Extending service discovery protocols with support for context information

Buchina, N.G.

Award date:
2014

Extending service discovery protocols with support for context information

Author:
Nina Buchina
Student number: 0871926
n.buchina@student.tue.nl

Supervisor:
Dr. Ir. Pieter J.L. Cuijpers
p.j.l.cuijpers@tue.nl

Tutor:
Milosh Stolikj MSc
m.stolikj@tue.nl

Eindhoven University of Technology
Department of Mathematics and Computer Science
Chair of Systems Architecture and Networking

October 27, 2014

COMMIT/

Abstract

Applications in wireless sensor networks often require information about the context of individual sensors. This information can be used to analyse the situation globally, or to select a node which provides a required service. For example, the closest service provider may need to be selected, or decisions about the situation in a particular region may need to be taken.

Service oriented architecture allows for services to be automatically discovered in the network. Currently service discovery is associated with considerable amount of messages sent over the network, as all the services have to be surveyed in order to find few with desired context. In low-capacity wireless sensor networks the number of network messages is important, because communication consumes a lot of energy and reduces sensors battery life.

With the ability to mention the context in the service discovery request the client might be able to address the more narrow set of services. It means that only satisfying nodes would have to send their data via the network. Including context information in service discovery protocol should reduce the network load and prolong sensors life.

The service discovery technology under consideration is DNS-SD - a DNS-based service discovery protocol [1]. This protocol is widely used for service discovery in LANs. Popular implementations include Bonjour [2] by Apple and Avahi [3] for Linux.

The goal of the project is to extend the DNS-SD protocol with support for context information. A particular context of interest is device location. Location information can be of different types, like GPS coordinates or logical location (i.e. building, floor, etc.), and can change during time, which makes its inclusion especially challenging.

The project includes studying and specifying possible ways to include context information in the DNS-SD protocol. This involves enumerating possible protocol extension alternatives, evaluating them with defined metrics, and building a demo application. One of most promising alternatives is to employ the resource naming scheme that would provide required context information for each sensor.

This work is supported in part by the Dutch P08 SenSafety Project, as part of the COMMIT program. [4].

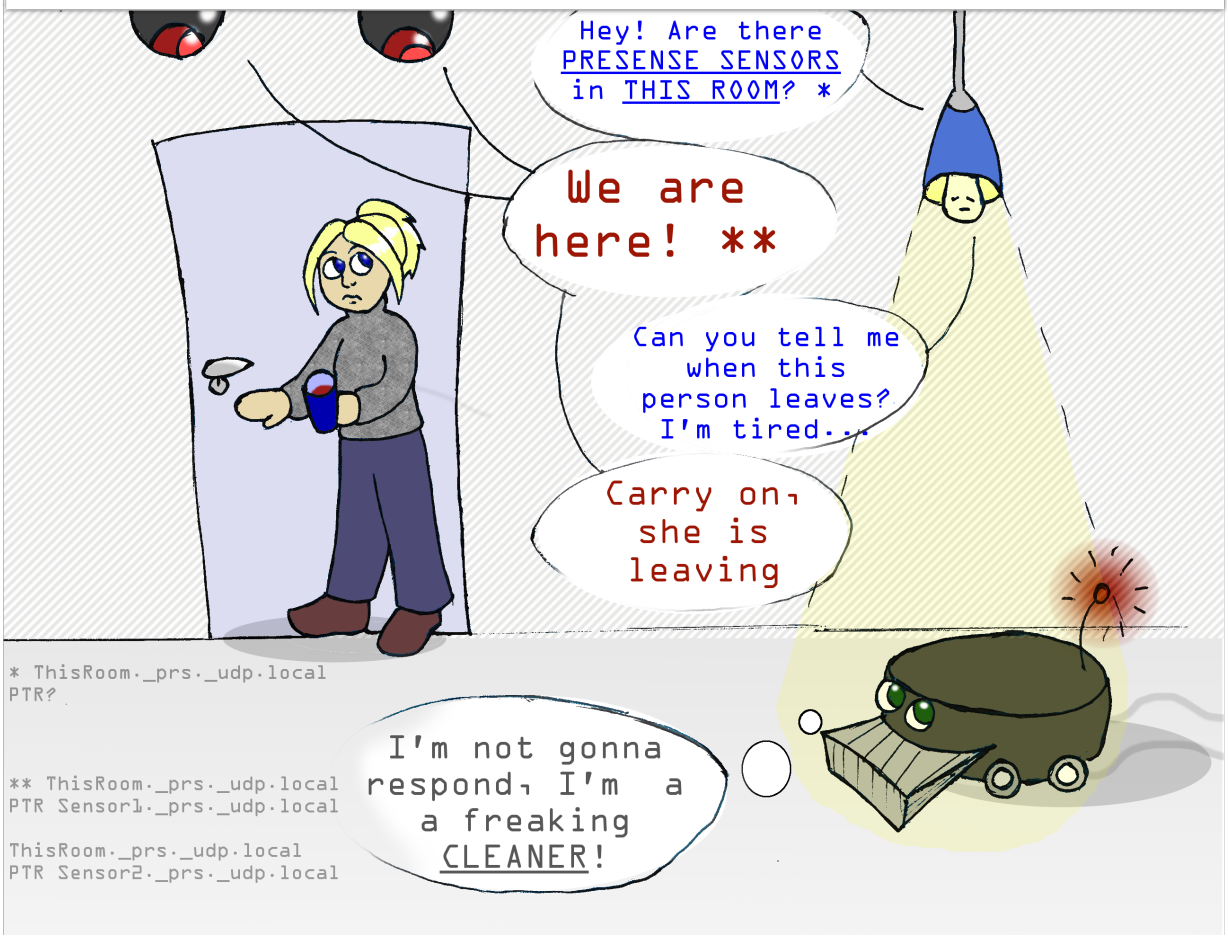
Keywords: Service discovery, DNS-SD, context, naming, wireless sensor networks

Contents

List of Figures

List of Tables

This is how the automatic context-based service discovery works.



Chapter 1

Introduction

1.1 Domain description

Wireless sensor networks (WSN) are networks of devices and sensors that are capable of communicating with each other using a wireless connection. Sensors in such networks measure a wide range of physical phenomena - temperature, moisture, light levels and many more. WSNs are widely used in military and environmental applications, health care and other areas [7] to gather, transfer and process physical information about the environment. A WSN may also contain actuators that can affect the environment. For these applications the information about context of individual sensors, such as their battery status, location and other properties, may be of great use.

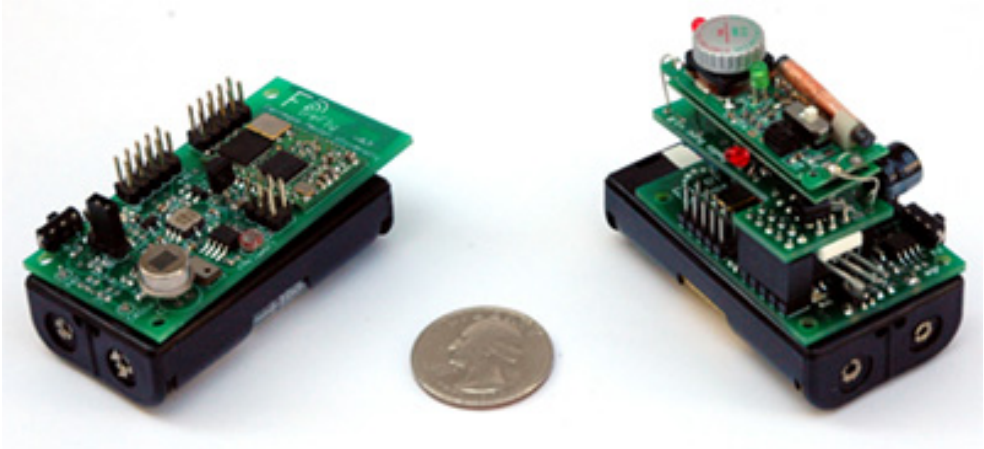


Figure 1.1: Wireless sensors developed in FireFly project [5].

Main concerns associated with WSN include resource limitations, network scalability and self-organisation, power constraints, and heterogeneity of nodes both on hardware and software levels. Energy and resource limitations motivate the need for optimization of network interaction, measuring operations and data processing. On the other hand, scalability

and heterogeneity problems require a flexible and platform-independent approach to the network utilisation.

Service oriented architecture (SOA) addresses the concern of network components heterogeneity by aiming to achieve loose coupling among interacting software agents [8]. It allows the application to abstract from implementation details of individual system components and build very complex systems from loosely coupled, interchangeable functional units, called services.

A service is an isolated piece of functionality with well defined inputs and outputs. Services can use other services in their work. In order to deal with the possible heterogeneity of the nodes, applications in SOA are not required to be aware of the details of individual services implementation. It should not matter for the service-oriented application on what platform services are executed, in which programming language they are written, on what operating system they are running or whichever underlying technologies they are using. To ensure interconnectivity between services, the application should employ SOA-specific protocols of the following types:

- **Service description protocol**. This protocol defines the way a service is specified, in particular, its inputs and outputs, name and available functionality.
  Examples: DNS-SD [1] WSDL [9];

- **Service discovery protocol**. The protocol specifies how available services can be automatically discovered by the application or other services. The context of services is addressed and represented in this type of protocols.
  Examples: DNS-SD [1], UPnP [10];

- **Service composition protocol**. This protocol describes the way to compose several services together to form an application.
  Examples: FLUENT [11], eFlow [12];

- **Service access protocol**. The protocol is used to actually access a service functionality.
  Examples: CoAP [13] HTTP [14];

Services in WSN are spread over separate devices and communicate via the network. This means that their discovery and usage will create a certain network load. As wireless communication significantly contributes in the energy consumption [15], in WSNs this network overhead should be minimised.

In this work we focus on the service discovery, that contributes to the network usage. In particular, we address one of most popular protocols of this type - the DNS based service discovery protocol (DNS-SD) [1].

DNS-SD was introduced by Apple in 2001 and became a standard in 2013 [1]. It allows to describe and discover services using Domain Name System (DNS) [16] features. Unfortunately, it does not allow to discover services by their context.

The benefits of using DNS, a technology mainly designed for name resolution, for service discovery, are that existing DNS servers, clients, protocols, infrastructure, and expertise can be reused, hence there is no need to create and deploy a new system just for service discovery ([1], appendix A). In case a setup has no DNS servers, clients still can take advantage of using an existing well-described and standardized protocol instead of implementing a new one, by employing the Multicast DNS [17] protocol for communication. As this work is focused on distributed wireless sensor networks, this combination will be he main one to address.

A service in DNS-SD is described through a unique human-readable name, a type and a domain name. A service also has the information about its access point. Namely, the network address and the port number used by the service are included. There also may exist some metadata associated with a service name. However, the possibility to the properties of the service in the metadata does not allow to query for these properties. The discovery is performed by requesting for services of a specific service type, protocol and domain. This mechanism is described in detail in section 2.2.4.1. There are no means to query for other properties.

1.2 Problem description

As we stated above, the DNS-SD stores services context in its metadata, which can not be used in service discovery queries. This means that a context-aware application cannot directly discover services with a particular context: it needs to survey all available services first and then filter out those with the matching metadata. However, receiving the data from every device that stores services information creates a lot of unneeded traffic, which is unacceptable for WSNs. Therefore, we have to find the way to discover services with DNS-SD with respect to their context. For the sake of wide applicability we would like to avoid without breaking the original DNS-SD protocol.

Another aspect of the problem is that different applications may need to address the context information of very different nature. Specific combinations of context properties for service-enabled wireless sensor networks cannot be predicted. Physical location, environment conditions and many application-specific properties of nodes may need to be expressed. All of these have to be expressible with the service discovery protocol.

1.2.1 Goals

The goal of this work is to extend the DNS-SD protocol with support for context information, such as devices location. The project implies studying, specifying and evaluating possible ways to include and discover context in the DNS-SD protocol. The solution will

consist of one or several described techniques for encoding context properties in the DNS-SD service description.

We describe and formalize a way to express context properties, that would be both flexible suitable for the DNS-SD protocol. We present and justify the requirements for possible solutions with respect to our model constraints. We also provide the metrics to evaluate these requirements against possible solutions. Next we enumerate and discuss several context extension alternatives, including four naming schemes designed to encode context tags in DNS-SD. We evaluate them with the metrics described before and calculate the amount of resources they will take. Several solutions are implemented in a demo application demonstrating the discovery process. The architecture and overview of this software are presented in the work.

1.2.2 Requirements

Our solution has to satisfy several requirements. We list them here together with the metrics that will be used to evaluate them. We motivate our choice of requirements in the chapter 2 of this work.

- Minimize network load associated with service discovery.

  Metrics: UDP datagram size, number of DNS records per request and response, number of UDP datagrams in the network;

- Minimize additional client code to support context-aware service discovery.

  Metrics: COSMIC [18] points;

- Minimize the amount amount of memory needed to store the context information.

  Metrics: The number of DNS records depending on the number of context properties per service, the size of DNS requests and responses associated with service discovery;

- Discovery features, allowing to express any required combination(s) of context properties.

  Metrics: Supported discovery operations;

- Maximize the amount of expressible context properties.

  Metrics: Maximum number of context properties per service;

- Compatibility with the original DNS-SD protocol.

1.2.3 Hypotheses

Our main hypothesis is that including the context information in service description, rather than in service metadata, will enable service discovery with less network requests and hence optimize the network load. Instead of querying for all services and afterwards filtering

results for services with a desired context , the client makes a specific request mentioning all the context it is interested in. Nodes without the requested context will not respond to this request, thus the network load will decrease.

We expect a naming scheme to be an appropriate solution with respect to all requirements mentioned above. This approach does not require us to change the original DNS-SD protocol, and can be tuned and optimized depending on the applications needs. Naming schemes are described in detail in section 3.2.

The main contribution of this work is the described approach to include context information of various nature in the existing DNS-SD protocol, cancelling the need to design and implement a separate context-aware service discovery protocol.

1.3 Related work

In this section we present two types of works related to our study. These are solutions specific to DNS and DNS-SD protocols, and new methods for the context-aware service discovery. All of them provide some means to use context information for the discovery and addressing of services. We argue that yet they do not solve the stated problem.

1.3.1 DNS-SD features

There are existing in the DNS protocol that can be used for context embedding. Some of these DNS features can be employed for this purpose in a non-straightforward way, for example by using naming conventions.

1.3.1.1 DNS resource records

NAPTR record NAPTR is a special type of DNS resource record that allows regular expression based rewriting of domain names [19]. The result of the execution of regular expression can then be used as a URI to process by the application, or as a domain name for a further lookup.

NAPTR is a quite powerful mechanism. One of the ways to solve the problem of embedding the context information could be the following. Every node can return the "rule" that must hold for the set of request tags to comply with the given node. For discovery, the application can make a request with the context it is interested in. All nodes that receive a request then reply with their NAPTR records. The application parses all regular expressions of retrieved records and this way obtains the list of service names for further service lookup.

One problem of this method is that all clients need to reply with the NAPTR record to the context discovery request. This practically defeats the purpose of the work in minimisation of the network load. Another disadvantage is that the regular expression stored in the NAPTR record is calculated by the receiving party. Additional calculations like these are not desirable for resource-constrained devices.

LOC record LOC is a DNS record that specifies a geographical location associated with a domain name [20]. The LOC record contains Latitude, Longitude and Altitude information together with the physical size of the area and accuracy of coordinates. This resource record type can be used to store location of nodes that reside outside of buildings, or when location is not expected to be very precise.

Unfortunately, LOC resource records can deal only with location information and do not support any other kind of context. Also, it is not possible to query for LOC records with specific values: the user needs to retrieve the record first to be able to read the actual location information. This makes the LOC record an inefficient tool for the service discovery.

1.3.1.2 DNS-SD extensions

The internet draft [21] by P. van der Stok, K. Lynn and A. Brandt describes a way to include one kind of location information to the DNS-SD service description. The work is concentrated on building control, hence the proposal is to describe location hierarchically, from broader context (i.e. building) to more narrow (floor, room). This hierarchy is to be included into the name of the service or the group of services. For example, the name `all-light.o4.b8.example.com` can be assigned to the group of all light services in building 8, office 4. Services can then be discovered with requests of different granularity.

Encoding non-hierarchical data or several hierarchies for one service is not discussed. There is also no information on non-location context embedding presented in the work. However, the proposed idea of including context information in domain name is a perspective one. This work further extends this approach, formalizing the means to express and encode arbitrary context.

The recent internet draft [22] proposes to extend DNS-SD protocol so that TXT keys could participate in the discovery. The author addresses the problem of a big number of service discovery responses due to the broad search context. The suggested solution is to include TXT keys with desired context in the request and discover only services with the same values of these keys as specified in the request.

The implementation of this proposal would require to alter the behavior of DNS responsers, as they currently only process domain names in the request, and not TXT key-value pairs. For heterogeneous sensor network this would require updating the DNS-SD software for all used platforms.

1.3.2 New methods

Many authors choose to implement a new service discovery system to address services context. This approach allows to implement any required features of service discovery, including context awareness. However, we state that all of these solutions have a considerable disadvantage comparing to DNS-SD. Namely, they are not as widely-used and ubiquitous, they have not been fixed in any standard and require additional effort to implement and maintain, whereas DNS-SD operates with concepts that have been well-known for years.

Hence for a community it would be more beneficial to enhance this existing and widely-used protocol.

In [23], authors propose the design and the implementation of a naming system, a resource discovery and a service location for dynamic and mobile networks of heterogeneous devices. The proposed system supports services mobility by adding a level of indirection; as a result, a client addresses services by their set of properties and not by their network addresses. The system supports anycast (selecting any appropriate resource) or multicast (addressing all resources matching the context) requests at the application level.

Context properties are organized in a hierarchical structure of attribute-value pairs. The element that is dependent on another on is a descendant of it. For example, the pair "building=whitehouse" depends on the pair "city=washington" that represents a broader context.

Service providers advertise themselves periodically to special resolver devices that replicate data among each other and form an overlay network. These devices keep track of other devices current network addresses and context, and act as proxies resolving properties requests.

Authors do not address the network load minimisation in their work. In the experiment they use the messages of size 586 bytes, which may be too large for embedded networks.

In [24], a design and architecture for naming systems working in Building control scenario have been proposed. The authors adopt the hierarchical approach to naming inspired by DNS. They adapted the approach for usage in distributed systems by grouping devices in zones and using multicast requests for the name resolution.

The architecture is location-centric, and names of devices are based on their location inside a building. The work solely concentrates on building control and does not propose any means to include context information of different nature in devices names.

In [25] the authors propose a centralized service discovery protocol for 6LoWPANs. The architecture is based on a location-based grouping of services. This approach enables execution of group-local commands, for example, switching off the room lights or reducing the level of heating in the area. The service registry keeps track of the services and their availability.

Each device registers itself in the Directory agent, providing the location, the battery information and available services. Then the device has to update its status once in a while, otherwise the directory agent will mark it as inactive.

The work does not describe a way to address context other than location and the service type, or any arbitrary context. Also, dividing services into fixed location-based groups may be less appropriate for some applications, for example, those that need to address the more general or more precise location than the one defining the group.

In [26] sensor discovery in a service oriented architecture is described. Unlike other works, this paper discusses querying for services with an arbitrary context. The context information is stored by directory proxy agents (DPA), that are responsible for wireless devices in a certain location, i.e. a certain room. Context is stored in a key-value pairs, for example, "`time=10:00`"

Though the work assumes the possibility to query for an arbitrary context, it is con-

centrated on finding closest services, which may not be suitable for some applications, that would rather prefer a further located service with some specific features. Also, as the rest of the works in this section, the approach requires the implementation and deployment of a novel system and acquiring the appropriate expertise.

1.4 Outline

We already described the problem and set up goals of work in the section 1.2 (Problem description). In chapter 2 we set up our model and constraints, as well as employed technologies, and formulate requirements in detail. We also introduce a leading example that will be used further to illustrate discovery mechanisms.

Chapter3 discusses and describes several solutions alternatives.

Chapter 4 is devoted to evaluating requirements defined earlier with measurements and experiments. It also presents some practical examples and comparison to the solution presented in one of related studies.

Finally, chapter 6 summaries the work and shows some of further research directions.

Chapter 2

Context-based service discovery

2.1 Context-aware applications

These days software applications operate a wide range of real world objects. Software is managing nuclear power plants, assembly of cars, robots that play football and many, many more.

As the Internet of Things concept emerges, a new kind of applications that involve numerous devices of different types, manufacturers and platforms, become realizable. For example, an application may use heaters, air conditioners and temperature sensors altogether to dynamically calibrate the climate in a given room, or in the whole building. Another use case is to apply energy saving policies, switching distinct smart lights on and off depending on the detected presence of people in certain areas.

Sometimes this kind of behaviour is configured by hand, on hardware or application level, binding inputs of one device to outputs of other device in a predefined way. For example, smart lights in a certain room can be associated with presence sensors in the same room. This solution is not very flexible and requires significant changes as new use cases emerge.

However, the binding can be as well performed automatically, at run time. If the application can make decisions and perform operations that are based on the current context of individual devices, this might enable more flexible solutions and easier configurations. This way instead of addressing individual devices or predefined groups, the application concentrates on situations, locations or other phenomena it needs to measure or affect.

For the purpose of our work we define the notion of a "context" as a set of properties, relevant for given application. The context can change over time. Examples of context properties are the device battery status, the neighbourhood to environmental phenomena, the time of deployment, etc. These properties may help an application to select devices that satisfy certain requirements. For example, an application can select the device that has the largest battery charge.

One of the most practically demanded properties is a device location. It is addressed in most papers in the Related Work section (1.3). For example, in [26] the location infor-

mation is used to choose the closest service provider for the user, and [21] employs it for performing distributed building control.

Depending on the needs of the application, location can be expressed in a different manner. For building control purposes, location can be represented as a hierarchical "address" of device, that may include levels such as building name, floor and number of room. Location can also be expressed as GPS coordinates; through proximity to other devices with known coordinates; via the name of the sector, or cell; and, possibly with other techniques.

In our work we address a variety of context properties in order to make the solution universal and suitable for a wide range of applications. However, we face the limitations originating from the domain specifics and the employed technologies. We discuss these in the next section.

2.2 Setup, scope, used technologies

In this section, the predefined settings and assumptions of the work are described. We shortly enumerate the important assumptions and concerns, and list the resulting architectural and technological decisions. We elaborate on each of them, explaining their principles and relevant specifics.

Following are the assumptions:

- We deal with a distributed wireless sensor network. The number of devices in the network can be quite large, an order of thousands or even millions [7] of devices.
- The network consists of heterogeneous devices. Heterogeneity on software and hardware level can be observed, meaning that devices may have different hardware architectures, capacities and software. Some of these devices are resource-constrained.
- Applications that use the network may require information about the context of individual nodes, such as location, battery status and other properties of various nature. An application may need to select and utilize only devices that have a specific context.

To address all these specifics, the following technological setup has been chosen:

- Service oriented architecture (SOA) is employed for the seamless integration of applications and to abstract from hardware and software heterogeneity of devices.
- Multicast DNS protocol (mDNS) is used as a communication protocol for service discovery. This decision is motivated by the distributed nature of the network. However, the compatibility with the classical DNS has to be preserved as well. The reason is that for some networks it may be beneficial to use proxy DNS servers that store the information about context and services instead of fully-distributed data storage.

- DNS based service discovery protocol (DNS-SD) is used to perform service discovery because of its popularity, which increases the potential number of devices and software systems supporting it.

Some specifics of these technologies are important for our work. They are explained in following subsections.

2.2.1 Wireless sensor networks

A sensor is a device that is able to measure some physical phenomena, like temperature or light levels. Sensors transform analogue data from the physical world in digital form and make them processable by computer applications.

A device that can both sense and transmit data is called a wireless sensor node. It usually has the following components: a microcontroller, a radio module with external or internal antenna, sensing component and an energy source, a battery or an embedded form of energy harvesting like solar panels. When an ability of wireless communication is added to the sensor, additional opportunities and use cases emerge. Sensors now can form huge networks that can measure phenomena in otherwise inaccessible places, like cattle stomachs [27].

Usually nodes in the network do not process the sensed data or at least do not perform the full processing required to compute the end result required by users. Instead, nodes transmit data to a processing machine through the special exit node called a sink. While the sensor can send its data to the sink directly, in WSNs it is usually not the case. Wireless nodes are often deployed far away from the sink or even any network devices except for other nodes. To cope with this problem, sensor nodes use each other as intermediaries to pass their messages to the sink, forming a mesh topology. It is also possible that groups of nodes send their data to local routing nodes, that are then responsible for forwarding data to the sink. Routing in WSNs is a subject of intensive research, and new solutions emerge every year.

We list here the main WSN-specific concerns that are relevant for our research.

Main concerns

Energy One of the main concerns in WSNs is power limitation. Wireless sensors often have no constant power supply and use batteries or energy harvesting. Sometimes recharging of sensing devices is not even possible [15]. This makes energy a precious resource that should not be wasted.

Power consumption can be divided into three domains: sensing, communication, and data processing [15]. In order to prolong sensors life, an application has to avoid performing operations that require a lot of energy. For different devices and applications the overall power consumption is distributed over domains differently. For nodes that spend most of their energy on wireless communication, the amount of network interaction required to serve a task should be minimized.

Scalability and adaptability Wireless sensor networks can consist of a large number of devices, covering huge spaces. Depending on the application, the number of nodes can reach an order of hundreds, thousands or even millions [15]. Some of these devices can be located out of the personnel reach. Furthermore, devices disconnect from the network due to the malfunctioning, low battery or a number of other reasons. New devices can connect to the network as well. In most cases the network is required to keep on working if these events happen.

Therefore, managing the network by hand may be virtually impossible. This means that the network has to have some means to self-organize, adapt to failures, changes in the network structure or environmental changes. An automatic discovery of the node that provides the required functionality by another node is an example of such behaviour.

Resources Because of their power constraints and their tiny size, individual nodes often face severe limitations of their computational resources. This means that, unlike most desktop applications, WSN applications have to optimize the usage of CPU and internal storage, avoid heavy computations and minimise the amounts of data stored in working memory or on the secondary storage. This applies for the discovery procedures as well.

Heterogeneity Often wireless sensor networks consist of a large number of very different devices: they have different vendors, different operating systems and different sensing capabilities. In other words, they can be heterogeneous on both software and hardware levels. If this is the case, applications have to be aware of these details, which is quite difficult, or abstract from specifics, for example, by adopting service oriented architecture or other abstraction [28].

Two of these problems (heterogeneity and adaptability) can be addressed by adopting a specific architectural pattern, which is described in the next section.

2.2.2 Service Oriented Architecture

Service oriented architecture is a design pattern, that describes a way to create complex applications from separate interchangeable building blocks, called services. These blocks can be reused in different operations and throughout the application, so that only the order of invocation or composition of services is changed, but not their code. With SOA, applications can be built almost entirely from existing software blocks.

A service-oriented application is a set of loosely coupled interacting services.[29]. A service is an isolated unit of functionality with well defined inputs and outputs. Details of its implementation are encapsulated and not known to the application or other services. These parties only know what the service does and what inputs need to be provided. A service can be both a service provider and a service consumer. Thereby a service can use other services to perform a required operation.

Loose coupling means that components of an application have little or no knowledge of each other. A component with a strong coupling relies on the existence of the other

component, that cannot be substituted by a component with the analogous functionality. A component with loose coupling, in contrast, only has knowledge about the interface of the required component. This interface can be implemented by one or more different components. SOA requires for a loose coupling between applications and services.

As SOA implies the necessity of communication between different interchangeable pieces of functionality irrespective of their implementation details, all parts of the application should support some common protocol. In fact, there are four different types of protocols that can be derived for a service-oriented architecture. We briefly explain them here.

Service access A Service access protocol provides a standardized way to connect to the service and use its functionality. It allows the application to abstract from implementation details of every individual service. The service can be written in any programming language and be executed on any device, but as long as it supports a service assess protocol, the application and other services will be able to use it. For services that require input data or return a result of their work back to the application, the service access protocol must define a standard way of encoding different types of data. For example, in the HTTP [14] protocol all non-Roman characters that are passed through the query string are encoded in hex representation, and non-ASCII characters are first encoded as UTF-8. In CoAP [13], a message can include a number of options in one of four formats: uint, string, empty or opaque.

Service description Service description provides an interface for the application and other services to use. It specifies what a service does, and not how it does it. The service description typically includes the name of the service, its inputs and outputs, type, and access point. For example, the DNS-SD [1] protocol describes the service with its name, type, transport protocol for interaction, and includes the address and port of the service. WSDL [9] additionally allows to specify inputs and outputs, entry points and several functions that can be performed by one service.

Service composition The composition protocol describes a way to build applications with an automatic coupling of available services. This activity involves translating user requests to processes and composing appropriate services in a way that they fulfill the users demand. Service composition is a topic of numerous researches [30].

Fundamentally there are two classes of service composition techniques: service orchestration with a dedicated software that performs composition, and the distributed technique of service orchestration.

Examples are FLUENT [11], that provides a dedicated orchestrator for each application in the system and a centralized repository, and eFlow [12], orchestrating services based on manually created workflow templates, that are accessed in a centralised manner, through dedicated information brokers.

Service discovery The loose coupling principle implies that the application is initially unaware whether components of required type exist or are available. It means that these components need to be located before they can actually be used. While for desktop applications, services can in principle be bound at compile time, for dynamic and self-configuring wireless networks this technique is inappropriate. Isolation and abstraction from implementation details allows for services to be found and selected by the application automatically at run time. This process is called Service discovery. Discovery can be performed by the application or other services. There are both centralized solutions with a dedicated service repository, that clients address, and distributed models. Of course, it is important that the application knows what kind of services it requires. For example, the service should provide a certain function and have certain inputs and outputs. For context-aware applications the context properties of services are of interest as well, meaning that they have to be addressed in the service discovery protocol.

Examples of service discovery protocols are DNS-SD [1] and UPnP [10]. In the next subsections we will examine the DNS-SD protocol in more detail.

2.2.3 Multicast DNS

Multicast DNS (mDNS) is typically used as a carrier protocol for DNS based service discovery. We discuss it here in order to give an idea on how the service discovery data might be delivered in distributed networks. mDNS is a protocol for name resolution in local networks that do not have a local name server. The protocol utilizes mostly the same packet format and semantics as the classical DNS. It is possible to use DNS alongside mDNS in one network.

As there is no name server to store the naming information, this data is distributed among devices in the network. The device itself stores records with its own name-to-address mapping and other information, like DNS-SD service descriptions. Also mDNS clients may have caches to store some information about other network participants.

mDNS only resolves names that end with the `.local` top-level domain. When an mDNS client needs to resolve a domain name, it sends a multicast request with this name to the local network. A node that has an information about this name sends back a response. Eventually this information is expected to reach all other devices in the network, and update the data for this name in their caches. The time records are stored in the cache depends on the Time To Live (TTL) value, that is contained in every record.

This allows for a technique to remove the information about a name from the network: a host that leaves the network may free the domain name by sending a response for the name with TTL value equal to zero.

mDNS messages are UDP packets sent to a specific multicast address. Addressing details are specified in Table 2.1.

In practice mDNS allows to provide the discovery of services in a distributed system of nodes without the central server. Discovery cn be performed regardless of the network structure or the presence of specific nodes. The next section describes the service discovery protocol that addresses this matter.

	IPv4	IPv6
Mac address	01:00:5E:00:00:FB	33:33:00:00:00:FB
IP address	224.0.0.251	FF02::FB
UDP port	5353	

Table 2.1: Multicast addresses used by the mDNS protocol.

2.2.4 DNS based service discovery

DNS based service discovery, or DNS-SD, is a service discovery protocol that allows services to be described and resolved using DNS resource records. The re-use of the DNS protocol is beneficial, because it allows to avoid deployment and configuring of a separate service discovery system.

The DNS protocol is widely used in networks of all sizes for many years and has gained an exceptional popularity. Virtually every company that has an IP network also has a DNS server or other infrastructure, meaning that it also has a corresponding expertise. DNS-SD allows to reuse these resources to provide an additional value.

Another advantage is that DNS-SD can work both as a centralised service discovery protocol and as a distributed one. In the first case the information about services is stored on the DNS server and obtained with usual DNS requests to this server. In the second case mDNS is used to provide distributed service discovery. In this work the main accent is made on the DNS-SD discovery protocol on top of the mDNS.

Here we present capabilities of DNS-SD and the specifics of service discovery with this protocol. We start with the service description part to provide the understanding of what information about the service can be stored. Then we explain the process of discovery, illustrating it with two examples. We expect the reader to be familiar with such DNS-related concepts as *resource record* and *domain name*.

2.2.4.1 Service description

Services in DNS-SD are described using two DNS resource entry types: SRV and TXT.

SRV records Entries of type SRV are used to link type and domain of a service to the service instance. A SRV record has the following form:

```
<service>.<apptype>.<protocol>.<domain>.  <TTL> <class> SRV
<priority> <weight> <port> <target>
```

- `<service>` - the human-readable name of the service. It may contain any characters that can be represented using Net-Unicode [31], including spaces, uppercase, dots or non-Roman text. The service name may be at most 63 octets long, which, depending on the encoding, may result in 15 Unicode symbols in the worst case.
- `<apptype>` - the application type of the service. Describes what the service does and the application-level protocol it uses. For example, `_ipp` type is often used for network printers.

- `<protocol>` - the protocol used for accessing the service. Possible values are `_tcp` and `_udp`. `_tcp` is used in the case service protocol is implemented on top of TCP, `_udp` is used in all other cases;
- `<domain>` - the domain name for the service. It may be any domain name suitable for the task. For example, domain name may represent the location of device: `Floor2.Building1.example.com`. If mDNS is used, the only domain name allowed is `.local.`.
- `TTL` - Time To Live, in seconds. Specifies the period of time the record is valid for;
- `class` - DNS class, usually IN;
- `priority` - Priority, smaller value indicates higher priority;
- `weight` - Relative weight for entries with the same `priority` value;
- `port` - TCP or UDP port number the service listens to;
- `target` - The host name of the host providing the service.

For the sake of conciseness, the `<type>.<protocol>.<domain>` portion of SRV name is further referred to as "Service type". The `<type>` portion is denoted as "Application type".

TXT keys DNS TXT records are designed to store key-value pairs. Such key-value pair is called a TXT key. The usage of TXT is optional, and the specification advises to make service usable even if a client does not have the information from this record. TXT keys are used to further describe the service and provide all kinds of additional information, including context. For example, for network printer they could provide the information about whether the printer can print in colors.

There exists a list of service types [32], that specifies what TXT keys should be provided by services of each type. Among these, there is one service type that provides location information about itself. The `airprojector` type, illustrated in the Figure 2.1, specifies the "note" TXT key for location information. However, we cannot query for entries that have a specific TXT keys value, which makes this feature hardly applicable for the service discovery.

2.2.4.2 Service discovery

PTR records Records of type PTR are used in DNS to map one domain name to the other. A PTR record has the following format:

```
<name> <TTL> <class> PTR <alias>
```

- `<name>` - the source name;

```
airprojector  AirProjector
              Yoshinori Nakayama <yoshinori_nakayama at komatsu-trilink.jp>
              Protocol description: http://www.komatsu-trilink.com
              Defined TXT keys:
                mac=<MAC address>
                ip=<IP address>
                note=<Location>
                use=<Status>
                mainprog=<Main program version>
                bootprog=<Boot program version>
```

Figure 2.1: The service type that specifies location in its TXT keys.

- `TTL` - Time To Live, in seconds. Specifies the period of time the record is valid for;
- `class` - DNS class, usually IN;
- `alias` - The destination name.

Service discovery in DNS-SD is performed by sending a request for PTR records with a specific name. This name should contain the application type, protocol and domain of required services. These are all context properties that can be addressed with DNS-SD. It is implied that the client knows exactly these parameters.

Let us have a closer look on the DNS-SD request. The request has a following structure:

```
<apptype>.<protocol>.<domain> PTR
```

The discovery starts with the client making a request for PTR entries that contain desired service type and domain in their names. The result of this PTR lookup is a set of zero or more PTR records containing Service Instance Names of the form

```
<service>.<apptype>.<protocol>.<domain>
```

in their aliases.

For example, to get the list of services with the service type "_ipp._tcp.example.com.", the client requests records of type "PTR" with this type as a name. The result is a set of zero or more PTR records of the form:

```
_ipp._tcp.example.com.  PTR <service>_ipp._tcp.example.com.
```

Now to obtain SRV-type records that contain service access points, the client must query for each of these names with an SRV request.

Example 1 Let us examine the work of the system with a simple example. In this one, DNS-SD is working on top of mDNS.

Say the node named NodeA joins a network and tries to find a smart light that uses UDP communication protocol. Suppose there is a node NodeB in the network that, indeed, hosts a smart light service named `nodeB._lgt._udp.local`. Let us assume there is a service type `lgt` for smart light services.

NodeA issues a request for PTR entries that point to this kind of services:

```
_lgt._udp.local PTR
```

Then NodeA should receive the name of service and perform an SRV lookup by this name. This process is shown on the Figure 2.2.

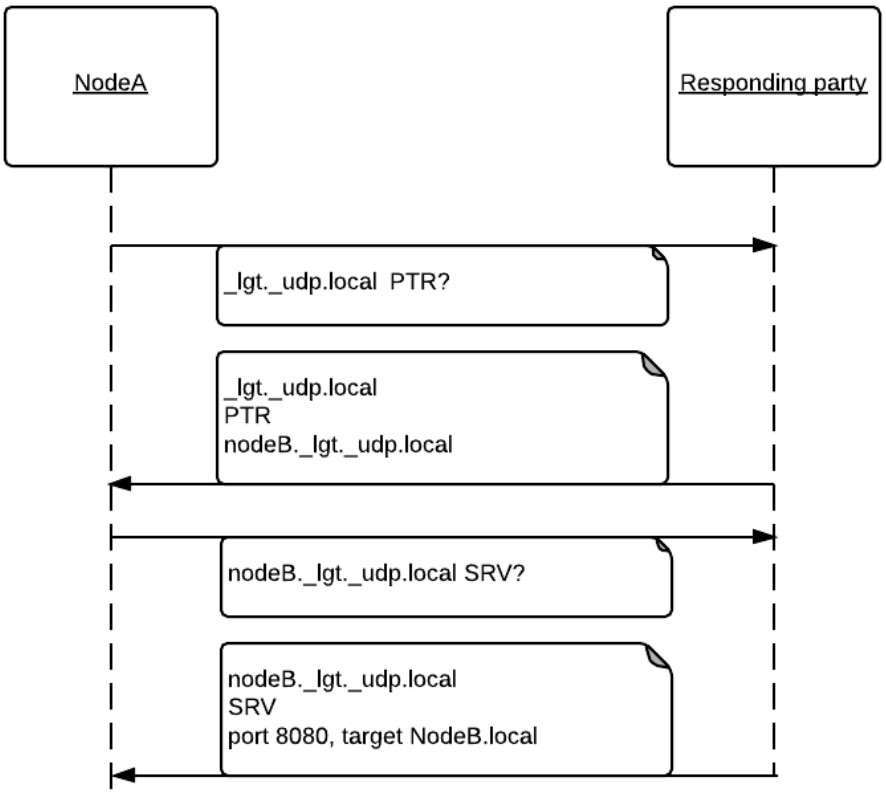


Figure 2.2: The sequence diagram for the example of service discovery with DNS-SD.

Additionally the NodeB service description may include other information required to access the service, such as IP addresses and TXT keys:

```
nodeB.local AAAA fe80::762f:68ff:fe0d:4e1d
nodeB.local A 192.168.1.15
nodeB._lgt._udp.local:  type TXT
txtvers=1
key=value
```

Now the node NodeA has all the required information to connect to the smart light. In the case these additional data and TXT keys were not transferred together with the SRV recors, NodeA requests them explicitly.

The question is, what node should send all these responses? It can be either NodeB itself, or some other node that stores this information. We could, for example, set up a small DNS server on one of the nodes for this purpose.

Example 2 There is another option for mDNS networks: nodes may send out the information about their services upon connecting to the network or periodically. This information ends up in other nodes DNS caches, so they do not even have to send network requests to obtain it.

Again, let us consider a smart light service running on node NodeB. This is the record this node sends out:

```
_lgt._udp.local PTR nodeB._lgt._udp.local
```

Once this information is in clients caches, they can proceed by requesting SRV record for `nodeB._lgt._udp.local` without further PTR requests.

If NodeB is not ready to serve clients anymore or just about to leave the network, it may send the response record (`_lgt._udp.local PTR nodeB._lgt._udp.local`) with Time To Live equaling zero to make sure this entry is removed from clients caches.

2.2.4.3 Implementations

The protocol is widely used and has been implemented for several platforms, including embedded ones. The list of implementations can be found in Table 2.2.

In this work, we use JmDNS library as a foundation to build an API and demo application. We develop additional functionality associated with context-aware discovery on top of the existing library.

Name	**Platform**
Apple Bonjour	Apple, Windows [2]
Avahi	Linux [3]
JmDNS	Java [33]
mDNS for Arduino	Arduino [34]
mDNS for Contiki	Contiki OS [35]
mdns	Python [36]

Table 2.2: Implementations of DNS-SD protocol for different platforms.

2.2.5 Example of WSN usage: SenSafety project

The given concepts and technologies are applied in Sensafety project [4].

SenSafety is a public safety project that relies on the wireless sensors technology. Its aim to offer real-time automatic analyses of potential hazardous situations and support first responders.

An example of SenSafety use case is to ensure people safety on the festival. In particular, there might be a group of aggressive people that should be detected and reported, possibly captured with photo or cameras. Another use case example might be to predict unsafe crowd movement or stampede, and notify authorities to help them safely direct people and resolve the problem before the situation becomes critical.

The SenSafety network may consist of numerous heterogeneous devices, including sound sensors, video cameras, chemicals detectors or even user smartphones. These devices can move, appear or disappear, sometimes unpredictably.

The SenSafety project employs SOA architecture, where a single service addresses a single sensor functionality. mDNS and DNS-SD protocols are used for service discovery. As wireless nodes may be put to sleep for saving energy, the architecture includes active and passive proxies for discovering and addressing services. The project provides a light-weight implementation of mDNS/DNS-SD protocol for Contiki OS [35], which is suitable for resource-constrained devices.

For this kind of applications it is essential to know in what context devices are operating, to realize what measures can be taken and how the network may or may not influence the situation. For example, it is essential to know the location of sensors that detected the stampede, so that authorities could arrive at the spot and resolve the problem. Context-based discovery is the part of the problem that is addressed in this work.

2.3 Problem of context in DNS-SD

As we stated before, applications that use wireless sensor networks can gain a great value from information regarding the context of individual nodes. When SOA with distributed service discovery is deployed, context-aware discovery brings yet one more benefit.

Due to the loose coupling principle, services and applications have to find the functionality they need at run-time. With the mDNS protocol they do so by sending a multicast local request that propagates through all the network. In general, the more precisely a required service is described, the less nodes will reply to the request. For example, assuming that a network contains smart lights of white and red colors, the request for red lights will result in less responses than the request for all lights irregardless of color. Obviously, less responses means less network load.

As we work with a WSN, that has a network load listed as one of its main concerns, the minimisation in network load is highly desirable. This means we have to perform service discovery with as much context detail in the request as possible. Unfortunately, DNS-SD does not allow us to do that.

Imagine the office building with a smart light control system. The lights in the room are only enabled when there is a person inside, detected by presence sensors. There might be thousands of such sensors in the building. If the application will send a discovery request for presence sensors with DNS-SD over mDNS, all these sensors will have to reply, loading the network. The application, however, might only need dozen of sensors on the third floor, or even the one in a specific office. If there was a way to specify, that only sensors in a specific location should reply, this might improve the network performance and sensors battery life.

The goal of the current work is to enable DNS-SD to perform context-aware service discovery, so that other service properties than type, protocol and domain could be addressed. The amount of expressable context should be enough to create the full wireless node description in terms of any particular application.

Now that we went through the setup components, their limitations, and specified the problem, we can formulate the set of requirements for the desired solution of the context-aware service discovery problem.

2.4 Requirements to the solution

The DNS-SD specification states one of the properties for service discovery:

> *The ability to query for services of a certain type in a certain logical domain, and receive in response a list of named instances (network browsing or "Service Instance Enumeration").* [1]

The goal of this work is to add a context to this requirement; hence, not only certain type and domain should be included in query, but also context properties. This is our functional requirement.

However, the specifics of the setup and of the used technologies produce certain limitations, that lead us to formulating more non-functional requirements. We list them here, motivating their usage and the connection with the described setup. These requirements will later be evaluated with the set of metrics, which are presented in Chapter 4.

2.4.1 Minimise network load associated with service discovery

Due to the energy constraints, network load associated with the service discovery should be minimised. The requirement is motivated by the fact that the setup includes very resource-limited devices and devices with the limited power supply. We noted earlier that network communication is one of domains that significantly contribute to the energy consumption. Hence the minimisation of network interactions can prolong life of the energy-constrained sensors.

2.4.2 Provide rich discovery features

Applications might need to discover services with varying context. Sometimes the required context can be expressed as a combination of several atomic context properties. This allows to deploy applications with different context requirements without changing the set of context properties assigned to services. For example, one application might request for service providers with full battery, and another one - for those with full battery or constant power supply. The first requirement can be expressed as an atomic context property "full battery", and the second one - as a combination of two properties, "full battery" and "power supply", of which at least one should be present.

To ensure that context properties can be reused by different applications, and that applications are able to address the required context as precisely as possible, we require that the solution had means to compose complex requests with several atomic context properties. For example, the application should be able to specify that for a desired service several context properties should present all at once, or only one of them might hold.

2.4.3 Minimise the amount of additional client code

We want to limit changes to the DNS-SD protocol usage and introduce as little additional functionality in software of nodes as possible. In the perfect case, the client that implements DNS-SD over mDNS should be able to request services by their context without introducing any additional protocol-level code on client side. There are two reasons for this requirement. First, the network is heterogeneous on several levels, and reimplementing additional code for several architectures can be troublesome. Second, some devices in the network are extremely limited in resources and storage, which means that it might be hard or impossible to introduce additional supportive software.

2.4.4 Minimise the amount of memory required for context

The amount of memory required to store context of services should me minimised. Both working memory and secondary storage are addressed in this requirement. The motivation is again the need for extremely resource constrained devices to support the service discovery protocol. Such hardware may have very limited amount of storage, and it should be possible for these devices to hold all the data required for the service discovery.

2.4.5 Maximise the possible number of expressible context properties

While we do not limit our work applicability with one domain or area, the need to express any sets and combinations of context properties may arise. It is important to know how much context information can be associated with one service. We would like to be able to address as many properties as possible.

Chapter 3

Methods

In this section we describe our methods and solutions. We start with the model to express various context properties. After that, we introduce the leading example which illustrates any following parties interaction. Then we discuss possible means to include the context, expressed according to the model, to the DNS-SD protocol. We give examples of the DNS message flow for each of the proposed solutions.

3.1 Context model

The context information of the service may be of different nature or precision, and depending on the application, may be expressed in different forms or notations. The human-readable context properties might be desired, as well as compact notations for the machine-to-machine interaction. This justifies the need for a flexible application-independent context model.

To address this problem, we define "Context tags" - pieces of context information similar to tags in Web 2.0 blogs and web pages. One tag defines one atomic context property. Even though these properties can have similar meaning or even can belong to some hierarchy (for example, the office is located in a specific building, which is a part of a specific campus etc.), but functionally there is going to be no connection between them: every tag can be assigned, removed and addressed individually. In that sense we can say that tags are mutually independent.

The combination of tags will create a full node description. For example, a node might have a set of human-readable tags: "2nd floor" or "on the wall" or "on maintenance". For machine-to-machine interaction a service might rather have tags like "f2", "w", and "status=mnt" to express the same meaning. This scheme allows us to encode a very wide range of properties of different nature with a varying level of detail.

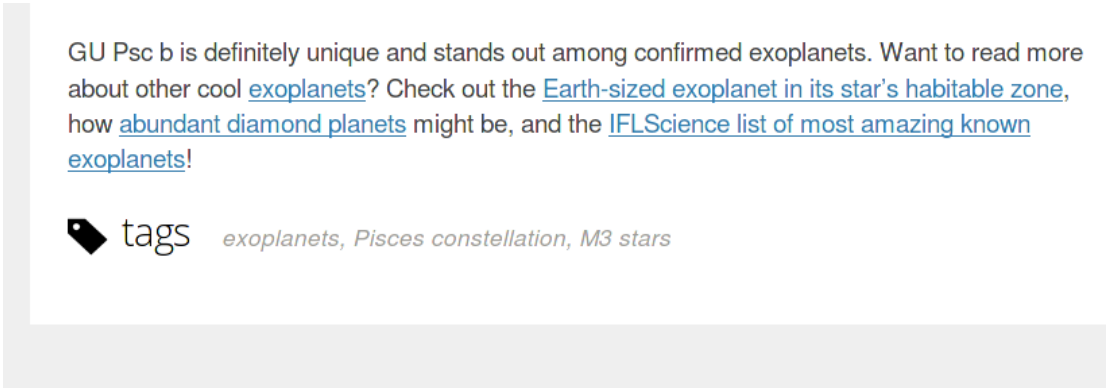


Figure 3.1: The example of tags applied to an article on a popular scientific website [6].

3.1.1 Formal description

In this subsection we give a formal description of service discovery with context tags and describe which entities exist in the discovery model.

The context of every service is described by the set of tags it has. Hence, each service can be mapped to a certain set of tags. The discovery of services includes making a *query* for a desired combination of context tags and returning the set of services whose set of tags satisfies the query. We use boolean logic to express context queries. Basic operations of boolean algebra are: AND (conjunction), denoted as $\wedge$; OR (disjunction), denoted as $\vee$; and NOT (negation), denoted as $\neg$. The context query can contain tags and any of those operations.

Therefore we define the following elements of the context model:

- A set of services R;
- A set of tags T;
- A mapping function $\pi : R \to \mathcal{P}(T)$;
- A set of context queries, defined by a grammar $Q := t|Q \wedge Q|Q \vee Q|\neg Q$ with atoms $t \in T$.

In the context model, a *discovery function* is the mapping from the context query Q to the set of matching services: $\sigma : Q \to \mathcal{P}(R)$. If the query only contains one context tag, then the discovery function simply returns all of the services that have this tag assigned. For a conjunction of tags, the function returns all of the services that have all of the tags at once, and for disjuncture of tags the function returns all services that have any of the tags. The negation of the tag in the query results in the set of services that do not have this tag applied.

Let t be an individual context tag, p and q are context queries, $\pi(r)$ is the function that maps services to their sets of context tags, then the discovery function $\sigma(Q)$ is defined as follows:

$$\sigma(t) = \{r \in R | t \in \pi(r)\}$$
$$\sigma(p \wedge q) = \sigma(p) \cap \sigma(q)$$
$$\sigma(p \vee q) = \sigma(p) \cup \sigma(q)$$

$$\sigma(\neg t) = \{r \in R | t \notin \pi(r)\}$$
$$\sigma(\neg[p \wedge q]) = \sigma(\neg p) \cup \sigma(\neg q)$$
$$\sigma(\neg[p \vee q]) = \sigma(\neg p) \cap \sigma(\neg q)$$

The described model of context allows for creating context descriptions for services by assigning context tags, and discovering services with the desired context by querying for corresponding context tags combinations. To deliver any utility, this model has to be used in a service discovery protocol. However, the selected service discovery protocol (DNS-SD) does not describe such kind of queries, compelling us to explore the possibility to extend the protocol with the specified context model.

In the following sections we discuss possible implementations of the context model with the DNS-SD protocol.

3.2 Context in DNS-SD

In this work we use DNS-SD as a service discovery protocol. As DNS-SD does not provide means to use context of services in discovery, we have to introduce certain changes in this protocol to enable discovery of services with context tags.

To make sure the changes do not break DNS-SD protocol, we need to include context tags of services to fields of DNS messages that are allowed to carry the user information. As the core concept of DNS is a name lookup, and for a DNS responder the only available operation is to return a requested resource record by its name, searchable parameters like context tags should in the name field of one of those records. The work [22] describes a way to perform filtering of service lookup results on the server side using TXT keys, but this requires a changing of DNS lookup procedure.

The mechanism used by the DNS-SD protocol to discover services, described in section 2.2.4.1, uses the name of the PTR record to express properties of a service. While in DNS-SD only service type is included in this name, we can as well encode context tags there. Then to query for services with a particular context the client will have to send a PTR request, where the PTR name contains a (part of) context query alongside a service type.

However, there are some particularities of expressing boolean algebra with DNS. One interesting property of DNS requests is that they may contain several separate questions. If the responding party has a piece of information that answers any of questions, this information will be sent in a response. This behavior can be used to express the boolean OR operation on context tags.

Another particularity is related to the negation operation. DNS lookup-based responders cannot process negations in requests, as it would require to store records with all

possible combinations of negated and non-negated tags in the system. However, the negation can be implemented if either the requester or responder performs a processing of queries. For example, to process the query `tag_1` $\wedge \neg$`tag_2` the requester may separately request for services with `tag_1` and for those with a negated tag `tag_2`, and then exclude services with both `tag_1` and `tag_2` from the returned set. This implementation of negation cannot be used on individual tags, but rather on tags that participate in conjunction, because the requester needs to have some set of services to filter out those with negated tags. As an example of responder-side negation processing, a responder might parse the query in PTR name and check this query against known services, instead of performing a name lookup.

Though these specifics of DNS provide ideas and constraints of expressing context tags, there is still some choice in an exact format of service description and discovery queries. In this section we demonstrate and discuss several methods of performing association of context with the service name using PTR records. We define four *naming schemes* that implement the $\sigma(Q)$ discovery function through requests and responses for PTR records. Naming schemes differ with: the format and number of messages; the party responsible for evaluating context queries; and the amount of supportive code. As a result, different naming schemes also have different influence on usage of memory, network load and other resources. This influence may matter for a specific setup and the set of available services. Therefore there is no single best solution; properties of several naming schemes should be investigated in detail. For example, a setup with a legacy DNS server will require a scheme that does not perform anything but DNS lookups on the server side. On the other hand, a setup with a dedicated service discovery server might employ server-side query evaluation in order to conserve resources of clients.

In this report we restrict ourselves to the following choices of naming schemes. The **Formula in PTR** naming scheme delegates the selection of services to the responder side. The requester encodes the context query in the PTR question, and the responder has to parse it and send back appropriate service names. The **Tag to PTR** naming scheme, in contrast, delegates the selection of services to the requester. The requester asks for a list of services for every tag in the query formula and filters out services that do not satisfy the query. The responder just performs a lookup for requested tag names. The **Sorted tags in PTR** naming scheme also delegates the selection of services to a requester, but reduces the processing at cost of more complex requests. The scheme breaks the context query into conjunctions and places each in a separate DNS question. Post-processing of results is required only to perform negation of tags. The responder, again, just performs a lookup. The **Nested tags combinations** scheme is similar to previous one, but it requires more DNS requests and additionally provides the client with a full context of each service.

The next subsection describes these schemes in more detail. For each naming scheme we give the its description, the algorithm that describes behavior of requesting and responding sides, and compare schemes using a running example.

3.2.1 Running example

In our running example we demonstrate the sequence of DNS messages that leads to discovery of a service with required type and context tags. Suppose we have to perform maintenance of the organisations' printers, that includes changing of cartridges for laser printers and refilling paper supplies for all printers with no paper. Paper should not be delivered to printers that belong to the designer Zoe, who needs a specific kind of paper and reloads her printers herself.

Descriptions of services for this example are listed in Table 3.1. There is one commonly used printer that currently has no paper to print on. There is another commonly used printer: this one is lazer and also has no paper. There is a printer that belongs to Zoe and has no paper as well. Zoe also has a smart light in her room, which is of no interest to the application, and demonstrates the need to filter services by their service type. The requester is interested in services of type `_printer._tcp.local`, that satisfy a context query $(\texttt{laser} \wedge \texttt{noToner}) \vee (\texttt{noPaper} \wedge \neg\texttt{Zoes})$, in other words, all lazer printers with no ink and all printers with no paper, except for those of Zoe.

We assume that mDNS technology is used for service discovery. Every device hosts its own service and replies to context discovery messages for this service.

Service	**DNS-SD SRV record name**	**Context tags**
Common printer	`Printer1._printer._tcp.local`	`noPaper`
Common lazer printer	`Printer2._printer._tcp.local`	`laser`, `noPaper`
Zoe's printer	`PrinterZ._printer._tcp.local`	`noPaper`, `Zoes`
Zoe's smart light	`SmartL._lgt._udp.local`	`Zoes`

Table 3.1: Entities that participate in examples of this section. The context query is $(\texttt{laser} \wedge \texttt{noToner}) \vee (\texttt{noPaper} \wedge \neg\texttt{Zoes})$.

3.2.2 Naming schemes

3.2.2.1 Formula in PTR

This naming scheme addresses context queries by including the whole query in the name of a PTR request. The query is parsed and evaluated on the responder side.

No resource records are created for the service in advance; they will only be generated as responses to clients' requests. Instead, the responder can store known services and tags associated with them in flat lists or other structures. The responder should be able to iterate over the set of services and retrieve a set of tags associated with any known service.

The naming scheme allows for including the whole range of boolean operators in a query. The discovery is performed by requesting for the name that contains the whole context query concatenated with a label divisor and a service type. However, the DNS standard limits the range of symbols that can be used in PTR records. Therefore, it is necessary to select the characters that will denote boolean operators between context tags. The actual

choice depends on the application and the kind of formulas that have to be addressed. In this work, the conjunction is represented by the asterisk ("*") and the disjunction is represented by the label divisor (".""). The hyphen ("-") symbol is used to express negation. We assume that the query is short enough to fit into one pointer name together with a service type[1]. Also we demand a context query to be in Disjunctive Normal Form (DNF) [37], because it allows to avoid using brackets and to break long formulas into several DNS questions.

The requester composes a string according to these rules, concatenates the string with a label divisor and a desired service type, and uses the result as a record name in a PTR request. The responder receives a request and parses the record name, thus obtaining the original query. If the responder can provide the information about the service with the satisfying set of tags, it responds with a PTR record with the original query as a name and the service name as a target.

Algorithm

Requester DiscoverServices(Context query Q in DNF)

1. $A \leftarrow$ Empty DNS request;
2. $n \leftarrow$ encode Q with symbols allowed by DNS;
3. $p \leftarrow$ PTR question for n;
4. Add p to A;
5. Send(A);
6. $E \leftarrow$ Empty set of services;
7. **while** (No appropriate response in E **and not** timeout reached):
8. Receive response e;
9. Place e in the set of responses E;
10. **end while**

Responder RespondToRequest(DNS Request R, Services storage G)

1. $Q' \leftarrow$ Boolean query restored from R;
2. $A \leftarrow$ Empty DNS response;
3. **for** Service $r \in G$:
4. $T \leftarrow$ set of tags for r;
5. **if** (T **satisfies** Q'):
6. $p \leftarrow$ PTR record for r;
7. Add p to A;

[1]In the case the query is especially long, it can be divided into several DNS questions; however, for the purpose of this work we assume that the query is transmitted into one DNS question.

8. **endif**
9. **endfor**
10. SendBack (A).

The "**satisfies**" function on line 5 of the responder algorithm corresponds to the procedure of checking the value of the boolean formula in DNF given the set of tags. In our implementation, for each service its tags are replaced by "True" in the query formula; the rest of the formula's tags are replaced by "False". Then, the value of a resulting expression is calculated. This algorithm is described in detail in chapter 4. The satisfying set of services is generated on the responder side by selecting services that satisfy the query, which guarantees that all received services are indeed in a set $\sigma(Q)$.

Example The generated DNS question is the following:

```
laser*noToner.noPaper-Zoes._printer._tcp.local PTR
```

The response will then be

```
laser*noToner.noPaper-Zoes._printer._tcp.local PTR Printer1._printer._tcp.local
laser*noToner.noPaper-Zoes._printer._tcp.local PTR Printer2._printer._tcp.local
```

In our case, an mDNS is used, so these two records arrive separately. If a DNS server is used to store all records, these two records arrive together in one DNS response.

The sequence diagram in Figure 3.2 summarises the discovery process with the **Formula in PTR** scheme.

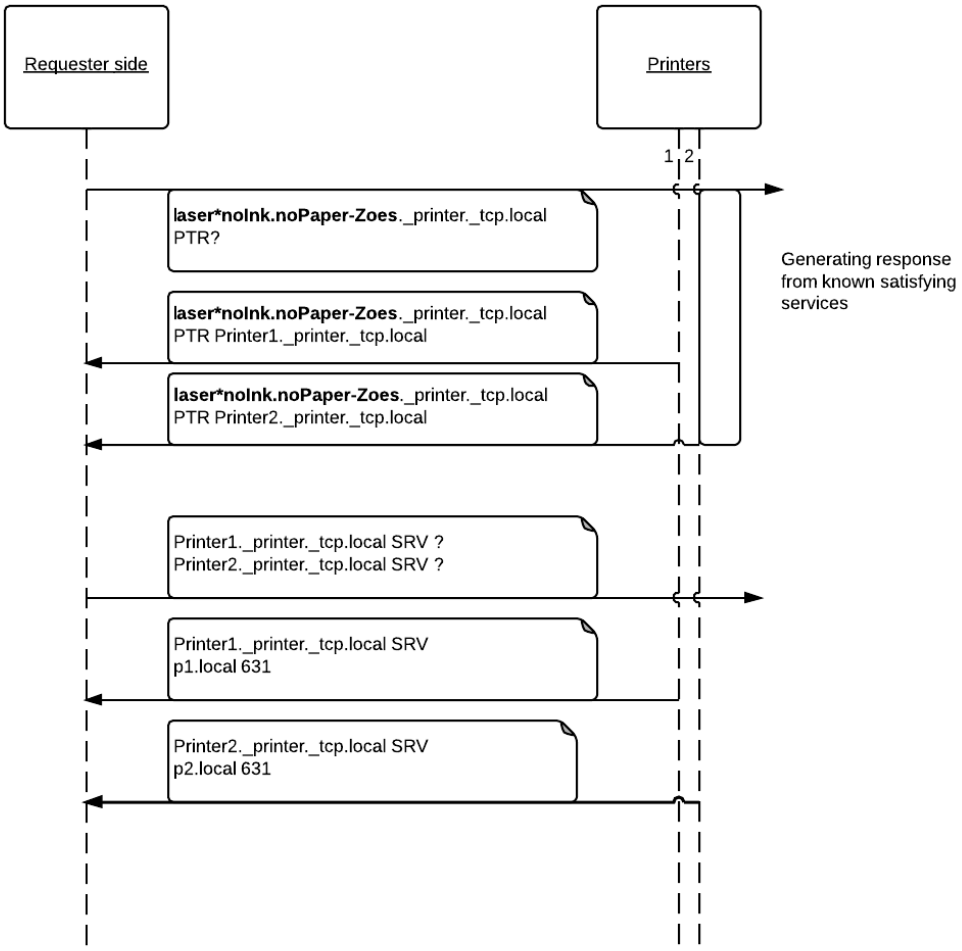


Figure 3.2: The sequence diagram for the service discovery process with the Formula in PTR scheme.

3.2.2.2 Tag to PTR

The Tag to PTR naming scheme creates one PTR record for each context tag on a service. This way the client that sends a PTR question with a specific context tag and service type will receive a response of a record that points to the SRV record name. The query is evaluated on the requester side.

For each context tag associated with a service, a pointer record is created. The name of the pointer is equal to the tag value concatenated with the service type by the label divisor. The pointer of the PTR record is the name of the service.

The naming scheme allows for all boolean operations in a query, but this time the query is evaluated on the requester side. The client has to query for all tags in the context query formula individually. For every tag in a query, one PTR-type question is created. A name of a question is a tag concatenated with a label divisor and the service type. When

responses arrive, the client learns which service has which tag. With this knowledge, the client can validate the received set of services against the context query.

Algorithm

Requester DiscoverServices(Context query Q)

1. $A \leftarrow$ Empty DNS request;
2. $T \leftarrow$ Set of tags in Q;
3. **foreach** t in T:
4. $p \leftarrow$ PTR question for n;
5. Add p to A;
6. **end foreach**
7. Send(A);
8. $X \leftarrow$ Empty set of pairs $< Service, SetOfTags >$;
9. $E \leftarrow$ Empty set of services;
10. **while** (No appropriate response in E **and not** timeout reached):
11. Receive response e;
12. **foreach** DNS Answer p in e:
13. $Name \leftarrow$ Name(p);
14. $Alias \leftarrow$ Alias(p);
15. **if** ($Alias$ **not in** X):
16. $X_M \leftarrow \{Name\}$;
17. Add $< Alias, X_M >$ to X
18. **else:**
19. $< Alias, X_M >:= X[Alias]$;
20. Add $Name$ to X_M
21. **end if**
22. **foreach** Alias $Alias$ in X:
23. **if** (Set of tags $X[Alias]$ **satisfies** Q):
24. Add $Alias$ to E
25. **end if**
26. **end for**
27. Try selecting the appropriate service(s) from E.
28. **end while**

Responder CreateRecordsForService(Service S, set of tags T)

1. $P \leftarrow$ Empty set of DNS pointer records;
2. **foreach** t in T:
3. $p \leftarrow$ new PTR record;
4. $p.Name \leftarrow t$;
5. $p.Alias \leftarrow S$;
6. Add p to P;
7. **end for**
8. **return** P

RespondToRequest(DNS request R, Records storage G)

1. $A \leftarrow$ Empty DNS response;
2. **for** DNS question $p \in R$:
3. **if** G contains a record for p:
4. Add $G[p]$ to A;
5. **endif**
6. **endfor**
7. SendBack (A).

The "**satisfies**" function represents the same procedure as for **Formula in PTR** scheme (3.2.2.1). The satisfying set of services is calculated on the requester side, based on the obtained mapping of services to tags.

This algorithm returns valid results for queries without negations. If negations are present in the context query, the algorithm requires that, for every service, its whole context was stored altogether by one party. The responder sends DNS answers to all questions in DNS requests altogether, in one DNS response. The reason of that is a necessity of processing negation on the client side. If the requester receives the message with all relevant tag-service mappings, it will be able to determine if the service satisfies the context query. Otherwise, there is always a possibility that a service has some tags that make it unsatisfying, but the party storing these tags failed to deliver this information. In this case, the algorithm will return an unsatisfying service. This situation is illustrated in the following example.

Example The whole context information for services is stored in the following records:

```
noPaper._printer._tcp.local PTR Printer1._printer._tcp.local

   laser._printer._tcp.local PTR Printer2._printer._tcp.local
 noPaper._printer._tcp.local PTR Printer2._printer._tcp.local

 noPaper._printer._tcp.local PTR PrinterZ._printer._tcp.local
    Zoes._printer._tcp.local PTR PrinterZ._printer._tcp.local

              Zoes._lgt._udp.local PTR SmartL._lgt._udp.local
```

Then the client composes and sends three DNS questions, one for each tag. As a response the client is expected to receive five PTR records: one for the common printer, two for the common lazer printer and two for Zoe's printer. The client needs to detect the fact that some records point at the same service and note the fact that the received service names are associated with several tags at once. The client has to remember that `Zoes` is a negated tag and services with this context should be filtered out (unless they satisfy some other conjunction in a formula). Now the client checks the received set of tags against the formula for each service and finds that common printers, indeed, satisfy the context query.

Now imagine that the record `Zoes._printer._tcp.local PTR PrinterZ._printer._tcp.local` is stored on a separate device, which happens to go offline at the moment the client makes its request. In this case, the client will receive only four records, and assume that Zoe's printer does not have a tag `Zoes`. The following conclusion is that Zoe's printer satisfies the conjunction $\texttt{noPaper} \wedge \neg\texttt{Zoes}$ and hence should be presented as a valid result. Zoe will certainly not like this. Therefore, transmitting records for one service separately is not safe for queries with negations.

The sequence diagram on the Figure 3.3 shows the discovery process with the **tag to PTR** scheme.

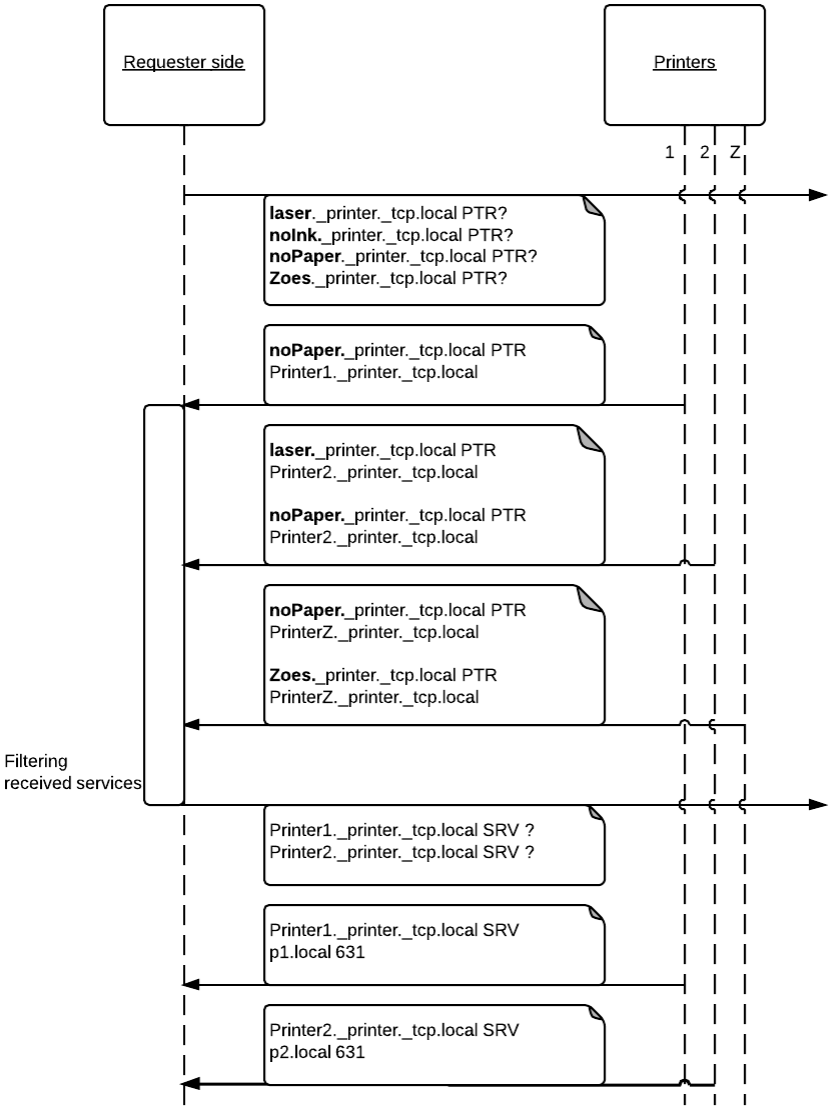


Figure 3.3: The sequence diagram for the service discovery process with the Tag to PTR scheme.

3.2.2.3 Conjunctions in PTR

The Conjunctions in PTR naming scheme places each conjunction of tags from the query to a separate PTR record. The negation is evaluated on the requester side, while the conjunction operation is expressed in messages.

To store the context information for a service, it is required to create a PTR record for each conjunction of tags that the service satisfies. This will ensure that the service is discovered when one of these conjunctions is requested. Thus, the PTR record that

contains all tags associated with this service is created. Additionally PTRs containing all combinations of tags in their names are created. To ensure unambiguity in combinations of tags, all tags in the pointer name are required to be lexicographically ordered. This way we make sure that every party produces exactly the same names out of the same sets of context tags. As before, every PTR name also contains a service type.

This scheme supports all boolean operations. While conjunction is expressed in the message format and does not require post-processing, the negation has to be evaluated on the client side after receiving responses. To query for a formula, the requesting side first brings the formula in the DNF. Then for every conjunction, the DNS PTR question containing all non-negated tags in the conjunction is created. This request can be resolved by the responder with the DNS lookup for each of DNS questions. For each conjunction with negations, a PTR question with all non-negated tags in its name is created. For every negated tag, a separate PTR question containing this tag is created.

The responder performs a lookup for all requested names and replies with matching PTR records. For each PTR record in response, the requester checks if the name of a record corresponds to any of a query conjunctions without negations. If it does, a pointer of a record is accepted as a satisfying service name. Otherwise, the responder has to check if the context of the service in the pointer of record satisfies any conjunctions with negations. If the service name has all non-negated tags of any conjunction, and has none of its negated tags, the service name is accepted as a satisfying name.

Algorithm

Requester DiscoverServicesSorted(Context query Q in DNF)

1. $A \leftarrow$ Empty DNS request;
2. $C \leftarrow$ Set of conjunctions in Q;
3. $C^- \leftarrow$ Set of conjunctions with negations from Q;
4. $Satisy \leftarrow$ Empty set of strings;

   Create questions for all conjunctions of tags:
5. **foreach** c in $C \setminus C^-$:
6. $T \leftarrow$ Set of tags in c;
7. Sort(T);
8. $n \leftarrow$ Concatenate items from T with ".";
9. $Satisfy.Add(n)$;
10. $p \leftarrow$ PTR question for n;
11. Add p to A;
12. **end foreach**

    Create questions for conjunctions with negations:
13. **foreach** c in C^-:

14. $T^+ \leftarrow$ Set of non-negated tags in c;
15. $T^- \leftarrow$ Set of negated tags in c;
16. Sort(T^+);
17. $n \leftarrow$ Concatenate items from T^+ with ".";
18. $p \leftarrow$ PTR question for n;
19. Add p to A;
20. foreach t in T^-:
21. $p \leftarrow$ PTR question for n;
22. Add p to A;
23. **end foreach**
24. Send(A);
25. **end foreach**
26. **while** (No appropriate response in E **and not** timeout reached):
27. Receive response e;
28. **foreach** DNS Answer p in e:
29. $Name \leftarrow$ Name(p);
30. $Alias \leftarrow$ Alias(p);
31. **if** ($Alias$ **not in** X):
32. $X_M \leftarrow \{Name\}$;
33. Add $< Alias, X_M >$ to X
34. **else:**
35. $< Alias, X_M >:= X[Alias]$;
36. Add $Name$ to X_M
37. **end if**
38. **foreach** Alias $Alias$ in X:
39. $< Alias, X_M >:= X[Alias]$;
40. **if** ($X_M \cap Satisfy$):Add $Alias$ to E

The name of no record is an acceptable conjunction - they must belong to conjunction(s) with negation!

41. **else**
42. **foreach** c in C^-:
43. $T^+ \leftarrow$ Set of non-negated tags in c;
44. $T^- \leftarrow$ Set of negated tags in c;
45. Sort(T^+);
46. $n \leftarrow$ Concatenate items from T^+ with ".";
47. **if** ($n \in X_M$) **and not** ($X_M \cap T^-$):Add $Alias$ to E
48. **end for**
49. Try selecting the appropriate service(s) from E.
50. **end while**

Responder CreateRecordsForService(Service S, set of tags T)

1. $P \leftarrow$ Empty set of DNS pointer records;
2. $C \leftarrow \mathcal{P}(T)$
3. **foreach** c in C:
4. Sort(c)
5. $n \leftarrow$ Concatenate items from c with ".";
6. **if not** (P contains $p'|p'.Name = n$)
7. $p \leftarrow$ new PTR record;
8. $p.Name \leftarrow n$;
9. $p.Alias \leftarrow S$;
10. Add p to P;
11. **end if**
12. **end for**
13. **return** P

RespondToRequest(DNS request R, Records storage G)

1. $A \leftarrow$ Empty DNS response;
2. **for** DNS question $p \in R$:
3. **if** G contains a record for p:
4. Add $G[p]$ to A;
5. **endif**
6. **endfor**
7. SendBack (A).

The service name may end up in the resulting set for two reasons: first, if it has all tags of some conjunction from the query; second, if it has all non-negated tags from some conjunction and none of negated tags from the same conjunction. This way all of service names in the resulting set satisfy the context query in DNF and, hence, belong to $\sigma(Q)$. Again, to guarantee that query negations are satisfied for all services in the resulting set, we require all answers for one service to be sent in a same DNS response.

Example For nodes the following pointers have to be created in the system:
For common printer:

```
noPaper PTR Printer1._printer._tcp.local
```

For common lazer printer:

```
laser._printer._tcp.local PTR Printer2._printer._tcp.local
noPaper._printer._tcp.local PTR Printer2._printer._tcp.local
laser.noPaper._printer._tcp.local PTR Printer2._printer._tcp.local
```

For Zoe's printer:

```
noPaper._printer._tcp.local PTR PrinterZ._printer._tcp.local
   Zoes._printer._tcp.local PTR PrinterZ._printer._tcp.local
noPaper.Zoes._printer._tcp.local PTR PrinterZ._printer._tcp.local
```

For Zoe's smart light:

```
Zoes._lgt._udp.local PTR SmartL._lgt._udp.local
```

Then to query for services of type `_printer._tcp.local` with satisfying tags, three DNS questions need to be asked: one for a conjunction of `laser` and `noToner`, one for non-negated conjunction part `noPaper` and one for a negated tag `Zoes`. As follows:

```
laser.noToner._printer._tcp.local PTR
      noPaper._printer._tcp.local PTR
         Zoes._printer._tcp.local PTR
```

The responses are filtered and non-satisfying services are filtered out. The activity diagram in Figure 3.4 summarises the discovery process with the **Conjunctions in PTR** scheme.

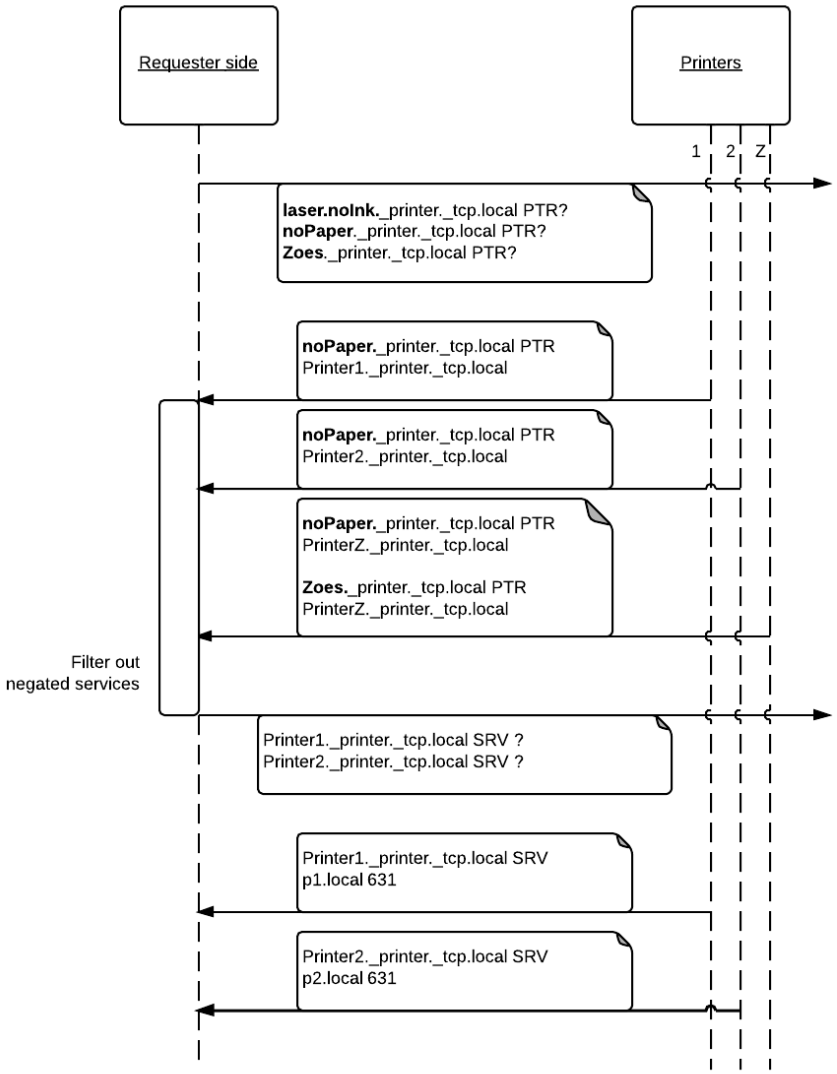


Figure 3.4: The activity diagram for the service discovery process with the Conjunctions in PTR scheme.

3.2.2.4 Nested tag combinations

This naming scheme is very similar to the previous one, but instead of making all combinations of tags point to the service, it creates pointers from less specific tag combinations to the most specific one (the one containing all tags for the service). This most specific combination becomes the name of the PTR record that points to the actual service name. As with **Conjunctions in PTR** scheme, boolean formulas can be expressed with this naming scheme.

One of advantages of this scheme, comparing to the **Conjunctions in PTR** scheme,

is that **Nested tag combinations** can allow to store less DNS records network-wise, i.e. decrease an overall number of stored records in a network. This is because PTR records with commonly used names no longer point to the specific service name, but rather to the most specific tag combination that is resolved to both service names. This technique can be of great use if the resolution with a DNS server is used, as commonly used pointers are stored altogether and do not need to be distributed among devices. In the best case (if all the services have exactly the same combinations of tags) for x services in the system we can decrease the required storage by x times.

Another advantage of this scheme is that it allows a requester to receive a list of all tags for a particular service on the last step of the discovery process. This information can be useful for some applications; also, the knowledge of all tags of a service might cancel the need for a discovery of a service with the same set of tags.

Requests for services are equivalent to those of **Conjunctions in PTR**: each conjunction without negations becomes a PTR question, conjunction with negations is split into a conjunction of non-negated tags and individual negated tags. However, this time the pointer of an answer record may contain either a service name or the conjunction of all tags assigned to a service name. To obtain a service name from this conjunction, the requester has to make a second PTR request for the conjunction. As a result, it will receive a record with the conjunction as a name and a service name as a pointer.

Algorithm

Requester DiscoverServicesNested(Context query Q)

1. $S \leftarrow$ Set of records obtained with **DiscoverServicesNested** algorithm;
2. $A \leftarrow$ Empty DNS request;
3. **foreach** s in S:
4. $\quad p \leftarrow$ PTR question for Alias(s);
5. $\quad$ Add p to A;
6. **end foreach**
7. Send(A);
8. Receive set of services (E).

Responder CreateRecordsForService(Service S, set of tags T)

1. $P \leftarrow$ Empty set of DNS pointer records;
2. $C \leftarrow \mathcal{P}(T)$
3. $T' \leftarrow$ Sort(T);
4. **foreach** c in C:
5. $\quad$ Sort(c)
6. $\quad n \leftarrow$ Concatenate items from c with ".";

7. **if not** (P contains $p'|p'.Name = n$):
8. $p \leftarrow$ new PTR record;
9. **if** ($c = T$):
10. $p.Name \leftarrow n$;
11. $p.Alias \leftarrow S$;
12. **else**:
13. $n' \leftarrow$ Concatenate items from T' with ".";
14. $p.Name \leftarrow n$
15. $p.Alias \leftarrow n'$;
16. **end if**
17. Add p to P;
18. **end if**
19. **end for**
20. **return** P

RespondToRequest(DNS request R, Records storage G)

1. $A \leftarrow$ Empty DNS response;
2. **for** DNS question $p \in R$:
3. **if** G contains a record for p:
4. Add $G[p]$ to A;
5. **endif**
6. **endfor**
7. SendBack (A).

Services from responses are filtered as in the previous naming scheme (3.2.2.3). Again, we require from the responder to place all answers for one service in one response if negations are present in the query.

Example The following pointers have to be present in the system:
For common printer

```
noPaperPTR Printer1._printer._tcp.local
```

For common lazer printer:

```
laser._printer._tcp.local PTR laser.noPaper._printer._tcp.local
noPaper._printer._tcp.local PTR laser.noPaper._printer._tcp.local

laser.noPaper._printer._tcp.local PTR Printer2._printer._tcp.local
```

For Zoe's printer:

```
noPaper._printer._tcp.local PTR noPaper.Zoes._printer._tcp.local
   Zoes._printer._tcp.local PTR noPaper.Zoes._printer._tcp.local

noPaper.Zoes._printer._tcp.local PTR PrinterZ._printer._tcp.local
```

For Zoe's smart light:

```
Zoes._lgt._udp.local PTR SmartL._lgt._udp.local
```

Then to discover services of type `_printer._tcp.local` with a satisfying set of tags, three DNS questions have to be asked, same as those from the previous naming scheme (**Conjunctions in PTR**). The response contains three records. They have to be filtered out to satisfy negations in the query. One of remaining records is an SRV that points to the service name NodeA. The alias of another record is used to make a second PTR record and thus obtain a second service name.

The activity diagram in Figure 3.5 summarises the discovery process with the **Nested tags combinations** scheme.

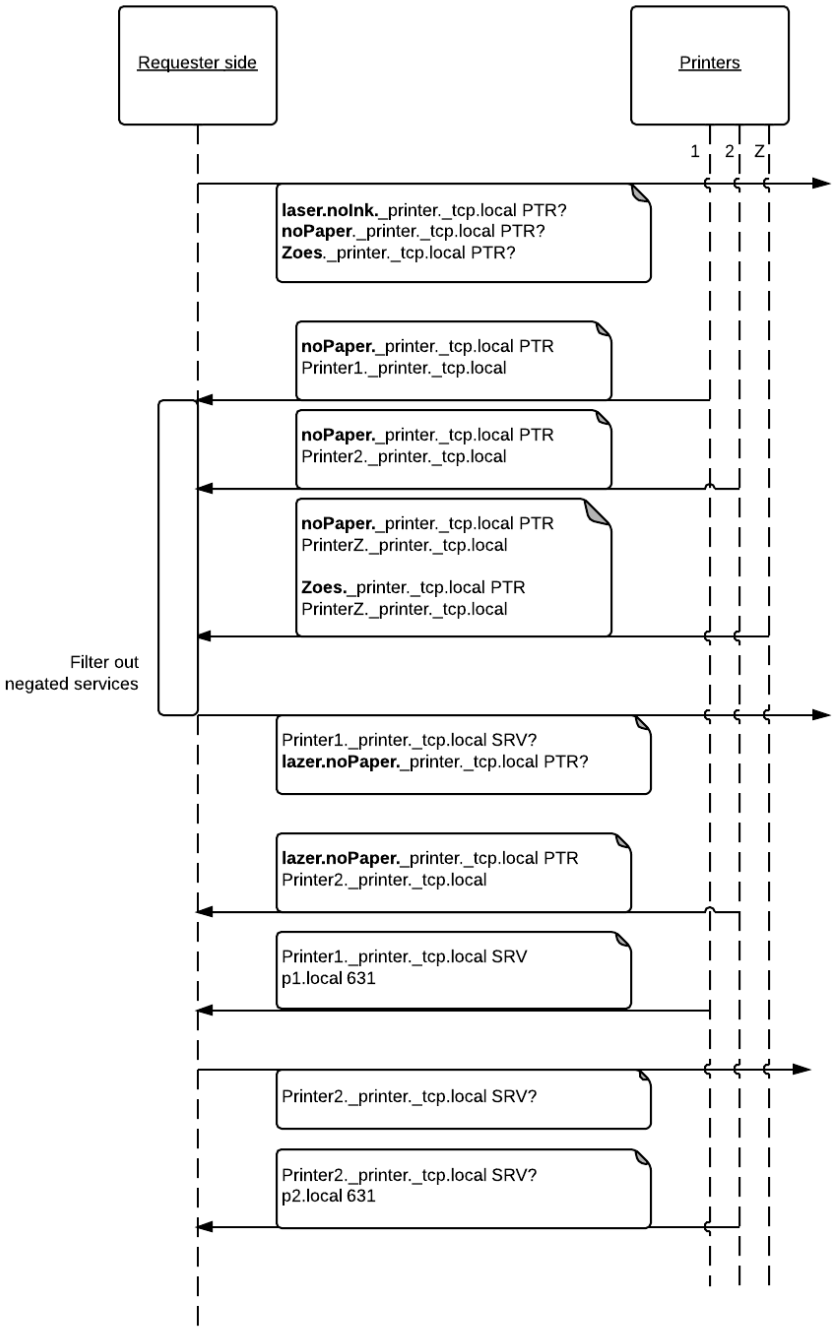


Figure 3.5: The activity diagram for the service discovery process with the Nested tags combinations scheme.

3.2.3 Summary

In this section we presented four naming schemes. Though they serve the same task, the format and number of messages, the necessary calculations and the party responsible for the services selection are very different. For example, **Formula in PTR** creates compact messages, but requires a lot of calculations on the responding side. **Tag to PTR** produces bulky messages and needs to perform client-side post-processing. **Conjunctions in PTR** and **Nested tags combinations** partly express a query in the request and partly check them on the client side, thus balancing between two previous approaches. Differences like these make some of the schemes more useful and effective than others.

In the following section we examine these differences and rank naming schemes according to several important criteria, to select the optimal solution for the given setup.

Chapter 4

Design evaluation

In this chapter we evaluate the naming schemes that are presented and described in the previous chapter. For each requirement, formulated in section 2.4, we calculate values of one or more metrics and compare these values for all naming schemes. We present an overall rating of the solutions and identify the optimal ones.

4.1 Memory

In this section we evaluate naming schemes against the **Memory** requirement. The requirement is to minimise the amount of memory (either primary memory or secondary storage) needed to store the services context. The memory amount required to store context is important for resource-constrained devices that participate in WSNs, as such devices are typically very limited in available memory.

In this section we consider two models for context storage: DNS resource records and a flat list of context tags. While the information about context of services is transmitted between interacting parties in DNS PTR resource records, that does not mean that the context should be locally stored in DNS records as well. However, keeping the context information in such form may be convenient, as it eliminates the need to dynamically construct resource records every time the application needs to make a request or reply to requests from other nodes.

Actual storage formats and structures might vary depending on the used platform, programming language and programmers preference. We abstract from these details by accepting a format that takes into account relevant data fields, such as context tags, for each storage model.

4.1.1 Resource records

In this section we give the number of DNS records required for each naming scheme to encode the service context of one service. We also estimate the size of every individual record. We take n as the number of tags associated with the service, s as a service type

length, m as a service name length and individual lengths of tag names $t_i \in t_1...t_n$.

While the format of DNS record is well specified and described, the internal representation of records may vary. For example, in the mDNS library [36] the type of the record is represented as a number constant, and in JmDNS [33] a special class is created to hold the record type. These design decisions affect the memory size occupied by the program.

To abstract from these implementation details, we assume that DNS records are stored in the format described in the DNS specification [38]. We assume that no names compression is taking place.

In general, a DNS record has the structure shown in the table 4.1. For every record, the amount of bytes required is then:

$$(t_i + 1) * n + 2 * s + m + 13$$

.

Stored data	**Length in bytes**
Set of tags	$(t_i + 1) * n$
Service type	$s + 1$
Type (PTR)	2
Class	2
Time to live	4
Data length	2
Service name and Service Type	$m + s + 2$

Table 4.1: Structure of a DNS record used for expressing the context of a service (without name compression).

Formula in PTR The exact combination of context tags in this naming scheme is not supposed to be stored in DNS records, and the request has to be parsed by the receiving node in order to decide whether the context of the node satisfies the query or not. Hence, no resource records need to be constantly stored. Instead, tags can be stored in another storage structure, for example, a flat list.This kind of storage is discussed further in section 4.1.2.

Tag to PTR For one service with n tags, we need to create n PTR records - one for each tag. Hence, the number of records per service is n. The overall size of DNS records for a service with n context tags is

$$\sum_{i=1}^{n} (t_i + 2 * s + m + 13)$$

bytes.

Conjunctions in PTR For one context tag the situation is trivial and we only need one record.

For 2 context tags we have 3 different combinations of tags.

Say the context tags are `tag_1` and `tag_2`. Then PTRs we will need to store all combinations are:

```
      tag_1._printer._tcp.local PTR Printer._printer._tcp.local
      tag_2._printer._tcp.local PTR Printer._printer._tcp.local
tag_1.tag_2._printer._tcp.local PTR Printer._printer._tcp.local
```

Total of 3 records.

In general, this scheme will require

$$\sum_{k=1}^{n} \binom{n}{k} = \sum_{k=0}^{n} \binom{n}{k} - 1 = 2^n - 1$$

PTR records for one service. Hence the number of records per service is $2^n - 1$.

Every context tag together with a label divisor appears in this set of combinations exactly 2^{n-1} times. Additionally, every record includes a service name and service type. As a result, an overall number of bytes required to store these records is:

$$(2^n - 1) * (2 * s + m + 13) + \sum_{i=1}^{n} ([t_i + 1] * 2^{n-1})$$

Nested tag combinations As for the previous scheme, for each service with n context tags, we need to create at most $2^n - 1$ resource records. These records are almost identical to those generated for **Conjunctions in PTR**, except that PTR records for tag combinations now point to the most descriptive combination of tags, not to the service name. The calculations are the same as for the **Conjunctions in PTR** scheme above. However, now this is the worst case estimation, as depending on the actual tags assigned to services, some combinations of records can be shared among several services (see section 3.2.2.4 for details).

For example, consider two printer services with an identical set of tags: `tag_1` and `tag_2`. The records required to store this information are the following:

```
tag_1.tag_2._printer._tcp.local PTR Printer1._printer._tcp.local
tag_1.tag_2._printer._tcp.local PTR Printer2._printer._tcp.local

tag_1._printer._tcp.localPTR tag_1.tag_2._printer._tcp.local
tag_2._printer._tcp.localPTR tag_1.tag_2._printer._tcp.local
```

This gives a total of 4 records. To store the same information for two services in **Conjunctions in PTR** scheme, $2 * (2^2 - 1) = 6$ records have to be created.

The actual number of stored records in this scheme depends on the number of tags and their mapping to services. In this work, we rather consider a worst case: when the number of records is $2^n - 1$.

Every context tag together with a label divisor appears in the name of these records exactly 2^{n-1} times. Hence, the whole amount of memory taken by the context tags is $\sum_{i=1}^{n}([t_i+1]*2^{n-1})$. Only one record includes a service name, which takes $2*s+m+13$ bytes of space. The length of the other 2^n-2 records depends on the length of the most descriptive tags combination. To address these records we multiply 2^n-2 to the sum of all the lengths of tags, service type with label divisor and 10 bytes of DNS fields, which results in $(2^n-2)*(\sum_{i=1}^{n}[t_i+1]+s+11)$.

Therefore, the overall amount of bytes required to store the context for a service, is:

$$\sum_{i=1}^{n}([t_i+1]*2^{n-1})+(2^n-2)*(\sum_{i=1}^{n}[t_i+1]+s+11)+2*s+m+13$$

4.1.2 Dynamic PTR creation

The number of records required for the **Nested tag combinations** and the **Conjunctions in PTR** naming schemes can be very large even with moderate number of context tags. For example, for just 8 tags these schemes would require $2^8-1=255$ resource records! Along with storage issues, it can be impossible to announce all these PTRs on the device startup with a single DNS message.

To deal with these issues we can apply Dynamic PTR creation technique. That is, we store all context tags as a flat list and dynamically create and send PTR records with specific tags combinations as they are being requested. Of course, this will require writing some supporting code.

The size of the array list with context tags is then just a sum of their lengths, plus overhead for storing a data structure. As a result, the total storage required is

$$\sum_{i=1}^{n}t_i+h$$

bytes, where h is the storage structure overhead.

This optimisation can be applied to all naming schemes, except for the **Formula in PTR**, which, by default, processes requests dynamically.

4.1.3 Comparison with DNS-SD

In classic DNS-SD without enhancements, the context can still be associated with a service. Context tags can be stored in a TXT record with its name equal to a service name. For example, context tags may be stored in a TXT entry with a key named "tags", and a string containing comma-separated tags as a value:

```
Printer._printer._tcp.local TXT tags=tag_1,tag_2,tag3
```

.

This way of storage would take $\sum_{i=1}^{n}t_i+1-1+10$ bytes, where 10 bytes are required to store DNS information about a record, and -1 byte is required because in fact the number

of divisors is 1 less than the number of tags. This is still a bit worse than the flat list with a constant storage overhead, as it requires to store 1 extra byte per context tag.

4.1.4 Summary

When context information is stored in DNS records, **Conjunctions in PTR** and **Nested tags combinations** show exponential growth of the space required to store them. The **Tag to Pointer** scheme, however, only shows linear growth of memory requirements. These dynamics are shown on the Figure 4.1.

The figure presents the amount of memory that is required to store DNS records with context information for different naming schemes. The service name length is 5 bytes, service type length is 16 bytes, and each context tag is 5 bytes long. The overhead for storage structure is taken as zero.

If context tags are stored as a flat list, then requirements for storage do not differ between naming schemes. This technique is also the most beneficial in terms of occupied memory. Even the **Tag to PTR** scheme adds up $t_i + s + m + 13$ bytes for every tag length of t_i, while it only takes t_i more bytes to store the same tag in the list.

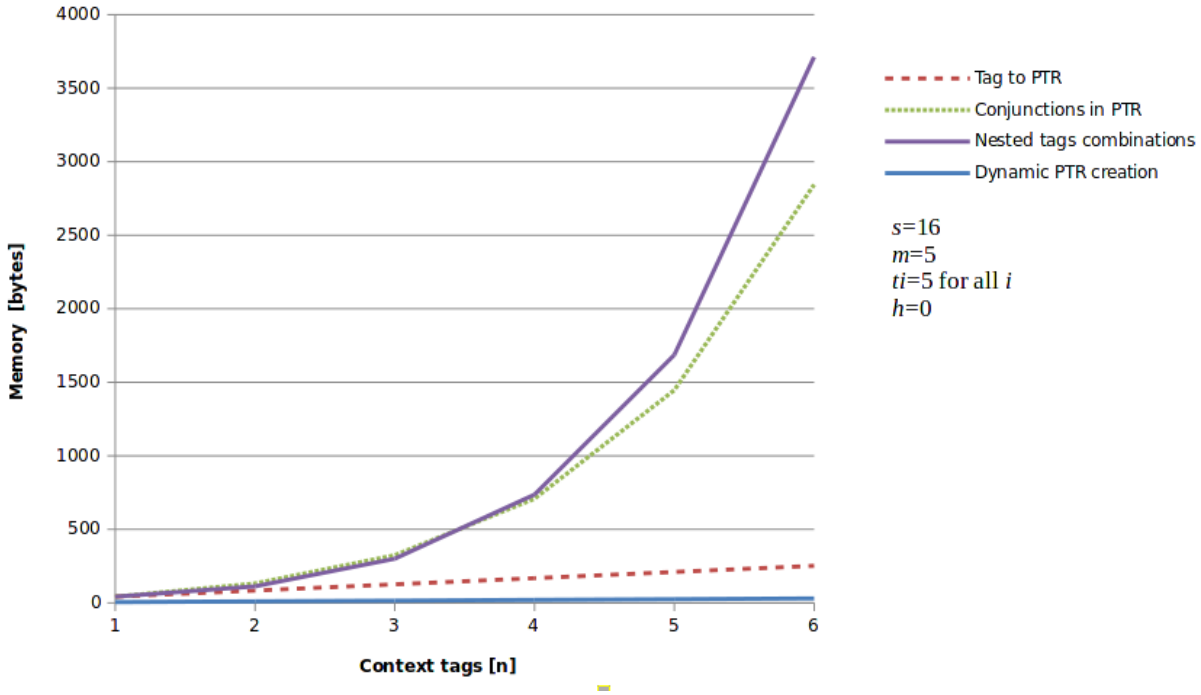


Figure 4.1: Worst-case amount of memory required for DNS records with different naming schemes.

Table 4.2 presents the ranking of the naming schemes by the memory criterion. It is assumed that schemes with Dynamic PTR optimisation do not store DNS records. This fact makes dynamic schemes the most memory-efficient ones, because the flat list requires

the least amount of memory. The second best scheme is **Tag to PTR**, as it only creates one record for each assigned tag. Finally, **Conjunctions in PTR** and **Nested tags in PTR** are the worst, as they require a large amount of records with tags combinations to be stored. The latter one additionally stores the most descriptive tags combination in every record, which contributes to the taken space as well.

Rank	Naming scheme
I	Formula in PTR
I	Naming schemes with Dynamic PTR optimisation
II	Unmodified DNS-SD
III	Tag to PTR
IV	Conjunctions in PTR
V	Nested tags in PTR

Table 4.2: Ranking of different naming scheme by the worst-case memory usage criterion. Smaller rank is better.

4.2 Operations on tags

In this section the support of different operations is discussed. The corresponding requirement is to provide rich query operations.

In the section 3.1.1 we established that context queries can be effectively expressed with the boolean algebra. This algebra contains three basic operations: AND, OR and NOT. To express any boolean formula, the naming scheme should support these operations.

The basic operators are, indeed, supported by all naming schemes. However, there are differences in the negation processing. Some of the schemes require that in order to process negations correctly, all records for one service should arrive in one DNS response. Other schemes do not have this requirement.

The only tags-related feature that is not available for all naming schemes is receiving the list of context tags for a service. This feature is only provided by the **Nested tags combinations** scheme. The **Tag to PTR** scheme also allows to retrieve a set of tags that correspond to a service name, but this set is limited with the set of tags in the context query. In other words, if the tag is not in a context query, it will not appear in the list of service tags.

Further we briefly list specifics of operation support for each naming scheme.

Formula in PTR The requester of this naming scheme sends a boolean formula of a context query to the responder, which resolves it; therefore, every response received by the requester contains a satisfying service. There is no way of obtaining a list of services tags.

Tag to PTR In this naming scheme the requester creates a PTR question for every tag in the formula, obtaining the list of services with relevant tags. The client then filters received service names according to their tags and the context query. To guarantee that this process returns correct results, it is required to deliver all records for a service in one DNS response.

Conjunctions in PTR In this naming scheme the requester creates a PTR question for every conjunction in the context query formula. Unless this formula consists of a single conjunction, there is no way to receive a list of tags for a service. The scheme also requires that all records for a service should be delivered in one DNS response for guaranteed negations satisfaction.

Nested tags combinations As in the previous scheme, the requester creates a PTR question for every conjunction in the context query formula. However, this scheme delivers the whole set of context tags that are assigned to each returned service. This list also allows to justify if the service satisfies the query without a need to deliver all records for a service in one response.

4.2.1 Comparison with DNS-SD

All context tags stored in one TXT record. The client has to query for this record to obtain the context of the service. The client always receives a full list of tags, which can be used to reliably check if the service satisfies a context query, including queries with negations.

4.2.2 Summary

Table 4.3 summarizes the subsection and presents the capabilities of different schemes in expressing context requirements.

As it can be seen from the table, the **Nested tags combinations** provides with the most complete set of available operations.

Naming sceme	Boolean operators	**Query for service tags**	**Negation with separate responses**
Formula in PTR	+	−	+
Tag to PTR	+	+	−
Conjunctions in PTR	+	−	−
Nested tags combination	+	+	+
Unmodified DNS-SD	+	+	+

Table 4.3: Operations on tags supported by different naming schemes.

Table 4.4 presents the ranking of naming schemes by the memory criterion. The best naming scheme is the **Nested tags in PTR**, as it allows to retrieve a full list of tags

for each service. The last place is taken by **Conjunctions in PTR** that cannot provide with the ability to return full services context, and cannot guarantee to process negation correctly if services records arrive in different responses.

Rank	Naming scheme
I	Unmodified DNS-SD
I	Nested tags in PTR
II	Formula in PTR
II	Tag to PTR
III	Conjunctions in PTR

Table 4.4: Ranking of different naming scheme by the available operations criterion. Smaller rank is better.

4.3 Network

This subsection is devoted to evaluating the network load minimisation requirement. For wireless nodes, the network communication is one of main factors that contribute to the energy usage. Minimising the network load is therefore expected to prolong a battery life of sensors.

The task of estimating the network load caused by service discovery is not easy to solve, because there is no clear correlation between instances described in the DNS-SD protocol and the network load. Besides, there are differences in DNS and mDNS networking, that can also affect the load.

In this section we use several metrics to estimate the network load for each naming scheme. First, we calculate the length of one DNS service discovery message depending on requested context and message structure. Second, we estimate the number of produced DNS messages depending on the number of services satisfying a context query. Third, we derive the number of network packets per one service discovery query. Finally, we compare the number of DNS messages produced by DNS-SD protocol with and without the usage of context tags.

4.3.1 Assumptions

Networking is a complicated area, due to a big number of possible factors that influence data transmission and network load. For example, the actual load of the network largely depends on the network topology, tags distribution over services in the network, context query, and selected naming scheme. To simplify the analysis, we adopt several assumptions, which are listed further.

4.3.1.1 Calculation of traffic generated on nodes

Networks have varying topologies, which influences the network load and throughput. For solutions with a centralised DNS server, it also matters where in the network topology such a server is located. The topology can be influenced and defined by a specific usage scenario, and the number of possibilities to analyse can be very large. Therefore, we abstract from these details. In our analysis we calculate the number and size of requests and responses generated by the nodes of a network, and take them as metrics of network load. Our assumption is that regardless of exact topology or network conditions, the less data to transmit the better.

4.3.1.2 No negation in queries

Negation operation is problematic for all schemes except for **Formula in PTR**. Even with a requirement of all answers for a service arriving as a single response, there might be a situation when the information about negated tags is lost. For example, depending on the protocol, if the DNS message is broken into several packets, and one of those packets is missing, such a message can still be accepted as a valid one. If there is information about the negated tags in the missing part, the service can be falsely qualified as satisfying. Hence, the introduction of negation does not allow the algorithm to guarantee the returning of only correct services.

Another problem is services that do not satisfy the formula but still answer the request to be filtered on the client side later. It is not easy to estimate the number of such services and to present them in terms of a context query. The number of such services also depends on the distribution of tags over services.

Finally, it might be conceptually better to ask for services that have some property, rather than for those that do not. For example, there is little use to ask for every device that is not "blue".

To summarise, the negation operation is hard to validate, it introduces excessive complexity and analysis and conceptually arguable. With this in mind, we do not consider this operation in further analysis.

4.3.1.3 One service type per request

Though technically the requester can produce a DNS request with two DNS-SD questions for different service types, we do not consider such requests here. One reason is to simplify the analysis. Another reason is to examine the network load for a single context query, which implies a single service type. Finally, this assumption makes it easier to compare naming schemes with classical DNS-SD.

4.3.2 Network load with naming schemes

4.3.2.1 UDP datagram size

Presented below are calculations for the size of DNS requests and responses, required to discover services, in bytes. The result is a formula that solves to the size of an actual DNS questions or answers section, given the lengths of every individual tag, services name(s) and service type. To obtain the whole length of a UDP datagram for the operation, one should add 8 bytes of a UDP header [39] and 12 bytes of a DNS header [38]. The formulas were checked against the lengths of actual packets produced by the demo application (see Section 5 for the application details) and logged with the Wireshark tool [40]. The traces of these packets are presented on figure 4.2.

DNS request First we list items that appear in a DNS question with n tag names, service type length of s and individual lengths of tag names $t_i \in t_1...t_n$. No service was resolved at this point yet, and no service name appears in any of the DNS questions. To address requests with several DNS questions, we introduce several more variables. The length of service type is denoted as s. The variable q represents the number of DNS questions in the request. Each tag name is accompanied by the divisor byte that separates parts of the domain name and stores the length of the tag. This is represented by adding one byte to the tag name in the list and formula.

The components that contribute to the length of DNS questions are presented in Table 4.5. For every table row, the left cell is a description of the data stored, and the right one is the length of this data. Due to the name compression mechanism, some of the data may become compressed and shrink in length. If the data can be compressed, the additional cell with the length after compression is added to the right part of the row.

Data	**Length in bytes**	
List of tags	$(t_i + 1) * n$	
Service type	$s + 1$	2
Type (PTR)	2	
Class	2	

Table 4.5: The contents of DNS question used for context-aware service discovery.

The total length of one question without name compression is $\sum_{i=1}^{n} (t_i + 1) + s + 5$ bytes. If there are 2 or more questions for the same service type in the request, for subsequent questions the compression of the service type takes place. This results in the total of $\sum_{i=1}^{n} (t_i + 1) + 6$ bytes.

Now that we have the sizes of the individual DNS questions of different types, we can calculate sizes of context requests for all naming schemes. Note that we cannot give the precise value for complex requests of several questions, because depending on the context tags addressed, the name compression will or will not take place, influencing the size of

the question. Therefore, here we give only the upper bound for complex several-question requests.

In the general case, the request includes all context tags with their divisors ($\sum_{i=1}^{n}(t_i+1)$), a service type with divisor of total length $s+1$ bytes, and 4 bytes of DNS overhead for each question ($q*4$). Additionally, all questions but one contain the pointer to the service type of length 2 bytes. To keep the formula simple, we add these bytes to the DNS overhead and subtract 2 bytes for the first question from s. The resulting formula is presented in Formula 4.1.

$$\sum_{i=1}^{n}(t_i+1)+s-1+q*6 \tag{4.1}$$

With this formula, a length of request for any naming scheme and any context query can be calculated. While a size of one DNS question depends on the context tags and service type, the size of the whole request is influenced by the structure of the request, which is defined by the naming scheme.

Formula in PTR This naming scheme transports the whole formula in one question, hence minimising the transporting overhead.

Tag to PTR This naming scheme creates a separate question for each tag in a formula. An additional overhead of 4 bytes is required to transport each additional question; hence, an additional overhead of $4*n$ bytes is generated, where n is the number of tags in a formula.

Conjunctions in PTR This naming scheme creates a separate question for each conjunction in a formula. Hence, an additional overhead of $4*x$ bytes is generated, where x is the number of conjunctions in a formula.

Nested tags combinations Requests for this naming scheme are equivalent to those of **Conjunctions in PTR**.

It can be seen that the **Formula in PTR** scheme minimises an overhead to request for the same context query. **Conjunctions in PTR** can be as good as **Formula in PTR**, if the query contains the only conjunction, or it can be worse. **Conjunctions in PTR** can in principle be worse than **Tag to PTR**, as the number of answerable questions that it can produce from the redundant query with the same set of tags is up to 2^n-1, against n possible requests for **Tag to PTR**. Even if we assume that the context query does not contain redundant conjunctions (e.g. `tag_1` and `tag_1` $\wedge$ `tag_2` at the same time), in the worst case the maximum number of conjunctions is $\binom{n}{\lfloor \frac{n}{2} \rfloor}$ (see Appendix A for the proof). Finally, **Nested tags combinations** is equivalent to **Conjunctons in PTR**.

DNS response As before, we begin by defining variables for entities that appear in the response. For a response with a total of n tag name occurrences, individual lengths of tag names are denoted as $t_i \in t_1...t_n$. The length of a service type is denoted as s. For each service type the response may contain one or more service names. Service names are denoted as $m_x \in m_1...m_y$, where y is the total number of service names in the response. For each of these names, one or more DNS answers may be included. The number of answers for a service name m_x is denoted as q_x.

The **Nested tags in PTR** naming scheme operates with responses that do not contain the service name, but rather the most specific tags combination. In this case, lengths of tags with divisors in the pointer alias are just added to the lengths of tags in the pointer name, and the length of the service name is set to zero.

Each tag name is accompanied by the divisor byte that separates parts of the domain name and stores the length of the tag. This is represented by adding one byte to the tag name in the list and formula.

Data	**Length in bytes**	
Set of tags	$(t_i + 1) * n$	2
Service type	$s + 1$	2
Type (PTR)	2	
Class	2	
Time to live	4	
Data length	2	
Service name and pointer to the Service Type	$m + 1 + 2$	2

Table 4.6: Structure of a DNS answer used for context-aware service discovery.

There is more variance in the DNS answer size than in the question size, because the set of tags, service type and service name can become subjects to name compression. It might happen if there are other answers with the same values in the DNS responce. Possible alternatives are briefly listed here:

- If there are 2 or more answers for the same **service type** in the response, the compression of the Service type field takes place. The total size of the answer is $\sum_{i=1}^{n}(t_i + 1) + m + 14$ bytes;
- If there is more than one answer for the same **service** in the response, the compression of both service name and service type takes place. This results in a total of $\sum_{i=1}^{n}(t_i + 1) + 14$ bytes;
- If there are 2 or more answers for the same **service type with the same set of tags** in the response, the compression of the tags sequence and service type fields takes place. The result is that the compressed answers have a total of $15 + m$ bytes length;

- Finally, if there are several answers for the same **expression on tags, service and service type**, the length of an answer is just 14 bytes.

In the general case, the response includes a set of tag names with their divisors ($\sum_{i=1}^{n}(t_i+1)$), a name of a service type with a divisor ($s+1$), a set of service names with their divisors and 2-byte pointers to their service type ($\sum_{x=1}^{y}(m_x+3)$), and each answer contains a DNS protocol information of size 10 bytes ($\sum_{x=1}^{y}(q_x*10)$). Additionally, for every service name all answers but one include the pointer to the service name, and for every service type all answers but one include the pointer to the service type. To keep the formula simple, we add these 4 bytes to the 10 bytes of DNS overhead for every record, and later subtract 2 bytes from s and m_x. The resulting summation is represented by Formula 4.2.

$$s - 1 + \sum_{i=1}^{n}(t_i + 1) + \sum_{x=1}^{y}(m_x + 1 + q_x * 14) \tag{4.2}$$

As before, this formula can be used to describe responses for every naming scheme. The difference between schemes is in the structure of a response. However, this time it is not clear how many records the response contains, as it depends on the size of each record and DNS responder implementation. It is also hard to compare responses of naming schemes, because for some naming schemes all tags from the request will repeat in the response, and for others just a subset of these tags will appear in the response, and this subset depends on the actual assignment of tags to services in a network.

As the size of a response depends mostly on the context query and the assignment of tags to services, we do not analyse the link between the chosen naming scheme and the size of a response.

Example Here we check our calculations against three kinds of context query: a disjunction of multiple tags, a single conjunction of same tags and a complex request, containing both operations.

Say we have 2 services, `SmartL._lgt._udp.local` with tag `tag_1` and `nodeB2._lgt._udp.local` with tags `tag_1`, `tag_2`. We first request for services that have `tag_1` OR `tag_2`, then for those that have both tags. Finally, we request for services that have either both of these tags, or `tag_3`. We have actual requests produced by the application discussed in the section 5, using **Conjunctions in PTR** naming scheme. Network traces of resulting requests and responses can be found in Figure 4.2.

Disjunction request Let us consider the disjunction request. For two tags, two DNS questions are created:

```
tag_1._lgt._udp.local.  PTR
tag_2._lgt._udp.local.  PTR
```

The variables values are: $t_1 = t_2 = 5, s = 16$.

These 2 questions take 39 bytes (see the network trace for union request in Figure 4.2a). The size value calculated with a formula is

$$\sum_{i=1}^{n}(t_i + 1) + s - 1 + q * 6 = 2 * 6 + 15 + 12 = 24 + 15 = 39.$$

The calculated length is the same as for the actual request.

Disjunction response As a response, three records are sent, two for each of the services.

```
tag_1._lgt._udp.local PTR NodeB._lgt._udp.local
tag_1._lgt._udp.local PTR NodeB2._lgt._udp.local
tag_2._lgt._udp.local PTR NodeB2._lgt._udp.local
```

The variables values are: $t_1 = t_2 = t_3 = t_4 = 5, s = 16, m_1 = 5, m_2 = 6$. Note that we count all occurrences of context tags, even if some of them repeat each other.

The request results in a total of 82 bytes in the network trace (Figure 4.2b). The value calculated with the formula is only an upper bound and not an exact value. This upper bound is:

$$\begin{gathered} s_a - 1 + \sum_{i=1}^{n}(t_i + 1) + \sum_{x=1}^{y}(m_x + 1 + q_x * 14) = \\ 15 + 3 * 6 + (6 + 1 * 14) + (7 + 2 * 14) \\ = 15 + 18 + 6 + 14 + 7 + 28 = 88 \end{gathered}$$

bytes.

As $88 \geq 82$, we state that our formula indeed provides an upper bound for this union response length. Let us now take a closer look at the name compression that causes the imprecision of the calculations. In this example, the `tag_2` name gets compressed in one record. So, instead of the tag name with the divisor and pointer to the service type, one pointer to the whole name gets inserted. We can express it by adding $2 - (t_i + 1 + 2)$ to our calculations. As a result, we get $88 + 2 - 8 = 82$ bytes, which is the exact length of the example data.

Conjunction request For the conjunction request, one question containing two tags is created.

```
tag_1.tag_2._lgt._udp.local.  PTR
```

The variables values are: $t_1 = t_2 = 5, s = 16$.

The total length of the network trace (Figure 4.2c) is 33 bytes.

The value in bytes calculated with formula is:

$$\sum_{i=1}^{n}(t_i + 1) + s - 1 + q * 6 = \sum_{i=1}^{2}(5 + 1) + 15 + 6 = 12 + 21 = 33.$$

The formula produces the same value as the actual data takes.

Conjunction response For this conjunction response, one record is returned.

```
tag_1.tag_2._lgt._udp.local.  PTR NodeB2._lgt._udp.local.
```

Variables values are: $t_1 = t_2 = 5, s = 16, m_1 = 6$
Total size of network trace (Figure 4.2d) is 48 bytes.
Using the formula, we obtain a value of

$$s_a - 1 + \sum_{i=1}^{n}(t_i + 1) + \sum_{x=1}^{y}(m_x + 1 + q_x * 14) = 15 + 2 * 6 + 7 + 14 = 48$$

bytes, the same value as observed.

Request and response calculations and data lengths for **Formula in PTR** scheme are identical to those of Conjunction requests and responses.

Complex request Let us consider a request for a context query, expressed by the following boolean formula: $(tag_1 \wedge tag_2) \vee tag_3$.

Two DNS questions are created by the client:

```
tag_1.tag_2._lgt._udp.local PTR
     tag_3._lgt._udp.local PTR
```

The variables values are: $t_1 = t_2 = t_3 = 5, s = 16$.

The size of the network request (Figure 4.2e) is 45 bytes. All tags are found in the request only once, hence no name compression for tags is taking place, and the generic formula for requests should result in the exact value.

$$\sum_{i=1}^{n}(t_i + 1) + s - 1 + q * 6 = 3 * 6 + 15 + 12 = 18 + 15 + 12 = 45$$

Complex response The previous request results in discovery of the service, that satisfies the $(tag_1 \wedge tag_2)$ part of the formula. This is represented by the following record in the response:

```
tag_1.tag_2._lgt._udp.local PTR NodeB2._lgt._udp.local
```

The variables values are: $t_1 = t_2 = 5, s = 16, m = 6$.

The answers section shown in Figure 4.2f is 48 bytes long. As before, no name compression is taking place, hence the general formula should produce exactly the same result.

$$s - 1 + \sum_{i=1}^{n}(t_i + 1) + \sum_{x=1}^{y}(m_x + 1 + q_x * 14) =$$

$$15 + 2 * 6 + 6 + 1 + 14 = 30 + 12 + 6 = 48$$

```
0000   05 74 61 67 5f 31 04 5f 6c 67 74 04 5f 75 64 70  .tag_1._lgt._udp
0010   05 6c 6f 63 61 6c 00 00 0c 00 ff 05 74 61 67 5f  .local......tag_
0020   32 c0 12 00 0c 00 ff                             2......
```

(a) The network trace of the disjunction request with 2 tags, 39 bytes.

```
0000   05 74 61 67 5f 31 04 5f 6c 67 74 04 5f 75 64 70  .tag_1._lgt._udp
0010   05 6c 6f 63 61 6c 00 00 0c 00 ff 00 00 0e 0e 00  .local..........
0020   08 05 4e 6f 64 65 42 c0 12 c0 0c 00 0c 00 ff 00  ..NodeB.........
0030   00 0e 0d 00 09 06 4e 6f 64 65 42 32 c0 12 05 74  ......NodeB2...t
0040   61 67 5f 32 c0 12 00 0c 00 ff 00 00 0e 0e 00 02  ag_2............
0050   c0 41                                            .A
```

(b) The network trace of disjunction response with 3 records, 82 bytes.

```
0000   05 74 61 67 5f 31 05 74 61 67 5f 32 04 5f 6c 67  .tag_1.tag_2._lg
0010   74 04 5f 75 64 70 05 6c 6f 63 61 6c 00 00 0c 00  t._udp.local....
0020   ff                                               .
```

(c) The network trace of conjunction request with 2 tags, 33 bytes.

```
0000   05 74 61 67 5f 31 05 74 61 67 5f 32 04 5f 6c 67  .tag_1.tag_2._lg
0010   74 04 5f 75 64 70 05 6c 6f 63 61 6c 00 00 0c 00  t._udp.local....
0020   ff 00 00 0e 0f 00 09 06 4e 6f 64 65 42 32 c0 18  ........NodeB2..
```

(d) The network trace of conjunction response with 1 record, 48 bytes.

```
0000   05 74 61 67 5f 31 05 74 61 67 5f 32 04 5f 6c 67  .tag_1.tag_2._lg
0010   74 04 5f 75 64 70 05 6c 6f 63 61 6c 00 00 0c 00  t._udp.local....
0020   ff 05 74 61 67 5f 33 c0 18 00 0c 00 ff           ..tag_3......
```

(e) The network trace of complex request with 3 tags, 45 bytes.

```
0000   05 74 61 67 5f 31 05 74 61 67 5f 32 04 5f 6c 67  .tag_1.tag_2._lg
0010   74 04 5f 75 64 70 05 6c 6f 63 61 6c 00 00 0c 00  t._udp.local....
0020   ff 00 00 0e 0f 00 09 06 4e 6f 64 65 42 32 c0 18  ........NodeB2..
```

(f) The network trace of the response on complex request, 2 tags, 48 bytes.

Figure 4.2: Network traces of requests and responses used in to evaluate packet length formulas.

4.3.2.2 Number of mDNS messages

Here we calculate the number of mDNS requests and responses that need to be sent by devices in a network in order to discover services. Different naming schemes produce different numbers of such messages. While the number of network packets required for the operation cannot be directly derived from the number of generated DNS messages, minimising the number of DNS messages gives a strong indication that a network load will be reduced as well.

Unless otherwise specified, we assume that every DNS answer for a service can be distributed in a separate DNS response. This gives us a worst-case estimation that is independent of the actual DNS responder implementation.

We use: the Q variable to address the context query being transmitted as one or more DNS questions, σ as the function that maps queries (Q) to services (see section 3.1.1), U as a set of services of a requested type in a network, $c_1...c_x \in C$ is a set of conjunctions in a formula, and T as a set of tags in a context query.

Formula in PTR One DNS request with the boolean formula is sent and $|\sigma(Q)|$ perfectly satisfying responses are received. Hence the total number of DNS messages transmitted is $1 + |\sigma(Q)|$. In the worst case this number is equal to $1 + |U|$.

Tag to PTR The formula evaluation for this scheme is performed on the client side. One DNS question is placed in the request for each required tag, and for each of these questions the client may receive one or more answer records.

Let T be the set of tags the client queries for. If mDNS is used, the DNS response is generated for each service that has at least one of those tags. Hence the number of messages transmitted is $1 + |\sigma(t_1)| + |\sigma(t_2)| + ... + |\sigma(t_n)|$ for $t_1...t_n \in T$. In the worst case each service in a network has each tag from a query, and hence the number of messages is equal to $1 + |T| * |U|$.

Conjunctions in PTR One DNS request is composed for a context query. Then the requester may receive one or more responses of different structure. An overall number of DNS records per service discovery depends on the number of conjunctions in a query. Suppose there are x conjunctions, then i-th tag in a j-th conjunction is denoted as t_i^j. The number of responses to a context query is then $\sum_{j=1}^{x} | \cap t_i \in c_j|$. In the worst case each service in a network has each conjunction from a query, and hence the number of messages is equal to $1 + |C| * |U|$.

Nested tag combinations The requester starts by requesting a pointer for every conjunction. As a response it gets zero or more descriptive pointers, or service names. In the worst case, none of the returned names is a service name, so the requester needs to query for them one more time to get pointers to the actual services. Due to a non-deterministic nature of the network, it will result in a separate request for each arrived response. The

reason is, the requester can never tell if all of responses arrived, and hence it can never decide that the time has come to compose a second request for all the services. The worst-case overall number of requests is thus $1 + 3 * (\sum_{j=1}^{x} | \cap t_i \in c_j|)$. In the worst case this number is equal to $1 + 3 * |C| * |U|$.

We can, however, apply an optimization on the responder side and send out the service name as an answer to the initial request together with the most descriptive name. In that case, we need to request for the most-descriptive names one less time. However, this also means that the non-satisfying services will send one more set of records, as the filtering happens on the client side after receiving the responses. Then the overall number of messages will be $1 + 2 * |C| * |U|$.

Comparison It is easy to notice that each naming scheme introduces its own multiplier to the worst-case formula for the number of messages. For **Formula in PTR** the number of services in a network is multiplied by 1; for **Tag to PTR** the multiplier is $|T|$, for **Conjunctions in PTR** it is $|C|$, and for **Nested tags combinations** it is $3 * |C|$. Here we try to compare these numbers to determine which naming scheme produces less network messages.

First of all, we notice that **Nested tags combinations** will always be worse than **Conjunctions in PTR**, because $3 * |C| \geq |C|$ and $|C| \geq 1$. **Conjunctions in PTR** can be as good or worse than **Formula in PTR**, as $|C| \geq 1$. Same holds for **Tag to PTR**, as $|T| \geq 1$.

Again, **Conjunctions in PTR** can in principle be worse than **Tag to PTR**. If we assume that the context query does not contain redundant conjunctions, then in the worst case **Conjunctions in PTR** contains at most $\binom{n}{\lceil \frac{n}{2} \rceil}$ conjunctions and is comparable to **Tag to PTR**.

Finally, **Tag to PTR** is better than **Nested tags combinations** when $|T| < 3 * |C|$, and worse when it is otherwise.

4.3.2.3 Number of DNS messages

In the case of a setup with a dedicated DNS server, DNS requests do not change. However, in this case one DNS response may contain the data of several services. The questions are, how many services can such a response contain and how many responses does it take to return service names?

The answer to these questions does not depend solely on the structure and components of the messages. The responder faces a problem of distributing resource records of different sizes into DNS responses, that, without additional optimisations and add-ons, have a maximum size of 255 bytes. The actual number of DNS responses depends on the algorithm that is used to solve this problem.

The study of such an algorithm is out of the scope of this work. Here we only establish worst-case estimations and a lower bound for an optimal number of messages. In the worst case for every naming scheme, only one record fits the response. This is possible because every domain name can be up to 127 bytes long, and some additional information should

also be included in a response. Hence, in the worst case the number of DNS responses is equal to the number of mDNS responses.

The lower bound for an optimal solution is defined as an even distribution of all bytes of a response over several 255-byte-long packets. The size of a response can be calculated with Formula 4.2. Note that every DNS response should include a 12-byte header. Also, every DNS response has to specify a service type exactly once, as all other occurrences of a service type will be compressed. Hence, the lower bound for an optimal number of DNS responses is calculated as follows:

$$\frac{[s - 1 + \sum_{i=1}^{n}(t_i + 1) + \sum_{x=1}^{y}(m_x + 1 + q_x * 14)] - s}{243 - s} \tag{4.3}$$

The value of this formula does not strictly depend on the naming scheme, but rather on the actual tags, service types and service names.

4.3.3 Comparison with DNS-SD

4.3.3.1 UDP datagram size

The size of a DNS-SD request and response can be estimated with formulas 4.1 and 4.2, omitting the missing fields. Such fields are, namely, context tags. Hence, a DNS-SD request has a size of $s-1+q*6$ bytes, and a DNS response - a size of $s-1+\sum_{x=1}^{y}(m_x+1+q_x*14)$ bytes. Note that for each service we additionally have to transmit a TXT record with the service context, which takes $\sum_{i=1}^{n} t_i + 1 - 1 + 10$ bytes, for n tags assigned to a service, $t_i \in t_1...t_n$.

DNS-SD request will typically be smaller than one of naming schemes. A DNS-SD request only contains one question with no tags; hence, it will be smaller than a request of **Formula in PTR** by $\sum_{i=1}^{n}(t_i + 1)$ bytes, and hence smaller than a request of any other naming schemes, that may introduce additional overhead.

The relation between sizes of responses is, again, hard to estimate because of its strong connection with context query and actual assignment of tags.

4.3.3.2 Number of mDNS messages

In this subsection we estimate the gain of our approach compared to the classical DNS-SD by comparing the number of produced DNS responses. We assume that each unique record is sent only once and do not take DNS records caching into account.

First of all, we note that both for naming schemes and for classical DNS-SD context requests contain the service type part. Therefore, any request will be answered only with services of the corresponding type.

Let U be the set of all services of a type s, Q is a context query expressed with boolean formula on context tags, $\sigma(Q) \subseteq U$ is a set of services that satisfy the context query. By definition, $|\sigma(Q)| \leq |U|$. Let us assume that the application is only interested in all services from $\sigma(Q)$. In order to obtain a list of services, a DNS request must be made.

Let the application make a request, specifying the required service type s. For classical DNS-SD, descriptions of all the services in U will be sent back. If TXT records containing the services context are sent together with pointers to service names, the resulting number of records sent as a response is $|U|$. If not, the application has to query for context of all services separately, creating $|U|$ requests and $|U|$ responses. Therefore, DNS-SD can generate either $|U|$, or $3 * |U|$ responses.

Now let us consider the numbers of responses for different naming schemes. These numbers have been calculated in previous subsections. The **Formula in PTR** scheme requires $1 + |\sigma(Q)|$ response records to be sent.
For **Tag to PTR** it is required to transmit $|\sigma(t_1)| + |\sigma(t_2)| + ... + |\sigma(t_n)|$ responses for $t_1...t_n \in T$. At most this number can become as large as $|T| * |U|$.

The **Conjunctions in PTR** scheme requires $|\sigma(c_1)| + |\sigma(c_2)| + ... + |\sigma(c_x)|$ response records to be sent, and at most this number can become as large as $|C| * |U|$.

Finally, the **Nested tags in PTR** scheme requires $2 * |C| * |U|$ to $3 * |C| * |U|$ responses for each individual DNS request.

In the worst case, all services in U will satisfy the context query Q, which will result in $|U| = |\sigma(Q)|$. In this case, the **Formula in PTR** scheme guarantees that the number of responses will not become larger than with classical DNS-SD requests. **Tag to PTR** and **Conjunctions in PTR** schemes can only guarantee that if $|C|$ or $|T|$ are equal to one. Finally, the **Nested tags in PTR** scheme can make the number of messages larger than those of DNS-SD for 2 to 3 times C times, if we deal with DNS-SD responder that sends TXT records together with the discovery response.

To prevent the number of responses from becoming larger that $|U|$ for all naming schemes, we can reintroduce a requirement of placing all answers for one service to one DNS response. This way the number of responses will be at most $|\sigma(Q)|$ and will never become larger than the actual number of services in a network. An exception is for **Nested tags combinations**, that will still require up to $3 * |\sigma(Q)|$ responses.

To summarise, the number of messages generated with naming schemes can become considerably larger in a worst case, unless all the information for each service is sent as one DNS message. In this case, most naming schemes guarantee that the number of messages will never become larger than that of DNS-SD.

4.3.3.3 Number of DNS messages

As with naming schemes, the number of produced DNS messages depends largely on the actual situation with services context and the algorithm for distributing responses to DNS responses. In the worst case, the number of DNS messages is equal to one of mDNS messages, that has been analysed before.

4.3.4 Number of network packets

In previous subsections we determined lengths of DNS-SD context messages of different types, and the number of messages that have to be transmitted to fulfill the discovery

request. In this subsection, we calculate the number of network packets that one request or response will take. With the knowledge of this number, the context request made by the application, the number of satisfying services, and their context, one can calculate the number of network packages that will be sent via the network as a result of the context request.

In the previous section we calculated the number of DNS messages that have to be sent for each naming scheme to perform service discovery. Here, we note that for one DNS request we may receive several DNS responses of different structures and of different size, which has to be taken into account when calculating the resulting number of packets. If the context of every individual service is known, then for every context request we can determine all all of the produced responses.

With the general formulas 4.1 and 4.2, we can then calculate the length of the DNS questions and answers sections of the request and responses. By adding 8 bytes of UDP header and 12 bytes of DNS header, we get the whole size of each message. The UDP length of the request is then divided by the available space in the network packet. The operation for responses depends on the service discovery implementation. If the DNS server is used, all DNS records for suitable services may be placed in one response, and the length of this response is divided by the available space. For mDNS, every device replies to the request with matching records, and the length of each of these responses should be divided by the available space. In either case, the result of dividing is rounded up to the next natural number.

Of course, performing this operation for every service in the network is labor-intensive. To solve this problem, we can leave out service names during our calculations, and come up with the parametric equation $\lceil \frac{x+m}{c} \rceil = P$, where x is the overall length of the response without a service name, c is the available space in the network packet, P is the resulting number of packets, and m is the sum of all service names lengths in the response. This way we can come up with classes of services, for which the number of packets per response will be the same due to the close lengths of their names.

Here is the algorithm to find the number of network packets for a specific discovery request:

1. Calculate the UDP length for the request with the formula for requests length;
2. Calculate the number of packets for this request;
3. For every service that satisfies the request, determine the DNS response that will be generated and calculate its UDP length;
4. Calculate classes of equivalence, i.e. services that produce 1-packet responses, 2-packet responses, etc;
5. Multiply the size of each class of equivalence by the number of packets its members produce. Sum up the results for all classes of equivalence.
6. Add the number of packets for the request to the sum.

Further we illustrate the process of calculating the number of packets with an example. We also measure the number of packets for different naming schemes and try to find a possible correlation of this number with a number of transmitted DNS messages.

The size of the network packet depends on the used standard. In WSN the 6LoWPAN standard is often used. This standard specifies that the maximum length for a physical layer packet is 127 bytes. The header can be up to 25 bytes long, which leaves 102 octets. Link-layer security imposes further overhead, which in the worst case leaves 81 octets for data packets. Full IPv6 header takes another 40 bytes, leaving just 41 bytes for the UDP packet.

Fortunately, 6LoWPAN employs a header compression technique, which in the best case can compress an IP header down to 2 bytes. Hence the available space for a UDP packet is about 41-79 bytes.

To examine the number of packets generated by different naming schemes under different conditions, we performed a simulation. We randomly generated names of 40 services of type `_lgt._udp.local` (each name has length from 3 to 10 bytes). We randomly generated 3 different tags, names vary from 3 to 10 bytes. We assigned each service a random number of tags from 0 to 3. We then generated all possible non-redundant DNF queries on these 3 tags and calculated a number of 70-byte packets that will be produced if these queries are made by each naming schemes. We assumed that the **Nested tag combinations** scheme may require a second request for a service name. We calculated a number of packets both for the case of separate records delivery, and for the case of delivering all records for one service in one DNS response.

We performed this simulation 10 times to determine a linear equation(s) that could estimate the number of packets for a given number of messages and other parameters. This gave us a total of 180 observations for each naming scheme. A required sample size for multiple regressions studies is 134, provided that a linear regression has 3 variables, $(1-\beta) = 0.9, \alpha = 0.01, f^2 = 0.15$ [41]. Hence, we have enough of data for a regression study.

Our results are presented in Table 4.7. Numbers are minimal, maximal and average observed number of packets over 10 simulations. It can be seen that including all response records of a service in one DNS message helps to reduce the average and the maximal number of packets, especially for **Tag to Pointer** and **Nested tags combinations** schemes.

Naming scheme	**Formula in PTR**	**Conjunctions in PTR**	**Tag to PTR**	**Nested tags combinations**	**DNS-SD**
Minimum	9	8	14	7	41
Average	31.14	29.01	53.66	75.85	41
Maximum	65	75	151	214	41

Table 4.7: Number of packets produced by different naming schemes on a random setup. Every response record is transmitted separately.

Using linear regression analysis, we determined the equations to predict the number of packets produced by different naming schemes with the packet size of 70. We suggested

Naming scheme	Formula in PTR	Conjunctions in PTR	Tag to PTR	Nested tags combinations	DNS-SD
Minimum	9	8	14	7	41
Average	31.14	27.86	43.46	73.11	41
Maximum	65	61	81	182	41

Table 4.8: Number of packets produced by different naming schemes on a random setup. Response records for one service are transmitted in one DNS response.

that the number of packets depends on the number of resource records that have to be transmitted, and on the lengths of tags and service names. Hence, we investigated the dependency of the number of packets on three variables: the number of one-record messages, discussed in section 4.3.2.2, the average length of a tag, and the average length of a service name.

Most of the resulting regressions have an R^2 value very close to one, which means that they are very successful in predicting the number of packets. Unfortunately, we did not manage to include the size of the network packet in the equations, as it decreases the R^2 value to around 0.80. Variable coefficients in equations do not seem to depend on the packet size either. However, for a fixed number of packets, linear regression analysis produced equations with R^2 value very close to 1. Tables 4.9 and 4.10 represent a statistical model for the number of network packets of size 70 for different naming schemes. It can be seen that R^2 value for the **Formula in PTR** scheme is quite low, meaning that the number of packets cannot quite be predicted with the given variables. It could be a result of including the whole formula in DNS messages, thus making the number of packets depend on the length of a query.

Naming scheme	Number of DNS messages slope	Average service name length slope	Average length of tag slope	Intersection	R^2
Formula in PTR	1.351	0.754	2.031	-15.155	0.4582
Tag to PTR	1.008	-0.096	0.171	0.374	0.9998
Conjunctons in PTR	0.984	-0.147	0.189	1.518	0.9922
Nested tags combinations	1.036	3.123	2.973	-33.73	0.9964

Table 4.9: Coefficients of linear equations describing the number of generated network packets for different schemes. Every response record is transmitted separately.

To summarise this subsection, we note that the results of an example calculation seem to correlate to the calculated number of messages per service discovery. All naming schemes except for **Formula in PTR** have shown a reliable linear dependence on the number of transmitted records. This allows us to state that the number of records can be used for the primary evaluation of a naming scheme applicability, while the precise calculations of the number of packets allow us to estimate the actual network load.

Naming scheme	Number of DNS messages slope	Average service name length slope	Average length of tag slope	Intersection	R^2
Formula in PTR	1.35	0.754	2.031	-15.155	0.4582
Tag to PTR	0.49	0.898	0.063	12.0322	0.8557
Conjunctons in PTR	0.837	0.067	0.08	3.781	0.9551
Nested tags combinations	0.937	1.176	1.039	-9.529	0.9936

Table 4.10: Coefficients of linear equations describing the number of generated network packets for different schemes. Response records for one service are transmitted in one DNS response.

4.3.5 Summary

In this section we estimated the network load produced by different naming schemes. It is possible to compare the size of DNS requests: **Formula in PTR** produces the smallest requests, while other schemes can produce comparable requests. The size of the response mostly depends on a tags assignment and a context query.

We also compared the number of DNS messages that need to be sent in the mDNS network in order to complete a service discovery. In the worst case, every answer record is sent in its own response. In this case, **Formula in PTR** produces the smallest number of messages, **Conjunctions in PTR** and **Tag to PTR** the second smallest, and **Nested tags combinations** can produce the largest number of messages. The actual performance also depends on the structure of the context query. It is also possible that all records for one service fit on one DNS response. Our calculations show that for a given DNS request, all of the naming schemes can produce the same results except for **Nested tags combination**, which requires two to three times more messages to discover a single service.

Finally, we attempted to estimate the number of messages produced when using a classical DNS server as a responder for all services. We stated that this number largely depends not only on the context query and tags assignment, but also on the algorithm used to distribute answer records in DNS responses. In the worst case, each record ends up in a separate algorithm, and the number of messages is equivalent to the one of the mDNS model.

We have shown the difference in message size and number of messages between naming schemes and DNS-SD. The classical DNS-SD produces the number of messages limited by one or three sizes of the service type universe, i.e. the set of services with the requested type. Other naming schemes, except for **Nested tags combination**, can limit the number of messages by one size of the universe. Hence introducing naming schemes in the worst case does not change the number of DNS messages, and in other cases it decreases this number. With the **Nested tags combination** scheme a possible decrease depends on the applied optimisations and an implementation of a DNS-SD responder.

We gave the algorithm to calculate a number of network packets knowing the context query and the network setting. We also gave linear equations that can be used to accurately estimate the number of produced network packets given the number of transmitted records,

average tag length and average length of a service name. Thus, we have shown that the number of packets depends on the number of transmitted records for all schemes but **Formula in PTR**. From the number of network packets calculated in the example, it is also seen that for the same context query all naming schemes show roughly the same result, and the **Nested tags combination** scheme can produce several times more packets. The main reason for this is the necessity to transmit triple the amount of messages for each discovered service.

The usage of the **Formula in PTR** naming scheme is preferable in terms of the network load optimisation, as it produces less packets, though it is less predictable. The second best scheme is **Conjunctions in PTR**. Table 4.11 presents ranking of the naming schemes by the network footprint criterion. In the ranking we took into account the number of mDNS messages, as this is the only metric that allows us to compare schemes amongst each other.

Rank	**Naming scheme**
I	Formula in PTR
II	Conjunctions in PTR
II	Tag to PTR
III	Nested tags in PTR
III	Unmodified DNS-SD

Table 4.11: Ranking of different naming scheme by the network footprint criterion. Smaller rank is better.

4.4 Context tags restrictions

In this section we specify the maximum amount of context tags that can be assigned to one service with different naming schemes. It is important to know how many context properties a naming scheme can address.

A full domain name in DNS is limited to 255 octets (including 2 bytes for the total length field and the separator). Hence the maximum number of tags that can be encoded in one domain name is $(253-s)/(t+1)$, where s is the service type, and t is the length of one tag. This number is maximised if we choose $t = 1$ and do not include a service type at all, for example, to implement the optimisation described in subsection 6.2.1.1. The maximum number of tags in one domain name is then 126. However, this is an undesirable situation, because of a possible human-readability requirement of the application. The maximum length of a label in DNS is 63, hence we can always have at least $(253 - s)/64 = 3$ tags in one record (again, assuming we do not include service type here).

All naming schemes allow for placing an undefined number of tags in the query, provided that the query can be divided into separate DNS names of the allowed length. A DNS request with two DNS questions, say, q_1 and q_2, is functionally equivalent to the request with one DNS question $q_1 \vee q_2$. The number of DNS questions we can employ depends on the request size limitations; even if these limitations have been exceeded, the application

can divide DNS questions in several DNS requests and send them separately with the same set of discovered services.

In the following paragraphs we present scheme-specific boundaries for context tags that can be encoded in certain naming schemes.

Formula in PTR As this naming scheme creates records with context information dynamically, in principle the service could have any number of context tags. The space for requests and responses is, however, limited by the DNS protocol.

If we consider a formula to be one DNS label, then the most we can get is 63 bytes for the formula. This is quite a waste of bytes, as the total length of a domain name can be much larger. However, if we bring a formula to the disjunctive normal form and replace all disjunctions by dots, we can use up to 126 1-byte labels, or 126 1-byte tags in one question. Hence the maximum number of tags on the service for this scheme is undefined. However, the maximum number of tags that can participate in a DNS question or answer is 126.

Tag to PTR As each pointer record only contains one tag in this scheme, it allows for any number of tags associated with the service. Only one tag can be mentioned in an individual DNS question or answer.

Conjunctions in PTR This naming scheme assumes that for each service there exists the most descriptive combination of tags, containing all the context tags assigned to the service. This combination should fit in a single domain name. The maximum number of labels in one name is 126, hence this is the maximum number of tags that can be assigned to a service in this naming scheme. The size of every tag in this case should be no more than 1 byte.

If the Dynamic PTR creation optimisation technique is applied, the number of tags associated with the service could be any and the length of every tag could be increased to up to 63 bytes, as the DNS records no longer have to be created for all tags combinations. However, DNS requests and responses still have to fit in 253 bytes [38], and not all desired tags combination may be expressed.

In either case, the maximum number of context tags in one DNS question or answer is 126.

Nested tags combinations This naming scheme relies on the pointer record containing the most descriptive combination of context tags. This record has to be retrieved at one point of a service discovery. Hence the service cannot have more than 126 context tags applied. As well as with the previous scheme, there is only space for 126 1-byte tags in one DNS question or answer.

4.4.1 Comparison with DNS-SD

When context tags are stored in a TXT entry, they have to be concatenated with label divisors to form one string. The maximum length of one TXT entry value is 255 bytes [38], hence with respect to label divisors only 128 1-byte tags are allowed per one service. This is also the maximum number of tags in one answer. No tags are allowed in a request.

4.4.2 Summary

Table 4.12 presents the restrictions on the number of addressible context tags for one service with different naming schemes. We can state that the best scheme by this criterion is **Conjunctions in PTR** with the Dynamic PTR optimisation applied. Table 4.13 presents

Naming scheme	**Tags in request**	**Tags in one question or answer**	**Tags on the service**	
			With no optimisation	**Dynamic PTR optimisation**
Formula in PTR	undefined	126	undefined	undefined
Tag to PTR	undefined	1	undefined	undefined
Conjunctions in PTR	undefined	126	126	undefined
Nested tags combination	undefined	126	126	126
Unmodified DNS-SD	—	128	128	—

Table 4.12: Number of context tags that can be applied to a service with different naming schemes.

ranking of naming schemes by the number of context tags criterion. The best schemes by this criterion are **Formula in PTR** and dynamic **Conjunctions in PTR**, that allow for maximum 126 tags per query and an undefined number of tags on a service. The second best are **Tag to PTR** with or without dynamic optimisation, and static **Conjunctions in PTR**. The **Nested tags in PTR** is last because of inevitable limitation on the number of tags per service.

Rank	**Naming scheme**
I	Formula in PTR
II	Tag to PTR
III	Conjunctions in PTR
III	Unmodified DNS-SD
IV	Nested tags combinations

Table 4.13: Ranking of different naming schemes by the number of context tags criterion. Smaller rank is better.

4.5 Supportive code estimation

To estimate the size of software features required to support every solution, we employ the COSMIC [18] method. It allows to measure the size of software before it can actually be implemented, based on defined interfaces.

The measurement process consists of three phases: Measurement strategy phase, Mapping phase and Measurement phase. The first phase includes identifying the purpose of measurement, the scope and users of software being measured, and the level of measurement granularity. The second phase implies representing the software in terms of the Generic Software Model. The key concepts of this model are events, processes and subprocesses. The latter can be of two types: data movements and data manipulation. Finally, on the last phase the number of different types of data movements is calculated.

Data movement is an essential concept of the COSMIC method. Data that moves from users to the software is called an Entry. Data that goes back from software to the user is an Exit. Additionally, the software may need to read and write data to some kind of storage during its functioning. Such data movements are called Reads and Writes respectively.

We will go briefly through all these phases for all naming schemes. We will also calculate software sizes for some optimisations, for which the COSMIC analysis is applicable. The results of the first phase are equal for all schemes. The resulting sizes are given in Cosmic Functional Points (CFP).

4.5.1 Measurement strategy

The first thing required for the measurement is the measurement purpose. In this work, we perform the measurement to compare the size and complexity of implementation of different alternatives of DNS-SD context discovery extension tools.

The second concept that we need to specify is the scope. We select the whole implementation of the context-aware DNS-SD discovery tool as our scope. We examine on the requester and responder sides of the service discovery process separately. We assume that the external application provides the requester with a context query and expects to receive access points for appropriate services.

Users of the discovery tool are likely to be other pieces of software rather than actual people. They can be services in the network, service composition tools, or other tools capable of sending DNS requests and receive DNS responses. Therefore we define our Users as DNS-enabled software instances.

Finally, we describe the level of granularity. We operate with DNS requests and responses as atomic messages, and describe the process of responding in terms of constructing DNS messages from the set of context tags or performing a lookup in the local table.

4.5.2 Requesters

The process is initiated once the request for the service discovery is received from the User by the requester. The requester composes and sends the DNS service discovery request.

Depending on the naming scheme, the requester may need to perform a post-processing of results.

Formula in PTR requester In this paragraph we estimate the complexity for **Formula in PTR** requesters and requesters with Dynamic PTR optimisation. This kind of requesters do not need post-processing of results.

The model for a dynamic requester is depicted on the Figure 4.3.

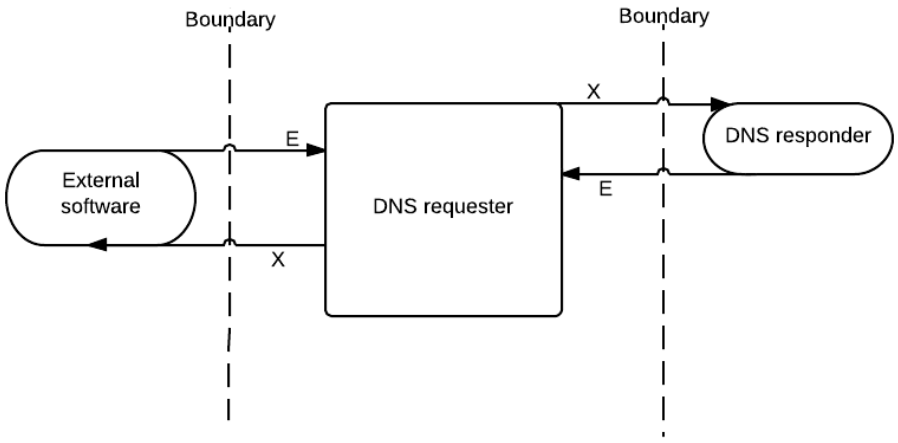


Figure 4.3: Generic software model for a dynamic context-aware DNS-SD requester.

1. Entry: Receiving of the signal from an external application;
2. Exit: Request for services to a DNS responder;
3. Entry: Receiving services from a DNS responder;
4. Exit: Passing the data to an external application.

This sums up to 4 CFP.

Tag to PTR requester Tag to PTR requester receives the list of services and their context tags, which has to be checked against context queries on the client side. In particular, it has to be checked if the service context satisfies necessary conjunctions and negations, which are represented by two different functions.

Figure 4.4 contains the Generic Software Model for this process.

Data movements include the following:

1. Entry: Receiving of the signal from an external application;

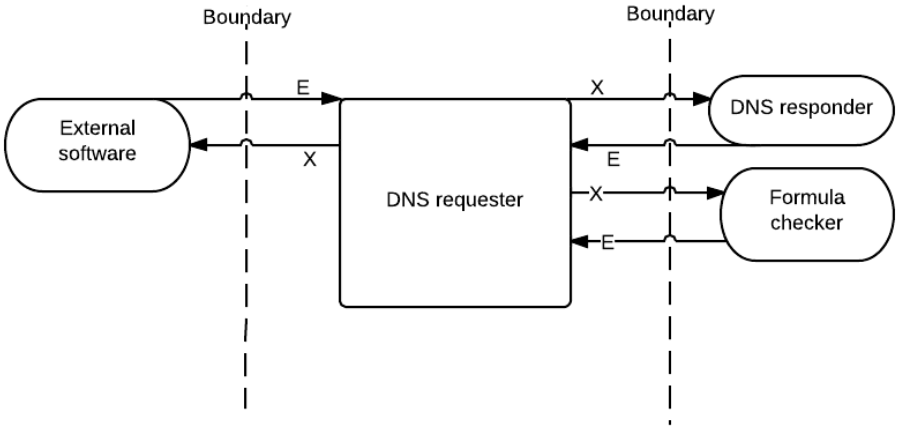


Figure 4.4: Generic software model for a context-aware Tag to PTR requester.

2. Exit: Request for services to a DNS responder;
3. Entry: Receiving services from a DNS responder;
4. Exit: Passing services to the formula checker;
5. Entry: Receiving services from the formula checker;
6. Exit: Passing the data to an external application.

The result is 6 CFP.

With the Dynamic PTR creation optimisation, this scheme would still require a post-processing of received services to perform an AND operation, which would result in 6 CFP for a requester.

Conjunctions in PTR requester The process is initiated once the request for the service discovery is received from the User. The query specifies the set of context tags and the service type for the discovery. The tool combines this data into the DNS request and sends it out. When responses come back, the tool filters out services that have negated tags.

Figure 4.5 contains the Generic Software Model for this process.

Data movement subprocesses include the following:

1. Entry: Receiving of the signal from an external application;
2. Exit: Request for services to a DNS responder;
3. Entry: Receiving services from a DNS responder;

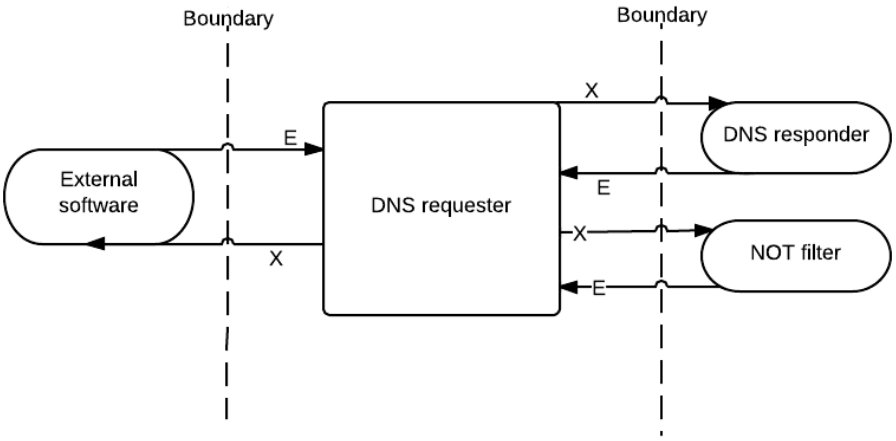


Figure 4.5: Generic software model for a context-aware DNS-SD requester.

4. Exit: Passing services to the NOT filter;
5. Entry: Receiving services from the NOT filter;
6. Exit: Passing the data to an external application.

This sums up to 6 CFP.

The possible necessity to query the received name again with the **Nested tags combinations** does not matter in the analysis, because subsequent name requests and responses are just instances of the same Exit and Entry data movements respectively, and the programmer would not need to reimplement them. The COSMIC analysis only considers different types of data movements.

If the Dynamic PTR optimisation has been applied, processes are identical to those of **Formula in PTR** scheme.

4.5.3 Responders

Lookup responders The responder process is initiated once the discovery DNS request is received. The tool performs a lookup in the internal DNS storage; if the matching record is found, the tool sends it back to the requester and the process ends.

We illustrate the model of the tool with the context diagram in Figure 4.6.

Data movement subprocesses include following:

1. Entry: Receiving of the request from a DNS requester;
2. Read: Reading DNS records from the internal storage;
3. Exit: Returning found DNS records to a DNS requester.

This sums up to 3 CFP.

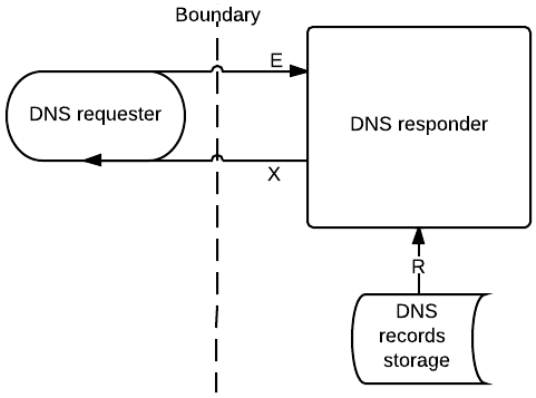


Figure 4.6: Generic software model for a static context-aware DNS-SD responder.

Dynamic responders On receiving the request, the responder reads context tags for known services from its internal storage. Next it compares the requested query against these context tags. If the satisfying service is found, the tool generates the DNS response and sends it back to the Requester. The model for a dynamic responder is depicted on the Figure 4.7.

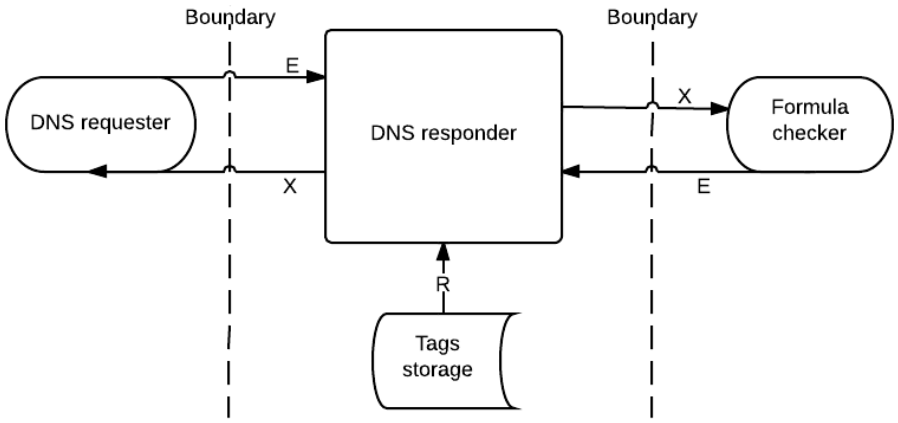


Figure 4.7: Generic software model for a dynamic context-aware DNS-SD responder.

Data movement subprocesses include following:

1. Entry: Receiving of the signal from a DNS requester;
2. Read: Reading available context tags from the internal storage;
3. Exit: Passing these tags and the request to the formula checker, in order to filter services that satisfy conjunctions;

4. Entry: Getting the result services from the formula checker, if any;

5. Exit: Returning DNS records for resulting services to a DNS requester.

Which sums up to 5 CFP.

4.5.4 Complexity analysis

The COSMIC method does not provide means to estimate the complexity of individual components. However, the method allows to assign extra points based on the expert ranking. Here we briefly estimate the complexity of the services resolution for each naming scheme, and assign extra COSMIC points based on this estimation.

Formula in PTR The complexity of a requester process is linear. The responder needs to parse and validate the received formula to return the list of services. Validation of a formula in DNF can take up to n comparison operations, where n is the number of tags in a query. The maximum number of operations is unbounded, as the query in DNF can be split to several DNS questions or even several DNS requests.

Conjunctions in PTR The complexity of a requester process is linear. The number of operations required to create pointers with no negations is bounded by the number of conjunctions in the formula. For a single conjunction with negated tags, it is required to create x PTR questions, where x is the number of negated tags in a conjunction. However, x is limited to the value of 126, due to DNS name length restrictions.

The checking of the negation is linear and also depends on the number of conjunctions, as the requester needs to check if the name of a received pointer is in the list of negation-free conjunctions, conjunctions with negations, or negated tags.

A responder is linear. All it has to do is to search for DNS records that match a request and return them. Again, in the worst case there will be 126 lookups for each conjunction in the context query formula, while in the best case the number of lookups will be equal to the number of conjunctions.

Tag to PTR The requester has to compose a request from a formula, collect all tags for all services and validate the formula. As with **Formula in PTR**, this process is linear to the number of tags assigned to services. The responder performs a linear-time DNS lookup for each requested context tag. In the worst case a lookup for each tag in a context query formula will be required.

Nested tags combinations The complexity analysis for this scheme is similar to the one for **Conjunctions in PTR** scheme, as the necessity for a second context request does not influence the complexity.

Dynamic PTR calculation This optimisation requires the responder to parse the DNS request and compare its tags to those of known services. As with **Formula in PTR**, in the worst case it would take n operations.

We established that all service discovery processes have a linear complexity; however, the running time of **Conjunctions in PTR** and **Nested tags combinations** depend on the number of conjunctions in the context query formula, and not on the number of tags. In the worst case, these schemes would produce the same running time as **Formula in PTR** and **Tag to PTR** because of the possible multiplication factor introduced by negation processing.

To express these differences, we assign an extra functional point to schemes whose running time depends on the number of tags. These are **Formula in PTR**, **Tag to PTR** and schemes with **Dynamic PTR optimisation**.

4.5.5 Additional features measurement

Employing additional optimising features will also result in size and complexity overhead. Here we list these features that can be applied to any naming scheme and estimate their size in COSMIC points.

Records caching The records caching means that DNS messages, calculated or received from the network, may be saved in internal storage for future use. For static scheme this requires one more Write operation to write records to the internal storage. For requester and dynamic schemes, two more data movements are required: one Read to check the cache before trying to resolve the request and one Write to write down the calculated record.

Changing the context If the service has to change its context, it has to either change its context tags or alter its stored DNS records. In either case this feature will require one COSMIC point to change the context of internal storage.

Preparing the storage In this work we did not discuss the ways for the service to obtain context tags. However, should this operation be performed, it will take one Write operation to populate the tags storage or records storage with the required context.

4.5.6 Summary

Table 4.14 summarises the estimations of the supportive code required by different naming schemes. The size of a requester was added to size of a responder for each naming scheme. No optimisation techniques except for Dynamic PTR were applied. Sizes of implementations for these techniques are presented in Table 4.15.

Unchanged DNS-SD does not require any supportive code introduction, hence it is ranked best of all. The second place is taken by static **Conjunctions in PTR** and

Nested tags in PTR schemes, the third one by schemes with support for dynamism and the static **Tag to PTR**. Finally, dynamic **Tag to PTR** scheme requires additional code both on requester and responder sides and takes the fourth place.

Rank	**Naming scheme**	**COSMIC Points**
I	Unmodified DNS-SD	0
II	Conjunctions in PTR	9
II	Nested tags combinations	9
III	Formula in PTR	10
III	Tag to PTR	10
III	Conjunctions in PTR (Dynamic PTR)	10
III	Nested tags combinations (Dynamic PTR)	10
IV	Tag to PTR (Dynamic PTR)	13

Table 4.14: Estimation on the implementation sizes of different naming schemes and their ranking. Smaller rank is better.

Optimisation	**COSMIC Points**
Caching	2
Changing context	1
Assigning of initial context	1

Table 4.15: Estimation on the implementation sizes of different optimisation techniques.

4.6 Evaluation summary

In Table 4.16 we provide the overall ranking of all naming schemes. As the dynamism introduced by the Dynamic PTR optimisation affects results of other metrics, we chose to consider naming schemes with this optimisation applied separately. Numbers in the tables columns represent the rank of the naming scheme when compared by the requirement in the column header. The lower number is the better solution. Naming schemes are sorted by their overall score, which is the sum of all its rankings. Figures 4.8 and 4.9 demonstrate the same data with spiderweb diagrams.

While Table 4.16 does not address the importance of different requirements, we can still use its data to select solutions by the criteria that are of most interest. As the main motivation of our work was to minimise the network load, we chose to accept only solutions

with ranks 1 or 2. This excludes the **Nested tags combinations** scheme from the set of solutions. The second important requirement may depend on the specific application. For example, we can choose the amount of additional code as such requirement, as the variety of resource-constrained heterogeneous nodes can be difficult to reprogram. The "Code amount" column helps us to select the **Conjunctions in PTR** as a solution for such nodes.

Note that static and dynamic variants of one naming scheme are compatible with each other and can be used together in one application. This means that the static variant can be used for nodes that have a small amount of tags or are hard to reprogram, while the dynamic variant can be used for nodes that are easier to reprogram, or have a larger number of tags, or limited memory. With this in mind, we can recommend **Conjunctions in PTR** as an optimal scheme. Its network performance is only slightly worse than one of the **Formula in PTR**, and it can be employed in static or dynamic variants for nodes with different capabilities. Its network performance is also more predictable.

However, as can be seen in Appendix B, other schemes can still be applicable, as applications may require specific features addressed only in particular naming schemes.

	Network	**Operations**	**Code amount**	**Memory usage**	**Number of tags**	**Total**
Unmodified DNS-SD	3	1	1	2	3	10
Dynamic schemes						
Formula in PTR	1	2	3	1	1	8
Tag to PTR	2	2	4	1	2	11
Conjunctions in PTR	2	3	3	1	3	12
Nested tags in PTR	3	1	3	1	4	12
Static schemes						
Tag to PTR	2	2	3	3	2	12
Conjunctions in PTR	2	3	2	4	3	14
Nested tags in PTR	3	1	2	5	4	15

Table 4.16: The overall ranking of all naming schemes with different metrics.

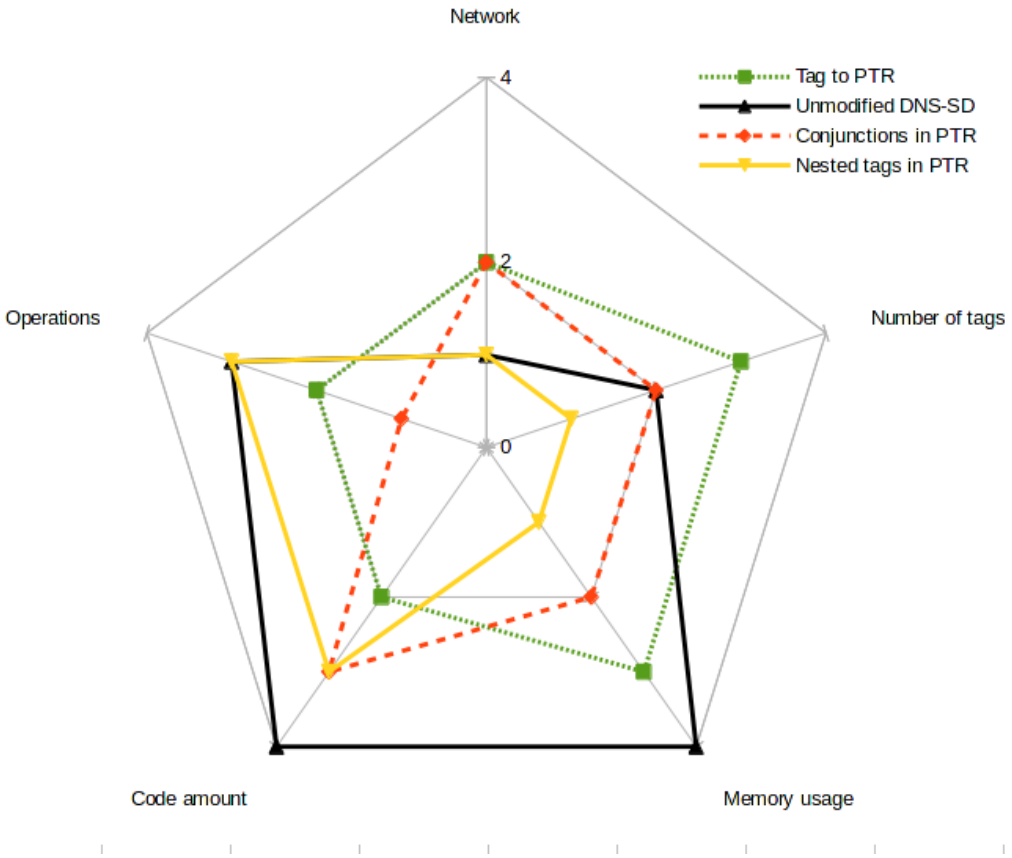


Figure 4.8: Web diagram comparing naming schemes with static responders. Larger value is better.

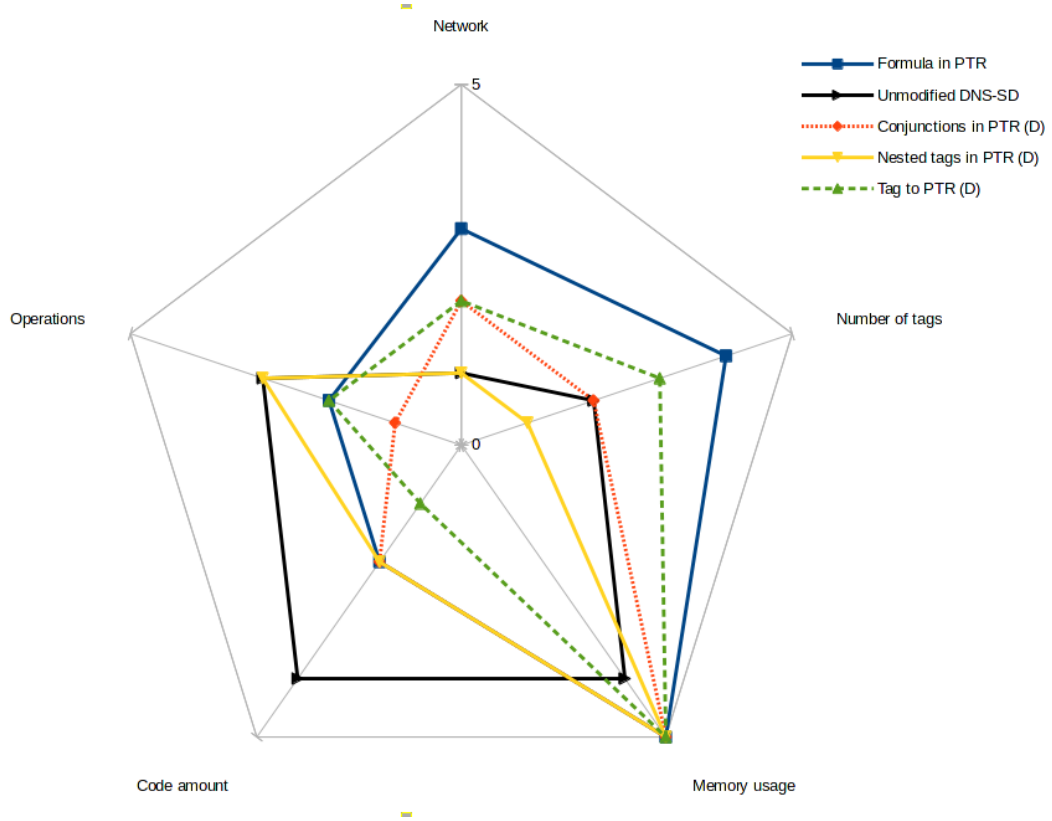


Figure 4.9: Web diagram comparing naming schemes with dynamic responders. Larger value is better.

Chapter 5

Implementation

To illustrate our work and as a proof of concept we built a Java library that provides context-aware service discovery using mDNS/DND-SD.[1] The library was written using the Java programming language version 7 and is based on the JmDNS library [33] of version 3.4.1. JmDNS provides means for publishing and discovering services with mDNS/DND-SD. However, a number of important changes had to be made in order to make a context-aware discovery possible. We also created a demo application to demonstrate the library capabilities and compare different naming schemes.

5.1 Base library: JmDNS

JmDNS [33] is a Java library that provides service discovery with mDNS and DNS-SD protocols. It runs on most of JDK 1.6 compatible machines. JmDNS ships with documentation and a lot of useful examples, which made it easy to extend.

The library is quite service-centric: in spite of the fact that the mDNS protocol can in principle be used for purposes other than DNS-SD service discovery, JmDNS only provides service registering and discovering features. For example, there are no interfaces to resolve device names, search for PTR resource records, or create separate PTR records for other clients to find.

The interfaces that the developer has to operate with are the `ServiceInfo` interface, which contains the information about the service, and the `JmDNS` interface, which allows publishing and discovering services. Figure 5.1 depicts the public methods of these interfaces. These interfaces rely on a multitude of classes with different levels of abstraction. Most of these classes are limited to support only service discovery related needs: for example, when the DNS entry representation is constructed, it is automatically assumed that the name of the record should contain service type, protocol and domain. In order to implement context-aware service discovery, we had to make changes to the library on several levels. These changes are introduced further.

[1] The documentation and examples are available online [42].

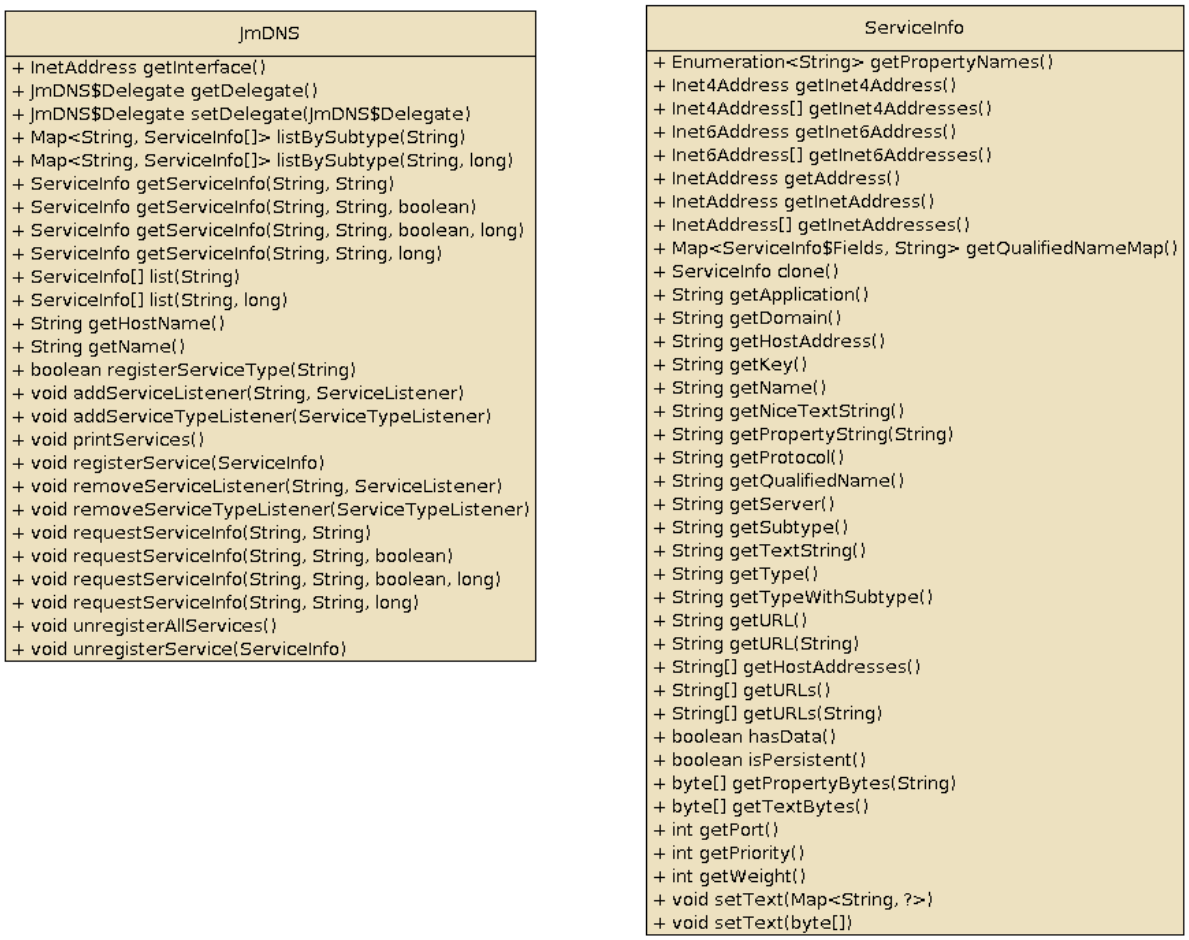


Figure 5.1: Public methods of two main JmDNS interfaces.

5.2 Changes and extensions

The updated version of a library includes following new features:

- Service publishing and discovery with one of naming schemes;
- Dynamic PTR creation mode for publishing;
- Discovery of services that satisfy given context formula.

To implement these features, we had to introduce a number of changes, shortly described here.

5.2.1 Naming schemes

Newly introduced classes include the Naming Scheme abstract class and four implementations for Formula in PTR (3.2.2.1) Tag to PTR (3.2.2.2), Conjunctions in PTR (3.2.2.3)

and Nested tag combinations (3.2.2.4) naming schemes. The methods of the classes are shown in Figure 5.3. All naming schemes allow usage of Dynamic PTR optimisation.

5.2.2 Context-aware service discovery

The library was modified to be able to publish and store any DNS resource record, even if it is not associated with a particular service. These records can now be discovered by other services. JmDNS interface now has a new method for discovering any type of resource record.

One of the most important changes is the extension of the `ServiceInfo` class with methods to access, add and remove context tag of the service. The Formula and Conjunction classes are introduced in order to enable service discovery using boolean context queries. The Formula class has a single method for checking if the query is satisfied, as shown in Figure 5.2.

Another introduction is a set of two classes that allow easy service publishing and discovery. These classes work as wrappers for the modified JmDNS interface and hide its complexity. Both classes take a `NamingScheme` class as a constructor parameter; that is, both classes perform discovery and publishing with respect to some naming scheme. The public methods of these classes are presented in Figure 5.4.

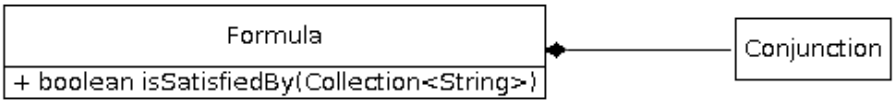


Figure 5.2: Classes for representing DNF queries.

The component diagram shown in Figure 5.5 summarizes the changes made to JmDNS interfaces and the newly added interfaces. The interface elements of `JmDNS` and `ServiceInfo` contain new methods.

5.3 Demo application

We created an application that uses the extended JmDNS library as a proof of concept and to demonstrate the discovery process. The application provides a visual interface for an end-user to publish and discover services with the help of context tags. The application consists of two forms for service publishing and service discovery. In the publish form, the user can specify an arbitrary name for a service, its protocol, service type and its port. Context tags can be applied or removed. In the discovery form the user can specify arbitrary protocol and service type. Context tags can be applied or removed.

Here we provide an illustration of the discovery process performed with our tool. Suppose the service NodeB with context tags `tag_1` and `tag_2` is being provided. The user

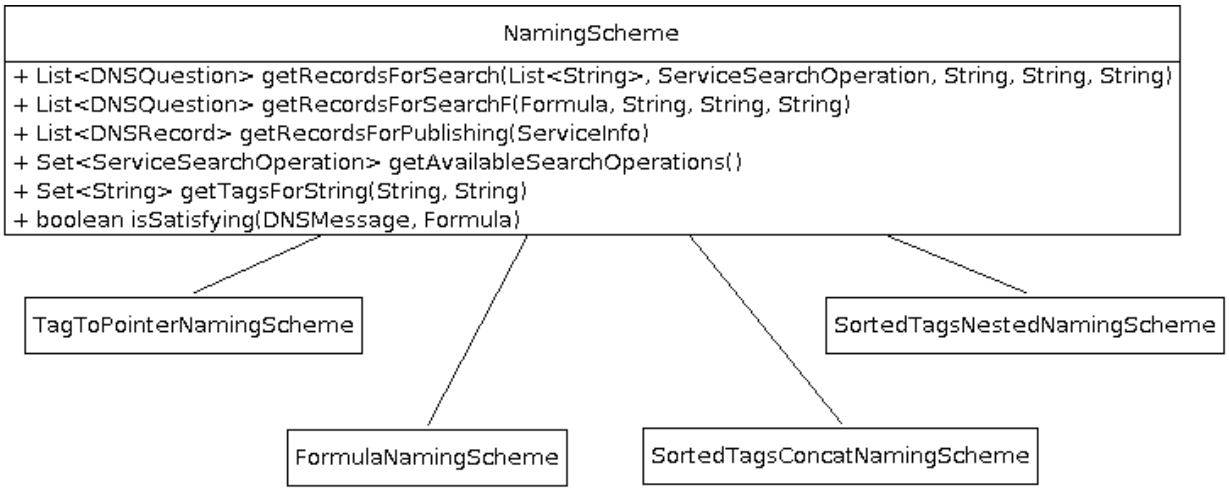


Figure 5.3: Classes for the implementations of naming schemes.

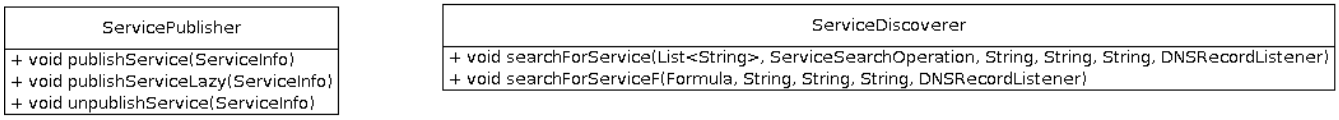


Figure 5.4: Public methods of context-aware Publish and Subscribe classes.

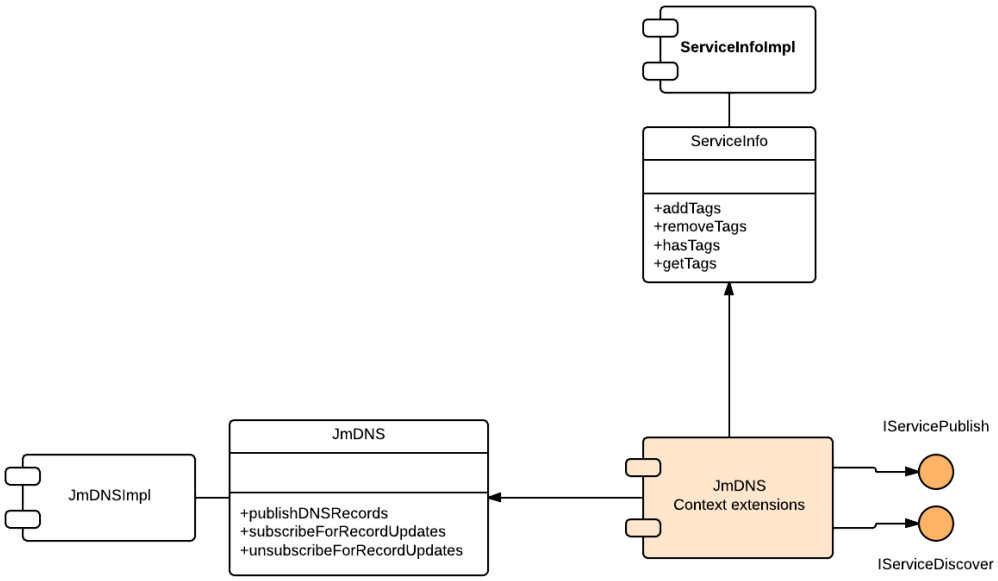


Figure 5.5: The component diagram of JmDNS context extensions.

Figure 5.6: Publish form.

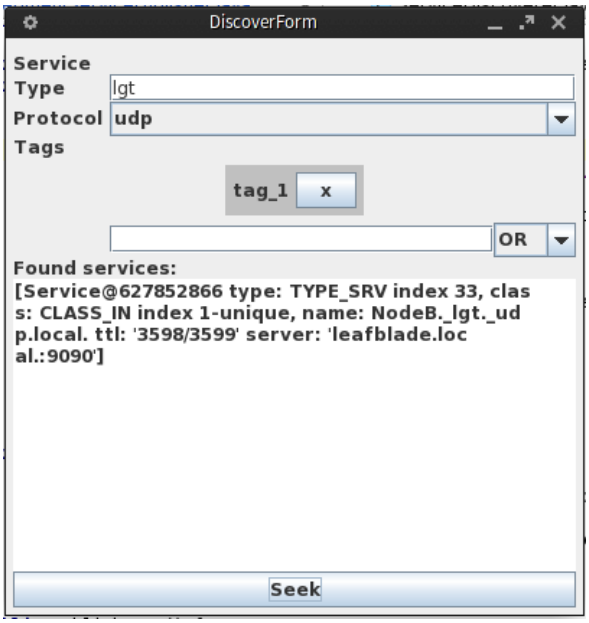


Figure 5.7: Discovery form.

wants to find services of type `_lgt._udp.local` with context tag `tag_1`. The naming scheme used by the tool is **Conjunctions in PTR**. The Dynamic PTR optimisation is used.

Figures 5.6 and 5.7 show filled application forms to perform service publishing and discovery. Table 5.1 shows the network trace produced by the application and captured with Wireshark. Messages 1-10 refer to the tool publisher and discoverer parts trying to obtain domain names in .local domain. Messages 11-15 correspond to the publisher ensuring that the service has a unique name. Messages 16-19 are most interesting, because they refer to the actual discovery process. We present DNS questions and answers that are contained in those messages below.

Message 16 is the question asked by the discovery part of the tool. It contains the query for tag `tag_1` and type `_lgt._udp.local`.

```
tag1._lgt._udp.local:  type PTR, class ANY, "QM" question
```

Message 17 contains the answer on the question from message 16. This is a PTR record with a service name in its alias.

```
tag_1._lgt._udp.local:  type PTR, class ANY, NodeB._lgt._udp.local
```

Then message 18 represents the process of resolving the service name. The discoverer tool tries to obtain Pointer and Service records for the name it received in the previous step. The request for Pointer record is currently required to support **Nested tags combinations** scheme.

```
NodeB._lgt._udp.local:  type PTR, class ANY, "QM" question
NodeB._lgt._udp.local:  type SRV, class ANY, "QM" question
```

Finally, in message 19 the service description, access point and metadata is being sent. The tool successfully discovered a service with required context.

```
_lgt._udp.local:  type PTR, class IN, NodeB._lgt._udp.local

leafblade.local:  type A, class IN, cache flush, addr 131.155.220.67

NodeB._lgt._udp.local:  type TXT, class IN, cache flush

NodeB._lgt._udp.local:  type SRV, class IN, cache flush, priority 0, weight 0, port
9090, target leafblade.local
```

The source code for the library and the demo application can be obtained online [42].

5.4 Summary

Implementation of context-aware service discovery requires to introduce changes to the original library on several levels of abstraction. It turned out that JmDNS contains a lot of assumptions regarding names of records and discovery process. Some of those assumptions were implemented at a very low level, with no possibility to employ an alternative implementation. Hence the existing implementation had to be altered. As a result, a new form of the original library has been created, which is not compatible with the original one. This might be a motivation to reimplement the whole stack of technologies in the future, taking care of maximal functional scalability and allowing for multiple alternative implementations on all levels of abstraction.

No.	Length	Info
1	91	Standard query 0x0000 ANY leafblade.local, "QM" question
2	91	Standard query 0x0000 ANY leafblade.local, "QM" question
3	91	Standard query 0x0000 ANY leafblade.local, "QM" question
4	85	Standard query response 0x0000 A, cache flush 131.155.220.67
5	85	Standard query response 0x0000 A, cache flush 131.155.220.67
6	91	Standard query 0x0000 ANY leafblade.local, "QM" question
7	91	Standard query 0x0000 ANY leafblade.local, "QM" question
8	91	Standard query 0x0000 ANY leafblade.local, "QM" question
9	85	Standard query response 0x0000 A, cache flush 131.155.220.67
10	85	Standard query response 0x0000 A, cache flush 131.155.220.67
11	111	Standard query 0x0000 ANY NodeB._lgt._udp.local, "QM" question
12	111	Standard query 0x0000 ANY NodeB._lgt._udp.local, "QM" question
13	111	Standard query 0x0000 ANY NodeB._lgt._udp.local, "QM" question
14	132	Standard query response 0x0000 PTR NodeB._lgt._udp.local SRV, cache flush 0 0 9090 leafblade.local TXT, cache flush
15	132	Standard query response 0x0000 PTR NodeB._lgt._udp.local SRV, cache flush 0 0 9090 leafblade.local TXT, cache flush
16	81	Standard query 0x0000 PTR tag_1._lgt._udp.local, "QM" question
17	95	Standard query response 0x0000 PTR NodeB._lgt._udp.local
18	87	Standard query 0x0000 PTR NodeB._lgt._udp.local, "QM" question SRV NodeB._lgt._udp.local, "QM" question
19	148	Standard query response 0x0000 TXT, cache flush SRV, cache flush 0 0 9090 leafblade.local PTR NodeB._lgt._udp.local A, cache flush 131.155.220.67

Table 5.1: Network trace produced by the demo application and captured by Wireshark.

Chapter 6

Conclusions and future work

6.1 Conclusions

Application in WSNs can employ the information about context of individual devices. This context information allows to address specific situations through devices with a given property, as location, capabilities, etc.

The goal of our work was to find a way to include context properties in the DNS-SD service description and make services discoverable using DNS-SD with respect to their context properties. This extension was expected to optimise the network load. We derived extra-functional requirements to the solution from the setup in order to address its specifics and make the solution suitable for WSNs and current task. These requirements are the following:

- Minimize network load associated with service discovery;
- Minimize additional client code to support context-aware discovery;
- Minimize the amount amount of memory needed to store the context information;
- Introduce discovery features, allowing to express any required combination(s) of context properties;
- Maximize the amount of expressible context properties.

We described a technique to encode context as a set of independent atomic context tags. We proposed four different naming conventions, or naming schemes, that allow to express context inside DNS-SD and subsequently, use context information as discovery parameter. The following naming schemes were proposed:

- **Formula in PTR**. This naming scheme allows to include a boolean formula for context query into one PTR record. The responder checks whether this formula holds for any service.

- **Tag to PTR**. This naming scheme creates a pointer for every tag in the query. It does not allow conjunctions of tags to be expressed. Services are resolved by a lookup for records matching individual tags in the context query. The results are then checked against the actual query on the client side.
- **Conjunctions in PTR**. This naming scheme creates a DNS label for every tag in the query. Conjunctions of tags are expressed by sorting corresponding labels and placing them in one domain name.
- **Nested tags combinations**. This naming scheme allows to resolve not only the name of the service, but also its full context. The scheme is similar to **Conjunctions in PTR**, but may require one more network request for service resolution. The resolver first gives out lists of all tags for all discovered services, allowing a client to make a new request with a full service context. In addition, this naming scheme can save some storage by sharing some resource records between several services.

To evaluate the proposed solutions we derived metrics from our requirements, and calculated metrics values for each naming scheme. Those metrics are, namely:

- The amount of memory occupied by the context information;
- Search operations allowable in queries;
- UDP datagram size;
- Number of DNS/mDNS messages;
- Maximum number of context tags that can be assigned to a service and used in queries;
- The amount of supportive code.

It turned out that the network load produced by naming schemes depends a lot on the context query and distribution of tags over services. It was also established that in certain cases naming schemes can produce as much or even more network load than the original protocol.

The main conclusion is that no naming scheme can be selected as optimal. The choice is to be made depending on the relative importance of metrics in a specific case. However, in our opinion, the **Conjunctions in PTR** scheme is a good compromise, due to the simpler complexity and improved network load. Another important advantage of this scheme is its larger flexibility: the scheme can be implemented in static and dynamic variants for two types of nodes.

The contribution of this work includes both theoretical and practical parts. As a theoretical part we presented several naming schemes for including the context, as well as their ranking by relevant criteria. This allows for easy selection of the most appropriate naming scheme depending on individual requirements of the specific application. The practical part of contribution is a library that implements service discovery with the presented naming schemes.

6.2 Future work directions

In this work we gave the best- and worst-case boundaries for memory requirements of the **Nested tags combinations** naming scheme. However, an estimation of the expected number of stored records might be useful, as in certain cases this scheme might optimise the memory usage. Therefore, one of possible future work directions is a more detailed analysis of the **Nested tags combinations** scheme in the memory part. It involves estimating the general-case boundaries of the number of stored records depending on the number of services, their context and context tags in the system.

Another work direction is to perform a more detailed statistical analysis of naming schemes, especially **Formula in PTR**. Such analysis might include experiments with formulas on a larger number of tags and building a model that includes the size of the network packet or other factors. This will help to derive better equations for predicting the generated network load based on a multitude of factors.

Evaluating the approach against the realistic setup might be a promising task, as it will allow to check the ranking provided by metrics results and experimentally estimate the network load produced by service discovery with different methods.

The process of assigning context to services was not discussed in this work. It is hard to manually maintain a context of each device in a large network, therefore the automation of context management could be a relevant task.

Another aspect of interest could be minimisation of memory and network load associated with the context information. There might exist applicable compression techniques. For example, context tags could be encoded in a bit string, where each bit represents if a certain tag is set or not.

6.2.1 Optimisations

In this section we discuss possible optimisations of a context-aware service discovery implementation. These optimisations help to decrease the amount of supportive code and the amount of required memory. These features might be useful for certain applications.

6.2.1.1 Tags without a type

If the application requires an access to all the services in a particular context, irrespective of their type, the service type suffix in PTR names can be omitted. For example, instead of the record

```
tag_1.tag_2._printer._tcp.local PTR Printer._printer._tcp.local
```

The following one may be used:

```
tag_1.tag_2 PTR Printer._printer._tcp.local
```

This optimisation decreases the payload for DNS questions, and also eliminates the necessity to make several requests with different service types to obtain all the services with a specific context.

However, this optimisation might be impractical if the application requires services of a particular type.

6.2.1.2 Dynamic negation calculation

For most naming schemes, the negation operation is performed on the client side, as a DNS resolver can only perform a records lookup for requests. However, we still can implement this operation on the server side if the responder is allowed to parse and process request tags rather than perform a DNS lookup. In this case, the client marks negated tags or conjunctions with a special negation symbol (for example, the hyphen) inside the DNS request. The responder recognises this symbol and checks the whole set of negated tags in the query against the set of context tags applied to the service.

This optimisation implies that either the **Formula in PTR** naming scheme is used, or the Dynamic PTR creation optimisation (4.1.2) is applied. These techniques enable to parse tags in the request and check them against the list of the service tags dynamically, which is required for the dynamic negation calculation. The advantage of a responder-side negation evaluation is that with little additional effort on the responder side, the client gets completely spared from the necessity to process negation.

Acknowledgements

This thesis was created as a part of the master project that I performed in the System Architecture and Networking group in Eindhoven University of Technology, the Netherlands. I would like to thank my supervisors, Pieter Cuijpers and Milosh Stolikj, for guiding me through the master project and providing a lot of insights. My research and technical writing skills have improved a lot under their supervision.

I would also like to express my gratitude to my fellow students who supported my work with their help and attention. I especially thank Mathijs Vermeulen for his infinite care and personal support. Many thanks to Andy Burstein for proof-reading my paper and checking my English.

This project would be impossible without the help from my family. Therefore, I would like to thank my parents Natalia Buchina and Gennady Gurov for supporting my decision to study here in TU/e and perform this master project.

Finally, I thank Anna Kuranova, Oleg Svyato, Natalia Kuznetsova, Roy Berkeveld, Emilija Lazdanaitė and other friends of mine for their cheers and moral support.

Appendix A

Maximal number of conjunctions in redundancy-free DNF formula

Consider a set of tags $t_1 ... t_n$. We argue that any non-redundant boolean formula on this set in DNF can contain a maximum number of conjunctions of $\binom{n}{\lfloor n/2 \rfloor}$.

First of all, we state that a formula of the maximal length should contain only conjunctions of a same length (i.e. conjunctions, containing the same number of variables). We name conditions for this statement to hold and prove that either of these conditions always holds. Then we find the maximal number of conjunction in a formula consisting of conjunctions of a same length.

Suppose that there is a DNF formula of maximal length with no redundancy, containing conjunctions of different length. Then one of the following is possible:

- A conjunction with the maximal length can be replaced by a set of its subconjunctions of a smaller length with the formula still being non-redundant and in DNF.
- A conjunction with the minimal length can be replaced by one or more conjunctions of a larger length that contain a minimal conjunction as their subconjunction. For example, a conjunction t_1 can be replaced by $t_1 \wedge t_2, t_1 \wedge t_3 ... t_1 \wedge t_n$.

A.1 Condition 1

A non-redundant formula in DNF does not contain any of subconjunctions of a maximal conjunction; otherwise, it would contain both the maximal conjunction and its subconjunctions, which would make it redundant. Hence, a maximal conjunction of the length c can be replaced by c conjunctions of the length $c - 1$.

Each of these conjunctions, however, might make another conjunction of the length c in a formula redundant. They cannot make any longer conjunctions redundant, as c is the maximum length of a conjunction in a formula; and they cannot be redundant themselves, because there may be no subconjunctions of a maximal conjunction in a formula. A number of conjunctions of length c that can be made redundant by each subconjunction is $n - c$:

this is the number of variables that do not participate in a subconjunction. An overall number of conjunctions of length c that can be made redundant is therefore $c * (n - c)$.

Now we note that, unless all of conjunctions in a formula are of the same size, there will be at least one conjunction of the length $c-a, a \geq 1$. An existence of such conjunction makes it impossible for its superconjunctions of the length c to exist in a formula, because that would make a formula redundant. The number of such impossible conjunctions of length c is $\binom{n-(c-a)}{a}$.

Now for Condition 1 to hold, the number of newly generated subconjunctions must be larger than the number of the conjunctions of the length c, that can possibly be made redundant by this operation, minus the number of conjunctions of the length c that cannot exist due to a necessity to have at least one conjunction of length $c - a$. In other words, a following inequality should hold:

$$c \geq c * (n - c) - \binom{n - (c - a)}{a}$$

.

A.2 Condition 2

A non-redundant formula in DNF does not contain any of superconjunctions of a minimal conjunction; otherwise, it would contain both minimal conjunction and its superconjunctions, which would make it redundant. Hence, a minimal conjunction of the length c can be replaced by $n - c$ conjunctions of a the length $c + 1$.

Each of these conjunctions, however, might be made redundant by another conjunction of the length c in a formula. Shorter conjunctions cannot cause this, as c is the minimum the length of a conjunction in a formula; and longer conjunctions cannot be made redundant, because there may be no superconjunctions of a minimal conjunction in a formula. A number of conjunctions of the length c that make redundant each subconjunction is c: this is the number of subconjunctions in a new conjunction, excluding the initial minimal conjunction. An overall number of conjunctions of the length c that can make newly generated conjunctions redundant is therefore $c * (n - c)$.

Now we note that, unless all of conjunctions in a formula are of a same size, there will be at least one conjunction of a the length $c + a, a \geq 1$. Existence of such conjunction makes it impossible for its subconjunctions of the length c to exist in a formula, because that would make a formula redundant. The number of such impossible conjunctions of the length c is $\binom{(c+a)}{c}$.

Now for Condition 2 to hold, the number of newly generated superconjunctions must be larger than a number of the conjunctions of the length c, that can possibly make newly generated conjunctions redundant, minus the number of conjunctions of the length c that cannot exist due to a necessity to have at least one conjunction of the length $c + a$. In other words, a following inequality should hold:

$$n - c \geq c * (n - c) - \binom{n - (c + a)}{c}$$

.

A.3 Proof

Suppose there is a formula with a maximal conjunction length c and minimal conjunction length $c - b, b > 0$. Inequalities for two condition can be rewritten as:

$$\begin{aligned} c &\geq c * (n - c) - \binom{n-(c-b)}{b} \\ n - (c - b) &\geq c * (n - (c - b)) - \binom{n-c}{(c-b)} \end{aligned} \tag{A.1}$$

At least one of these inequalities always holds with $c \geq 2$ [1]. Hence, indeed, for every formula with different lengths of its conjunctions, there is a formula of same length or longer, with conjunctions of a same size.

The number of conjunctions of a same size that can be placed in a formula can be calculated as a number of combinations of that size: $\binom{n}{c}$ represents a number of possible conjunctions of the length c with n variables. This function is maximised when $c = \lfloor \frac{n}{2} \rfloor$, hence the maximal possible number of conjunctions in a DNF formula with n tags is $\binom{n}{\lfloor \frac{n}{2} \rfloor}$.

[1] Checked with WolfamAlpha [43]

Appendix B

Examples

This section illustrates the applicability of different naming schemes to different application and provides an insight on how specific properties of naming schemes can be utilised in various setups. We will apply our approach on practical examples and demonstrate the discovery process for their specific applications.

The section covers two scenarios: Building control and the SenSafety festival setup. For each scenario we give two examples of applications. For every application we derive relevant context properties that it addresses, express them with context tags and argue on the best naming scheme applicable to the example. We illustrate the service discovery of the application with the sequence of DNS messages. For the Building control scenario we compare our approach against the alternative scheme described in [21]. For the SenSafety scenario, for each application we provide two solutions based on different naming schemes and compare them with each other.

B.1 Building control

Consider a network of sensors and actuators deployed in building. These sensors and actuators provide an interface for building automation and control applications. Every device in the network is defined through the following contexts:

- Hierarchical location: campus/street/address → building → block → floor → room;
- Type: luminosity sensor, light actuator, presence sensor, shade actuator, temperature sensor etc.
- Access interface: application protocol and URI.

Here we will demonstrate how the proposed context extensions will handle several applications.

B.1.1 Energy saving

The setup consists of a presence sensor, a luminosity sensor and a light actuator that have methods for switching light on and off. They are described using DNS-SD as in Table B.1. The application receives data from the presence and the luminosity sensors periodically, or as events when their measurements have changed. If there are people in the room and light is off, the light should be switched on with the light actuator. If there are no people in the room and the light is on, the light should be switched off. For this purpose we assume that all devices are located in the office number 80, on floor 6, in the MetaForum (MF) building, in the TU/e campus.

Device	**DNS-SD SRV records**
Presence sensor	`ps._presence._sub._coap._udp.local SRV ps1.local 80`
Luminosity sensor	`ls._lumsensor._sub._coap._udp.local SRV ls1.local 8081`
Light actuator	`l1._light._sub._coap._udp.local SRV la1.local 8082`

Table B.1: Service records for the energy saving scenario.

Encoding context The application in this example needs to discovery services located in exact room, exact building, exact floor etc. Moreover, it is not interested in acquiring all context tags for each service, and all services are of different type. As a result, we employ the **Conjunctions in PTR** naming scheme (see subsection 3.2.2.3). It is adapted for querying for conjunctions and is not as complex as **Nested tags combinations**.

We encode location information with the following tags: `r80` for the room number, `f6` for the floor number, `mf` for the building. Type and protocol information are appended at the end of pointer record, just as in standard DNS-SD.

As a result, we need to create following DNS records to express context:

```
f6.mf.r80._presence._sub._coap._udp.local PTR ps._presence._sub._coap._udp.local

f6.mf.r80._lumsensor._sub._coap._udp.local PTR ls._lumsensor._sub._coap._udp.local

f6.mf.r80._light._sub._coap._udp.local PTR l1._light._sub._coap._udp.local
```

The application that wants to find presence sensor and luminosity sensor in 80 office on 6th floor of MF building makes a PTR request with questions of two service types with appropriate tags. Then application receives pointers with service names as a response. After that, application queries for SRV records by received names, and, finally receives service access points. This process is depicted on Figure B.1.

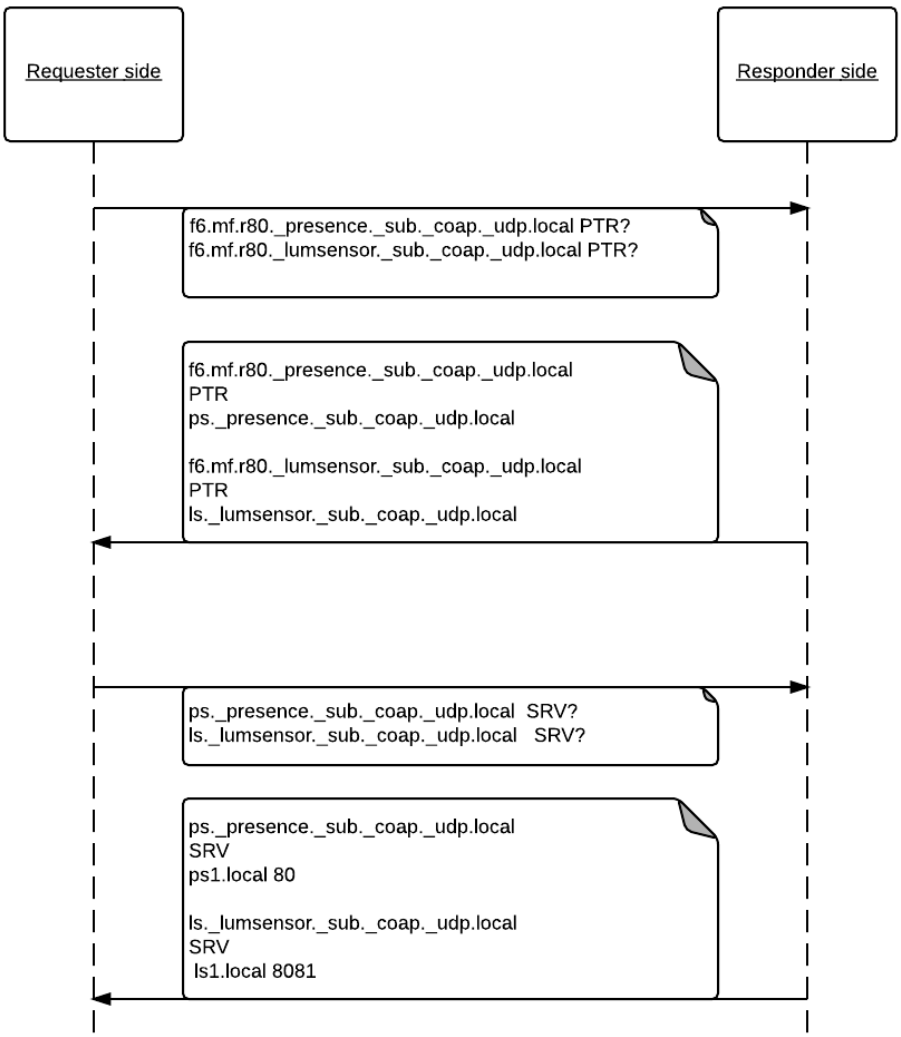


Figure B.1: Sequence diagram for the Energy saving scenario.

After all measurements have been made, the light actuator is discovered the same way.

For the needs of this application, we don't need any more records for context. However, if later we may want to modify it, for example, to turn off light in corridor if there is no one on the floor, we will need to create more records to make devices discoverable without mentioning their office:

```
f6.mf._presence._sub._coap._udp.local PTR ps._presence._sub._coap._udp.local

f6.mf._lumsensor._sub._coap._udp.local PTR ls._lumsensor._sub._coap._udp.local

f6.mf._light._sub._coap._udp.local PTR l1._light._sub._coap._udp.local
```

However, this leads to necessity to store additional records. To save space, the **Dynamic PTR creation** optimisation (4.1.2) can be applied. In this case, no records will be

stored, and responses to DNS questions from the client will be generated by the responding side.

Following is the example of the application that discovers all smart lights in the building with the Dynamic PTR creation optimisation applied. The application makes a request specifying only the building and the device type. The responder side, instead of looking up its DNS records storage, parses the request for context tags and service types, obtains the list of services that satisfy these restrictions and generates DNS records for them. These records are sent out:

```
mf._light._sub._coap._udp.local PTR
mf._light._sub._coap._udp.local PTR l1._light._sub._coap._udp.local
```

The same principle may be used to construct requests of any granularity.

B.1.2 Fire department

In the fire department scenario, an application needs to discover all presence sensors in a given area (room, floor, building). This can be useful for security check or for better understanding of evacuation process in case of fire.

Encoding context For this application it might be useful to know full context of discovered services, for example, to build an interactive map of people presence. In addition, we are only interested in sensors of one type, and would like save some space on DNS records. That's why here we employ **Nested tags combinations** naming scheme as it allows such behavior. See section 3.2.2.4 for details on this behavior.

We assume that all sensors are located in one building, on arbitrary floors and in arbitrary offices. Services for presence sensors are encoded in DNS-SD as described in B.2.

Device	DNS-SD SRV record	Location	Tags
Sensor 1	`ps_1._presence._sub._coap._udp.local SRV ps1.local 8081`	MF building, floor 6, room 80	`mf`, `f6`, `r80`
Sensor 2	`ps_2._presence._sub._coap._udp.local SRV ps2.local 8082`	MF building, floor 6, room 13	`mf`, `f6`, `r13`
Sensor 3	`ps_3._presence._sub._coap._udp.local SRV ps2.local 8082`	MF building, floor 1, room 1	`mf`, `f1`, `r1`

Table B.2: Service records for the fire department scenario.

Now to encode context we need to create one pointer containing full set of context tags for each sensor. Following are examples of such records.

```
f6.mf.r80._presence._sub._coap._udp.local PTR ps_1._presence._sub._coap._udp.local
f6.mf.r13._presence._sub._coap._udp.local PTR ps_2._presence._sub._coap._udp.local
 f1.mf.r1._presence._sub._coap._udp.local PTR ps_3._presence._sub._coap._udp.local
```

Now for each building hierarchical level (floor, building, etc) we need to create pointer records that point to these most fully-descriptive records.

```
f6.mf._presence._sub._coap._udp.local PTR f6.mf.r80._presence._sub._coap._udp.local
f6.mf._presence._sub._coap._udp.local PTR f6.mf.r13._presence._sub._coap._udp.local
f1.mf._presence._sub._coap._udp.local PTR f1.mf.r1._presence._sub._coap._udp.local
```

As we know that all sensors are located in same building, we don't need to create any more records. Now for the application to discover all sensors located on floor 6, the following request is needed:

```
f6.mf._presence._sub._coap._udp.local PTR
```

As a response, two records with the full nodes context arrive. Now the application knows exact rooms sensors are located in. It makes another request to obtain service names. After the response arrived, the application needs only to make SRV request with obtained service names. The message flow for this process is represented by Figure B.2.

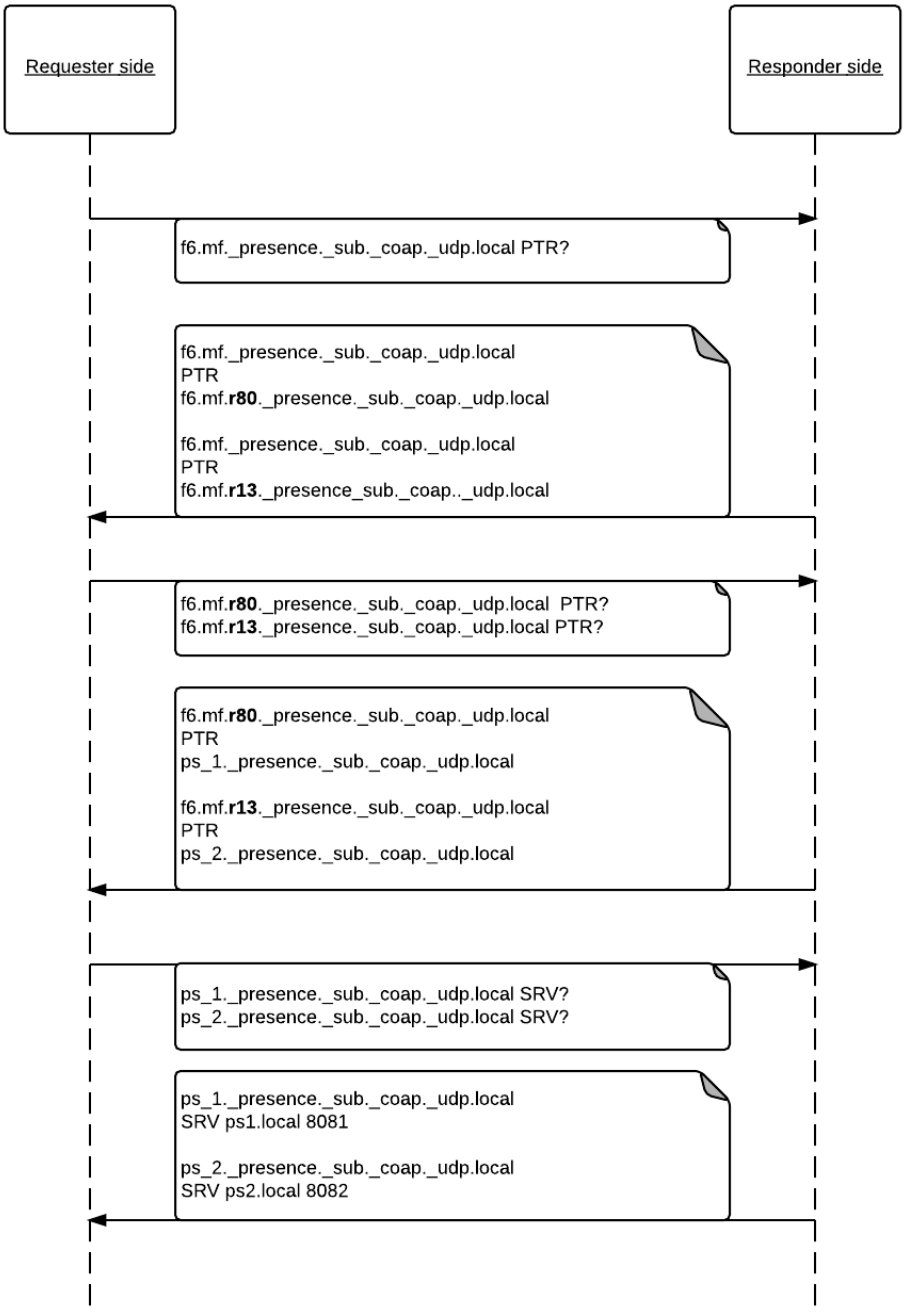


Figure B.2: Sequence diagram for the Fire Department scenario.

B.1.3 Climate control

The function of the application in this example is to control average temperature per area by measuring it and controlling HVAC devices. To measure average temperature in

certain area, we need do discover all thermometers and HVAC devices in this area. We adopt **Conjunctions in PTR** naming scheme for these sensors.

Services for temperature sensors are encoded in DNS-SD as follows:

Suppose we have four sensors in two different rooms, and for each of these rooms there is one climate control device working with TCP protocol, located next door. Locations and context tags for these devices are listed in the table B.3.

Device	**DNS-SD SRV records**	**Location**	**Tags**
Thermometer 1	`ts_1._temp._sub._coap._udp.local SRV ts1.local 80`	MF building, floor 6, room 80	`mf`, `f6`, `r80`
Thermometer 2	`ts_2._temp._sub._coap._udp.local SRV ts2.local 80`	MF building, floor 6, room 80	`mf`, `f6`, `r80`
HVAC 1	`cond_1._hvac._sub._coap._udp.local SRV hvac1.local 80`	MF building, floor 6, room 79	`mf`, `f6`, `r79`
Thermometer 3	`ts_3._temp._sub._coap._udp.local SRV ts2.local 80`	MF building, floor 3, room 14	`mf`, `f3`, `r14`
Thermometer 4	`ts_4._temp._sub._coap._udp.local SRV ts2.local 80`	MF building, floor 3, room 14	`mf`, `f3`, `r14`
HVAC 2	`cond_2._hvac._sub._coap._udp.local SRV hvac2.local 81`	MF building, floor 3, room 15	`mf`, `f3`, `r15`

Table B.3: Sensors, actuators and their context for the climate control scenario.

Now to encode context we need to create one pointer containing full set of context tags for each sensor. Following are examples of such records.

```
f6.mf.r80._temp._sub._coap._udp.local PTR ts_1._temp._sub._coap._udp.local
f6.mf.r80._temp._sub._coap._udp.local PTR ts_2._temp._sub._coap._udp.local
f6.mf.r79._hvac._sub._coap._udp.local PTR hvac_1._hvac_sub._coap._udp.local

f3.mf.r14._temp._sub._coap._udp.local PTR ts_3._temp._sub._coap._udp.local
f3.mf.r14._temp._sub._coap._udp.local PTR ts_4._temp._sub._coap._udp.local
f3.mf.r15._hvac._sub._coap._udp.local PTR hvac_2._hvac_sub._coap._udp.local
```

Suppose the application wants to discover all temperature sensors in the room 14 on the floor 3. The application makes a request with the following DNS question:

```
f3.mf.r14._temp._sub._coap._udp.local PTR
```

The application receives a list of such service names as a response.

```
f3.mf.r14._temp._sub._coap._udp.local PTR ts_3._temp._sub._coap._udp.local
f3.mf.r14._temp._sub._coap._udp.local PTR ts_4._temp._sub._coap._udp.local
```

Now the application can query for received service names and obtain SRV records.

If measures show that the temperature needs to be corrected, the application may search for climate control devices in the given room or in one of the neighboring rooms.

```
f6.mf.r79._hvac._sub._coap._udp.local PTR
f6.mf.r80._hvac._sub._coap._udp.local PTR
f6.mf.r81._hvac._sub._coap._udp.local PTR
```

The node that is aware of HVAC service in room 79 will return the following record:

```
f6.mf.r79._hvac._sub._coap._udp.local PTR hvac_1._hvac_sub._coap._udp.local
```

After resolving returned name to SRV record and service name, the application gains access to climate control service close to the area with unsatisfying conditions.

B.1.4 Comparison with alternative approach

Let us have a look at an alternative approach to building control location encoding described in [21]. This approach assumes creating pointers with different sets of context properties to individual service names. For example, for light actuators in office 80 on floor 6 from **Energy saving** scenario, the following records are created:

```
_light._sub._coap._udp.80.f6.mf PTR l1._light._sub._coap._udp.local
                    80.f6.mf PTR l1._light._sub._coap._udp.local
```

Obviously, these records allow to query devices only by their full location, including room number. To enable more general queries, like queries by floor, the authors propose creating named network groups of devices. For example, the following domain names can resolve to network groups, containing all devices on specific floor:

```
all.f7.mf
all.f6.mf
all.f5.mf
    ...
all.f1.mf
```

In a similar way groups that contain only lights can be created; domain name like `all-lights.f7.mf.tue.example.com` may be given to such group.

To make these groups discoverable special SRV and PTR records are created. In our example with lights on floor 7 these will be following:

```
_all_light.f7.mf SRV all-lights.f7.mf.tue.nl 8080
 _light._sub._coap._udp.f7.mf PTR _all_light.f7.mf
```

Energy saving To discover luminosity sensor and presence sensor in one room with this alternative approach, the application needs to send out DNS request with following questions:

```
_lumsensor._sub._coap._udp.80.f6.mf PTR
 _presence._sub._coap._udp.80.f6.mf PTR
```

As a response it will receive names for SRV lookup.

```
_lumsensor._sub._coap._udp.80.f6.mf PTR l1._light._sub._coap._udp.local
_presence._sub._coap._udp.80.f6.mf PTR ps._light._sub._coap._udp.local
```

For this scenario, the process of service discovery does not differ much from the naming scheme approach we propose, as both naming schemes contain full path to the office.

Fire department To repeat the same behavior as described above for this scenario, network operator needs to create network groups for presence sensor on each floor, in each building etc. Suppose we have such network group for floor 6 of MetaForum building named `all-presence.f6.mf.example.com`. The following records will be created to support discovery of this group:

```
_presence._sub._coap._udp.f6.mf PTR _all_presence.f6.mf
_all_presence.f6.mf SRV all-presence.f6.mf.tue.nl 8080
```

First the client makes a query to find all presence sensors on floor 6:

```
_presence._sub._coap._udp.f6.mf PTR
```

As a response it receives the name of SRV record to look up.

```
_presence._sub._coap._udp.f6.mf PTR _all_presence.f6.mf
```

After SRV lookup, client receives the group name.

```
_all_presence.f6.mf SRV
_all_presence.f6.mf SRV all-presence.f6.mf.tue.nl 8080
```

Now the client sends a message to the group members to use their services. Note that in this case contexts of individual sensors remain unknown. This means, that if in the case of fire one of sensors indicates people presence, the application will need to use IP mapping or other additional techniques to locate this sensor. Our approach, on the other hand, allows to obtain the full context of sensors right during the service discovery.

Climate control For a climate control scenario, we may need to create groups for temperature sensors. However, for HVAC devices it is not necessary.

Device group for temperature sensors requires following records:

```
_temp._sub._coap._udp.80.f6.mf PTR _all_temp.80.f6.mf
_all_temp.80.f6.mf SRV all-temp.80.f6.mf.tue.nl 8080
```

The climate control service in room 79 needs following records:

```
_hvac._sub._coap._udp.79.f6.mf PTR hvac1._hvac._sub._coap._udp.local
                    79.f6.mf PTR hvac1._hvac._sub._coap._udp.local
```

Resolving of group name is done in the same way as in previous example. After the data has been collected, the application may need to find a climate control service in nearby rooms:

```
                                    _hvac._sub._coap._udp.79.f6.mf PTR
                                    _hvac._sub._coap._udp.80.f6.mf PTR
                                    _hvac._sub._coap._udp.81.f6.mf PTR

_hvac._sub._coap._udp.79.f6.mf PTR hvac1._hvac._sub._coap._udp.local
```

The received name is then resolved to SRV record as described above.

B.2 SenSafety: festival

B.2.1 Scenario description

A network of wireless sensors is deployed in a festival environment. This can be the place where a festival, sport event or parade takes place. Such crowded event region can be secured by the network of wireless sensors that are used by applications to detect (potentially) dangerous situations and inform personnel about them.

Sensors There are two types of sensors: ambient, i.e. part of the SenSafety infrastructure, and opportunistic sensors from the visitors' smartphones. Ambient sensors that participate in this example are pressure sensors on the fence around the area that are able to detect a breach, and security cameras. User smartphones may also act as cameras, and also they can be used by users to notify the personnel on some unusual events taking place. These features may be enabled if the user agrees to install the special festival application on her device.

We are interested in the Security check application, that counts the number of visitors in a given area, and Perimeter breach application, that detects disturbance in the fence using a pressure sensor, and uses the appropriate camera(s) to spot the intruder.

B.2.2 Location representation

This scenario assumes that the festival is taking place on a certain area, fenced off from outer world. In this case, it would be convenient to break the whole terrain into squares or regions of the size appropriate for the specific application. The device location will be then determined by the number of square or region it resides at. This approach can often be seen on the maps.

Another alternative would be to use GPS coordinates. But this makes discovery more tricky. An application that needs all devices around some point, or just "nearby" and has no prior knowledge of devices location has no means to express this request in one of naming schemes, as the discovery is only done by the exact combination of context tags, and the application does not know exact coordinates of services. The solution might be to use less precise GPS coordinates, that correspond to the area rather than the point. The size of the area is again to be justified by specific terrain and application needs.

In both cases there are security cameras involved. There are a lot of optional parameters and features that may or may not be supported by cameras, such as focus distance and the ability to rotate. For the sake of simplicity here we assume that all security cameras can only face one direction, and, possibly, different focus distances.

For our purposes we will require cameras that can record some specific point on the terrain; there is no use for applications in camera parameters as such. Hence in addition to the physical location for each camera we can encode the direction it is facing. We can represent this direction as a cardinal direction (south, north, east, west) with a special context tag. Of course, the certain level of precision could be chosen: for example, we decide to encode direction with four possible ordinal directions and four intermediate directions, i.e. to values like NW for north-west or SE for south-east. For the camera, facing north-east this context tag may be the following: `direction_NE`. If the angle of the camera is broad enough, we can apply several of these context tags: for instance, `directio_NE` and `direction_N`.

However, from this information it is not easy to tell if the camera is able to record the objects in a specific location. Suppose there is a static object in the middle of the room. If the camera is hanging on the north wall of the room, it should have the context

tag `direction_S` to be selected for recording the object. However, if the camera is on the south wall, the direction it should face is exactly opposite. Therefore to discover both cameras that can see the object, creation of the complex request consisting of two parts is required:

```
(direction_S AND location_NorthWall) OR (direction_N AND location_SouthWall)
```

If some of cameras cannot see the object because something is blocking the view, additional terms may have to be introduced.

We can also encode the range and direction of the camera by specifying sectors it can observe, for example, with the tag starting with "`CanSee_`" This may not be convenient for square sectors, but in camera-centric applications the space may be logically divided in such a way that for each camera there is an area that can be well observed with it. This approach allows for an easier service discovery requests composition than the previous one. The application only needs to know the location that cameras have to point at. If some object is preventing the camera from seeing the location, then the camera service simply should not be tagged with the `CanSee` tag for this location.

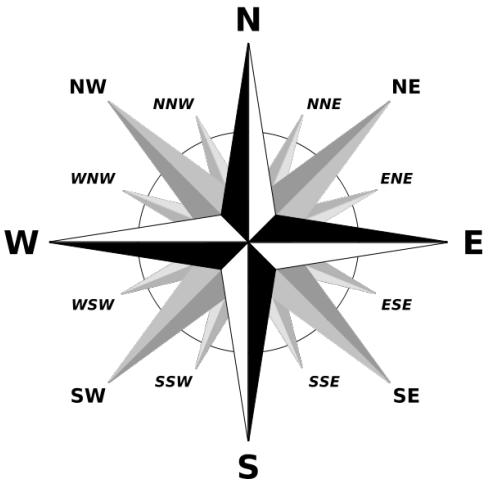


Figure B.3: The compass rose with ordinal, intermediate and further divisions.

B.2.3 Security check

In this scenario the application is required to find all festival visitors in a certain area. This task might be also useful for detecting the crowded areas that need manual regulation by the personnel.

While this task is quite simple if all users have a special festival application on their smartphones, this condition will probably never hold. There are many reasons people might prefer not to install or use the application, or they may just not own the compatible device.

Approaches To find the number of people in specific area automatically is quite a complicated task by itself, leave alone the automatic identification of each person. We present several approaches to this task that use wireless sensing devices.

- **Face recognition with cameras.** If the powerful face recognition software and high-definition cameras are available, the application that recognizes and counts faces may be built. To discover all users in the specific location, such application should discover cameras aimed at this location and process images from these cameras, recognizing all faces. The application can use a simple algorithm that just spots the face-like objects for people counting, or the full face recognition algorithm that recognizes different faces and may count them.

- **Heat detection.** Thermographic cameras can be used to detect heat from bodies and other objects. These observations may help to estimate the number of people in the area by recognizing spots of higher temperature and of specific shape and counting them.

  Unfortunately, this technology depends on the assumption that human body temperature is higher than the environment. Hence its applicability depends on the climatic conditions. Also it does not allow to discover and distinguish individual users.

- **Smart cloth.** If the budget and the area of the festival allow that, the smart cloth that detects pressure may be placed on the floor. Then spots of high pressure will correspond to persons.

  The drawback of this technique is that the people in certain area cannot be identified.

For this scenario we will employ face recognition technique. For that we need to discover all cameras in specific area. We will employ less precise GPS coordinates to address location and cardinal directions to encode orientation of cameras.

Our illustrative face recognition application needs to use the images from two opposite nearby cameras in order to count the number of people in the area. Knowing that, we adopt the **Formula in PTR** scheme, as we need to query for binary formulas that have both disjunction and conjunction operations. Conjunctions are represented by hyphen signs, disjunctions as label divisors. As an alternative approach, we adopt **Conjunctions in PTR** scheme with Dynamic tags optimisation, as it allows for complex formulas too and has the second best result for the network efficiency.

To encode GPS coordinates we introduce two types of context tags. Latitude values are prefixed with "`LA`" and longitude with "`LO`". The decimal part of the value is separated by the underscore. Directions are prefixed with the character "`d`".

Suppose the alert from the user smartphone has arrived. The alert contains GPS coordinates: `52.068769,5.161695`. The application rounds coordinates up to `52.068770,5.161700` and drops the last digit of each coordinate to obtain the area to search for cameras. Then it starts searching for opposite facing cameras in the nearby area. It chooses to search for north- and south-facing cameras. The application composes the following binary formula to address its context requirements:

$$([Latitude = 52.06877]\&[Longitude = 5.16170]\&[Direction = S])\|$$
$$([Latitude = 52.06877]\&[Longitude = 5.16170]\&[Direction = N]).$$

Formula in PTR approach The application converts the formula to the form that allows it to be put in PTR name. As a result the following PTR question is sent:

```
dN-LA_52-06877-LO_5-16170.dS-LA_52-06877-LO_5-16170._cam._sub._coap._udp.local PTR
```

In response the application receives set of one or more resource records pointing at cameras service names:

```
dN-LA_52-06877-LO_5-16170.dS-LA_52-06877-LO_5-16170._cam._sub._coap._udp.local PTR
Cam01._cam._sub._coap._udp.local
...
dN-LA_52-06877-LO_5-16170.dS-LA_52-06877-LO_5-16170._cam._sub._coap._udp.local PTR
CamNN._cam._sub._coap._udp.local
```

Note that this set should not necessarily contain cameras with both directions. If there is no such pairs in the region, the application may try to search for west- and east-facing cameras instead. However, from the returned records the application cannot see if there are pairs of opposite-facing cameras.

Conjunctions in PTR approach To express the same formula in this naming scheme, the application will require two DNS questions. The following DNS request is being sent:

```
dN.LA_52-06877.LO_5-16170._cam._sub._coap._udp.local PTR
dS-LA_52-06877-LO_5-16170._cam._sub._coap._udp.local PTR
```

Responses, apart from service record names, contain the information of actual orientation of the camera: a PTR record has either the name starting with `dN` (north-facing camera), or `dS` (south-facing camera).

```
dN.LA_52-06877.LO_5-16170._cam._sub._coap._udp.local PTR Cam01._cam._sub._coap._udp.local
                                                                                        ...
dS-LA_52-06877-LO_5-16170._cam._sub._coap._udp.local PTR CamNN._cam._sub._coap._udp.local
```

The drawback of this scheme is that two times more questions have to be sent by client in order to discover cameras.

B.2.4 Perimeter breach

This scenario is initiated when pressure sensor installed on the fence detects the disturbance. The application needs to discover cameras that are located nearby the initiating sensor and can record the area around it.

For this scenario we choose to divide the terrain into squared logical areas to address the location. The direction of cameras is determined by squares of that cameras can see more than a half.

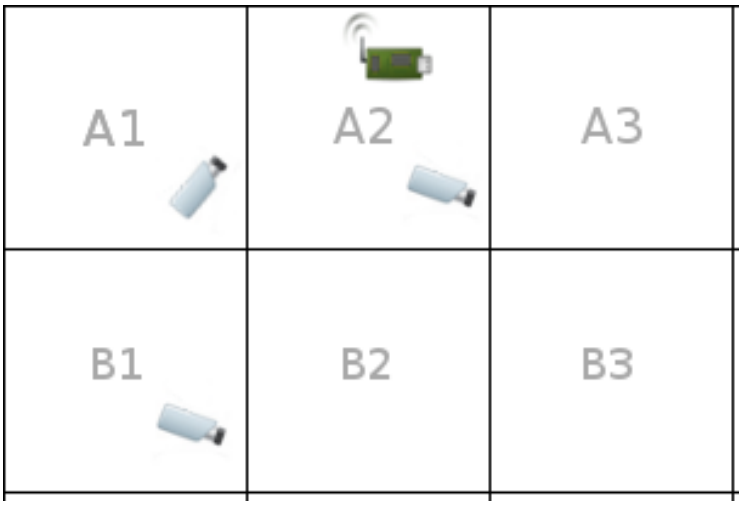


Figure B.4: The scheme of terrain logical division and devices location. The pressure sensor is located in square A2.

Suppose the initiating sensor is located on the north section of the fence in the square named A2. There are cameras in squares A1, A2 and B2, located and rotated as shown on figure B.4.

We would like to discover camera(s) that can help us record the phenomena that caused the pressure sensor to go off. We assume that the application already knows the location of the pressure sensor that generated an alert. To find what happened, we search for all cameras that can see the square A2. In our example, only the camera in the square A1 is facing the needed direction.

Following are the participating devices and their services.

Device	**DNS-SD SRV records**
Pressure sensor in the square A2	`PressSensor01._press._sub._coap._udp.local SRV ps01.local 80`
Camera in the square A1	`Cam01._cam._sub._coap._udp.local SRV cam01.local 8080`
Camera in the square A2	`Cam02._cam._sub._coap._udp.local SRV cam02.local 8080`
Camera in the square B1	`Cam03._cam._sub._coap._udp.local SRV cam03.local 8080`

Table B.4: Service records for the perimeter breach scenario.

Scenario context tags For each camera we include the information about squares it can observe and the square it resides at.

The location is encoded as the context tag starting with the string "`Loc_`", for example, `Loc_A2`.

The direction is encoded as the context tag starting with the string "`CanSee_`", for example, `CanSee_A1`.

As a result, we are going to have following context tags for participating devices:

Device	Context tags
Pressure sensor in the square A2	Loc_A2
Camera in the square A1	CanSee_A2, Loc_A1
Camera in the square A2	CanSee_B3, Loc_A2
Camera in the square B1	Loc_B3

Table B.5: Context tags for the perimeter breach scenario.

Tag to PTR approach For this application we require filtering by one tag type only (the camera visibility area). Therefore **Tag to PTR** naming scheme (3.2.2.2) is sufficient.

To include context tags described above we create the following set of DNS pointers:

```
Loc_A2._press._sub._coap._udp.local PTR PressSensor01._press._sub._coap._udp.local


   Loc_A1._cam._sub._coap._udp.local PTR Cam01._cam._sub._coap._udp.local
CanSee_A2._cam._sub._coap._udp.local PTR Cam01._cam._sub._coap._udp.local

   Loc_A2._cam._sub._coap._udp.local PTR Cam02._cam._sub._coap._udp.local
CanSee_B3._cam._sub._coap._udp.local PTR Cam02._cam._sub._coap._udp.local


Loc_B3._cam._sub._coap._udp.local PTR Cam03._cam._sub._coap._udp.local
```

Now for the application to discover the camera that can see this sensor, the following DNS request has to be made:

```
CanSee_A2._cam._sub._coap._udp.local
```

The name for the required camera is then returned:

```
CanSee_A2._cam._sub._coap._udp.local PTR Cam01._cam._sub._coap._udp.local
```

Then the client searches for SRV record with this name and finds the entry point for he service.

```
Cam01._cam._sub._coap._udp.local SRV cam01.local 8080
```

Nested tags in PTR approach The alternative approach allows us to discover locations of cameras apart from their visibility area. With that information, the application may want to filter out some results after discovery phase, to select the camera closest to the breach.

Unfortunately, the given scheme requires quite a lot of storage (see section 4.1.1). To cope this, we can employ the Dynamic PTR creation optimisation. It has an additional advantage for the **Nested tags in PTR** scheme, as the processing of requests allows to manipulate the data being transmitted, and not just reply with whatever is saved in DNS records storage. This way the responder side may send the pointer containing the service name together with the pointer containing full service context, thus avoiding additional network load. Further we assume that the optimisation is applied and both pointers are sent together.

The application that needs to find a camera makes the following PTR request:

```
CanSee_A2._cam._sub._coap._udp.local PTR
```

In response it receives the name of the service and its full context:

```
CanSee_A2._cam._sub._coap._udp.local PTR CanSee_A2.Loc_A1_cam._sub._coap._udp.local
CanSee_A2.Loc_A1_cam._sub._coap._udp.local PTR Cam01._cam._sub._coap._udp.local
```

Now the application can use this information to choose the best camera (in our example we only have one though) and request for its SRV record.

Bibliography

[1] S. Cheshire and M. Krochmal, *DNS-Based Service Discovery*, Febrary 2013. IETF, RFC6763.

[2] "Apple bonjour." `http://developer.apple.com/bonjour/`. Accessed: 2014-03-16.

[3] "Avahi." `http://avahi.org/`. Accessed: 2014-02-17.

[4] "Sensafety project." `http://sensafety.nl/`. The SenSafety consortium. Accessed: 2014-02-17.

[5] C. M. U. D. of Electrical and C. Engineering, "Real-time wireless sensor network platform." `http://www.ece.cmu.edu/firefly/index.html`. Accessed: 2014-08-07.

[6] "Unexpected giant exoplanet discovered." `http://www.iflscience.com/space/unexpected-giant-exoplanet-discovered`. Accessed: 2014-05-26.

[7] I. F. Akyildiz and M. C. Vuran, *Wireless sensor networks*, vol. 4. John Wiley & Sons, 2010.

[8] H. He, "What is service-oriented architecture," *Publicação eletrônica em*, vol. 30, pp. 1–5, 2003.

[9] R. Chinnici, J. Moreau, A. Ryman, and S. Weerawarana, "Web services description language (wsdl)." `http://www.w3.org/TR/wsdl20/`, June 2007. Version 2.0. Accessed: 2014-06-21.

[10] "Information technology – upnp device architecture – part 1: Upnp device architecture version 1.0," ISO 29341-1:2011, International Organization for Standardization, Geneva, Switzerland, 2011.

[11] B. Orlic, I. David, R. Mak, and J. Lukkien, "Dynamically reconfigurable resource-aware component framework: Architecture and concepts," in *Software Architecture* (I. Crnkovic, V. Gruhn, and M. Book, eds.), vol. 6903 of *Lecture Notes in Computer Science*, pp. 212–215, Springer Berlin Heidelberg, 2011.

[12] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M.-C. Shan, "Adaptive and dynamic service composition in eFlow," in *Advanced Information Systems Engineering*, pp. 13–31, Springer, 2000.

[13] Z. Shelby, K. Hartke, and C. Bormann, "Constrained application protocol (coap)," *CoRE Working Group*, June 2013. Internet-Draft.

[14] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol–http/1.1," 1999.

[15] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey," *Computer networks*, vol. 38, no. 4, pp. 393–422, 2002.

[16] P. V. Mockapetris, "Domain names-concepts and facilities," *RFC1034*, November 1987.

[17] S. Cheshire and M. Krochmal, *Multicast DNS*, Febrary 2013. IETF, RFC6762.

[18] "COSMIC Method v4.0 Introduction to COSMIC ." `http://www.cosmicon.com/portal/public/Introductionv4.0.pdf`. Accessed: 2014-05-26.

[19] M. Mealling and R. Daniel, *The Naming Authority Pointer (NAPTR) DNS Resource Record*, September 2000. IETF, RFC2915.

[20] C. Davis, P. Vixie, T. Goodwin, and I. Dickinson, *A Means for Expressing Location Information in the Domain Name System*, January 1996. IETF, RFC1876.

[21] P. van der Stok, K. Lynn, and A. Brandt, "Coap utilization for building control," *CoRE*, July 2012. Internet-Draft.

[22] A. Aggarwal, "Optimizing dns-sd query using txt records," July 2014. Internet-Draft.

[23] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley, "The design and implementation of an intentional naming system," *ACM SIGOPS Operating Systems Review*, vol. 33, no. 5, pp. 186–201, 1999.

[24] G. Bovet and J. Hennebert, "A distributed web-based naming system for smart buildings," *The third IoT-SoS workshop*, June 2014.

[25] T. A. Butt, I. Phillips, L. Guan, and G. Oikonomou, "Trendy: an adaptive and context-aware service discovery protocol for 6lowpans," in *Proceedings of the Third International Workshop on the Web of Things*, p. 2, ACM, 2012.

[26] C. Perera, A. Zaslavsky, C. Liu, M. Compton, P. Christen, and D. Georgakopoulos, "Sensor search techniques for sensing as a service architecture for the internet of things," *IEEE SENSORS JOURNAL*, vol. 14, Febrary 2014.

[27] A. Martinez, S. Schoenig, D. Andresen, and S. Warren, "Ingestible pill for heart rate and core temperature measurement in cattle," in *Engineering in Medicine and Biology Society, 2006. EMBS'06. 28th Annual International Conference of the IEEE*, pp. 3190–3193, IEEE, 2006.

[28] K. Channabasavaiah, K. Holley, and E. Tuggle, "Migrating to a service-oriented architecture," *IBM DeveloperWorks*, vol. 16, 2003.

[29] M. P. Papazoglou, "Service-oriented computing: Concepts, characteristics and directions," in *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pp. 3–12, IEEE, 2003.

[30] J. Rao and X. Su, "A survey of automated web service composition methods," in *Semantic Web Services and Web Process Composition*, pp. 43–54, Springer, 2005.

[31] J. Klensin and M. Padlipsky, *Unicode Format for Network Interchange*. Apple Inc., March 2008. IETF, RFC5198.

[32] DNS-SD.org, *DNS SRV (RFC 2782) Service Types*. Available at http://www.dns-sd.org/ServiceTypes.html.

[33] "JmDNS." `http://jmdns.sourceforge.net/`. Accessed: 2014-06-11.

[34] "Bonjour/zeroconf with arduino." `http://gkaindl.com/software/arduino-ethernet/bonjour`. Accessed: 2014-07-29.

[35] "mdns/dns-sd for contiki." `https://github.com/mstolikj/contiki/`. Accessed: 2014-06-11.

[36] "mdns library." `https://github.com/svinota/mdns`. Accessed: 2014-06-11.

[37] E. Mendelson, *Introduction to Mathematical Logic, Fourth Edition*. Discrete Mathematics and Its Applications, Taylor & Francis, 1997.

[38] P. V. Mockapetris, "Domain names-implementation and specification," *RFC1035*, November 1987.

[39] J. Postel, "User datagram protocol," *RFC768*, August 1980. Internet standard.

[40] "Wireshark." `http://www.wireshark.org/`. Accessed: 2014-06-17.

[41] M. Abramowitz, I. A. Stegun, *et al.*, *Handbook of mathematical functions*, vol. 1. Dover New York, 1972.

[42] "Dns-sd extensions website." `http://dnssdext.net`. Accessed: 2014-07-13.

[43] Wolfram Alpha LLC—A Wolfram Research Company, "Computational knowledge engine." `http://www.wolframalpha.com/`. Accessed: 2014-10-06.

[44] S. Luke, *Essentials of Metaheuristics*. Lulu, second ed., 2013. Available at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[45] A. Gulbrandsen, P. Vixie, and L. Esibov, *A DNS RR for specifying the location of services (DNS SRV)*, Febrary 2000. IETF, RFC2782.

[46] S. H. Chauhdary, M. Y. Cui, J. H. Kim, A. K. Bashir, and M.-S. Park, "A context-aware service discovery consideration in 6lowpan," in *Convergence and Hybrid Information Technology, 2008. ICCIT'08. Third International Conference on*, vol. 1, pp. 21–26, IEEE, 2008.

[47] W. Dargie and C. Poellabauer, *Fundamentals of wireless sensor networks: theory and practice*. John Wiley & Sons, 2010.