An Architecture for a Secure Service Discovery Service

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

Abstract

The widespread deployment of inexpensive communications technology, computational resources in the networking infrastructure, and network-enabled end devices poses an interesting problem for end users: how to locate a particular network service or device out of hundreds of thousands of accessible services and devices. This paper presents the architecture and implementation of a secure Service Discovery Service (SDS). Service providers use the SDS to advertise complex 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 highly-available, fault-tolerant, incrementally scalable service for locating services in the wide-area. Security is a core component of the SDS and, where necessary, communications are both encrypted and authenticated. Furthermore, the SDS uses an hybrid access control list and capability system to control access to service information.

1 Introduction

The decreasing cost of networking technology and network-enabled devices is enabling the large-scale deployment of such networks and devices [32]. Simultaneously, significant computational resources are being deployed within the network infrastructure; 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 client users. For example, an application that allows a user to control the lights in a room is a service. Other examples of services are printers, fax machines, and music servers.

Ultimately, we expect that, just as there are hundreds of thousands of web servers, there will be hundreds of thousands (or millions) of services available to end users. Given this assumption, a key challenge for these end users will be *locating* the appropriate service for a given task, where "appropriate" has a user-specific definition (e.g., cost, location, accessibility, etc.). In addition, trustworthy and secure access to such services are critical requirements. 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 services that they are interested in using. We have built such a service, the Ninja[1] *Service Discovery Service* (SDS) to provide this functionality and enable clients to more effectively search for and use the services available in the network. Like the rest of the major components of Ninja, the SDS is implemented in Java [10].

The SDS is a scalable, fault-tolerant, and secure information repository, providing clients with directory-style access to all available services. It stores two types of information: descriptions of services that are available for execution at computational resources embedded in the network (so-called "unpinned" services), and services that are already running at a specific location. The SDS also supports both push-based and pull-based access; the former allows passive discovery, while the latter permits the use of a query-based 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

This work was supported in part by grants from Ericsson, Intel, Sprint, and Motorola, by DARPA under contract DABT63-98-C-0038, by the State of California under the MICRO program, and by NSF Research Infrastructure grant CDA-94-01156.

Mobicom '99 Seattle Washington USA
Copyright ACM 1999 1-58113-142-9/99/08...$5.00

[1] The Ninja project is developing a scalable, fault-tolerant, distributed, composable services platform [30].

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 rather than (or in addition to) disallowing access to a located service.

As a globally-distributed, wide-area service, the SDS addresses challenges beyond those of services that operate solely in the local area. The SDS architecture handles network partitions and component failures; addresses the potential bandwidth limitations between remote SDS entities; arranges the components into a hierarchy to distribute the workload; and provides application-level query routing between components.

This paper presents the design of the SDS, focusing on both the architecture of the directory service and the security features of the system. Section 2 begins our discussion by describing the design concepts used in order to achieve our goals. The SDS architecture is described in Section 3. Wide-area operation is discussed in Section 4. Performance measurements from the SDS prototype implementation are presented in Section 5, followed by a discussion of related systems in Section 6. Finally, we summarize and conclude 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, support for complex queries, and secure access for clients and services.

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. The SDS addresses this issue by using *periodic multicast announcements* as its primary information propagation technique, and through the use of 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 are tolerated in the normal mode of operation rather than addressed through a separate recovery procedure [1]: recovery is enabled by simply listening to channel announcements. The combination of periodicity and the use of multicast is often called the "announce/listen" model in the literature, and is appropriate where "eventual consistency" rather than a transactional semantic suffices. The announce/listen model initially appeared in IGMP [6], and was further developed and clarified in systems such as the MBone Session Announcement Protocol [15]. Refinement of the announce/listen idea to provide for tolerance of host faults (leveraging multicast's indirection along with cluster computing environments [2]) appeared in the context of the AS1 "Active Services" framework [1]. We will describe our use of announce/listen in Sections 3.1 and 3.2.

2.2 Hierarchical Organization

As a scalability mechanism, SDS servers organize into a hierarchical structure; service announcements and client queries are assigned to go to a particular SDS server. The "domain" of an SDS server is the network extent (e.g., the fractional subnet, subnet, or subnets) it covers. If a particular SDS server is overloaded, a new SDS server will be started as a "child" and assigned a portion of the network extent (and, thus, a portion of the load). See Figure 1 for an example configuration.

Section 3.1 discusses how domains are mapped to the multicast channels that are used by all services in the domain. Discussion of hierarchical organization is treated in Section 4.

2.3 XML Service Descriptions

Rather than use flat name-value pairs (as in, e.g., the Session Description Protocol [12]), 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, while additionally enabling "subtyping" via nesting of tags.

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, and Java RMI address. 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.

2.4 Privacy and Authentication

Unlike many other directory services, 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 and integrity are 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

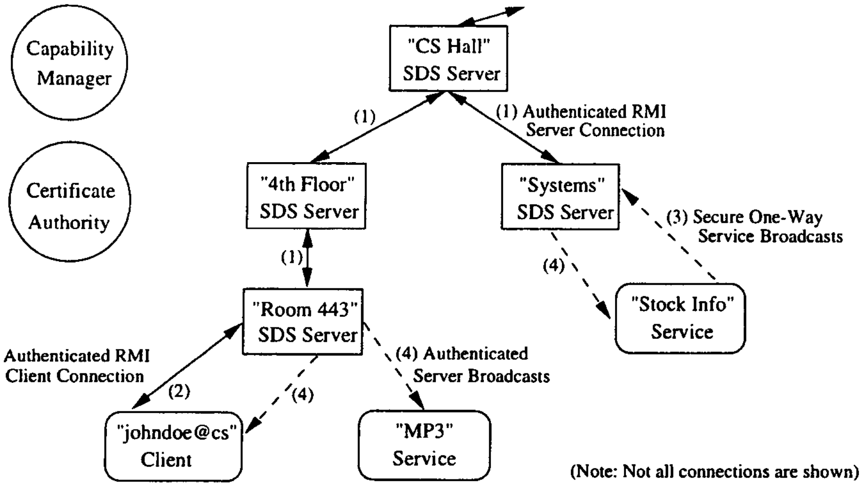

Figure 1: Components of the secure Service Discovery Service. Dashed lines correspond to periodic multicast communication between components, while solid lines correspond to one-time Java RMI connections.

```
<?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: An example XML query (A), matching service description (B), and failed match (C).

overhead of the encryption, a traditional hybrid of asymmetric and symmetric-key cryptography is used.

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). By making authentication an integral part of the SDS, we can incorporate trust 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, an SDS server will return only those services that are run by the specified principals.

For example, if a *CS Division* principal is used for CS division-wide services, then a client with access to all *CS Division* services looking for an "official" e-mail server would specify the *CS Division* principal. SDS servers would only return *CS Division* servers, instead of including e-mail servers being run by, e.g., individuals.

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. These capabilities are basically 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 3.4 elaborates on the use of capabilities.

Section 3.5 provides details of our use of authentication and encryption in the architecture, while Section 5.1 presents our measurements of the cost of these security components.

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

Each server is responsible for sending authenticated messages containing a list of the domains that it is responsible for on the well-known global 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 and 3.4). 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 [15] and RTCP [26]. (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.

In the SDS server hierarchy, when the service load reaches a certain threshold on an SDS server, one or more new "child" servers are spawned. Each new server is allocated a portion of the existing service load. Servers keep track of their child nodes through periodic "heartbeat" messages.

If a server goes down, its parent will notice the lapse in heartbeats and restart it (possibly elsewhere if the node itself has failed). As an additional measure of robustness, server crashes can also be recovered by a "peer" workstation in the same manner described by Amir *et.al.* [1]: have the peer workstations listen in on the announce/listen messages and, leveraging the multicast indirection, transparently select amongst themselves. Restarted servers populate their databases by listening to the existing service announcements, thereby avoiding the need for an explicit recovery mechanism. Additionally, because the 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.

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 3.5.2), 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 3.5.3) 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 XML [34] 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 below in Section 3.4).

3.3 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 the client to distribute the certificate to him; possessing a certificate does not benefit a client unless he also possesses the private key associated with it. Accepting new certificates and encryption key 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.

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

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

3.4 Capability Manager

The SDS uses capabilities as an access control mechanism to enable services to control the set of users that are allowed to discover their existence. In traditional access control, services would have to talk to a central server to verify a user's access rights. 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 principals allowed to access the service's description). The CM then generates the appropriate capabilities and saves them for later distribution to the clients. 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.

3.5 Secure SDS Communication

The communication methods used by the SDS balance information privacy and security against information dissemination efficiency. In the following sections, we discuss the various types of communication used by the SDS.

3.5.1 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.

3.5.2 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 is then used to encrypt 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.

3.5.3 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 [33][2]. ARMI allows applications to invoke methods on remote objects in a two-way authenticated and encrypted fashion.

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. This enables a client to set a policy that restricts access to only those remote SDS servers that have valid "sds-server" certificates. The performance of ARMI is discussed in Section 5.

3.6 Bootstrapping

Clients discover the SDS server for their domain by listening to a well-known SDS global multicast address. Alternatively, as a discovery latency optimization, a client can solicit an asynchronous SDS server announcement by using expanding ring search (ERS) [7]: TTL-

[2] The choice of ARMI for client-server communication is a function of our use of Java. This implementation choice is orthogonal to the system design; the necessary functionality can be mapped onto most other secure invocation protocols.

limited query messages are sent to the SDS global multicast address, and the TTL is increased until there is a response. This is analogous to "foreign agent solicitation" and "foreign agent advertisement" in Mobile IP [19] extended beyond the local subnet.

An SDS server's domain is specified as a list of CIDR-style network-address/mask pairs (e.g., 128.32.0.0/16 for the entire Berkeley campus). Newly spawned child SDS servers claim a portion of the parent's region, where the specific portion is specified by the parent. This syntax allows for complete flexibility in coverage space while providing efficient representation when domains align to the underlying topology.

4 Wide-Area Support

The previous section detailed the local interactions of SDS servers, SDS clients, and services. In this section, we focus on how SDS servers could interact with one another as a whole in order to support scalable, wide-area service discovery. (A caveat: this section is a design overview. We have implemented portions of it but not incorporated it with the local-area SDS code or measured it.)

Using the SDS as a globally-distributed service requires that it be able to scale to support a potentially huge number of clients and services using it while adapting as the underlying entities that comprise it change (e.g., due to network partitions and node failures). We would like clients to be able to discover all services on all SDS servers, so we cannot simply partition the information and queries. Additionally, neither maintaining all service information at all servers, nor sending queries to all servers, scales as the number of such servers grows. One approach to reducing this scaling problem is to leave service information partitioned amongst the individual SDS servers, while propagating summaries of the contents to one another. Even if such summarization were possible, there is quadratic growth in such messaging, and queries would still have to go to all servers. This is again a scaling problem. A further step, then, is to have the servers arrange themselves into a multi-level hierarchy. Summary information would be propagated only to parents, and queries partitioned amongst the servers for further forwarding. It is this latter approach that we employ for wide-area service discovery.

There are two major components to achieving this goal: the dynamic construction and adaptation of a hierarchy of SDS servers, and providing an application-level routing infrastructure that allows servers to propagate information through the hierarchy. The information propagation problem can be further decomposed into two sub-problems: providing lossy aggregation of service descriptions as they travel up the hierarchy (to prevent the root nodes from becoming a bottleneck for updates or queries), and dynamically routing client queries to the appropriate servers based on the local aggregate data. In short, the problems of building routing tables and then interpreting them.

We now discuss our proposed solutions to these problems.

4.1 Adaptive Server Hierarchy Management

Two key questions arise given the use of hierarchy. The first is the choice of what hierarchy, or hierarchies, to build. The second is determining how to build and maintain the chosen hierarchies given their semantics. The first question is a policy decision that must be determined by whoever is running the SDS server itself (or defaulted if no configuration is specified); the second is a choice of mechanism that will be shared by whoever participates in that hierarchy. Because the first decision is policy-based, we contend that the best solution to it is to allow for the use of *multiple* hierarchies. Examples of possible useful hierarchies include those based on administrative domains (e.g., company, government, etc.), network topology (e.g., network hops), network measurements (e.g., bandwidth, latency), and geographic location (e.g., using location and distance metrics). They are independently useful because they enable users to make queries that resolve based on them – i.e., querying for a service based on geographic proximity rather than ownership.

Individual SDS servers can choose to 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" for each link, as will be described below). Due to the fact that routing may be performed differently in each hierarchy, a single "primary" hierarchy is required to guarantee that queries can get to all portions of the tree. Our current choice is to use an administrative primary hierarchy, but a better choice would be one based on the underlying network characteristics – such as topology – because such a hierarchy requires no manual setup and is robust to network partitions.

Our previous descriptions of SDS client/server operation does not address how parent/child relationships are determined, only the mechanisms used to maintain them once they are known. Examples of possible mechanisms for constructing these parent/child relationships include using manual specification in configuration files (i.e., to indicate administrative hierarchies) or an approach based on external information. Such external information could be geographic data (e.g., through the use of GPS or DNS LOC records [5]), topological data (e.g., using topology discovery [14, 21], multicast administrative scoping [17], a variant of expanding ring search [7]), or network measurement (e.g., using a tool such as SPAND [27] to derive bandwidth and latency information). In these latter cases, such information can be shared via a global multicast address and the neighbor relationships (and resulting tree) inferred from it in a manner analogous to Internet link-state routing [16].

Individual node failures can be tolerated in the same manner as is used to tolerate single-server failure in the local-area case: have a cluster of workstations listen in on the announce/listen messages and leverage the indirection to transparently select amongst themselves.

4.2 Description Aggregation and Query Routing

To prevent the servers in the upper tiers of the hierarchy from being overloaded by update or query traffic, the SDS architecture must keep updates localized. This implies that individual service descriptions and queries must be *filtered* as they propagate up the hierarchy. We describe this process as the *lossy aggregation* function of the hierarchy. At the same time, the aggregation function must be designed such that the summarized information (aka *index*) can be queried as to whether a given piece of information is, or is not, contained in it. Applied to the SDS, this means that 1) service description data must be summarized/indexed as it travels up the hierarchy, allowing control over the rate of updates to the root, and 2) queries must be able to be compared against these indices to determine whether the branch they are summarizing contains a match for that query. Performing the former operations as updates occur at the leaves is called "description aggregation;" performing the latter function while iterating through the tree is called "query routing." In all cases, service descriptions are only stored at the servers where they are being periodically refreshed; only summaries are sent up the tree.

The SDS has a far more difficult problem than most systems that build an application-level routing infrastructure and use it. This is due to the allowance for multi-criteria selection (arbitrary attribute-value pairs) in queries. The novelty of the SDS is that it is attacking the wide-area discovery problem for the case where queries *do not have a hierarchical structure embedded in them.* Multiple systems have solved wide-area scaling and non-hierarchical queries independently [18, 31, 13]; none that we know of have succeeded at addressing both.

We now look at one approach to lossy aggregation and query routing. This approach is based on the use of *hashing* and hash summarization via Bloom filtering.

Hashing summarizes data by creating a unique N-to-M mapping of data vales, where M is a short fixed length bitstring and N can be arbitrarily long. Unfortunately, because SDS queries contain subsets of the tags rather than exact matches, just computing a hash for each service description is not sufficient: all possible matching query values hashes – so-called "subset 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 two obvious problems with computing all possible subset hashes: the amount of computation required, and the amount of space required to store the large number of hashes produced (seen as memory usage at local servers and update bandwidth on the network).

Our solution the computation problem is to limit the number of subset hashes by limiting the number of cross-products of tags that are hashed (e.g., only singles and pairs). Incoming queries must be similarly broken up into groups of tag combinations and checked to ensure there are no false misses. Limiting the computation in this manner increases the probability of false positives, but addresses the exponential computational growth in a manner that gives a "knob" that can trade reduced probability of false positives for additional computation and vice-versa. The knob is the "depth" of the cross product (number of tag combinations hashed).

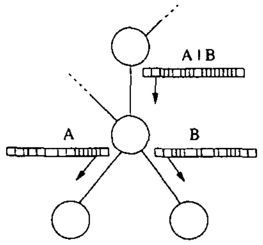

Figure 4: Aggregation of Bloom filter tables.

Even given a solution to the computation problem, there still remains the second problem: space. Each service has tens (or even hundreds) of hashes, and all these hashes must be stored locally; more worrisome, the hashes must be propagated up the hierarchy because they are our index. Our approach to solving this problem is to use Bloom filters [3, 8]. A Bloom filter is a summarization method that uses a bit vector with bits set in positions corresponding to a set of hash values of the data. The key property of Bloom filters is that they provide hashed summarization of a set of data, but collapse this data into a fixed-size table, trading off an increased probability of false positives for index size – exactly the knob we need to address the issue of long hash lists. This approach never causes false misses, thereby maintaining the correctness of our lossy aggregation function. The basic probability of false positives (independent of limiting the number of tag cross-products hashed) can be reduced by using more hash functions and/or longer bit vectors [9], though these numbers must be sized based on the root node – i.e., sized based on the acceptable false positive rate at the root, which knows about *all* service hashes and shouldn't just be "too full of ones."

To summarize how these ideas are applied to the SDS: each SDS server applies multiple hash functions (e.g., using keyed MD5 and/or groups of bits from a single MD5 as in [9]) to various subsets of tags in the service announcements, and uses the results to set bits in a bit vector. The resulting bit vector (the index) summarizes its collection of descriptions. A query against this bit vector is resolved by multiply hashing it and checking that all the matching bits are set. 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 (i.e., makes the routing non-optimal), but does not sacrifice correctness.

If an SDS server is also acting as an internal node in the hierarchy, it will have pointers to its 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 (in the form of differential updates to conserve bandwidth), who then associates it with that branch of the tree. This is illustrated in Figure 4. 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 chil-

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 1: Timings of cryptographic routines

dren; if not, it passes it upward. If it is coming down the hierarchy, the query's computed bit locations are checked against the local and child tables. If there is a hit in the local table, the query is resolved locally. If there is a hit in any children's tables, the query to routed down the matching children's links. If neither of these occur, it is a known miss.

A final problem to address: the bit vectors must be cleaned when a service dies (i.e., we would like to zero their matching bits). Bits cannot be unset, though, because another hash operation may have also set them, and zeroing them would not preserve correctness (i.e., could cause a false miss). To address this, the tables must either be periodically rebuilt, or per-bit counts must be tallied and propagated along with the tables.

5 System Performance

In this section, we examine the performance of the SDS and the XSet XML search engine. The results we present are for single client to single SDS server interactions, and are used to calculate the number of clients an SDS system can handle and to verify that the security features of the system do not greatly reduce performance.

The measurements we will present were averaged over 100 trials and were made using Intel Pentium II 350Mhz machines with 128 MB of RAM, running Slackware Linux 2.0.36. We used Sun's JDK 1.1.7 with the TYA JIT compiler to run each component of the SDS system. 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 assert that the majority of SDS queries will contain a small number of search constraints, and use that model for our performance tests. Our tests on XSet were done on a large set of small XML files, similar in complexity to typical service descriptions. Finally, SDS uses an authenticated RMI implementation developed by the Ninja research group [33], which we modified to use Blowfish [24] (instead of Triple DES) for encrypting the data sent over the network.

5.1 Security Component

In this section, we take a closer look at the cost of the security mechanisms used in the SDS. Specifically, we examine the individual costs of using Java implementations of certificates and asymmetric/symmetric cryptography. As our results show, the costs are relatively high, but for the most commonly used component, symmetric encryption, the cost is small enough to allow the system to scale reasonably well.

Files	*ms / query*
1000	1.17
5000	1.43
10000	2.64
20000	2.76
40000	4.40
80000	5.64
160000	6.24

Table 2: XSet Query Performance

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

Table 3: Query Latencies for Various Configurations

Table 1 lists the various costs of the security mechanisms. As it shows, we profiled the use the DSA certificates [23] for both signing and verifying information, RSA [23] encryption and decryption as used in the service broadcasts, and Blowfish which is used in authenticated RMI. Note, 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 1 KB 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 is an especially important result for our system, since the SDS was designed to leverage the fast performance of symmetric key algorithms. We also note that the DSA verification time is especially high because it verifies multiple signatures per certificate: the certificate owner's signature and the certificate authority's signature.

5.2 XML Search Component Using XSet

We use the XSet XML [34] search engine to perform the query processing functionality needed by the SDS. To maximize 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 2. To reduce the cost of query processing, validation of a service description against its associated Document Type Definition is performed only once, the first time it is seen, not per query or per announcement.

5.3 Performance and Throughput

Table 3 lists the performance of handling various types of SDS queries: both null queries and full queries, in both secure and insecure versions of the SDS. We also measured the service announcement processing time to be 9.2 ms, which is the time the SDS takes to decrypt and process a single service announcement. This was measured using 1.2 KB service announcements

Description	*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 4: Secure Query Latency Breakdown

The other measurements in Table 3 show the performance of various components of the SDS system. For example, the null query time on an insecure SDS system demonstrates the RMI and network overhead, since no time is spent on encryption or searching. The difference between this time and the query time on an insecure SDS indicates the time spent on search overhead. Likewise, the null query time on the secure SDS demonstrates the amount of time spent on the security features. We should point out that the time of a secure SDS query is much higher than the search time plus secure non-query numbers because a secure query performs more encryption/decryption than the secure null query, and also it uses the client's capabilities to perform the search. Note that these times do not include session initialization, since this cost is amortized over multiple queries.

Table 4 shows the average performance breakdown of a secure SDS query from a single client. The SDS server was receiving service descriptions at a rate of 10 1.2 KB announcements per second, and the user performed a search using 7 different capabilities. The SDS was only searching twenty service descriptions, but as the XSet performance numbers show, additional search engine file-handling would contribute very 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, network traffic, and array copies. Not shown in table is that server processing time for the same operation takes about 60 ms.

Using these performance numbers, we approximate that a single SDS server can handle a user community of about five hundred clients sending queries at a rate of of one query per minute per client.

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 [18] and Globe [31] (both conceptual descendents of Grapevine [25]) 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 publicly discoverable; 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 simpler than that attacked by the SDS.

6.2 Condor Classads

The "classads" [20] (classified advertisements) 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 services, classads do not offer secure communication between parties.

6.3 JINI

The Jini [28] 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 [29], 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 [29], 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 believe Jini services and devices can be made to cooperate easily. By adding a Java object to XML generator proxy that speaks the Jini discovery protocol on one end, and SDS broadcasts on the other, we can integrate the Jini federation into the SDS hierarchy and benefit from the strengths of both systems.

6.4 SLP

The Service Location Protocol (SLP) [11], and its wide-area extension (WASRV) [22], 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 the IETF SLP draft [11], 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 the scaling beyond the local area, there are actually two different mechanisms: named scopes and brokering. SLPv2 [11] has a mechanism that allows the local administrative domain to be partitioned into named User Agent "scopes." This scheme was not designed to scale globally, as it has a flat scoping namespace. Scopes are optional, though, allowing evolutionary growth while retaining backward-compatibility with "unscoped" clients. To address wide-area usage, the WASRV draft extension has been proposed [22]. The suggested 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 selectively listen to multicasts from other 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 the basic problem of a lack of hierarchy imposed on the global set of available services. 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 quadratically in the number of AAs/BAs.

One of the most interesting 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 [11]. 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 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 a 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 such 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.

7 Conclusion

The continuing explosive 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 easily handles the addition of new servers and services, while also recognizing when existing services have failed or are otherwise no longer available. This feature will be very important in the future, where given the number of components, failures will be a frequent occurrence.

The use of XML to encode service descriptions and client queries also gives the SDS a unique advantage. 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 aids services in protecting sensitive information and clients in locating trustworthy services. In this age of integrated networks and digital commerce, this feature will be greatly appreciated by both clients and service providers.

Continuing work on the SDS includes finishing the wide area implementation and additional benchmarking. Once the infrastructure is in place, the SDS will be used with components of the Ninja system – and other Internet systems – allowing us to gain practical experience with real services and client applications.

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 Andrew Begel and the anonymous reviewers for their comments.

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 Proposed Recommendation, December 1997. http://www.w3.org/TR/PR-xml-971208.

[5] 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.

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

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

[8] Fan, L., Cao, P., Almeida, J., and Broder, A. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *Proceedings of SIGCOMM '98* (1998).

[9] 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.

[10] Gosling, J., and McGilton, H. The Java language environment, a white paper. http://java.sun.com/docs/white/langenv/, May 1996.

[11] Guttman, E., Perkins, C., Veizades, J., and Day, M. Service Location Protocol, Version 2. IETF, November 1998. RFC 2165.

[12] Handley, M., and Jacobson, V. *SDP: Session Description Protocol.* IETF, 1998. RFC-2327.

[13] Howes, T. A. The Lightweight Directory Access Protocol: X.500 Lite. Tech. Rep. 95-8, Center for Information Technology Integration, U. Mich., July 1995.

[14] 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).

[15] Maher, M. P., and Perkins, C. Session Announcement Protocol: Version 2. IETF Internet Draft, November 1998. draft-ietf-mmusic-sap-v2-00.txt.

[16] McQuillan, J., Richer, I., and Rosen, E. The New Routing Algorithm for the ARPANET. *IEEE Transactions on Communications 28*, 5 (May 1980), 711–719.

[17] Meyer, D. Administratively Scoped IP Multicast. IETF Internet Draft, June 1998. Internet Draft, work in progress.

[18] Mockapetris, P. V., and Dunlap, K. Development of the Domain Name System. *Proceedings of SIGCOMM '88* (August 1988).

[19] Perkins, C., et al. IP Mobility Support. IETF, October 1996. RFC 2002.

[20] 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).

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

[22] ROSENBERG, J., SCHULZRINNE, H., AND SUTER, B. Wide area network service location. IETF Internet Draft, November 1997.

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

[24] 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.

[25] 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.

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

[27] 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).

[28] SUN MICROSYSTEMS. Jini technology architectural overview. white paper. `http://www.sun.com/jini/whitepapers/architecture.html`.

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

[30] THE NINJA TEAM. The Ninja Project. `http://ninja.cs.berkeley.edu`.

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

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

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

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