A Taxonomy of Self-configuring Service Discovery Systems

Vasughi Sundramoorthy[1], Pieter Hartel[2], and Hans Scholten[2]

[1] Lancaster University
Lancaster, Lancashire, LA1 4WA, UK
v.sundramoorthy@lancaster.ac.uk
[2] University of Twente
Enschede, 7500AE, The Netherlands
(pieter.hartel, hans.scholten)@utwente.nl

Abstract. We analyze the fundamental concepts and issues in service discovery. This analysis places service discovery in the context of distributed systems by describing service discovery as a third generation naming system. We also describe the essential architectures and the functionalities in service discovery. We then proceed to show how service discovery fits into a system, by characterizing operational aspects. Subsequently, we describe how existing state of the art performs service discovery, in relation to the operational aspects and functionalities, and identify areas for improvement.

1 Introduction

Computer scientists can learn much from how the human body manages itself autonomously, and apply the same techniques to building distributed systems. This is how *autonomous computing*, an initiative of IBM [1] sees the future of computer systems. An autonomous system has at least one of the following four properties [2]:

1. Self-configuring. Systems that adapt automatically to dynamically changing environments. The systems can dynamically add ("on-the-fly") new hardware and software to the system infrastructure with no disruption of services.
2. Self-healing. Systems discover, diagnose and react to disruptions. The objective of self-healing is to minimize outages to keep applications available at all times.
3. Self-optimizing. Systems monitor and tune resources automatically. Self-optimization requires hardware and software systems to maximize resource utilization to meet end-user needs without human intervention. Resource allocation and workload management must allow dynamic redistribution of workloads to systems that have the necessary resources to meet workload requirements.

4. Self-protecting. Systems anticipate, detect, identify and protect themselves from attacks. Self-protecting systems must have the ability to define and manage access to computing resources, to protect against unauthorized access, to detect intrusions and report and prevent these activities as they occur.bu

Autonomy of distributed systems is also one of the fundamental characteristics of the more ambitious *pervasive (or ubiquitous) computing*. The vision of pervasive computing is elegantly articulated in Mark Weiser's acclaimed seminal paper published in 1991 by Scientific American [3]:

"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it."

To realize the Weiser vision of pervasive computing, Satyanarayanan [4] pose system *invisibility* as an important research challenge. Invisibility in the context of pervasive systems means human intervention is so minimal, that technology disappears into the background of everyday life. Therefore, a pervasive system continuously meets user expectations and rarely presents unpleasant surprises.

Invisibility is achieved when the system applies autonomous behaviors; self-configuration, self-healing, self-optimization and self-protection. One of the most widely used techniques contributing to self-configuration is *service discovery*. Service discovery is a fundamental step for intelligent applications, before they can collaborate to perform a certain function; entities are able to self-configure themselves by detecting other entities (hence, services), and are able to self-heal from failures, without the necessity for a professional system administrator.

An example of an application that uses service discovery is an ambience controller that uses time, temperature and location as inputs, to satisfy a particular condition: "in the *evening*, when the outdoor temperature is *hot* and *Alice is home*, set the mood of the home to *cool*". As a result of this condition, the ambience controller discovers a number of services (e.g. clock, location of Alice, temperature, lighting, audio, window) to receive inputs and to produce the output. Subsequently, when the time and temperature is appropriate, and Alice is home, the indoor temperature is reduced, lights are dimmed, windows are shut, curtains drawn and chamber music is played. Therefore, a service discovery system is necessary for autonomous, collaborating applications to arrive to the right inference, and produce the expected results.

Contributions: This work diverges from existing surveys [5–7] which categorize functional features of service discovery protocols, based on architectural and programming platform differences. We aim to:

1. Clarify the fundamental concepts of service discovery
2. Specify the operational aspects that impact the design of a service discovery system, and the resulting design solutions.
3. Provide a taxonomy that first analyzes state of the art solutions with respect to the operational aspects, before comparing functional behaviors.

This work focuses on the *communication aspect of service discovery systems*, such as the type of architecture, topology, functional behaviors of the entities in the system, functional principles for service discovery and the operational aspects that impacts the design of the system. How services are described and accessed are issues outside the scope of this paper. More information on service descriptions can be found by studying existing schemes such as WSDL [8], OWL [9], SSDL [10] and RDF [11]. Meanwhile, RPC [12] and service interface invocation as done in Java [13] are methods to access services once they are discovered. The description and access techniques can be integrated into the service discovery systems described in our work.

The paper is organized as follows. In Section 2, we provide a new understanding of service discovery as a third generation name discovery system. In Section 3, we describe the service discovery system in terms of the participating entities, architecture and topology. We define the service discovery objectives and functions in Section 4. In Section 5, we analyze the operational aspects for a service discovery system. We proceed to summarize a selection of widely known service discovery systems in Section 6. In Section 7, we give a taxonomy of state of the art solutions to the operational aspects, and compare the functional implementations. Finally we conclude and identify areas for improvement.

2 Service Discovery: Third Generation Name Discovery

Service discovery allows applications in a distributed system to discover and share network entities. This goal is shared by a class of distributed systems that we refer to as *name discovery systems*. A *name* is a string of bits that is used to identify a variety of entities, such as computers, peripherals, applications, remote objects, files, etc. In the context of name discovery systems, a name is not the address of an entity (addresses are uninterpreted bit patterns such as Ethernet addresses [14]), but a name is the persistent identifier for the entity, or human understandable textual description [15]. Consistently named entities enable computers to communicate with one another via a distributed system, and share (access to) the entities [16]. A name has a list of *attributes* associated with it. An attribute is a name-value pair that describes a property of the entity

We classify *name services*, such as Grapevine [17], GNS [18] and DNS [19], developed in the 1980s as first generation name discovery systems. A *name server* stores a set of bindings between the name and the attribute list of an entity, and resolves queries for the entity. A query based on the name simply returns the list of attributes that describes the entity.

Directory services such as Profile [20], Univers [21], X.500 [22], LDAP [23] and CORBA's Interface Repository [24] developed in the 1990s are classified as second generation name discovery systems. Directory services provide a more powerful mechanism for querying entities. Directory services perform *attribute-based queries* that return the names associated with the attribute. An example

is a query that on the basis of a given telephone number, returns the name of the associated employee.

The first two generations of name discovery systems share the following context and limitations:

- Context: Computer-based environment with predominantly wired connectivity, where nodes are mostly static, names and attributes rarely change, and the system is reliable
- Limitation: Static infrastructure of servers and directories, that requires configuration and maintenance by privileged system administrators.

Service discovery inherits the fundamental concepts of traditional name discovery systems, where entities are named according to a naming standard, and attribute-based queries return the names of the matching entities. A *service* is defined as the following:

A service is a distinct part of a computer system that manages a collection of related entities and presents their functionality to users and applications [16].

We classify service discovery as a third generation name discovery system because service discovery satisfies the requirements of pervasive computing [3], by *enlarging the context and relaxing the limitations of traditional name discovery systems.*

- Context: An environment consisting of a variety of embedded devices (not just PCs and servers), with wired and wireless connectivity, static and mobile nodes, which undergoes frequent changes of resource availability and attribute values, and is vulnerable to failures.
- Solution: Service discovery provides **self-configuring** and **self-healing** capabilities for a **spontaneous network of devices**. A service discovery system allows entities to enter and leave the system automatically, with *minimum supervision and maintenance.*

Therefore, service discovery is a solution for naming and discovering resources in a versatile, and spontaneous network of devices. Examples can be found in the relatively new areas of home entertainment networking and ambient intelligence.

3 Service Discovery Architecture

This section describes the entities and the different architectures of service discovery.

A service is specified by a *Service Description (SD)*, which typically describes the service in terms of: (1) device type (e.g. printer), (2) service type (e.g. color printing) and/or (3) attributes (e.g. location, paper size). There are three types of entities in a *service discovery system*; a *Manager* owns the SDs, a *User* has

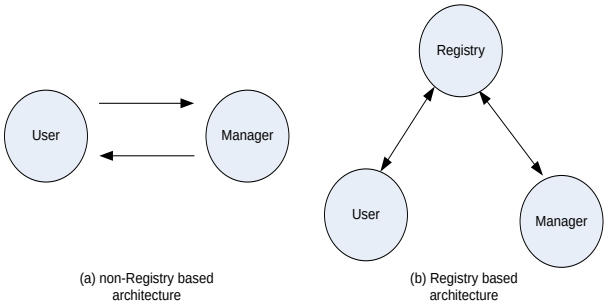

Fig. 1. (a) The non-Registry architecture consists of Users and Managers that multicast queries and service advertisements. (b) The Registry architecture uses unicast for registering services and sending queries.

a set of requirements for the services it needs, and a *Registry* caches available services so that Users can discover the services through *queries* to the Registry. A node can behave as a User, Manager, Registry or a combination of these roles.

Service discovery architectures are mostly formed based on how Users discover services; (1) Users can use a Registry (or a cluster of Registries) as the intermediate entity to discover services, therefore requiring a *Registry* based architecture. (2) Users can directly query Managers, and/or Managers can send multicast *advertisement* messages, therefore establishing a *non-Registry* based architecture. Figure 1 illustrates the two types of basic architectures. In the Registry-based architecture, Registries can be deployed by a system administrator, or automatically elected by the nodes in the system. Once the Registry is available, the rest of the nodes in the system will have to discover the Registry before services can be registered and queried. Unicast communication in the Registry-based architecture reduces network traffic, thus increasing scalability. In the non-Registry-based architecture, Users and Managers can perform multicast queries and service advertisements. Therefore, unlike the Registry-based architecture, the system is not vulnerable to single point of failure issues. However, since extensive multicast is used, network traffic increases, causing scalability issues. A combination of both these architecture can also be implemented, so that the system is more scalable and robust.

Service discovery entities habitually communicate with each other through *logical topologies*. A logical topology is typically used to optimize the way the system propagates and processes messages (such as queries and advertisements), thus optimizing one or more of the following: communication cost, energy efficiency, scalability, resource consumption, robustness, responsiveness and effectiveness (of the queries), etc. We identify four basic logical topologies, based on message propagation; (1) *Meshed topology*, where messages are sent to all listening entities, (2) *Clustered topology*, where messages are sent to a cluster of listening entities (Registries or Users), based on the type of service they provide, or the proximity to the source, (3) *Tree topology*, where messages are propagated

along a hierarchy of Registries, (4) *Unconnected Registries topology*, where messages are sent to any discovered Registry. We describe how the non-Registry and Registry-based architectures use these topologies, and their advantageous and disadvantageous in the next few paragraphs.

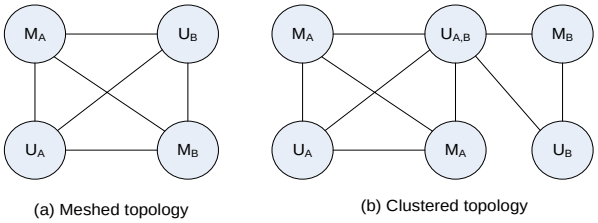

Fig. 2. Logical non-Registry topologies. (a) In the meshed topology, Users, U and Managers, M can listen to each other's queries and service advertisements. (b) In the cluster-based topology, Users, U and Managers, M may form a logical cluster according to some criteria. A and B denotes two different clusters, where $U_{A,B}$ belongs to both clusters, and is able to discover services of both clusters.

The non-Registry architecture is commonly based on two types of logical topologies: (1) Meshed topology, as shown in Figure 2(a), where all entities receive each other's multicast queries and service advertisements. (2) Clustered topology, where entities form clusters based on some criteria (e.g. service type or location). Members of a cluster communicate only with each other, thus service advertisements and queries are limited within the cluster. Figure 2(b) gives an example of the cluster-based topology.

The meshed topology improves the chances of discovering a service (queries are more effective), and the continued availability of the service, because it is not vulnerable to single point of failure issues [25]. However, it increases communication cost, and is less scalable because of the extensive use of multicast (queries and advertisements). On the other hand, the clustered topology makes the system more scalable, but increases the complexity of the system, because Managers and Users will have to establish clusters, and dynamically add and remove their membership. Furthermore, by limiting a query and advertisement to the members of a cluster, the effectiveness of the query is reduced. The scope of discovering a service can be widen by allowing the User to belong to a combination of clusters.

The Registry architecture has four types of logical topologies: (1) An unconnected Registry topology, as shown in Figure 3(a). In this topology, Registries do not communicate with each other, but Managers and Users may associate themselves with multiple Registries. (2) A meshed Registry topology, as shown in Figure 3(c), where Registries communicate with each other as peers. A Registry forwards queries and replicas of its cache to all its peers. (3) A tree-based Registry topology, where Registries form a parent-child relationship, based on

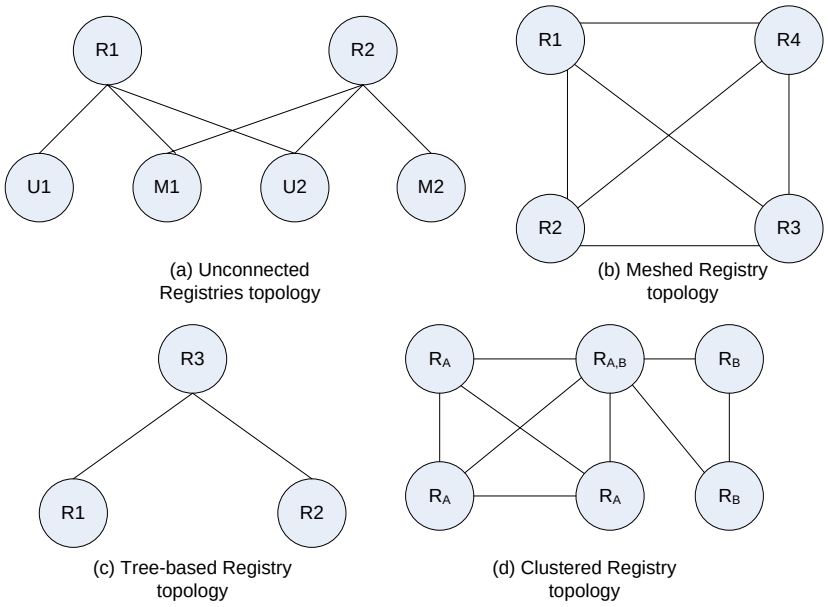

Fig. 3. Logical Registry topologies. (a) In the unconnected Registry topology, Registries, $R1$ and $R2$ do not communicate with each other, but User, $U1$ and Manager, $M2$ may register and discover services from both $R1$ and $R2$. (b) In the meshed Registry topology, Registries are peers to each other, and forward messages to all their peers. (c) In the tree-based Registry topology, Registries $R1$ and $R2$ are child Registries of $R3$. Child Registries may forward messages to parent Registries. (d) In the clustered Registry topology, Registries optimize the tree or mesh topology by limiting query processing to a select few Registries. A and B are two clusters, where Registries, R_A and R_B can only communicate with the members of their own cluster. $R_{A,B}$ is able to communicate within both clusters.

some criteria (such as location or resource-constraints). A child Registry, such as shown in Figure 3(b) forwards queries to its parent when it does not find matching services within its own cache. The query may traverse all or parts of the hierarchy, depending on some query processing optimization criteria. (4) A clustered Registry topology, as shown in Figure 3(d), where Registries form clusters based on service type or location. This topology optimizes the tree-based and meshed topology, where query processing is done only by a select few Registries.

The unconnected Registry topology does not require Registries to synchronize registration data among each other. Therefore, adding a new Registry is not a complicated task. Redundant Registries also provide increased robustness against communication and Registry failures. However, when Users and Managers redundantly communicate with each discovered Registry, communication cost is increased, and scalability is reduced. The other three topologies allow the Manager and the User to communicate with a single Registry only, keeping the service discovery task simple on the side of the Managers and Users (especially suitable for resource constrained nodes). The meshed Registry topology allows all the Registries in the system to communicate with each other, so that the scope of service discovery is system-wide. However this topology is not practical for large systems that span long distances. The tree-based Registry and clustered Registry topology would be more suited for large systems. The tree-based Registry topology allows each Registry to store only a part of the available services, therefore allowing load balancing for systems with high density of nodes (such as by spawning child Registries to share the load). However, the system becomes more vulnerable to single points of failure issues when Registries fail. Registries will have to check continuously on the availability of the parent or child Registry. The clustered Registry topology is more robust against Registry failures, and has better query effectiveness because multiple Registries can respond to the query, from different areas of the system. However, Registries will have to be able to establish and dynamically add and remove memberships to the clusters, causing additional protocol overhead and complexity.

A service discovery system may implement one or a combination of several logical topologies. Therefore, it is necessary to define and prioritize the optimization parameters early in the system design stage, to satisfy the requirements of the system. We will discuss some of these parameters in Section 5

4 Service Discovery Functions

We state the main objectives of service discovery as:

1. *Discover services that match requirements*
2. *Detect changes in service availability and attributes*

Toward accomplishing these objectives, we classify service discovery tasks into four main *functions*; Configuration Discovery, Service Registration, SD Discovery and Configuration Update. The term “configuration” refers to the entities

in the system: Manager, User and Registry. The Configuration Discovery and Service Registration (for Registry based architectures) functions are required for entities in the system to gather knowledge on the availability of nodes and services. The first objective is accomplished by the SD Discovery function, while the second objective is accomplished by the Configuration Update function. Each of the four functions can be accomplished using several different *methods*. We use italics to indicate the methods:

1. **Configuration Discovery** - This function allows Registries to be setup, and identities of entities (e.g. Registries or cluster members) in the system to be discovered. There are two sub-functions of Configuration Discovery:
   (a) Registry auto-configuration - Allows the system to configure one or more Registries automatically through (a) *Registry election* algorithms, or (b) *Registry reproduction*, where a parent Registry spawns a child Registry. The Registry election or reproduction is done based on some criteria such as resource superiority, load threshold, service type or location. Registry auto-configuration is done on the fly, without supervision.
   (b) Entity discovery - Allows entities in the system to discover a Registry or cluster members through (a) *active discovery*, where nodes initiate the discovery by sending announcements, or (b) *passive discovery*, where nodes discover the required entities by listening for announcements. In some systems, discovery via active and passive methods is integrated with the underlying routing protocol to optimize bandwidth utilization.
2. **Service Registration** - This function allows Managers to register their services at a Registry. Registration methods include (a) *unsolicited registration*, where nodes request the Registry to register their services and (b) *solicited registration*, where Registries request new nodes to register. The Registry keeps a cache of available SDs, and updates them according to requests from the Managers.
3. **SD Discovery** - This function allows Users to obtain SDs that satisfy their set of requirements. Users may cache the discovered SDs to reduce access time to the service, and reduce bandwidth utilization by avoiding multiple queries. There are two sub-functions in SD Discovery:
   (a) Query - This is a pull-based model where Users initiate (a) *unicast query* to a Registry, or (b) *multicast query*. The query specifies the requirements of the User. The Registry or Manager that holds the matching SD replies to the query.
   (b) Service notification - This is a push-based model, where Users receive (a) *unicast notification of new services* by the Registry, or (d) *multicast service advertisements* by Managers.
4. **Configuration Update** - This function monitors the node and service availability, and changes to the service attributes. There are two sub-functions in Configuration Update:
   (a) Configuration Purge - Allows detection of disconnected entities through (a) *leasing* and (b) *advertisement time-to-live (TTL)*. In leasing, the

Manager requests and maintains a lease with the Registry, and refreshes the lease periodically. The Registry assumes that the Manager who fails to refresh its lease has left the system, and purges its information. With TTL, the User monitors the TTL on the advertisement of a discovered Manager. The User assumes the Manager has left the system if the Manager fails to advertise before its TTL expires, and purges its information.

(b) Consistency Maintenance - Allows Users and Registries to detect updates on cached SDs. Updates can be propagated using (a) push-based *update notification*, where Users and Registries receive notifications from the Manager, or (b) pull-based *polling for updates* by the User to the Registry or Manager for a fresher SD. (c) In a multiple Registry topology, push-based *update notifications among Registries* can be done to achieve consistency.

Table 1 summarizes service discovery functions and the implementation methods for each function. Every service discovery system implements the functions according to its own design rationale. Furthermore, by clearly defining the functionalities for service discovery, applications can rely on the underlying service discovery protocol to perform without ambiguities.

Table 1. Service discovery functions, methods and related distributed system models

Function	**Subfunction**	**Method**
Configuration Discovery	Registry auto-configuration	(a) Registry election, (b) Registry reproduction
	Entity discovery	(a) active discovery, (b) passive discovery
Service Registration		(a) solicited registration, (b) unsolicited registration
SD Discovery	Query	(a) unicast query, (b) multicast query
	Service notification	(a) Registry notification, (b) multicast service advertisement
Configuration Update	Configuration Purge	(a) leasing, (b) advertisement TTL
	Consistency Maintenance	(a) pull-based polling for update by Users, (b) push-based update notification by Registry to Users, (c) push-based update notification among Registries

5 Operational Aspects of Service Discovery

This section analyzes the design aspects related to the *operational environment* of service discovery systems, and lists solutions found in the wider distributed system paradigm, but tailors them to the service discovery context.

The operational environment influences the design rationale of service discovery systems. For example, a stable, wired office environment, with good system administration may not require too much emphasis on fault-tolerance towards communication and node failures. However, in the context of a less controlled environment such as the home, it becomes a necessity, because home owners are not restricted in how they manage their appliances (unplugging, moving). In a wireless, mobile environment, the system becomes even more vulnerable to certain communication and node failures. We identify the following as design aspects for service discovery in pervasive computing:

1. ***System size*** - We define "system size" in terms of two dimensions: distance and the number of nodes. Small sized systems such as *Personal Area Networks (PAN)* and *Local Area Networks (LAN)* contain a limited number of nodes, and do not require a high degree of scalability. Large systems such as *Metropolitan Area Networks (MAN)* and *Wide Area Networks (WAN)* including the Internet require a scalable service discovery system. Scalability measures include setting up multiple Registries, whether in a tree or mesh topology, and applying query optimization and load balancing techniques to conserve bandwidth.
2. ***Lossy environment*** - Service discovery systems in wireless and mobile networks must assume that they will operate in a lossy environment with communication and node failures. *Communication failures* include message corruption, message loss and link failures. Message corruption is due to interference, noise or multipath fading. Message loss is due to loss of signal caused by physical obstacles, collisions, bandwidth limitations, etc. Link failures, especially in ad-hoc networks, are caused by mobile nodes losing radio contact with the destination node. *Node failures* include crash failures and interface failures. Crash failures are caused when nodes abruptly disappear from the system due to energy depletion, pulled out without warning, and overloaded processors (nodes simply stop communicating). Interface failures mean receiver and transmitter failure. Therefore, service discovery systems should be fault-tolerant. Some examples of fault-tolerant mechanisms in service discovery systems include redundant and replicated Registries, caching of alternate services, primary-based recovery protocols such as Registry monitoring and Registry backup [15], retransmissions and acknowledgments for reliable transmission, and containment of unreliable behaviors by blacklisting suspicious nodes.
3. ***Resource constraints*** - Nodes with hardware constraints are *resource-lean* (low memory, processing power and energy). Systems with resource-lean nodes require resource-aware service discovery functions. One solution is to delegate more tasks to more powerful nodes. In systems with *low bandwidth availability*, cross-layer dependencies such as service discovery with

routing knowledge, and efficient query processing among Registries (e.g. through DHTs) can help conserve bandwidth. Furthermore, load balancing techniques help scale Registry-based architectures so that Registries do not overload.

4. ***Security*** - A secure service discovery system must support *confidentiality, message integrity and availability* [26]. Methods to address these concerns include authentication of communicating entities, access control so only a select few are able to communicate, protection of sensitive service attributes (e.g. location) by hiding the value, data integrity, so that communicating entities can detect when data is tampered during transit, and detection and blacklisting of malicious nodes (including authorized entities). The challenge for a secure service discovery system is to maintain self-configuration of the system, because the owner of the devices will most probably be required to provide authentication and access control. Security also consumes resource due to encryption algorithms. Most service discovery systems assume participating nodes are secure by delegating security to the application layer. However, full fledged deployment of a service discovery system will eventually require some secure measures integrated into the service discovery functions [27, 28]

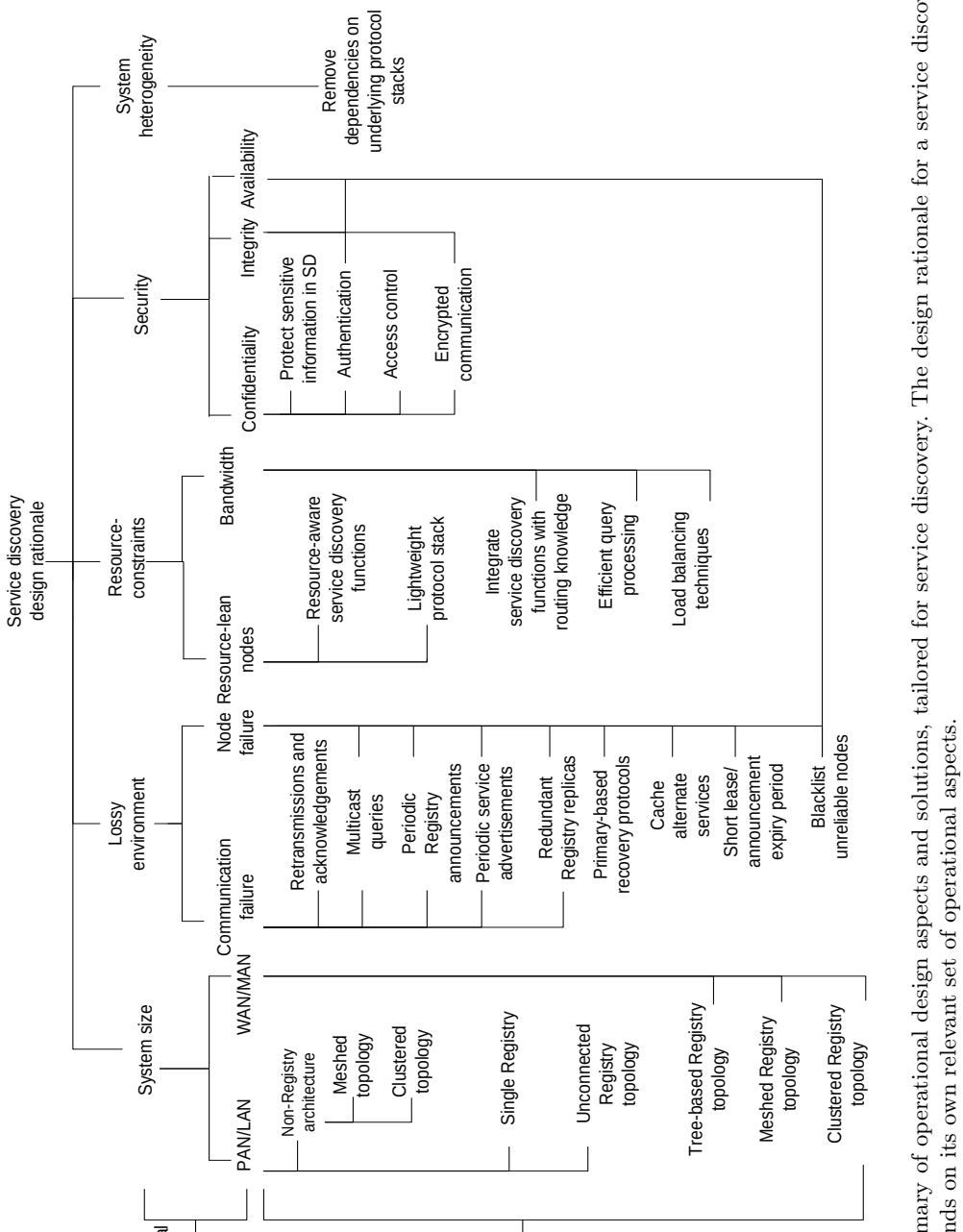

Fig. 4. Summary of operational design aspects and solutions, tailored for service discovery. The design rationale for a service discovery system depends on its own relevant set of operational aspects.

5. ***System heterogeneity*** - Nodes in heterogeneous systems contain different types of network connectivity (e.g. Ethernet, 802.11 a/b/g, IRDA), and a variety of network stacks (e.g. transport, routing, addressing). A service discovery protocol that abstracts away as much as possible the lower-layer protocol stacks, and can perform its functions with minimum dependencies allows easier deployment in a heterogeneous environment.

Figure 4 summarizes the five operational aspects and the respective solutions. By taking the operational aspects into consideration, it is possible to design a system that addresses more than one type of operational issue. For example, an architecture with replicated and redundant Registries supports a large and lossy system. State of the art systems usually base their design rationale on their own set of priorities for the design aspects, hence causing tradeoffs.

6 State of the Art

Having described the nature of service discovery in the context of distributed systems from the point of view of (a) the architecture, (b) the functionality, (c) the models underlying the implementations, and (d) the operational aspects, we now investigate how existing service discovery system fit into this mould.

We provide summaries of selected state of the art service discovery systems. We choose to describe these systems because of their popularity in the type of network and the size of system that they support. For ease of understanding, we will maintain the terms Manager, User and Registry to represent the protocol-dependent entities, even though the original papers use slightly different terms.

We divide the systems into two categories, based on the targeted system size: (1) *small systems*, which includes LAN and PAN, with limited number of nodes, and (2) *large systems*, which includes WAN and MAN, with a large number of nodes. The system size has the most influence on the design decisions in existing state of the art, where they implement similar service discovery functions and methods, and Registry topology (e.g. in large systems, the Registry-based architecture is chosen, where Registries are replicated, and arranged in either tree or mesh topology).

Unless mentioned specifically to support ad-hoc networks, the systems work in infrastructure-based wireless networks, and by default, also work on wired networks.

6.1 Small systems

Small systems are not usually concerned with scalability issues. The architecture type can be Registry, non-Registry or a combination of Registry with multicast query capability for Users (for resilience against single point of failure problems). Furthemore, as bandwidth utilization is not a critical issue in small systems, a strong support for consistency maintenance can be implemented, provided that the nodes in the system are not frequently moving in and out of the system.

1. Jini [29] - Jini was developed by Sun Microsystems, and is implemented using the Java programming language. Jini is a Registry architecture, where the Registry is called the *Lookup Service*. The Manager registers its service at the Registry, by uploading the service proxy code. The data stored is typically a part of a structured distributed shared memory, using tuple spaces, implemented through JavaSpaces [30]. The User queries the Registry for services matching its requirements, and receives the proxy code of the service. The User also requests the Registry to notify it if similar services register in the future. The Manager maintains a lease with the Registry, where it periodically refreshes the lease to indicate its continuous existence. The Registry automatically purges the information on the Manager that failed to refresh its lease. If the Manager updates its service description, it publishes an event to the Registry. The Registry propagates the event to interested Users. The use of Java allows code mobility and operating system flexibility for Jini devices. However, Jini uses the Java Virtual Machine (JVM) and Java Remote Method Invocation (RMI), and depends on TCP/IP for reliable communication. These technologies cause dependencies on the underlying protocol stacks.

2. UPnP [31] - Universal Plug and Play (UPnP) was developed by Microsoft, and is based on the Simple Service Discovery Protocol [32]. UPnP is a non-Registry based architecture. The User is called the *Control Device*, and the Manager is simply called the *Device*. Service Description is described in XML. The Manager sends multiple multicast messages periodically to announce its presence and its services. The User also sends multicast queries to request services matching its requirements. The Manager sends XML documents to the User. The XML document provides the device and service descriptions along with URLs to view the user interface of the Manager. The User controls the remote service through the Simple Object Protocol (SOAP) [33] and XML parsing of action requests. The Manager updates its service through General Event Notification Architecture Base (GENA) [34]. GENA publishes notifications to subscribers. The Manager periodically sends multicast announcements, which is used by the interested User to monitor the continued existence of the Manager. The non-Registry based architecture eliminates single point of failure issues, and supports mobility. However, it increases network traffic due to extensive use of multicast messaging. Like Jini, dependencies on IP technologies causes network dependencies.
3. SLP [35] - The Service Location Protocol (SLP) was developed by the IETF SvrLoc working group. It is a combination of Registry and non-Registry architecture. The User is called the *User Agent*, the Manager is the *Service Agent* and the Registry is the *Directory Agent*. When a Registry exists, the Manager registers its Service Description at the Registry and the User queries for services matching its requirement. SLP provides filters that allows attribute and predicate string search. When the Manager updates its Service Description, it re-registers at the Registry. The User has to query the

Registry periodically if it wants to discover the update. When the Registry is unavailable, the User can send multicast queries to discover the Manager. The Manager also periodically refreshes its registration by re-registering its data. If the Manager fails to refresh its registration on time, the Registry removes the data, and assumes that the Manager is no longer available in the system. A typical implementation of SLP depends on reliable TCP/IP. SLP provides basic service discovery functions, with limited consistency maintenance support. Unlike Jini, the combination of of Registry and non-Registry based architecture reduces single point of failure issues.

4. Bluetooth SDP [36] - Bluetooth was developed by the Bluetooth Special Interest Group, an industry consortium consisting of companies like Ericsson, Nokia and IBM. It is meant for low power, short range (within 10m), wireless (ad-hoc) radio system devices (PAN network) operating in the 2.4GHz ISM band. The Bluetooth Service Discovery Protocol (SDP) depends on the underlying connectivity, thus we first describe how devices establish their connectivity. Bluetooth devices periodically sniff for nearby Bluetooth devices and form a personal area network called piconets which has a maximum of 8 members. The member that initiates communication acts as the master of the piconet. Additional devices are supported by reusing addresses of a silenced existing member on a new member. Groups of piconets communicating with each other are called scatternets. The Bluetooth SDP [37] is a non-Registry based architecture. The Manager runs an *SDP server*, while the User runs an *SDP client*. Services are divided into classes. Each service is represented by a *service record*. The User sends a query for a particular service type, and receives a response from the Manager if it offers a matching service. The User detects that the Manager is no longer available when it does not receive a response to a request. The Bluetooth SDP has simple basic functions, where there is no leasing, or subscription to receive updates. The tight dependency on the underlying protocol layers makes the Bluetooth SDP unsuitable for stand-alone deployment.

Other state of the art that fall in the "small systems" category include FRODO [38], Salutation [39], Konark [40] and DEAPspace [41]. FRODO and Salutation has an architecture similar to SLP, where it is both Registry and non-Registry-based. FRODO is a single Registry architecture for the home environment, where the Registry is elected. FRODO emphasizes on robustness and resource-awareness, where guarantees of service delivery is offered, and entities perform service discovery based on their resource limitations. Unlike the Registries in SLP and Jini, the Registries in Salutation can query each other, forming a meshed Registry topology. Konark is similar to the non-Registry based UPnP architecture, but targets ad-hoc networks specifically. Users and Managers send multicast service advertisements and queries. Konark also uses HTTP, SOAP and XML like UPnP for service descriptions. However, Konark reduces multicast network traffic by using a gossip protocol for advertising services, instead of the more conventional periodic muticast advertisements used in UPnP. DEAPspace is targeted for a smaller system than in Konark or UPnP. DEAPspace is built

for a single-hop ad-hoc network, similar to Bluetooth SDP. However unlike Bluetooth SDP, Users in DEAPspace cache the service descriptions of all Managers within their vicinity, and periodically send multicast messages containing the list of cached services. Managers that find their services missing, or nearing their expiry periods in the advertised lists, readvertise their services sooner than previously scheduled. We do not explore these protocols further in this paper because of their architectural and functional similarities to the four state of the art that we have described in this section.

6.2 Large systems

Service discovery for large systems is designed with multiple Registry architecture because it reduces network traffic, thus increasing scalability. Since large systems are deployed over long distances, multiple, replicated Registries are available. To support a large number of nodes, Registries can do load balancing, and reproduce child Registries to help reduce load. To allow Registries to query and update each other efficiently, Registries are arranged in a logical mesh or tree topology.

1. Ninja SDS [42] - The Ninja project by the University of California, Berkeley developed the Service Discovery Service (SDS). SDS is a Registry architecture, where services are registered by Managers and discovered by Users through queries. The Registry in SDS is called the *SDS server*. For the purpose of scalability, Registries are organized into multiple shared tree-like hierarchies, so that tasks can be shared among several Registries. When a Registry is overloaded, it spawns a nearby node as a new Registry, which then becomes a child of the overloaded Registry. The new Registry is allocated a portion of the network extent, and thus, a portion of the load. Security is also supported by SDS, where service discovery functions are wrapped around steps to allow authentication, authorization and data integrity. For service queries and Service Descriptions, SDS uses an XML-based query and description language. SDS is implemented in Java and requires the use of Secure Remote Method Invocation (Secure RMI) to perform secure communication, hence it requires substantial resources.
2. INS/Twine [43] - INS/Twine was developed at Massachusetts Institute of Technology. Like SDS, The architecture is based on the Intentional Naming System [44], where a number of Registries, called *resolvers*, map queries to destination addresses, and also distribute service information. Unlike tree-like Registries structures in SDS, INS/Twine Registries have a mesh-like topology, where Registries are peers with each other. A Manager is simply referred to as a *resource*. The Manager advertises its Service Description to the nearest Registry. INS/Twine is built on top of a distributed hash table (DHT), such as Chord [45]. The Registry extracts prefix subsequences of attributes and values in the Service Description, into *strands*. The Registry then computes hash values for each of these strands, which constitutes numeric keys used to map resources to resolvers. To avoid being overwhelmed

with registrations, Registries in INS/Twine use keying mechanisms to limit registrations. The service information is stored redundantly in all Registries that correspond to the numeric keys. When a User queries the nearest Registry, the Registry splits the query similar to how the Service Description is split. The Registry then queries other Registries that are identified by one of the longest strands. The query is further processed by the Registry, which returns the matching service information.

3. Jxta [46] - Jxta was developed by Sun Microsystems. It is a combination of Registry and non-Registry based architecture. Registries in Jxta are known as Rendezvous peers. Managers (simply known as *peers*) send multicast advertisements to make their presence and services known to the network. Registries that receive the advertisements cache the service information. A User can send multicast queries, and Managers and Registries with matching service information respond to the queries. Each Manager periodically refreshes its service advertisements. Users and Registries purge service information when the Manager fails to refresh the advertisements at the expected time. When a service changes, the Manager sends another advertisement (either immediately, or at the next periodic refresh time) so that Users and Registries can detect the change. An additional entity called the *relay peer* acts as name resolver to map a service to its destination address. The relay peer stores routing information and relays messages across firewalls. In Jxta, Users, Managers and Registries can form groups, based on a certain criteria (such as location, service type, etc). An entity can only communicate with members of the groups that it has joined. Unlike SDS and INS/Twine, Jxta does not provide load-balancing techniques to unburden overloaded Registries. It also provides limited consistency maintenance support.

4. Ariadne [47]- Ariadne was developed by INRIA Rocquencourt in France. It targets mobile ad-hoc networks, comprising of at least 100 nodes, and integrates routing with service discovery. Like FRODO, the protocol uses an election algorithm to elect Registries. The node with the highest number of neighbors and the smallest number of Registries within its vicinity is selected as a new Registry. Registries periodically announce their presence to nodes within their vicinity. Managers register their service descriptions to Registries within a limited number hops, and Users query known Registries for services. WSDL is used for service description, and for specifying Quality of Service parameters such as availability of service within a particular time, and resource capacity of the Manager (e.g. memory, energy). If the Registry does not have the service description matching the User's query, it selectively forwards the query to other Registries, based on the distance between the sending and receiving Registries. Registries share each other's cached information (called as profiles) by using Bloom filters [48] to reduce memory and bandwidth utilization. Updates on service descriptions are not propagated among the Registries. Only when the number of false cache hits reaches a threshold, the Registry requests for replacements of the profiles.

Other state of the art that are categorized as "large systems" include Superstring [49], GloServ [50], and CDS [51]. The Registries in Superstring are a set of distributed query resolvers, arranged in a tree-based topology, like in SDS. Also, like in INS/ Twine, the resolvers route amongst themselves using a distributed hash table routing structure. A variant of Superstring, called SuperstringRep [49] uses a Bayesian reputation model to compute the reputation score of an entity. The reputation score reflects the quality of service, thus the trustworthiness of the entity. GloServ is a tree-based Registry architecture, that uses DNS-like hierarchy and RDF [11] for describing services. GloServ requires administration (services grouping and managed Registries), and assumes a stable environment, compared to the other systems described in this section. CDS uses a distributed hash-based Registry (resolver) system. Existing hash algorithms are used to distribute service information and queries to the Registries. CDS also does dynamic load balancing by clustering the Registries. Each cluster shrinks and expands according to the number of registrations and queries it supports (cluster size is determined individually by each Registry). Users select and query a Registry in the cluster. As with miscellaneous small systems, we do not explore these state of the art further in this paper.

7 Taxonomy of State of the Art

In this section, we analyze (1) how the selected state of the art service discovery systems address the operational aspects and (2) how they implement the service discovery functions.

7.1 Taxonomy of State of the Art Solutions to Operational Aspects

Figure 5 shows a summary of our analysis on the solutions provided by state of the art systems for the operational design issues. The shaded columns for each system expose the issues that the system considers, and the solutions. We also show which system provides the most support for each design issue by the number of shades for the issue across the systems.

For small sized, mobile PAN (less than ten nodes) the non-Registry architecture (UPnP and Bluetooth SDP) is the most suitable. This is because the number of nodes is small, and service discovery tasks will be accomplished faster, than if a Registry is required to be setup, and maintained. For LAN (tens to several hundred nodes), the Registry-based architecture would be more suitable, to help conserve bandwidth. The Registry can either be statically deployed (Jini, SLP), or dynamically elected (Ariadne, Jxta), depending on the degree of fault-tolerance required. For large systems, scalability is the primary concern. Registries are scoped according to location or services (Ariadne, SDS, Jxta), and arranged in a tree (SDS), or mesh (Ariadne, Jxta, INS/Twine) topology to optimize query processing and conserve bandwidth.

State of the art systems provide fault-tolerance for a lossy environment by implementing on-the-fly Registry setup (Ariadne, SDS), or multiple replicas of the Registry (as can be done in Jini, SLP, INS/Twine and Ariadne) to provide redundancy in the face of single point of failure problems caused by mobile Registries. However, redundancy increases design complexity and consumes additional resources. Registries can also be monitored by other nodes for node crash failures (SDS, Jxta and Ariadne). The non-Registry based architecture of UPnP is the most robust against crash failure and message loss, because Users and Managers can hear each others' multicast queries and announcements directly. However, the tradeoffs are scalability and conservation of resource consumption. A Registry-based architecture, with the ability for nodes to multicast queries when the Registry disappears (SLP, Jxta), increases robustness against message loss and crash failure, while also increasing scalability and resource consumption. To provide reliable transmission, TCP is used in Jini, SLP, and UPnP. None of the state of art protocols address message corruption (assume lower protocol layers will address this problem), and detect and recover from Byzantine failures.

To support resource-lean nodes, a Registry-based architecture that delegates heavier tasks to more powerful nodes allows resource-lean nodes to use less memory, and energy. None of the selected systems here provide explicit resource-awareness. Service discovery for large systems provide load balancing techniques so that Registries are not overloaded; SDS and INS/Twine allow overloaded Registries to spawn another to take over a portion of their tasks. For conserving bandwidth, some systems use efficient replication and query processing such as by using DHTs (INS/Twine) and Bloom filters (Ariadne).

To integrate nodes into different types of connectivity (e.g. wired Ethernet to wireless 802.11b), nodes are assumed to have the necessary interfaces to the different networks, or have connectivity via access points. SDS and Jxta abstract away underlying protocol stacks (transport layer and beyond), therefore providing more flexibility for deployment over different types of connectivity and protocol stacks. Service discovery functions in these systems are self-sufficient, with minimum network layer assumptions (multicast and unicast capabilities required). Systems built for the ad-hoc networks (Ariadne and Bluetooth SDP) integrate routing knowledge with service discovery, and become more dependent on the network stack. Some systems explicitly require a certain type of technology, such as TCP and IP (Jini, UPnP, SLP and INS/Twine). INS/Twine also uses the integrated INS as the underlying name mapping framework.

Most systems depend on higher layers in the protocol stack to provide authentication, authorization, privacy and data integrity. Additional steps are required in between service discovery functions. For example, once the Manager discovers a Registry, it decrypts the message and verifies the signature in the message to authenticate the Registry (as is possible in SLP, Jxta and SDS). These are intermediate steps, before service registration. Users who discover the service can only access the service if they are authorized by a capability manager (as in SDS), by applying for group membership (possible in Jxta), or if allowed by the Manager (can be performed by security applications in all systems). The

service discovery systems discussed here do not support detection and blacklisting of an authenticated and authorized entity that has turned malicious. As mentioned earlier, one protocol that does reputation-based service discovery is SuperstringRep [49]. Security measures consume substantial resources, increase complexity of the system extensively, and reduce self-configuration of the system. Due to these reasons, a full-fledged secure service discovery system is yet to be deployed successfully.

Operational Issues			Design Solutions		State of the art solutions to operational issues							
Issue	Specifics				Jini	SLP	UPnP	Bluetooth SDP	Jxta	SDS	INS/ Twine	Ariadne
System size	1	Small systems (PAN / LAN)	Non-Registry architecture	Meshed topology								
				Clustered topology								
			Registry architecture	Single Registry								
				Unconnected topology								
	2	Large systems (MAN / WAN)	Multiple Registry architecture	Meshed Registry topology								
				Tree-based Registry topology								
				Clustered Registry topology								
Lossy environment	3	Communication and node failure	Retransmissions and acknowledgements		TCP	TCP	TCP					
			Multicast queries									
			Periodic Registry announcements									
			Periodic service advertisement									
			Redundant Registry replicas									
			Primary-based recovery protocols									
			Cache alternate services		Appl	Appl	Appl	Appl	Appl	Appl	Appl	Appl
			Short lease/announcement expiry period		Appl	Appl	Appl	Appl	Appl	Appl	Appl	Appl
			Blacklist unreliable nodes									
Resource constraints	4	Resource-lean nodes	Resource-aware functions									
			Lightweight protocol stack									
	5	Bandwidth constraint	Integrate routing and service discovery									
			Efficient query processing									
			Load balancing techniques									
Security	6	Confidentiality, integrity and availability	Remove sensitive information from SD		Appl	Appl	Appl	Appl	Appl	Appl	Appl	Appl
			Authentication		Appl	Appl	Appl	LM	Appl		Appl	Appl
			Access control		Appl	Appl	Appl	LM			Appl	Appl
			Encrypted communication		Appl	Appl	Appl	LM			Appl	Appl
			Blacklist malicious nodes									
System heterogeneity	7	Heterogeneous protocol stack	Abstract away underlying protocol stacks		TCP/IP	TCP/IP	TCP/IP	Bluetooth			INS/IP	Routing
Operational issues that are given priority by the system					1,3	1,3	1,3	1,6	1,2, 3,6,7	2,3, 6,7	2,3,5	2, 3,5

Appl: The solution can be provided by the application layer
LM: The underlying Link Manager in the Bluetooth protocol stack provides authentication and encryption
RMI/JVM: Jini is supported by the security features in JVM and RMI/TCP: Reliable communication is provided using TCP
TCP/IP, Bluetooth, Routing, INS/IP: The system depends on the listed underlying protocol stack

Fig. 5. Taxonomy of state of the art solutions to operational aspects. Shaded service discovery systems support the proposed solutions. *Appl* means the solution to the operational aspect is supported by the application layer. Some systems depend on solutions provided by the underlying protocol stacks, such as TCP, IP, Bluetooth and ad-hoc routing protocols.

Service Discovery Functions		Methods	State of the art functional implementation							
			Jini	SLP	UPnP	Bluetooth SDP	Jxta	SDS	INS/ Twine	Ariadne
Configuration Discovery	Registry auto-configuration	Registry election			N/A	N/A				
		Registry reproduction			N/A	N/A				
	Registry or cluster discovery	Passive discovery				N/A				
		Active discovery	*		N/A	N/A				
Registration		Solicited registration			N/A	N/A				
		Unsolicited registration			N/A	N/A				
SD Discovery	Query	Unicast query				**				
		Multicast query								
	Service notification	Service notification by Registry			N/A	N/A				
		Multicast service advertisement								
Configuration Update	Configuration purge	Leasing expiry								
		Advertisement TTL expiry								
	Consistency maintenance	Poll for updates (by the User;)	Appl	Appl	Appl	Appl	Appl	Appl	Appl	Appl
		Notification of updates (by the Manager / Registry)								
		Update among Registries			N/A	N/A				

* Jini only does active discovery when the node initializes (powers on)
** Bluetooth SDP depends on the underlying Bluetooth network to detect neighboring nodes, so that it can query through unicast
N/A: the method is not relevant for the non-Registry architecture
Appl: the method can be supported by the application layer, by using the handles provided by the service discovery protocol

Fig. 6. Taxonomy of state of the art functional implementation. However, the choice of implementation method impacts other considerations, such as the efficiency, responsiveness and resource consumption.

7.2 Taxonomy of service discovery functions and methods

Once architectural decisions are made on how to address the operational design issues, the service discovery functions are designed. Some functions may be provided by different protocol layers (such as network and application). In such systems, the dependencies on different protocol layers means they are not easily portable, and may be *less effective* [52, 25]. For example, Bluetooth SDP cannot easily replace SLP on a printing device. Meanwhile, SLP expects the User application to poll for changes in services, and if the application fails to do so (overlooked by application developer, retransmission limit, etc), the system will not be effective in conveying changes in service information to interested parties.

We show in our previous work [52, 25] that the functional differences impacts system performance such as responsiveness, efficiency and resource consumption. Briefly, these performance outcome depends on the how exhaustive is the implementation of the function. For example, SLP and Jxta implement both unicast and multicast queries, therefore when Registry disappears, services can still be discovered by Users through multicast queries. In Jini, SDS, INS/Twine and Ariadne, Registries must recover before the service can be discovered, causing slower responsiveness.

Figure 6 summarizes the functional capabilities of state of the art systems. We now give a detailed analysis.

For the Configuration Discovery function, the Registry reproduction method in INS/Twine and SDS requires the first set of Registries in both systems to be manually deployed by a system administrator, unlike the more dynamic Registry election method in Ariadne. For discovering Registries and cluster members, systems that do periodic passive and active discovery (SLP,INS/Twine and SDS) have higher responsiveness than systems that implement only one of the methods. Passive and active discovery are especially useful to allow the system to recover from failures that cause network partitioning.

In the Service Registration function, none of these selected systems allow the Registry that receives messages from unknown Managers to solicit registrations. This method allows the Registry to speedily recover purged information of a Manager, after communication failures. The rest of the systems allow only unsolicited registration, after a Registry is discovered.

For the SD Discovery function, UPnP and Jxta allow both multicast queries and multicast service advertisements. The combination of these two methods gives the highest probability for successfully discovering a service, even after message loss and temporary node failures (e.g. mobile nodes getting temporarily disconnected). Among Registry systems, the probability of discovering services is increased if the Registry can notify the Users of newly registered services matching the requirements of the Users (Jini).

For the Configuration Update function, Jini uses leasing for Configuration Purge, where the Registry can request Managers to lengthen or shorten their lease period, according to the Registry's processing capability. The rest of the systems require Users and Registries to monitor the advertisement TTL of Managers to detect defunct services. Leasing is more efficient in terms of bandwidth

and resource utilization, compared to periodic multicast advertisements. For Consistency Maintenance, the state of the art systems provide handles to allow the application on the User to query the Registry or Manager periodically for updates. Only Jini and UPnP allow the Registry or the Manager to update the User directly on changes in the SD. In large systems (Jxta, INS/Twine, SDS and Ariadne), Registries achieve consistency by updating each other on changes in the cached SDs. Unlike small systems, updates are not propagated each time an SD changes, but in bulk (when a threshold is reached), thus providing weaker consistency maintenance (but necessary to conserve bandwidth).

8 Conclusion

We analyze the field of service discovery by first characterizing service discovery as a third generation name discovery system that solves the limitations of legacy naming systems for pervasive computing. We describe the different architectures and the main functions of service discovery that allows services to be discovered, and changes in service availability and attributes to be detected. We then classify the main operational design aspects for service discovery, and compare the state of the art solutions to these aspects.

Our analysis reveals which operational aspects, and functionalities are supported by the state of the art systems. Subsequently, system architects can choose the service discovery system that satisfies their requirements, and integrate functionalities to enhance service discovery at the application level, to produce a strong, working system.

There are still several interesting directions in which future research on service discovery can be taken.

- The semantics of device, service and attribute names still require much attention, to improve the context of a discovered service. For a truly unattended system deployment, different service discovery systems should adhere to a single, standardized method for describing services. This is important to ensure that applications that rely on discovered services can make the correct inference on the usage of the service.
- Service discovery systems should ensure that a service is discovered and accessed by authorized entities only. However, authorization should be dynamically allocated and revoked, as time progresses, and the requirement or capability of the entity changes. A service discovery system that assigns, monitors, revokes and reassigns access to services is necessary for establishing a truly self-protecting system.
- For a service discovery system in the pervasive environment to mature (as DNS has done in the Internet), applications that use service discovery need to be actively developed and promoted. One major hindrance to achieving this objective is the lack of agreement by manufacturers of devices and applications on a standard service discovery platform. A service discovery system

that unifies well-known service discovery protocols is a step towards this objective.
- Existing service discovery architectures for wide-area networks focus more on scalability issues (such as bandwidth efficiency, and supporting a large number of nodes). More work has to be done to produce a large-scale service discovery design, which is also evaluated against communication and node failures, such as done in smaller systems [53, 54, 25]. A scalable and robust architecture is especially important in mobile ad-hoc networks, because nodes are easily moved, uncertain wireless connectivity, low bandwidth availability, and energy constraints.

We conclude by stressing that a service discovery system is the medium for propelling the power of computing beyond the realm of personal computers, such that information and services are accessible anywhere, and anytime.

References

1. Murch, R.: Autonomic Computing. Prentice Hall (March 2004)
2. Ganek, A.G., Corbi, T.A.: The dawning of the autonomic computing era. IBM Systems Journal **42**(1) (2003) 5–18
3. Weiser, M.: The computer for the 21st century. **265**(3) (September 1991) 94–104
4. Satyanarayanan, M.: Pervasive computing: Vision and challenges. IEEE Personal Communications **8**(4) (August 2001) 10–17
5. Bettstetter, C., Renner, C.: A comparison of service discovery protocols and implementation of the service location protocol. In: Proceedings of 6th EUNICE Open European Summer School: Innovative Internet Applications, University of Twente (September 2000) 101–108
6. Vanthournout, K., Deconinck, G., Belmans, R.: A taxonomy for resource discovery. Personal Ubiquitous Comput. **9**(2) (2005) 81–89
7. Richard, G.G.: Service advertisement and discovery: enabling universal device cooperation. **4**(5) (September-October 2000) 18–26
8. W3C Working Group Note: Web services architecture. http://www.w3.org/TR/ws-arch (February 2004)
9. McGuinness, D.L., van Harmelen, F.: Owl web ontology language. http://www.w3.org/TR/owl-features (February 2004)
10. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web service description language. http://www.w3.org/TR/wsdl (March 2001)
11. RDF Core Working Group. Homepage at http://www.w3.org/RDF/ (2004)
12. Srinivasan, R.: Rpc: Remote procedure call protocol specification version 2 (1995)
13. Gosling, J., Joy, B., Steele, G.: The java language specification. Homepage at http://java.sun.com/java.sun.com/newdocs.html (1996)
14. Needham, R.: Names. In Mullender, S., ed.: An Advanced Course In Distributed Systems, Wokingham, England, ACM Press/Addison-Wesley Publishing Co. (1993) 315–326
15. Tanenbaum, A.S., Steen, M.V.: Distributed Systems: Principles and Paradigms. Prentice Hall PTR, Upper Saddle River, NJ, USA (2002)

16. Coulouris, G.F., Dollimore, J.: Distributed systems: concepts and design. fourth edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2005)
17. Birrell, A.D., Levin, R., Schroeder, M.D., Needham, R.M.: Grapevine: an exercise in distributed computing. Communications of the ACM **25**(4) (1982) 260–274
18. Lampson, B.W.: Designing a global name service. In: Proceedings of the fifth annual ACM symposium on Principles of distributed computing (PODC '86), New York, NY, USA, ACM Press (1986) 1–10
19. Mockapetris, P., Dunlap, K.J.: Development of the domain name system. In: SIGCOMM '88: Symposium proceedings on Communications architectures and protocols, New York, NY, USA, ACM Press (1988) 123–133
20. Peterson, L.L.: The profile naming service. ACM Trans. Comput. Syst. **6**(4) (1988) 341–364
21. Bowman, M., Peterson, L.L., Yeatts, A.: Univers: an attribute-based name server. Software-Practices and Experiences **20**(4) (1990) 403–424
22. Chadwick, D.: Understanding X.500 The Directory. Chapman & Hall, London (1994)
23. Howes, T., Smith, M., Good, G.S.: Understanding and Deploying LDAP Directory Services. Macmillan Technical Publishing, Indianapolis, Indiana (1999)
24. Object Management Group: OMG. The Common Object Request Broker: Architecture and Specification, Rev 1.2., OMG Document Number 93-12-43. (December 1993)
25. Sundramoorthy, V., Hartel, P.H., Scholten, J.: On consistency maintenance in service discovery. In: 20th IEEE Int. Parallel & Distributed Processing Symp. (IPDPS 2006), Los Alamitos, California, IEEE Computer Society Press (April 2006) 10pp in CD–ROM
26. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. In: IEEE Transactions on Dependable and Secure Computing. Volume 1., Los Alamitos, California, IEEE Computer Society Press (Jan-Mar 2004) 11–33
27. Elkhodary, A., Whittle, J.: A survey of approaches to adaptive application security. In: SEAMS '07: Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems, Washington, DC, USA, IEEE Computer Society (2007) 16
28. Merwe, J.V.D., Dawoud, D., McDonald, S.: A survey on peer-to-peer key management for mobile ad hoc networks. ACM Comput. Surv. **39**(1) (2007) 1
29. Sun Microsystems: The Jini Architecture Specification, version 2.0. (June 2003)
30. Sun Microsystems: JavaSpaces Service Specification , version 2.0. (June 2003)
31. Microsoft: Universal Plug and Play Architecture, V1.0. (Jun 2000)
32. Goland, Y., Cai, T., Leach, P., Y.Gu: Simple service discovery protocol, version 1.0. (2000)
33. Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J., Nielsen, H.: Simple Object Access Protocol (SOAP) V.1.2, Part 1: Messaging Framework. (June 2003)
34. Cohen, J., Aggarwal, S., Goland, Y.: General Event Notification Architecture Base: Client to Arbiter. (June 1994)
35. Guttman, E., Perkins, C., Veizades, J.C., Day, M.: Service Location Protocol, V.2, RFC-2608. Internet Engineering Task Force (IETF) (December 2003)
36. Bray, J., Sturman, C.F., Mandolia, J.: Bluetooth 1.1 Connect Without Cables, 2nd Edition. Prentice Hall (December 2001)
37. Bluetooth SIG: Specification of the Bluetooth System, Core, Vol. 1. (Feb 2001)

38. Sundramoorthy, V., Speelziek, M.D., van de Glind, G.J., Scholten, J.: Service discovery with FRODO. In: 12th IEEE Int. Conf. on Network Protocols (ICNP), Berlin, Germany, Computer Science Reports, BTU Cottbus (Oct 2004) 24–27
39. The Salutation Consortium Inc: Salutation Architecture Specification (Part 1), version 2.1. (1999)
40. Helal, S., Desai, N., Verma, V., Lee, C.: Konark-service discovery and delivery protocol for ad hoc networks. In: Proc. IEEE Wireless Communications Networking Conf. (2003)
41. Nidd, M.: Service discovery in deapspace. **8**(4) (2001) 39–45
42. Czerwinski, S., Zhao, B., Hodes, T., Joseph, A., Katz, R.: An architecture for a secure service discovery service. In: Proceedings of ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'99), Kluwer Academic Publishers (1999) 24–35
43. Stoica, I., R.Morris, D.Karger, M.F.Kaashoek, H.Balakrishnan: A scalable peer-to-peer lookup service for internet applications. In: Proceedings of the 2001 ACM SIGCOMM Conference. (2001)
44. Adjie-Winoto, W., Schwartz, E., Balakrishnan, H., Lilley, J.: The design and implementation of an intentional naming system. In: Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP), ACM Press (December 1999) 186–201
45. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Transactions on Networking (TON) **11**(1) (2003) 17–32
46. Gong, L.: Jxta: A network programming environment. IEEE Internet Computing **5**(3) (May-June 2001) 88–95
47. Sailhan, F., Issarny:, V.: Scalable service discovery for manet. In: 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom 2005), IEEE Computer Society (March 2005) 235–244
48. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communication of the ACM **13**(7) (1970) 422–426
49. Wishart, R., Robinson, R., Indulska, J., Josang, A.: Superstringrep: Reputation-enhanced service discovery. In Estivill-Castro, V., ed.: 28th Australasian Computer Science Conference (ACSC2005). Volume 38 of CRPIT., Newcastle, Australia, ACS (2005) 49–58
50. Arabshian, K., Schulzrinne, H.: Gloserv: Global service discovery architecture. In: First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (Mobiquitous), Los Alamitos, CA, USA, IEEE Computer Society (August 2004) 319–325
51. Gao, J., Steenkiste, P.: Rendezvous points-based scalable content discovery with load balancing. In: Networked Group Communication, New York, NY, USA, ACM Press (October 2002) 7178
52. Sundramoorthy, V., van de Glind, G.J., Hartel, P.H., Scholten, J.: The performance of a second generation service discovery protocol in response to message loss. In: 1st Int. Conf. on Communication System Software and Middleware, New Delhi, India, IEEE Computer Society Press (Jan 2006)
53. Dabrowski, C., Mills, K., Elder, J.: Understanding consistency maintenance in service discovery architectures during communication failure. In: Proceedings of the Third International Workshop on Software and Performance, ACM Press (July 2002) 168–178

54. Dabrowski, C., Mills, K., Elder, J.: Understanding consistency maintenance in service discovery architectures in response to message loss. In: Proceedings of the 4th International Workshop on Active Middleware Services, IEEE Computer Society (July 2002) 51–60