 and Criteria_2 maps to Nodes_2. Now consider a document satisfying both Criteria_1 and Criteria_2. For queries containing either Criteria_1 or Criteria_2 to return correct results, the documents would have to live at both Nodes_1 and Nodes_2, violating the non-duplication assumption. Thus, our only alternative is that $\text{Nodes}_1 = \text{Nodes}_2$. Taking this a step further, the transitive closure of overlapping criteria form cliques, and these cliques must all live at the same set of nodes. In other words, if Doc_1 satisfies Criteria_1 and Criteria_2, and Doc_2 satisfies Criteria_2 and Criteria_3, and Doc_3 satisfies Criteria_3 and Criteria_4, all documents Doc_1, Doc_2, and Doc_3 must be colocated, greatly constraining our ability to partition data. In the worst case, a certain criteria could be very popular and thus force most documents to one set of nodes. One way around this is by unnaturally biasing toward one criteria, and requiring all queries to contain it, as is done in DataSpace. Another way is to allow documents to reside in multiple partitions. In this latter case, though, using a similar argument as that above, each document in a clique would have to be duplicated at each related node, leading to excessive duplication. This defeats the purpose of partitioning.

Thus, the implication of supporting multi-criteria selection is that there is no natural data partition. Lack of partitioned data leads us to the next technique.

Flooding: An approach that avoids the listed limitations of centralization and mapping is flooding, the technique used by Gnutella [18] and link-state IP routing protocols [31]. Flooding addresses the lack of controlled data partitioning by sending queries to all nodes in the system. This has been shown to work at the "enterprise" level, and to a limited degree beyond that, but there are inherent limitations to the scalability of such an approach: the least-provisioned links limit the ability to propagate messages through the rest of the system [7,38]. This is not a problem for inter-domain IP routing table maintenance because the workload is controlled through the specification of the update periodicity. Location infrastructures cannot similarly bound the workload because it is not a system parameter – queries are user-generated.

Other strategies use a hybrid of one or more of these approaches. For example, the stalwart DNS [32] hybridizes mapping and centralization: data is partitioned, and names are mapped hop-by-hop based on name suffixes, while reliable "base pointers" for all names are centralized (at the root servers). The scheme works well through the use of extensive positive and negative caching and by keeping update rates low.

4.2. A New Approach: Query Filtering

We have now summarized the three classes of location techniques and their shortcomings. In the design of the SDS we have made a design decision that, in steady-state conditions, an advertised service should be found by a matching query. We call this property *full reachability.* This enables clients to access all services in all SDS servers, modulo access control provisions. Additionally, the SDS provides support for the type of multi-criteria selection enabled by local-area, centralized approaches. Given these decisions, an obvious next question is: how do we support this feature set in a manner that scales better than flooding?

Our answer is an approach called filtered query flooding, or more simply, *query filtering.* It hybridizes flooding, mapping, and when used in a hierarchy, centralization. There are two key ideas here. First, instead of using only a pull-based protocol, where a query initiates an exchange of information, we can also apply a push model, where state information is reported to nodes in the system via proactive *update* messages. Second, instead of proactively filling nodes with cached query responses from the information in updates, we instead propagate *summaries* of node contents, which are used as *filters* that are applied to queries. In this sense, updates are filter state updates rather than data cache updates.

A third idea is that, when used with nodes organized in a hierarchy, the approach utilizes centralization. Summaries are collapsed and aggregated as they move farther from their source, eventually all culminating at the top of the hierarchy. The centralization is not a requisite feature, though, only a byproduct of its use with a tree topology.

Filtered query flooding draws from existing approaches in its design. In the distributed database community, the notion of allowing data to be sent to the queries, in addition to vice-versa, is called "hybrid-shipping client-server query processing" or "cache investment" [27]. In the context of distributed Web caching, our approach looks like a combination of the use of the pull-based Internet Cache Protocol (ICP) [53] and push-based Cache Digests [41], modified to account for multiple-criteria queries.

To implement query filtering, we have to address its two major components: dynamic construction and adaptation of the neighbor relationships between SDS servers, and provisioning of an application-level filtering infrastructure allowing servers to propagate information through the topology. The information propagation problem can be further decomposed into two sub-problems: providing lossy aggregation of service descriptions as they travel farther from their source (setting up filter state along the way), and dynamically flooding client queries through the filters to the appropriate servers based on the local aggregate data. In short, we must build and use "query routing tables."

We now discuss our proposed solutions to these problems, and variations on the approach. We start with topology management, then cover information aggregation and query routing. With continue with details on range queries, wildcards, negative caching, and soft-state encoding of the system messages, and close with a description of our testbed. Experimental results are in Section 5.

4.3. Server Topology Management

The two most common topologies for peer-to-peer location systems are meshes and trees; all the systems discussed thus far in this paper use one or the other. OceanStore's architecture illustrates the tradeoffs and features of each through its separation of discovery into 1) a mesh-based probabilistic search, combined with 2) a deterministic approach that builds a hierarchy (tree) per data item inside a shared hypercube [28].

Following the example of DNS [32], the SDS runs over a set of hierarchical interconnections. In doing so, the SDS avoids the need for loop detection (which is left to be managed at a lower layer), and avoids the need for maintaining per-query state for unresolved queries – all path info is bundled into query metadata and passed along with it. In contrast, systems not guaranteed to be loop-free must either maintain a cache of unique identifiers for all queries that have been handled, and/or rely on decrementing a TTL field, in order to know when to drop queries. Additionally, the use of a tree for distribution provides an intrinsic notion of 'up,' thereby allowing us to (optionally) avoid maintaining filter state for one direction, a direction that is passed missed queries by default. The major disadvantage of a hierarchical topology is that a node cannot shortcut arbitrary combinations of paths, i.e., cannot be in two places in the hierarchy efficiently, as it could be in a mesh structure.[3]

Two key questions arise given the use of hierarchy: what trees to build, and how to construct them. The first question is a policy decision that we feel must be determined through experience rather than wired into the architecture; the second is a choice of mechanism that is dependent on the type of hierarchy to be maintained. Because the policy decision is left to be determined by operational experience, our solution to it is to allow for the use of *multiple* hierarchies, and thus support multiple policies. Examples of possible useful hierarchies include those based on administrative domains (school or company divisions), network topology (network hops), network metrics (bandwidth or delay), or geographic location (distance). The additional utility of supporting multiple hierarchies is that they are independently useful: users can choose to make queries that resolve based on a specific hierarchy, thereby allowing querying for a service based on, e.g., geographic proximity in one case and ownership in another. Additionally, as underlying network characteristics change, servers can gradually build new hierarchies aligned to the new circumstances, and transition to use of them.

Individual SDS servers participate in one or more of these hierarchies by maintaining separate pointers to parents and children for each hierarchy, along with any associated "routing table data" (described below) for each pointer. To guarantee that a query can reach all SDS servers, one particular hierarchy must be supported by all servers – the so-called "primary" hierarchy. Our current implementation uses an administrative primary hierarchy (called ADMIN), but a better choice would be one based on the underlying network characteristics – such as topology or bandwidth – because such a hierarchy requires no manual setup. Specification of a primary hierarchy is not a requirement for correct operation, but instead a optional, recommended way of supporting full reachability.

Our previous descriptions of SDS client/server operation does not address how parent/child relationships are determined. Examples of possible mechanisms for constructing these parent/child relationships include using manual specification in configuration files (e.g., to indicate administrative divisions), using geographic data (e.g., through the use of GPS or DNS LOC records [8]), using topological data (e.g., using topology discovery [29,37]), or using network measurements (e.g., using a tool such as SPAND [46] to derive bandwidth and latency information). A novel and robust approach for generating distribution trees (in our case, shared) is that taken by Gossamer [5]: build a resilient mesh at a lower layer, and run a routing protocol atop it to construct the trees. Leveraging the layering of the Gossamer protocol stack, with its clear distinction between Mesh Management, Routing, and Data Distribution, we can reuse their lower-layer functionality, replacing Gossamer's data distribution layer – which focuses on multipoint distribution – with our own for query and update distribution. Additionally, the SDS would benefit from replacing Gossamer's latency-based path metric with a bandwidth-based one.

Individual node failures can be tolerated in the same manner as is used to tolerate single-server failure in the local-area case: have multiple workstations listen in on the announce/listen messages and leverage the indirection to transparently select amongst themselves, a form of mirroring described in the Active Services framework [1].

4.4. Description Aggregation and Query Routing

Query filtering reduces the load on servers in the upper tiers of the hierarchy by localizing query traffic. Similarly, individual update operations are not propagated up the hierarchy; instead, information about these events is *summarized* into an "index." We call the summarization of service descriptions as they travel up the hierarchy "description

[3] Allowing 'cross-cutting paths' in our hierarchy – basically, additional connection and filter state between interior nodes – is a possible way to emulate mesh path shortcuts, but the utility of such an approach has not been verified.

Name	Description	Possible Responses
All-Pass/Null Filtering	no updates – equivalent to flooding	yes, false yes
Brokering	subset of list of service descriptions	yes, no, false no
Centroid-Indexed Terminals (CIT)	list of all tag values for each element	yes, no, false yes
Bloom-filtered Crossed Terminals (BCT)	criteria hashes put into a Bloom filter	yes, no, false yes

Table 1
Summary of Query Filtering Schemes. Filters determine whether to send queries through or turn them back. The 'Possible Responses' column indicates the nature of the information contained in the filter, any of four types: YES – can indicate a hit will occur if the query is sent through; FALSE YES – can indicate a hit will occur, while actually the query will result in a miss; NO – can indicate a miss will occur; FALSE NO – can indicate a miss will occur, while actually the query will result in a hit. "Terminals" and "Crossed Terminals" are defined in Section 4.4.1.

aggregation," and the process used to combine descriptions the *lossy aggregation* function of the hierarchy. We call the operation of iterating through the tree, comparing queries against the indices to determine whether the branch they are summarizing contains a match for that query, "query routing." Naturally, these operations are often inextricably combined, as the nature of the description aggregation defines how queries are routed.

Now that the context has been set, we can delve into some example query filtering schemes. A summary of these schemes is shown in Table 1. We start by describing the filtering scheme designed for use with the SDS – Bloom-filtered crossed terminals (BCT) – and then describe the others that we compare against it.

4.4.1. Bloom-filtered crossed terminals

The default SDS query filtering strategy is "Bloom-filtered crossed terminals" (BCT). BCT is based on the idea of breaking queries/services down to their constituent criteria, hashing them alone and in aggregate, and inserting these into a Bloom filter [3] to compress the list of hashes. The intuition and details are as follows.

Existing name-based mapping strategies hash an object identifier to decide its location (as might be done with URLs in web caching). Because we wish to locate services based on *subsets* of tags, just computing hashes over service descriptions and queries is not sufficient for correct operation: all possible matching query values hashes would have to be computed. To clarify the problem, imagine a service description with three tags. There are seven possible queries that should "hit" it: each tag individually, the three combinations of pairs of tags, and all three tags together. Each of these possible queries would need to be hashed and these hashes stored to guarantee correctness. There are obvious problems with computing all these possible hashes: the number of hashes scales exponentially with respect to the number of tags, and thus the amount of space required to store and transmit the hashes produced (seen as memory usage at local servers and update bandwidth on the network) would be excessive. Additionally, there is no way to bound the size of the resulting list.

Our solution to this problem is to limit the number of hashes by limiting the number of tag concatenations. We define the generation of hash entries from tags in terms of

$$\begin{aligned} CT_3 &= \bigcup_{i=1}^{3} \{A,B,C\}^i \\ &= \{A,B,C\} \cup \{A,B,C\}^2 \cup \{A,B,C\}^3 \\ &= \{A,B,C\} \cup \{AB,AC,BC\} \cup \{ABC\} \\ &= \{A,B,C,AB,AC,BC,ABC\} \end{aligned}$$

Figure 4. An example of computing the third-degree Crossed Terminal Set from a base terminal set (A, B, C).

a parameter that effectively trades an increased probability of false positives for hash-list size and vice versa. The parameter, N, is a measure of of the completeness of the tag concatenations relative to the original document. More formally, we define the initial base set of data from a description or query a *terminal set.* A terminal set is the linearization of an hierarchical XML document, comprised of the list of tags generated by walking from root-to-leaf for all nodes in the DOM tree [54] of the document. We then define a *Nth-degree crossed terminal set* as the set containing all combinations of elements from the terminal set of length less than or equal to N. Specifically: $\bigcup_{i=1}^{N} terminals^i$ where the product of set elements, $term_A \otimes term_B$, is defined as concatenation when $term_A < term_B$ and the empty set when $term_A \geq term_B$; comparisons are lexigraphic. An example is shown in Figure 4. Limiting the degree of the crossed terminal set (reducing N) increases the probability of false positives, but also limits the number of items to hash. Thus the degree addresses the exponential growth in a manner that gives a "knob" that can trade false positives for list size and vice-versa.

Incoming queries must be similarly broken up into crossed terminals (groups of tag combinations) and checked to avoid a false negative.

Given the use of crossed terminal sets, we would like to limit the total space that can be occupied by them. To do so, we insert them into a Bloom filter to compress them. The key property of Bloom filters is that they provide summarization of a set of data, but collapse this data into a fixed-size table, trading off an increased probability of false positives ("increased summarization") for index size – exactly the knob we need to address the issue of long hash lists. This use of Bloom filters is motivated by a similar use in Web caching [14]. A Bloom filter compresses

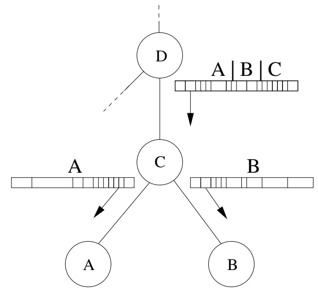

Figure 5. Aggregation of Bloom filters.

a list of data $d_1...d_n$ by using a given list of salts[4] $s_1...s_k$ to create a bit vector of length L. Bit x is set if and only if $hash(d_i + s_j) \bmod L = x$ for some i and j. Inclusion of data item d is queried by testing all the bit positions $hash(d + s_i) \bmod L$ for each $i \in 1...k$. If all are set, then the item is assumed to be a member of the compressed list of data, though it may or may not be (there can be false positive responses). If any are not set, then $d \notin d_1...d_n$. The basic probability of false positives (independent of table aggregation or degree of the crossed terminals that are hashed) can be reduced by using more salts and/or a longer bit vector [15]. This approach never causes false negatives, thereby maintaining the correctness of our lossy aggregation function in the face of the need for full reachability.

Now we explain how these ideas are applied to SDS query routing: upon receipt of a service announcement, an SDS server S_1 applies multiple hash functions (using keyed MD5 and/or groups of bits from a single MD5 depending on table size) to various subsets of tags in the service description (the crossed terminal set) and uses the results to set bits in a bit vector. The resulting bit vector (the filter) summarizes its collection of descriptions. This filter is given to neighbor S_2. When S_2 receives a query that it cannot resolve locally, it checks to see if the query should be forwarded to S_1 by similarly multiply hashing it and checking that all the matching bits are set in S_1's filter. If any are not set, then the service is definitely not there – it is a "true miss." If all are set, then either the query hit, or a "false positive" may have occurred due to aliasing in the table. The latter forces unneeded additional forwarding of the query, but does not sacrifice correctness.

If an SDS server is also acting as an internal node, it will have children. Associated with each child will be a similar bit vector. To perform index aggregation, each server takes all its children's bit vectors and ORs them together with each other and its own bit vector. This fixed-size table is passed to the parent (using a delta encoding to conserve bandwidth), who then associates it with that branch of the tree. This is illustrated in Figure 5.

To route queries, the algorithm is as follows: if a query is coming up the hierarchy, the receiving SDS server checks to see if it hits locally or in any of its children; if not, it passes it upward. If it is coming down the hierarchy, the query is checked locally and against the children's tables. If there is a hit locally, the query is resolved locally. If there is a hit in any of the children's tables, the query is routed down to the matching children either sequentially or in parallel. If neither of these occur, it is a known miss. We call this forwarding scheme *parent-based filtering (PBF)* because updates are propagated only up the hierarchy, not down to children. An implemented variation on the above, called *full indexing,* maintains filter information for parents in addition to children. In this case, the algorithm is simpler: instead of checking only children and then passing to the parent if they all miss, the server checks all neighbors' filters and acts accordingly.

A final problem to address: the bit vectors must be cleaned up when a service shuts down or moves – we would like to zero their matching bits. Bits cannot be zeroed directly, though, because another hash operation may have also set them, and zeroing them would not preserve full reachability (i.e., could cause a false negative). To address this, the tables must either be periodically rebuilt, or per-bit counts must be tallied and propagated along with the tables. We use per-bit reference counting, as is done in the Summary Cache [14] work.

4.4.2. Alternative filtering schemes

We now discuss the other three filtering schemes from Table 1.

The simplest possible filter is the "all-pass" or "null" filter, which lets everything through. This behavior is equivalent to flooding, as with Gnutella, except that due to the SDS's tree structure, queries eventually go to the root rather than circulate through a mesh. The benefits of a null filter is that no updates are required, while the disadvantage is that is promises maximal query load.

Another possible approach is "brokering." With brokering, a child decides some criteria for determining whether to pass service descriptions along to its parent. Those that do not match the criteria remain unreported, available only to those nodes locally attached (violating full reachability); those that do are sent in full to the neighbor. Depending on the selectivity of the criteria, this can arbitrarily reduce query load and update load. Also, passing the complete description is verbose – no compression occurs as the list grows – but it supports fast and correct operation (no false positives).

A more substantial filter is the "centroid-indexed terminals" scheme (CIT). The basis for this filter scheme is an approach for WHOIS++/LDAP server content trading called "centroids" [13]. Computing a centroid involves taking a list of key-value pairs and creating a concordance of all possible values for each key. An example is shown in Figure 6.

Because the SDS deals with hierarchical sets of key-value pairs (XML documents) instead of a flat list, we have implemented a modified form of this approach. To do so, we first create the terminal set (as defined above) of the service descriptions to be sent, and the centroid is then computed on

[4] The salts are used to produce multiple hash values from a single data value.

<first>Frank</first><last>Zappa</last>

<first>Moon</first><last>Zappa</last>

<first>John</first><last>Lennon</last>

⇓

<first>: Frank Moon John
<last>: Zappa Lennon

Figure 6. An example of computing the centroid of XML fragments from three documents. At top is the data to be summarized; below it is the resulting centroid.

the terminal set. The benefit of CIT is fact that updates decrease in size as they are aggregated (except in statistically unlikely worst-case workloads); the downside is that both aliasing (e.g., "Frank Lennon" in Figure 6) and the use of the terminal set can lead to false positives.

4.5. Range Queries, Wildcards, and Negative Caching

Various filtering techniques have different levels of support for searches with wildcards and/or range queries, e.g., those expressed as
<name comparison='*'>* Zappa</name> or
<size comparison='gt'>10</size>. Flooding, brokering, and CIT support both these query types naturally, with no additional performance degradation; BCT supports neither naturally. Full support for both of these more powerful query types is added by having forwarding nodes treat XML elements with the special comparison attributes differently: when making filtering decisions, the comparison attribute and element's value are elided. This maintains correctness but reduces the efficacy of the filters. (The attribute and value *are* used when querying against the cache of service descriptions, and this can done efficiently via XSet, XQL [39], XML-QL [11], or the like.)

BCT does not efficiently support such wildcarding or range queries because of a more general problem: it can't determine the cause of false positives. A way around this difficulty is to append information on known false positives (KFPs) to the metadata of failed queries. This technique is the equivalent of *negative caching* in the regime of query filtering; Mockapetris and Dunlap show the importance of such negative caching for name lookup in the context of the DNS [32]. KFP caching is implemented as follows. When returning a true miss, a server can optionally attempt to recognize the criteria combination that caused the false positive that got it there in the first place, and list it as a KFP on the response. Such a thing might occur due to the complexity of the query (having many common terms), the use of a wildcard, or the use of a range query. KFPs are cached on a per-filter basis, and used to allow more aggressive pruning of query propagation, and, more importantly, to address the problem of popular true misses.

KFPs cannot be passed further down a tree blindly because updates to neighbors are indications of the aggregate state of all outgoing links of a node, not the state of a single link. They can be passed, though, when all the other links in the node have either 1) a known negative or 2) an identical cached KFP – information that would be obtained after the first time a true miss is flooded throughout the tree.

4.6. Encoding Issues for Soft-State Messaging

Communications using a soft-state approach must not rely on state maintenance at the endpoints [35]. This means that either a complete set of information must be contained in application data units (ADUs), or information must be versioned and version mismatches must cause the soft-state cache to be flushed – the endpoints are implying that they no longer agree what the contents of any shared state may be. Following this model, the various soft-state encodings of the messages for wide-area query routing are as follows:

- *Updates:* In addition to including the deltas (differences) between the current table state and the previous table state, a fragment of the existing table is also included. The particular fragment changes with time, as can its size. The addition of these table fragments allows any errors or omissions in the local copy of the remote table to be eventually corrected – without requiring the endpoints to know the exact state of each other.
- *Queries:* Queries are inherently stateless, as all path information is maintained as metadata wrapped up along with the query. SDS servers along the path read and update this metadata at each hop to mark the progress of the query through the overall structure.
- *Query Replies:* Query replies, like queries, are basically inherently stateless – except for the optional inclusion of negative caching information in the form of known false positives (KFPs). To address this, KFP lists are encoded as deltas with associated version numbers. If a receiving server notes a jump in the version number that is not corrected via retransmissions, it flushes its cache of KFPs.

4.7. Summary of Node Internals

The complete operation of an SDS server node performing wide-area query filtering is summarized in Figure 7. The figure shows the path of queries as bold arrows and the path of filter updates at thin arrows. Query responses follow the reverse path of queries.

4.8. Testbed

We have implemented and simulated the components of our wide-area query routing solution and a suite of variations to better understand the design space. In addition to Bloom-based filtering (BCT), we have implemented all the filtering strategies from Table 1. In addition to parent-based query forwarding, we have implemented an update algorithm that propagates update information to all neighbors – "full" update forwarding. In addition to sequential (serial) query forwarding, we have implemented a query routing scheme that allows queries to "bifurcate" through the tree in parallel

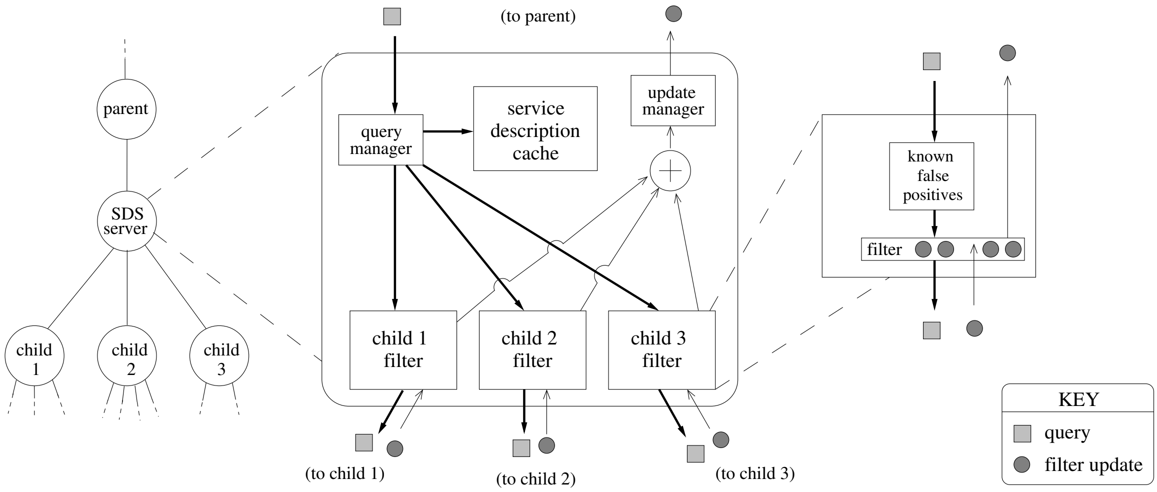

Figure 7. SDS Node Internals for a single hierarchy. Dark arrows are the path of a query; thin arrows are the path of filter updates. Replies follow the reverse path of queries. The service description cache returns hits; the known false positives cache and filter return misses.

rather than sequentially. Detailed quantitative results showing tradeoffs with the four indexing strategies are presented in Section 5.2.

5. System Performance

In this section, we examine the performance of the SDS and its underlying search capabilities.

5.1. Single-server Performance

Measurements of the local-area service-to-server and client-to-server interactions are averaged over 100 trials and were made using Intel Pentium II 350MHz machines with 128MB of RAM, running Slackware Linux 2.0.36. We used Sun's JDK 1.1.7 with the TYA JIT compiler. For security support, we use the java.security package, where possible, and otherwise we use the Cryptix security library. For the XML parser, we use Microsoft's MSXML version 1.9. We assume that the majority of SDS queries will contain a small number of search constraints, and use that model for our performance tests. The XML workload consists of XML files generated by converting other sources of data: printer configuration information and a subset of the CDs from the FreeDB CD database. For secure communications, SDS uses an authenticated RMI implementation developed by the Ninja research group [52], which we modified to use Blowfish [43] instead of TripleDES.

5.1.1. Security Component

Table 2 lists the various costs of the security mechanisms used in the SDS. We profile the use of DSA certificates [42] for both signing and verifying information, RSA [42] encryption and decryption as used in the service broadcasts, and Blowfish as used in authenticated RMI. Note that both DSA and RSA are asymmetric key algorithms, while Blowfish is a symmetric key algorithm. All execution times were determined by verifying/signing or encrypting/decrypting 1KB input blocks. The measurements verify what should be expected: the asymmetric algorithms, DSA and RSA, are much more computationally expensive than the symmetric key algorithm. This validates the design choice of providing symmetric-key crypto for the fast path. DSA verification time is especially high because it verifies two signatures per certificate: the certificate owner's signature and the certificate authority's signature.

Name	Time
DSA Signature	33.1 ms
DSA Verification	133.4 ms
RSA Encryption	15.5 ms
RSA Decryption	142.5 ms
Blowfish Encryption	2.0 ms
Blowfish Decryption	1.7 ms

Table 2
Timings of cryptographic routines

Files	Query Time
1000	1.17 ms
5000	1.43 ms
10000	2.64 ms
20000	2.76 ms
40000	4.40 ms
80000	5.64 ms
160000	6.24 ms

Table 3
XSet Query Performance

5.1.2. XML Search Component

We use the XSet XML [55] search engine to perform queries against the service description cache. To maximize

	Empty Query	Full Query
Insecure	24.5 ms	36.0 ms
Secure	40.5 ms	82.0 ms

Table 4
Query Latencies for Various Configurations

performance, XSet builds an evolutionary hierarchical tag index, with per-tag references stored in treaps (probabilistic self-balancing trees). As a result, XSet's query latency increases only logarithmically with increases in the size of the dataset. The performance results are shown in Table 3. To reduce the cost of query processing, validation of service descriptions against their associated Document Type Definition (DTD) is performed only once, the first time it is seen, not per query or per announcement.

5.1.3. Aggregate Search Performance

Table 4 lists the latencies for various SDS single-node queries: both empty and full queries, with security enabled and disabled. Times do not include the cost of session initialization, which is amortized over multiple queries. The basic (empty, insecure) query time includes RMI and network overhead; secure queries add encryption overhead, and non-null ("full") queries add search time and overhead due to their additional length. Service announcement processing time, which includes both decryption and processing of a single 1.2KB service announcement, averages 9.2ms.

Table 5 shows the average performance breakdown of a single-node secure SDS query from a single client. The SDS server was receiving service descriptions at a rate of 10 1.2KB announcements per second; XSet contained twenty service descriptions; and the search lists seven different capabilities to test. (As Table 3 shows, expanding the service description database contributes little additional latency.) Note that the table splits encryption time between its client and server components, and that RMI overhead includes the time spent reading from the network. The unaccounted overhead is probably due to context switches, competing network traffic, and object/array copying. As can be inferred from the table, security accounts for 27% of the total processing cost, a significant but not dominating percentage.

Extrapolating these performance numbers, we approximate that a single SDS server can handle approximately eighty clients sending queries at a rate of one query per second.

5.2. Wide-area Performance

Measurements of wide-area interaction were made using an Intel Pentium III 500MHz with 512KB of cache and 128MB of RAM running Red Hat Linux 2.2.12-20 and Sun's JDK 1.2. Our testbed runs on a single node, with messages sent between SDS servers via intra-JVM method calls. Every aspect of these "simulations" is identical to real operation except for the transport mechanism. For XML processing we use a non-validating parser written by ourselves. For the benchmarks, queries are sent up neighbor links in some serial order (not "bifurcated"), we use only a single hierarchy, encryption and authentication are turned off, and parent-based forwarding (rather than full forwarding) is used. Workloads are comprised of service announcements derived from CD descriptions from the FreeDB CD database (converted to XML); queries are generated by randomly selecting a single tag from a possible service description and asking for it. For the case of Brokering, all service descriptions are passed to neighbors (the brokering criteria is "send all services"), thereby maintaining full reachability and not artificially skewing results in favor of the scheme.

We now present the results of direct comparisons of the four implemented indexing strategies from Section 4.4. We attempt to tease out the fundamental trade-offs between the schemes through the use of focused "microbenchmark" workloads on small topologies. In assessing the approaches, there are two key components to account for: required update traffic (determined by the description aggregation

Query Component	Latency
Query Encryption (client-side)	5.3 ms
Query Decryption (server-side)	5.2 ms
RMI Overhead	18.3 ms
Query XML Processing	9.8 ms
Capability Checking	18.0 ms
Query Result Encryption (server-side)	5.6 ms
Query Result Decryption (client-side)	5.4 ms
Query Unaccounted Overhead	14.4 ms
Total (Secure XML Query)	82.0 ms

Table 5
Secure Query Latency Breakdown

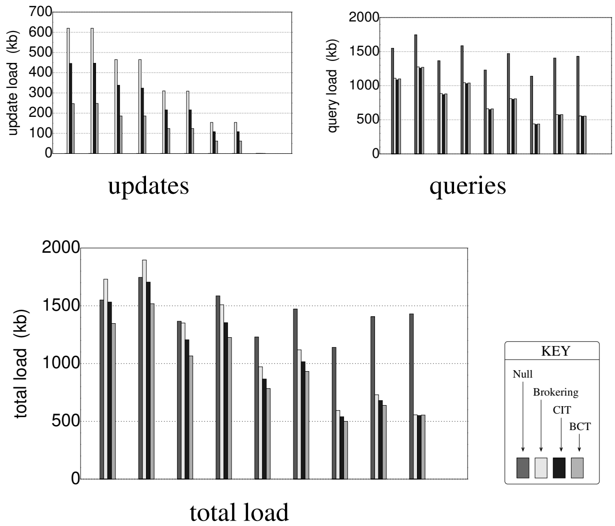

Figure 8. **Line, Q:U=2:1** Comparison of aggregate query bandwidth, update bandwidth, and total bandwidth for the four filtering schemes in a linear topology with twice as many queries as updates.

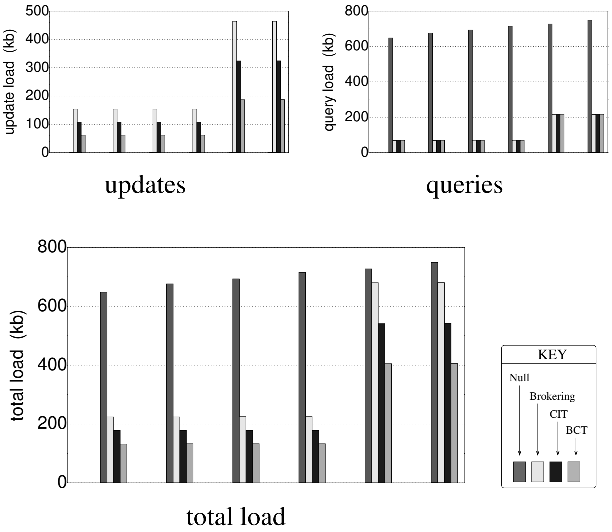

Figure 9. **Tree, Q:U=2:1** Comparison of aggregate query bandwidth, update bandwidth, and total bandwidth for the four filtering schemes in a binary tree topology with twice as many queries as updates.

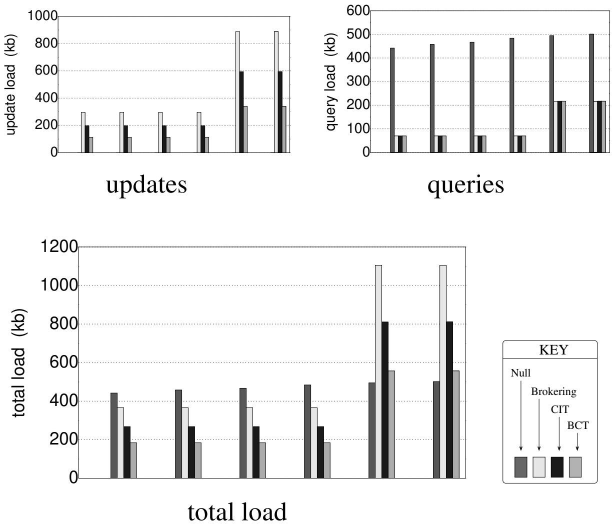

Figure 10. **Tree, Q:U=1:1** Comparison of aggregate query bandwidth, update bandwidth, and total bandwidth for the four filtering schemes in a binary tree topology with an equal number of updates and queries.

scheme) and its effect on query traffic. A given workload can be used to analyze filters by summing their total update message load and total query traffic load on a per-link basis. This aggregate metric – total load – can then be further compared by looking at averages, the maximum, etc. In a hierarchy, the roots will often be the scaling bottleneck, and thus we compare worst-case maximum total loads.

Our first benchmark looks at ten SDS servers in a linear topology, thereby investigating the basic properties of a sample leaf-to-root path. Each server in the line has one entity communicating with it, either a querier or service announcer alternating along the line. Queriers send periodic queries, while service announcers send periodic service registrations. There are twice as many queries as updates, and thus the query-to-update ratio is two (Q:U=2:1). Results are shown in Figure 8. The figure (and the others like it) is composed of three bar graphs. The top-left graph shows total update load on the y axis, and the various links in the topology sorted along the x axis. The top-right graph shows total query load on the y axis, and also has the various links in the topology along the y axis. The larger bar graph is the sum of the two smaller graphs, and it indicates per-link total load. As can be seen, null filtering requires no update traffic but pays the toll in much greater query traffic. Brokering, CIT, and BCT all send similar amounts of query traffic, but broker updates are larger than CIT updates, which in turn are larger than BCT updates. Thus, the worst-case total load is smallest for BCT, illustrating it works well for this workload and topology.

Our second and third benchmarks look at a seven-node binary tree topology. In this case there is a querier and service announcer at each node in the tree. Results are shown in Figure 9 and Figure 10. They are treated together to illustrate the importance of update-to-query ratio. The only difference between the two tests is that in Figure 9 the query-to-update ratio is 2:1, while in Figure 10 this ratio is 1:1. (Specifically, update load is doubled in the latter.) In the former case, the performance results maintain that the filtering strategies perform in the same rank order as with the linear topology (BCT performs best). In the latter, it is actually the *null* filter that exhibits minimum worst-case total load. What this illustrates is that the cost of updates, whatever the scheme, must be offset by enough of a query load to make the investment worthwhile. In short, with very high service join/leave rates, flooding may be the best policy. Of known workloads for location services (e.g., mp3 file sharing, DNS lookups), query rate dominates service join/leave rate, and thus our results generally suggest the use of query filtering rather than tree flooding. But Figure 10 is instructive: it argues for *workload-based* filter policy control, where the push/pull tradeoff is either based on an analysis of a static workload, or otherwise dynamically adapted as the workload varies.

6. Related Work

Service discovery is an area of research that has a long history. Many of the ideas in the SDS have been influenced by previous projects.

6.1. DNS and Globe

The Internet Domain Naming Service [32] and Globe [49] (conceptual descendents of Grapevine [44]) are examples of systems which perform global discovery of known services: in the former case, names are mapped to addresses; in the latter, object identifiers are mapped to the object broker that manages it. An assumption of this type of service discovery is that keys (DNS fully-qualified domain names or Globe unique object identifiers) uniquely map to a service, and that these keys are the query terms. Another assumption is that all resources are public; access control is done at the application level rather than in the discovery infrastructure.

The scalability and robustness of DNS and Globe derives from the hierarchical structure inherent in their unique service names. The resolution path to the service is embedded inside the name, establishing implicit query-routing, thus making the problem they address different from that of the SDS.

6.2. Condor Classads

The “classads” [34] service discovery model was designed to address resource allocation (primarily locating and using off-peak computing cycles) in the Condor system. Classads provides confidential service discovery and management using a flexible and complex description language. Descriptions of services are kept by a centralized matchmaker; the matcher maps clients’ requests to advertised services, and informs both parties of the pairing. Advertisements and requirements published by the client adhere to a classad specification, which is an extensible language similar to XML. The matchmaking protocol provides flexible matching policies. Because classads are designed to only provide hints for matching service owners and clients, a weak consistency model is sufficient and solves the stale data problem.

The classads model is not applicable to wide-area service discovery. The matchmaker is a single point of failure and performance bottleneck, limiting both scalability and fault-tolerance. Additionally, while the matchmaker ensures the authenticity and confidentiality of service, communication between parties is not secure.

6.3. Jini

The Jini [50] software package from Sun Microsystems provides the basis for both the Jini connection technology and the Jini distributed system. In order for clients to discover new hardware and software services, the system provides the Jini Lookup Service [48], which has functionality similar to the SDS.

When a new service or Jini device is first connected to a Jini connection system, it locates the local Lookup service using a combination of multicast announcement, request, and unicast response protocols (*discovery*). The service then sends a Java object to the Lookup service that implements its service interface (*join*), which is used as a search template for future client search requests (*lookup*). Freshness is maintained through the use of leases.

The query model in Jini is drastically different from that of the SDS. The Jini searching mechanism uses the Java serialized object matching mechanism from JavaSpaces [48], which is based on exact matching of serialized objects. As a result, it is prone to false negatives due to, e.g., class versioning problems. One benefit of the Jini approach is that it permits matching against subtypes, which is analogous to matching subtrees in XML. A detriment of the model is that it requires a Java interface object be sent over the network to the lookup service to act as the template; such representations cannot be stored or transported as efficiently as other approaches.

Security has not been a focus of Jini. Access control is checked upon attempting to register with a service, rather than when attempting to discover it; in other words, Jini protects access to the service but not discovery of the service. Furthermore, communication in the Jini Lookup service is done via Java RMI, which is non-encrypted and prone to snooping. Finally, the Jini Lookup Service specifies no mechanism for server-, client-, or service-side authentication.

A final point of distinction is the approach to wide area scalability. While the SDS has a notion of distributed hierarchies for data partitioning and an aggregation scheme among them, Jini uses a loose notion of federations, each corresponding to a local administrative domain. While Jini mentions the use of inter-lookup service registration, it’s unclear how Jini will use it to solve the wide-area scaling issue. In addition, the use of Java serialized objects makes aggregation difficult.

Despite the differences in architecture, we have created a Jini proxy that enables the SDS to discover Jini-enabled services and devices, similar to the SLP-Jini bridge [20]. In essence, we created a proxy that listens for Jini services using their discovery protocol, and upon finding new services, relays their descriptions (suitably transformed) to the SDS system.

6.4. SLP

The IETF Service Location Protocol (SLP) [21], and its wide-area extension (WASRV) [40], address many of the same issues as the SDS, and some that are not (e.g., internationalization). The design of the SDS has benefited from many of the ideas found in SLP, while attempting to make improvements in selected areas.

The SLP local-area discovery techniques are nearly identical to those of the SDS: Multicast is used for announcements and bootstrapping, and service information is cached in Directory Agents (DAs), a counterpart to the SDS server. Timeouts are used for implicit service deregistration.

As for scaling beyond the local area, there are actually two different mechanisms: named scopes and brokering. In the former scheme, the local administrative domain is partitioned into named User Agent “scopes” from a flat scoping namespace. The scheme is not designed to scale globally. In the latter scheme, the approach is to pick an entity in each SLP administrative domain (SLPD) to act as an Advertising Agent (AA), and for these AAs to multicast selected service information to a wide-area multicast group shared amongst them. Brokering Agents (BAs) in each SLPD listen to multicasts from SLPD AAs, and advertise those services to the local SLPD as if they were SAs in the local domain. While the WASRV strategy does succeed in bridging multiple SLPDs, it does not address a basic problem: the AAs must be configured to determine which service descriptions are propagated between SLPDs; in the worst case, everything is propagated, each domain will have a copy of all services, and thus there is no "lossy aggregation" of service information. This inhibits the scheme from scaling any better than linearly with the number of services advertised and the number of AAs/BAs. Additionally, WASRV's reliance on wide-area multicast is ill-advised given existing deployment difficulties with inter-domain multicast routing [12].

One of the most useful aspects of SLP is its structure for describing service information. Services are organized into service types, and each type is associated with a service template that defines the required attributes that a service description for that service type must contain [21]. The functionality and expressiveness of this framework is almost an exact mapping onto the functionality of XML: each template in SLP provides the same functionality as an XML schema or DTD. Queries in SLP return a service URL, whereas XML queries in the SDS returns the XML document itself (which can itself be a pointer using the XML XRef facility). There are some benefits to using XML rather than templates for this task. First, because of XML's flexible tag structure, service descriptions may, for example, have multiple location values or provide novel extensions (for example, encoding Java RMI stubs inside the XML document itself). Second, since references to DTDs reside within XML documents, SDS service descriptions are self-describing.

A final point of contrast between SLP and SDS is security. SLP provides authentication in the local administrative domain, but not cross-domain. Authentication blocks can be requested using an optional field in the service request, providing a guarantee of data integrity, but no mechanism is offered for authentication of User Agents. Additionally, because of a lack of access control, confidentiality of service information cannot be guaranteed.

Though the systems are disparate, we would like SLP and the SDS to cooperate rather than compete in providing information to clients. We believe that, as with Jini, this could be achieved through proxying.

6.5. Decentralized Distributed Location Services

Recent projects such as Tapestry [56], Chord [47], and Content-Addressable Networks (CAN) [36] have focused on providing name-to-location mapping services over the wide-area utilizing overlay networks. The systems provide a distributed hashtable interface, mapping an object's location given its global unique identifier.

These location services are novel in that they provide wide-area scalability in a decentralized manner by organizing nodes in the form of a hypercube or mesh, with each node maintaining routing state that scales sublinearly with the size of the network. Queries are routed based on the object identifier directly to the object location.

The key distinction between these location services and the SDS is support for multi-criteria searches. While Chord, CAN, and Tapestry provide efficient mappings from a single unique identifier to a location, they are insufficient when users are searching for an unknown resource or object based on descriptive requirements.

7. Conclusion

7.1. Summary

The continuing growth of networks, network-enabled devices, and network services is increasing the need for network directory services. The SDS provides network-enabled devices with an easy-to-use method for discovering services that are available. It is a directory-style service that provides a contact point for making complex queries against cached service descriptions advertised by services. The SDS automatically adapts its behavior to handle failures of both SDS servers and services, hiding the complexities of fault recovery from the client applications. The SDS is also security-minded; it ensures that all communication between components is secure and aids in determining the trustworthiness of particular services.

The SDS soft-state model and announcement-based architecture offers superior handling of faults and changes in the network topology. It handles the addition of new servers and services, while also recognizing when existing services have failed or are otherwise no longer available.

The use of XML to encode service descriptions and client queries also gives the SDS certain advantages. Service providers will be able to capitalize on the extensibility of XML by constructing service-specific tags to better describe the services that they offer. Likewise, XML will enable clients to make more powerful queries by taking advantage of the semantic-rich service descriptions.

Finally, the SDS integrated security model protects the sensitive information belonging to services, as well as assists clients in locating trustworthy services. By exploring design issues in the SDS, we hope to better understand the tradeoffs involved in offering this level of privacy.

7.2. Future Work

In ongoing work, we are incorporating various result caching strategies to enable short-cut routing from one interior node to another. Additionally, we are investigating an approach that allows indexing strategies to differ based on the workload presented to the system and the local traffic conditions. We call this approach "hybrid indexing:" given that particular filtering strategies perform better for differing workloads, and given no *a priori* knowledge of workload, allowing local optimizations rather than a static strategy should enable better overall performance. SDS servers could measure the query-to-update ratio, and vary the amount of information in updates and/or the underlying filtering strategy.

A more radical design change we are considering is to attempt query filtering over a mesh rather than in a shared hierarchy. One possible approach to accomplish this would be to generate a set of loop-free paths in the mesh (possibly reusing underlying BGP path vectors), and apply the update/filtering technique as before. This maintains the basic query filtering functionality, but generalizes it in a way where misses do not propagate to some shared root node – instead, each autonomous system (AS) would know the contents of its BGP neighbors, and queries would be passed from domain to domain.

We have generated performance results showing the tradeoff between update bandwidth and query routing efficiency exposed by full forwarding, but have not had the time to analyze them. Similarly, were are still investigating the results on the tradeoff between query response latency and total bandwidth used when queries bifurcate.

Finally, our approach to mobility support can be augmented with the use of forwarding pointers [25] to deal with especially high-mobility clients, and such pointers could elevate to stable positions in the hierarchy as is done in Globe [49].

Acknowledgments

We thank the students and faculty of the Ninja and Iceberg projects for their assistance in implementing the infrastructure and for their comments on earlier drafts. We also thank Ketan Mayer-Patel, Michelle Munson, Andrew Begel, and the anonymous reviewers for their insightful commentary and interest in this work.

References

[1] AMIR, E., MCCANNE, S., AND KATZ, R. An Active Services Framework and its Application to Real-Time Multimedia Transcoding. *Proceedings of SIGCOMM '98* (1998).

[2] ANDERSON, T., PATTERSON, D., CULLER, D., AND THE NOW TEAM. A Case for Networks of Workstations: NOW. *IEEE Micro* (February 1995).

[3] BLOOM, B. Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM 13*, 7 (July 1970), 422–426.

[4] BRAY, T., PAOLI, J., AND SPERBERG-MCQUEEN, C. M. eXtensible Markup Language (XML). W3C Recommendation, Feb 1998. `http://www.w3.org/XML`.

[5] CHAWATHE, Y., MCCANNE, S., AND BREWER, E. An architecture for internet content distribution as an infrastructure service, February 2000. Unpublished. `http://www.cs.berkeley.edu/~yatin/papers/`.

[6] CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. W. Freenet: A Distributed Anonymous Information Storage and Retrieval System. *ICSI Workshop on Design Issues in Anonymity and Unobservability* (July 2000).

[7] CLIP 2 DISTRIBUTED SEARCH SOLUTIONS. Bandwidth Barriers to Gnutella Network Scalability. `http://dss.clip2.com/dss_barrier.html`.

[8] DAVIS, C., VIXIE, P., GOODWIN, T., AND DICKINSON, I. *A Means for Expressing Location Information in the Domain Name System*. IETF, Jan 1996. RFC-1876.

[9] DEERING, S. *Host Extensions for IP Multicasting*. IETF, SRI International, Menlo Park, CA, Aug 1989. RFC-1112.

[10] DEERING, S. E. *Multicast Routing in a Datagram Internetwork*. PhD thesis, Stanford University, Dec. 1991.

[11] DEUTSCH, A., ET AL. XML-QL: A Query Language for XML, August 1998. `http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/`.

[12] DIOT, C., LEVINE, B. N., LYLES, B., KASSEM, H., AND BALENSIEFEN, D. Deployment Issues for the IP Multicast Service and Architecture. *IEEE Network, Special Issue on Multicasting* (January/February 2000).

[13] FALTSTROM, P., SCHOULTZ, R., AND WEIDER, C. *How to interact with a WHOIS++ mesh*. IETF, 1995. RFC-1914.

[14] FAN, L., CAO, P., ALMEIDA, J., AND BRODER, A. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *Proceedings of SIGCOMM '98* (1998).

[15] FAN, L., CAO, P., ALMEIDA, J., AND BRODER, A. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. Tech. Rep. 1361, Computer Sciences Department, Univ. of Wisconsin-Madison, Feb. 1999.

[16] FANNING, S. Napster. `http://www.napster.com`.

[17] FOX, A., GRIBBLE, S. D., CHAWATHE, Y., BREWER, E. A., AND GAUTHIER, P. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint-Malo, France, October 1997), vol. 16, ACM.

[18] FRANKEL, J., AND PEPPER, T. Gnutella. `http://gnutella.wego.com`.

[19] GRIBBLE, S., WELSH, M., ET AL. The Ninja Architecture for Robust Internet-Scale Systems and Services. *Special Issue of Computer Networks on Pervasive Computing* (2001). `http://ninja.cs.berkeley.edu`.

[20] GUTTMAN, E., AND KEMPF, J. Automatic Discovery of Thin Servers: SLP, Jini and the SLP-Jini Bridge. *Proceedings of the 25th Annual Conference of the IEEE Industrial Electronics Society* (1999), 722–727.

[21] GUTTMAN, E., PERKINS, C., VEIZADES, J., AND DAY, M. Service Location Protocol, Version 2. IETF, November 1998. RFC 2165.

[22] HANDLEY, M., AND JACOBSON, V. *SDP: Session Description Protocol*. IETF, 1998. RFC-2327.

[23] HODES, T., AND KATZ, R. H. Composable Ad hoc Location-based Services for Heterogeneous Mobile Clients. *ACM Wireless Networks Journal 5*, 5 (October 1999), 411–427. Special issue on Mobile Computing: selected papers from MobiCom '97.

[24] IMIELINSKI, T., AND GOEL, S. DataSpace - querying and monitoring deeply networked collections in physical space. *IEEE Personal Communications Magazine* (October 2000).

[25] JAIN, R., AND LIN, Y. An Auxiliary User Location Strategy Employing Forwarding Pointers to Reduce Network Impact of PCS. *ACM-Baltzer Journal of Wireless Networks 1*, 2 (July 1995), 197–210.

[26] KARGER, D. R., ET AL. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. *Proceedings of STOC '97* (1997), 654–663.

[27] KOSSMANN, D., FRANKLIN, M., AND DRASCH, G. Cache Investment: Integrating Query Optimization and Dynamic Data Placement. *ACM Transactions on Database Systems* (December 2000).

[28] KUBIATOWICZ, J., ET AL. OceanStore: An Architecture for Global-Scale Persistent Storage. *Proceeedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)* (November 2000).

[29] LEVINE, B., PAUL, S., AND GARCIA-LUNA-ACEVES, J. Organizing Multicast Receivers Deterministically According to Packet-Loss Correlation. *Proceedings of ACM Multimedia '98* (September 1998).

[30] MAHER, M. P., AND PERKINS, C. Session Announcement Protocol: Version 2. IETF Internet Draft, November 1998. draft-ietf-mmusic-sap-v2-00.txt.

[31] MCQUILLAN, J., RICHER, I., AND ROSEN, E. The New Routing Algorithm for the ARPANET. *IEEE Transactions on Communications 28*, 5 (May 1980), 711–719.

[32] MOCKAPETRIS, P. V., AND DUNLAP, K. Development of the Domain Name System. *Proceedings of SIGCOMM '88* (August 1988).

[33] PERKINS, C., ET AL. IP Mobility Support. IETF, October 1996. RFC 2002.

[34] RAMAN, R., LIVNY, M., AND SOLOMON, M. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing* (July 1998).

[35] RAMAN, S., AND MCCANNE, S. A Model, Analysis, and Protocol Framework for Soft State-based Communication. *Proceedings of ACM SIGCOMM '99* (September 1999).

[36] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SCHENKER, S. A Scalable Content-Addressable Network. In *Proceedings of SIGCOMM* (August 2001), ACM.

[37] RATNASAMY, S., AND MCCANNE, S. Inference of Multicast Routing Trees and Bottleneck Bandwidths using End-to-end Measurements. *Proceedings of INFOCOM '99* (March 1999).

[38] RITTER, J. Why Gnutella Can't Scale. No, Really. `http://www.darkridge.com/~jpr5/doc/gnutella.html`.

[39] ROBIE, J., LAPP, J., AND SCHACH, D. XML Query Language (XQL). In *QL '98 - The Query Languages Workshop* (December 1998), W3C. `http://www.w3.org/TandS/QL/QL98/pp/xql.html`.

[40] ROSENBERG, J., SCHULZRINNE, H., AND SUTER, B. Wide area network service location. IETF draft Request for Comments (RFC), December 1997. draft-ietf-svrloc-wasrv-01.txt.

[41] ROUSSKOV, A., AND WESSELS, D. Cache Digests. *Proceedings of the Third International Web Caching Workshop* (June 1998).

[42] SCHNEIER, B. *Applied Cryptography*, first ed. John Wiley and Sons, Inc., 1993.

[43] SCHNEIER, B. Description of a new variable-length key, 64-bit block cipher (Blowfish). In *Fast Software Encryption, Cambridge Security Workshop Proceedings* (December 1993), Springer-Verlag, pp. 191–204.

[44] SCHROEDER, M., BIRRELL, A., JR., R. C., AND NEEDHAM, R. Experience with Grapevine: the growth of a distributed system. *ACM Transactions on Computer Systems 2*, 1 (February 1984), 3–23.

[45] SCHULZRINNE, H., CASNER, S., FREDERICK, R., AND JACOBSON, V. RTP: A Transport Protocol for Real-Time Applications. *IETF RFC 1889* (January 1996).

[46] SESHAN, S., STEMM, M., AND KATZ, R. H. SPAND: Shared Passive Network Performance Discovery. In *1st Usenix Symposium on Internet Technologies and Systems (USITS '97)* (Monterey, CA, December 1997).

[47] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, F., AND BALAKRISHNAN, H. Chord: A Peer-to-Peer Lookup Service for Internet Applications. *Proc. ACM SIGCOMM 2001* (September 2001).

[48] SUN MICROSYSTEMS. Jini technology specifications. white paper. `http://www.sun.com/jini/specs/`.

[49] VAN STEEN, M., HAUCK, F., HOMBURG, P., AND TANENBAUM, A. Locating objects in wide-area systems. *IEEE Communications Magazine* (January 1998), 104–109.

[50] WALDO, J. The Jini Architecture for Network-centric Computing. *Communications of the ACM* (July 1999), 76–82.

[51] WEISER, M. The Computer for the 21st Century. *Scientific American 265*, 3 (September 1991), 94–104.

[52] WELSH, M. Ninja RMI. `http://www.cs.berkeley.edu/~mdw/proj/ninja/ninjarmi.html`.

[53] WESSELS, D., AND CLAFFY, K. ICP and the Squid Web Cache. *IEEE Journal on Selected Areas in Communication 16*, 3 (April 1998), 345–357.

[54] WOOD, L., APPARAO, V., ET AL. Document Object Model Level 1 Specification, October 1998. W3C DOM working group, `http://www.w3c.org/DOM/`.

[55] ZHAO, B. XSet. `http://www.cs.berkeley.edu/~ravenben/xset/`.

[56] ZHAO, B. Y., KUBIATOWICZ, J. D., AND JOSEPH, A. D. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, University of California at Berkeley, Computer Science Division, April 2001.