SAND: A Scalable, Distributed and Dynamic Active Networks Directory Service

M. Sifalakis, A. Mauthe, and D. Hutchison

Computing Department, Lancaster University, LA1 4WA, U.K.
{mjs,andreas,dh}@comp.lancs.ac.uk

Abstract. In the past a significant amount of work has been invested on architecting active node platforms that solve problems in various application areas by means of programmability. Yet, much less attention has been paid to the deployment aspects of these platforms in real networks. An open issue in particular is how active resources can be discovered and deployed. In this paper we present SAND, a scalable distributed and dynamic architecture that enables the discovery of active resources along and alongside a given network path. One of the main strengths of SAND is its customizability which renders it suitable to a multitude of network environments. As an active service, SAND does not have dependencies on any active platform and at the same time enables an active node to become part of a global infrastructure of discoverable active resources.

1 Introduction

Probably the most important application area for active and programmable networks and fundamental requirement of future networks is the ability to deliver flexible and customisable network infrastructures.

Research in active and programmable networks over the last decade has led to new developments in a number of areas ranging from secure programming languages [1,2], to mobile code techniques [3], execution environments [3,4], active node platforms [5,6,7,8,9], service composition models [9,10,11], etc. However, in spite of the unquestionable value of these advances, the equally important aspect of their real world deployment has received significantly less attention. This often provides fertile grounds to criticism and scepticism regarding the practical usability of active and programmable networks. Aspects such as the discovery of active resources, interoperability and interfacing of heterogeneous platforms (adhering to different programming paradigms), cooperation of different platforms at the control and signalling level, and so on, have been neglected for years.

This paper addresses the issue of service/function discovery in a network. It introduces SAND (Scalable Active Networks Directory), a distributed, scalable, and dynamic architecture that enables discovery and browsing of active resources[1] (subject to administrative policies) along or close to a network path, or within a given network neighbourhood. The two fundamental design goals of SAND are *customisability*

[1] In SAND, "*active resource*" refers to any type of resource related to network programmability such as software (active services, execution environments, NodeOS), hardware (memory and CPU capacity), or *policies* conditioning the use of an active node.

D. Hutchison et al. (Eds.): IWAN 2005, LNCS 4388, pp. 132–144, 2009.

which renders it suitable for a broad range of network environment (sensor networks, manets, mobile networks, fixed infrastructures, and GRIDs), and its cluster-based architecture which promotes *scalability* for different network sizes.

The remainder of this paper is organised as follows: Section 2 examines the requirements of such a service in an autonomic network environment. In section 3 the SAND architecture is presented and in section 4 the deployment and operational aspects of the service are considered. Finally, the paper is concluded in section 5, discussing related work and also providing an outlook. In this work we have focused on the design aspects of the SAND service in order to deliver the required functionality. An implementation of SAND is underway along with a simulated testbed that will allow us to evaluate its performance. The results will be published in future work.

2 Requirements Analysis

In programmable router platforms as well as most capsule-based active network architectures an active node must be able to dynamically find and load/fetch active services from the network (if the service code is not already available locally). This reveals the critical need for service (whole program) or function (library code) discovery.

In order to comprehend the type of functionality a resource discovery service should provide the needs and expectations from such a service have to be understood.

Let $P = <a_1, a_2, a_3, \ldots a_k>$ a vector, of a set of network locations where a user needs active networks support. This may be expressed as a set of IP addresses (assuming a uniform network transport), a sequence of autonomous system (AS) numbers identifying locations as administrative domains, a set of coordinates in the NPS [12] system, or a sequence of abstract namespace/object identifiers (as in OO languages), etc. Given vector P, the ultimate goal of a client is to obtain the available active resources along P. A simple but naïve interface between the client and the SAND service could be a function of the type:

$$ResourceList \quad FindResources \quad (<P>), \tag{1}$$

where *ResourceList* is a list of the available resources along P along with information about the hosting nodes of the resources. However, such an ill-defined interface is not very functional creating the following issues:

- The amount of information in *ResourceList* is proportional to the dimension k of vector P, consuming the end device's time and resources for filtering.
- The queried active nodes queried along P will have to generate larger amounts of data than what is actually needed, thus wasting useful network resources.

A better approach is to use a more expressive interface, which allows for more accurate and fine grained description of the information requested:

$$ResourceList \quad FindResources \; (<P>, <FilterSpec>), \tag{2}$$

The *FilterSpec* parameter corresponds to a filter description that (a) imposes constraints with regard to the type of information requested and (b) permits a metric-assisted ranking of the returned information. The objective here is that only a reduced subset of active nodes in a_i needs to respond to a query and the amount of information returned is minimised and ranked by relevance to the query. On the other hand very

often the active resources will be situated close to a specific data path rather than along it, thus necessitating the re-routing of the data path through them in order to perform active processing. This manifests the ability to discover resources that are within an acceptable proximity from to the data path. Moreover, knowing "*how far*", in network distance, from the actual data path the specific active resources lie, will allow us to assess the cost of a rerouting decision (number of hops, delay, congestion, etc). As a result it should be possible to support proximity metrics (hop count, path delay, etc) as heuristics to specify the range around a data path *D* where available active resources should be solicited. This can easily facilitated in the aforementioned interface as part of the *FilterSpec* parameter. E.g.:

ResourceList FindResources (<P>, FilterSpec (ResDistanceHops(<D>) = 5)),

Note also that it is possible to facilitate discovery in a "neighbourhood" instead of along a path if *P* consists of only one entry, the current locus.

On another subject information exchange between diverse and potentially non-compatible network transports has to be possible. Although addressing nodes between custom networks might be possible (using some sort of universal naming scheme), direct communication between them will not be feasible without "speaking" a common network transport protocol, or (dynamically) employing a protocol translator or adaptation layer. This condition can easily lead to a deadlock situation if the discovery process has to take place atomically along the complete end-to-path before deploying services (as is the case with other approaches). The SAND architecture needs to be flexible enough to allow the interplay between discovery and deployment steps, so that the discovery process completes "*gradually as a path is opened*" towards a destination, by deploying appropriate service elements. Although the cost (delay, overhead) of such a process seems high there will often be cases where the feasibility defies the trade-off, thus justifying such a design. In summary:

- If the discovery process cannot be completed atomically for *P*, it should not fail.
- The service nodes should support *query chaining* whereby they are able to (recursively) proxy queries across different network transports.
- The query protocol supported by our architecture must be transport independent.

3 The SAND Architecture

SAND[2] is an active service that enables an active node to participate in a scalable distributed and dynamic directory-lookup service, storing information about resources, services, policies and access control information (to govern the way the information is used and distributed). Being an active service itself it is generic and does not impose any special requirements on the platforms it is deployed on.

The SAND front-end interface is based on LDAP [14], which provides a simple, flexible and lightweight information access mechanism. LDAP allows easy interoperability and cooperation between heavily customised instances of SAND. The standard interface conforms to the one described earlier using two interchangeable APIs. Filter descriptions can be used to optimise a search. From a client perspective SAND is an opaque and ubiquitous service that can be used to discover active resources in the environment.

[2] Due to space limitation several details are omitted. A detailed description is available in [13].

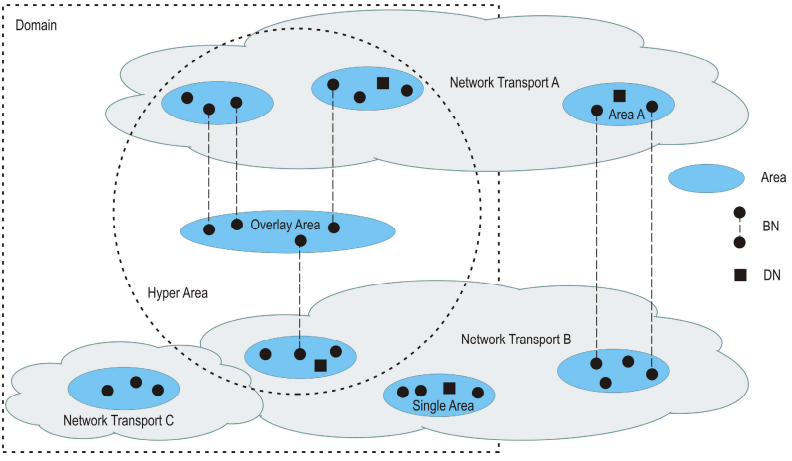

Fig. 1. The main concepts in SAND and their organisation in the Network

From the viewpoint of an active node SAND is a peer-based system, which it may join and register its resources with. Each participating node undertakes an "equal" role in the maintenance of the directory and is responsible of responding to client queries on behalf of the whole directory. This makes the service fault-tolerant and more resilient in episodic environments.

3.1 The SAND Framework

A SAND *node* $n \in \boldsymbol{S}$ ($\boldsymbol{S}$ the set of all SAND nodes), is any active node that runs the SAND service. Specifically such a node is one that has the following capabilities:

- Support of join/leave/peering functions in the SAND system and advertise its resources and services through it.
- Provides the client SAND APIs for answering client requests.
- Maintains a directory for its active resources, services and management/access policies.
- Maintenance of (unique) identification information and a set of hash functions to can generate a set of keys. These keys can be used to address the node.
- Support of an API for integrating information indexing functions and ability to communicate the index information to other SAND nodes.

These capabilities provide a SAND node with 2 very important properties: (a) *Identification* within the SAND system (and potentially beyond) which makes it discoverable and (b) *Accessibility* to its directory store for browsing and listing its resource and service information. These properties essentially provide the knowledge of *how* to reach the SAND system in a network location is and see *what* resources are available.

In SAND, the smallest functional entity is the SAND node. However, the smallest opaque service entity is an "*area*" *A*, which refers to a group of SAND nodes coupled in a peering relationship (figure 1):

$$A = \{ \bigcup n_i \Big| \, |i| \geq 1, n_i \in \boldsymbol{S} \} \tag{3}$$

A SAND node can simultaneously be a member of more than one area, participating in each one of them independently. For an area the following postulates hold:

- Each node can communicate with every other node in the same area. This implies a uniform network (overlay) transport across the area.
- The area members cooperate to provide a distributed, externally opaque, directory base that stores information collectively for the whole area.
- Externally the area is represented by (unique) identification information and there are functions used to generate area-wide IDs from that information.
- There are common procedures and methods among area members for summarizing the directory stored information and communicating it to other SAND nodes.
- Area members that are also members of other areas may (individually or collaboratively) volunteer to proxy requests and selectively advertise summary directory information from one area to another. Such nodes are *area border nodes (BN).* The *area border* representation can be uni or bi-directional.

It follows, from definition (3), that for $i = 1$ a single SAND node also constitutes an area. This also complies with the postulates for a SAND node and an area. Some very interesting developments derive from this observation. First of all every SAND node is member of at least one SAND area, the *self-area,* and therefore it is self sufficient. Seeing it in exactly the other way round, every area can be represented as a node within another area (by means of its *BN*s). This creates a mirage of locally concentrated resource information about a large number of networks (running in other areas), in a scalable way and therefore hiding complexity from the servicees.

If in definition (3) we replace n_i with A_i, we can produce the recursive definition of a SAND *hyper-area* (area of areas – figure 1) A^d, of dimension d.

$$A^d = \{ \bigcup A_i^{d-1} \Big| \, | i |, | d\text{-}1 | \geq 1 \} \tag{4}$$

The concept of a hyper-area is very important as it describes the construction of a global SAND service recursively, starting from the basic entity, the SAND node, and using the same primitives and functions, re-occurring at different scales.

The concept of an *area* is of key importance for the SAND service as it promotes scalability by dividing the network into independently configurable clusters, enables network and/or host mobility without impacting the stability and opaqueness of the service, and finally confines the distribution of the directory content.

Another abstraction used in SAND is the *SAND domain* (figure 1). This is used to signify the boundaries of a network administration authority. A domain is nothing more than a set of SAND areas under the same administrative management and in that sense it can serve as a unique common identification attribute for a group of areas.

3.2 SAND, under the Hood

The functionality of the service discovery mechanism can be decoupled in two abstract notions: (a) *Location* of required service and (b) *Type* of required service.

A client must have sufficient flexibility in describing *what* resources and services it needs and *where* within the network. On the other hand the server side must be able to *locate* service points at those network locations and *browse* the available resources.

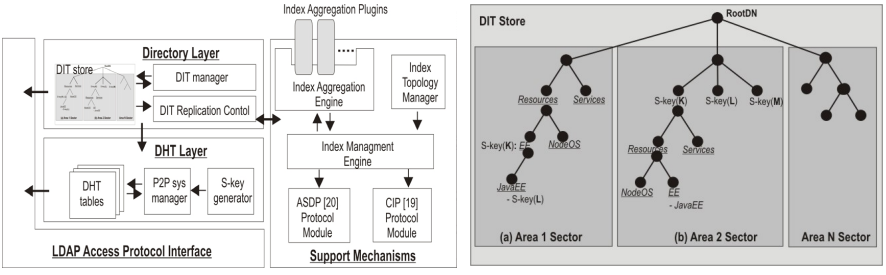

Fig. 2. The SAND Architecture and structure of the DIT in the Directory store

Internally in SAND these aspects are addressed independently by two different, yet complementary, mechanisms perceived as 2 distinct layers. Figure 2 is a block diagram of the SAND architecture in an active node, showing the most important components, their interactions and their relative positions within the architecture.

The DHT Layer. DHTs refer to a technology [15] that has become popular because it can provide a sophisticated and scalable (key) lookup facility, based on extensive use of hashes. In SAND we deployed DHTs as a means of looking up SAND entities. The DHT layer accommodates a set of structures that map keys to network transport-specific node addresses; thus providing a generic, abstract and scalable way for identifying, addressing and locating SAND entities, irrespective of network transport.

These keys, called *s-keys* are somewhat different from conventional hash keys (addressed here as *h-keys*). They derive from, and thus relate to, the identification information of a SAND entity. If an s-key corresponds to a SAND node, a transport address of the node is stored in the key map. If the s-key corresponds to a hyper-area the transport address of one or more *DN*s are kept in the key map. Different sets of key maps allow a SAND node to partake in more than one area.

The concept of an *s-key* is the fundamental primitive of customisability at this layer. In contrast to h-keys, s-keys are produced from a so called *S-function,* which combines a hash function *H(x)* and a "normalisation" function *N(x)*:

$$S(x) = a \cdot N(x) \oplus (1 - a) \cdot H(x), \quad where \;\; 0 \le a \le 1 \tag{5}$$

x indicates the network ID of a SAND node or hyper-area (e.g. the network address). The algebraic operation ⊕ determines how the two terms are combined to form the s-key. The coefficient *a* is the main amortizing factor and expresses the relative weight of the two terms in the generation of the s-key. Essentially, *a* determines how random or relevant to the network ID the s-key is. This is better realised through the following example: Assume IPv4 based transport, ⊕ to equal arithmetic addition, *H(x)* a randomising hash function that returns a random number from an IP address, and *N(x)* a function that masks an IP address to return its /28 netmask. Assuming *x* represents a continuous subset of IP addresses then in figure 3 graph (a) shows the distribution of the *H-term* (*a = 0*), and graph (b) shows the distribution of the *N-term* (*a = 1*). If a regression line would be drawn the correlation coefficient *r* for the

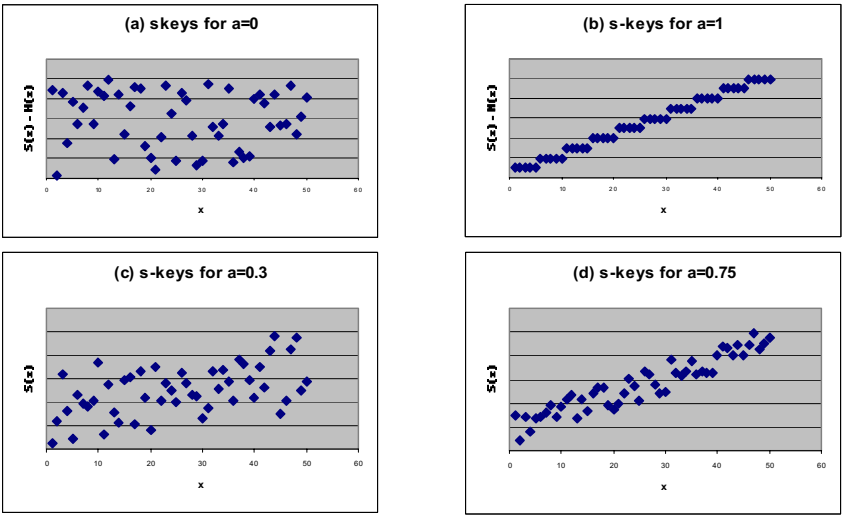

Fig. 3. Relationship between s-keys and network address for different values of *a*

H-term would be close to 0 showing the independence of the h-keys and the corresponding IPs, whereas for the *N-term, r* is close to 1, showing the dependence on the IP addresses. The graphs in (c, d) show the distribution of the s-keys *(S(x))* for various values of *a*, to imply a stronger (d) or weaker (c) relationship to the IP addresses. Therefore it is possible to customise the representation of s-keys in the DHT layer, in order to include/exclude locality information, and control the number of address resolution steps. This, combined with the flexibility to select a search algorithm allows the customisation of the DHT layer for various environments.

The Directory Access Layer. On top of the DHT layer in the SAND system is the directory layer providing access to information about the active resources and services in a SAND node or area, their exact locations and their access permissions.

Although it may be possible to integrate this function in the DHT layer, the not-always structured nature of the s-keys constrains the ability to aggregate and perform effective indexing. This creates scalability problems due to the amount of information that needs to be accumulated at the *BN*s, increasing the memory requirements and the search cost. For this reason a more structured form of identifiers is adopted for this layer, i.e. the ASN.1 *object identifiers (OIDs)* [16], which lends itself well to directory-based solutions (fast searching, aggregating).

As a data model for the resource information the LDAP system [14] has been selected. It builds upon the notion of *OIDs*, and provides a lightweight, flexible and extensible solution, optimised for fast read operations. In LDAP information is organised in objects and attributes describing those objects. Objects belong to classes defining their type. An extensible schema language describes how to interpret the information related to an object class and perform operations between attribute types. Objects follow a hierarchy (imposed by the *OID*s) for efficient searching, called *directory information tree (DIT)*. To improve search performance the scope of a search operation can be limited to specific parts of the DIT through search filters.

In SAND each node maintains its own complete, locally rooted DIT which is partitioned in a number of sectors equal to the number of areas that the SAND node is a member of (figure 2). Each sector represents a sub-tree dedicated to one area, holding specific information related to that area. All sectors converge at the RootDN of the SAND node[3] DIT. The sub-tree structure of each sector needs to be shared by all area members and is determined on a per area basis. This sub-tree forms the area *virtual DIT* and enables the SAND nodes in the area to have a common reference structure for their intra-area communications.

Since all members of an area need to have a synchronised view of the virtual DIT structure, we need an exchange mechanism for the area-wide graph along with s-key information depicting the nodes responsible for each part of the DIT. As of version 3 of LDAP, each entity maintains within the directory the so called *subschema* information that encodes among other things its DIT structure. This enables one to access it through the standard LDAP interface. In SAND the subschema is being extended to include s-key based attributes. The exchange/refresh of the virtual DIT across the whole area is facilitated by sequenced updates to a subset of s-keys in the key-tables.

The interfacing of the directory layer with the DHT layer requires the representation of a node in the LDAP data model. Easily a SAND entity (node or area) is described as a *SandIdObject* in the schema, storing all the node specific information.

The DIT structure as well as the information exchange model (push/pull, periodic/immediate) is customisable per area.

SAND Information Aggregation and Indexing. Consider the two virtual DIT structures in sector 1 and 2 in figure 2, and a SAND node that has received a query for a Java EE. If not enough information to answer the query is available locally by consulting the virtual DIT map, in case (a) it will propagate the query to node *K* (or respond with a referral to *K*). In case (b) though the process would be significantly more expensive as it would have to send the query in parallel to all active nodes, and expect to receive their responses. If however, some index information was available about the actual contents of the area in the above case (b), it would enable the SAND node to know that only *L* has a Java EE and so answer the query itself or have hints that only node *L* might have a Java EE and so prune and propagate the query only to *L*. In either case the amount of consumed resources would be significantly less.

SAND considers an optional indexing framework customisable to match different configurations trading-off memory for search efficiency. The *Index datasets*, which refer to record-sets of indexed data ("hints" depicting where more explicit information can be found) are stored within a node-local directory as directory objects.

To a certain degree the effectiveness of the indexing mechanism relies entirely on the ability to produce small and fast searchable datasets. The *Index aggregation mechanism,* i.e. the process of combining index records in a dataset is a two step operation on a dataset *D*: (a) collecting index information and (b) reducing it:

More important is the *Reduce* operation which has two types (and therefore two types of aggregation), namely the *implicit* and the *explicit* (algebraic) reduction.

In *implicit* aggregation the *Reduce* operation is defined as the intersection of the indexes in a dataset *D*:

[3] Note the differentiation between a "SAND node" which refers to the hosting device and the "DIT node" which refers to a location in the DIT.

$$\text{If } D = \{I_1, I_2, \ldots I_d\}, \qquad IReduce\ (D) = \bigcap_i^d I_i \tag{6}$$

Practically this means that the reduced index will have as an *OID* the common prefix of the *OID*s of all the indexes in the dataset, and as attributes only the common attributes types among all the indexes in the dataset (an example is given in [13]). On the other hand in *explicit* aggregation (*EReduce*) one can define any custom algebraic operation to modify the indexes in a dataset. Practically this may involve combining indexes, replacing an index with another, changing *OID* type, etc. Occasionally an *explicit* aggregation operation will simply modify the information in an index so as to make it *implicitly* reducible. Functions that perform explicit aggregation can be loaded at run-time as dynamic library plugins. Information on how the index can be aggregated with other indexes in a dataset is provided through the *eReduce* and *iReduce* attributes of the *IndexObject* class and may need to be area wide or area specific.

Finally for the *index exchange* one can use the standard LDAP interface as the datasets reside within the directory. However it is more efficient if this process occurs at a lower priority than actual query servicing, and so SAND uses the Common Index Protocol [17], which is a generic mechanism for establishing indexing topologies.

4 Deployment and Operation

4.1 Start Up and Initialisation

Once an active node has the SAND framework installed and initialised it will typically try to do one or more of the following (depending on the policy configuration):

a) Generate s-keys from its identification information for every area it participates (defined in the bootstrap configuration). This will involve at least the self-area.
b) Build the DHT structures in the bottom layer for every area it is a member of.
c) Construct the DIT and populate the directory with local resource information.
d) If indexing is enabled, initialise its local index structures, datasets, and plugins.
e) Start "beaconing" for the areas it is a member of, so as to announce its existence and invite other nodes to join.
f) Start listening for join invitations from other areas.

The mechanisms for performing the last two steps are outside of the scope of this work since semantically they are aspects related to self-association mechanisms.

4.2 Join and Leave Operations

When a SAND node in listen mode receives a "beacon" about the existence of an area it needs to do the following: (a) decide if it "wants" to join and (b) find out if it is allowed to join. Both are subject to bilateral policies. First there is need for a negotiation/authentication phase, during which a set of preconditions have to be checked. To comply with the overall philosophy, in the current prototype these preconditions are expressed a set of attributes in a *JoinInfoObject* object, at a fixed location in the DIT. The joining node can contact the beacon node using the standard LDAP interface to query and check the attributes of interest in this object and decide if it complies with its local policies for joining. The beaconing node can optionally do the same.

Once the joining node has decided to proceed it issues a join request at the DHT level, using the standard join method of whatever DHT system is used. At this point it is possible for the area node to reject the join request. If the newcomer completes the join successfully, this means that it has already initialised its DHT layer data structures and has a valid s-key. In order to participate in the SAND area at the directory level, it initially undergoes a quality assessment period during which it operates simply as a replicating node. During this process a *supervisor* node (typically the one that sent the "invite" beacons) is responsible to "keep an eye" and assess the new member by issuing occasionally requests as a client in order to verify the validity of its contents. After the assessment period is over (decided by the supervising area node), and depending on the area specific configuration, the new member may be assigned responsibility for a specific part of the DIT or remain a replicator.

If a node wants to gracefully leave the area, it deregisters its resources from the directory, removes its s-key from the virtual DIT and triggers an update, and finally informs accordingly its peers at the DHT level.

Finally, if a node is likely to become a *BN* or a *DN* the following actions are taken:

- As a *BN*, it establishes an index topology for the area's contents
- Generates and advertises an area s-key so as to respond on behalf of the area, to external requests.
- As a *DN*, it builds domain ID information so as to be able to respond to queries originating outside the domain.

4.3 Service Operation

When an entity wants to find an active resource it issues a request to its nearest SAND point through the abstract interface described in (2) or otherwise using the standard LDAP interface. The request is then decoupled in three parts: (a) *where* service must be provided, (b) w*hat* service needs to be found, and (c) heuristics for making the request more or less specific. Each of the first two questions essentially produces a part of a unique request identifier:

$$Request\ ID \equiv location\text{-}ID \mid resource\text{-}ID$$

$$Location\text{-}ID \equiv s\text{-}key = S(location)$$
$$Resource\text{-}ID \equiv resource\ OID = ASN.1(resource)$$

The location-ID (s-key) is resolved in the DHT (bottom) layer of the SAND architecture by contacting the SAND node responsible for the hyper-area and domain that corresponds to that s-key. This node, being part of that location-ID namespace can now produce the correct resource-ID (OID) and perform an internal query for it at the directory level. The resource-ID is looked up in the directory (upper) layer. One can see that for scalability and flexibility the resource-ID has local scope only within the location-ID namespace and therefore is *late-bound* to an OID only after the location-ID has been resolved. If there is not enough information available locally, there should be enough index information so as to know whom to contact next and so a recursion of s-key and OID lookup steps will follow within the hyper-area until the information can be obtained. The search will "halt" either when the information has been found, or if there is not enough information to continue the lookup process.

At every lookup step of the search process the server side might respond with a referral to a new location where the search should continue or act as a proxy in chain mode and perform the next search on behalf of the servicee.

5 Related Work and Outlook

This paper presented the design of a service discovery architecture called SAND. It provides a distributed and dynamic architecture that lends itself well to non-uniform and multi-transport network environments. It enables the discovery of active resources along or alongside a network path or within a given network neighbourhood. The main strengths of SAND are its *customisability*, and its cluster-based architecture which promotes *scalability* for different network sizes.

So far there has not been much work in the area of active resource discovery in heterogeneous networks. None of the approaches we are aware of considers a resource discovery infrastructure that is independent of, or customizable to diverse network environments. Moreover they have limited flexibility as they restrict the discovery process to be completed "atomically" before service deployment. SAND on the other hand is very flexible since:

- The discovery process can precede and succeed partial deployment of services.
- It can facilitate location discovery by using abstract and fuzzy expressions of a network location which can be independent of a specific network transport.

Service discovery mechanisms such as those advocated in [19, 20, 21] are not suitable for the application environments we consider because they are either non-generic for any network, or non-scalable (assume deployment of multicast), or else fairly static (rely on the existence of centralised registries in fixed locations). In [22] the authors consider a more dynamic approach, which enables the discovery of active nodes that are not in the data path. However it relies on other external services such as DNS to map consistently in the domain namespace, the network domain topology, thus assuming that proximity in the network is reflected in the domain names.

An approach which has inspired SAND in some aspects, is presented in [23]. The HIGS algorithm combines the discovery and deployment process in one, architecture. In HIGS, SAND could be providing the mechanism for performing the solicitation and summarisation steps.

The authors in [24] provide a mechanism for discovering active nodes in close vicinity. Although not a resource discovery mechanism per se, this work can be used in conjunction with SAND as a self-association mechanism for discovering SAND areas and nodes in close range.

In [25] the authors devised a sophisticated approach of optimally selecting service points in a network between two end nodes. Although the deployment of composite services in the network is the focus of this work, they do not address the discovery aspect, thus assuming an external mechanism like SAND to provide a number of available service nodes.

SAND is currently being implemented and refined. On-going work based on simulation aims to compare the performance of SAND against a fully DHT based solution. Future work will study more extensively the various aspects of unattended creation of

index topologies in order to further improve its customisability. We envision the proposed architecture to provide a sustainable solution for autonomic infrastructures.

Acknowledgements. Special thanks to Stefan Schmid (NEC labs, Europe), for the insightful discussions on the topic.

References

[1] Wakeman, I., Jeffrey, A., Owen, T., Pepper, D.: SafetyNet: A Language-Based Approach to Programmable Networks. Computer Networks and ISDN Systems 36(1) (2001)

[2] The Caml Language. Online Reference, INRIA, http://caml.inria.fr/

[3] Wetherall, D.J., Guttag, J., Tennenhouse, T.L.: ANTS: A toolkit for building and dynamically deploying network protocols. Proc. of IEEE Openarch (April 1998)

[4] Hicks, M.W., Kaddar, P., Moore, J.T., Gunter, C.A., Nettles, S.: PLAN: A Packet Language for Active Networks. In: Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming, pp. 86–93 (1998)

[5] Paterson, L., Gottlieb, Y., Hibler, M., Tullmann, P., Lepreau, J., Schwab, S., Dandelkar, H., Purtell, A., Hartman, J.: An OS Interface for Active Routers. IEEE Journal on Selected Areas in Communications 19(3), 473–487 (2001)

[6] Merugu, S., Bhattacharjee, S., Zegura, E., Calvert, K.: Bowman: A Node OS for Active Networks. In: Proceedings of IEEE INFOCOMM 2000, Tel Aviv, Israel, March 26-30 (2000)

[7] Keller, R., et al.: An Active Router Architecture for Multicast Video Distribution. In: Proc. of IEEE INFOCOM, vol. (3), pp. 1137–1146 (2000)

[8] Keller, R., et al.: PromethOS: A dynamically extensible router architecture supporting explicit routing. In: Sterbenz, J.P.G., Takada, O., Tschudin, C.F., Plattner, B. (eds.) IWAN 2002. LNCS, vol. 2546, pp. 20–31. Springer, Heidelberg (2002)

[9] Schmid, S., Finney, J., Scott, A.C., Shepherd, W.D.: Component-based Active Network Architecture. In: IEEE Symposium on Computers and Communications (July 2001)

[10] Merugu, S., Bhattacharjee, S., Chae, Y., Sanders, M., Calvert, K., Zegura, E.: Bowman and CANEs: Implementation of an Active Network. In: Proc. of 37th Conference on Communication, Control and Computing (September 1999)

[11] Bossardt, M., Antik, R.H., Moser, A., Plattner, B.: Chameleon: Realising Automatic Service Composition for Extensible Active Routers. In: Wakamiya, N., Solarski, M., Sterbenz, J.P.G. (eds.) IWAN 2003. LNCS, vol. 2982. Springer, Heidelberg (2004)

[12] Eugene Ng, T.S., Zhang, H.: A Network Positioning System for the Interne. In: USENIX Annual Technical Conference (2004)

[13] Sifalakis, M., Mauthe, A., Hutchison, D.: SAND: A Scalable, Distributed and Dynamic Active Networks Directory Service. Technical Report TR-COMP-008-2005, Lancaster University (July 2005)

[14] Wahl, M., Howes, T., Kille, S.: Lightweight Directory Access Protocol (v3). RFC 2251 (December 1997)

[15] Plaxton, C.G.: On the network complexity of selection. In: Proc. of Annual Symposium on Foundations of Computer Science (October 1989)

[16] Abstract Syntax Notation One (ASN.1) and ASN.1 Encoding Rules. ITU-T Rec. X.680–683 and X.690–693 (2002)

[17] Allen, J., Mealling, M.: The Architecture of the Common Indexing Protocol (CIP). RFC 2651 (August 1999)

[18] Sifalakis, M., Schmid, S., Chart, T., Hutchison, D.: A Generic Active Service Deployment Protocol. In: Proc. of ANTA 2003 (May 2003)
[19] Veizades, J., Guttman, E., Perkins, C., Kaplan, S.: Service Location Protocol. RFC 2165 (June 1997)
[20] Microsoft corporation. Windows Server 2003 Active Directory (2003)
[21] Gulbrandsen, A., Vixie, P., Esibov, L.: A DNS RR for specifying the location of services (DNS SRV). RFC 2782 (February 2000)
[22] Karrer, R., Gross, T.: Location Selection for Active Services. Cluster Comp.: Journal of Networks, Software and Applications (March 2002)
[23] Haas, R., Droz, P., Stiller, B.: Autonomic service deployment in networks. IBM Systems Journal 42(1), 150–164 (2003)
[24] Martin, S., Leduc, G.: Dynamic Neighbourhood Discovery Protocol for Active Overlay Networks. In: Wakamiya, N., Solarski, M., Sterbenz, J.P.G. (eds.) IWAN 2003. LNCS, vol. 2982. Springer, Heidelberg (2004)
[25] Keller, R., et al.: Active Pipes: Service Composition for Programmable Networks. In: Proc. of IEEE MILCOM 2001 (2001)