Computer Science and Information Systems 15(3):751–774 https://doi.org/10.2298/CSIS180131030B

A Novel Distributed Registry Approach for Efficient and Resilient Service Discovery in Megascale Distributed Systems*

Lars Braubach[1] and Kai Jander[2] Alexander Pokahr[3]

[1] City University of Applied Sciences Bremen
ZIMT
Flughafenallee 10
28199 Bremen
lars.braubach@hs-bremen.de
[2] University of Hamburg
Distributed Systems and Information Systems, Hamburg, Germany
jander@informatik.uni-hamburg.de
[3] Helmut-Schmidt-University / University of the Bundeswehr Hamburg
Industrial Data Processing and Systems Analysis Group, Hamburg, Germany
pokahr@hsu-hh.de

Abstract. Service discovery is a well-known but important aspect of dynamic service-based systems, which is rather unsolved for megascale systems with a huge number of dynamically appearing and vanishing service providers. In this paper first requirements for service discovery in megascale systems are identified from three perspectives: provider, client and system architecture side. Existing approaches are evaluated along these lines and it becomes apparent that modern solutions make advances with respect to the distributed system architecture but fail to support many aspects of client side requirements like persistent queries and elaborate query definitions. Based on these shortcomings a novel solution architecture is presented. It is based on the idea that service description data can be subdivided into static and dynamic properties. The first group remains constant over time while the second is valid only for shorter durations and has to be updated. Expressive service queries rely on both, e.g. service location as example for the first and response time for the latter category. In order to deal with this problem, our main idea is to also subdivide the architecture into two interconnected processing levels that work independently on static and dynamic query parts. Both processing levels consist of interconnected peers allowing to auto-scale the registry dynamically according to the current workload. The implementation using the Jadex middleware is explained and the approach is empirically evaluated using an example scenario.

Keywords: Service Discovery, Services, SOA, Cloud, Megascale, Jadex

1. Introduction

In recent years, many types of distributed applications have become increasingly large-scale. This trend is primarily driven by the number of users accessing such applications

* This paper is extended and revised version of "Service Discovery in Megascale Distributed Systems" published in IDC 2017 [9].

but complexity and intelligence of the applications themselves are driving factors as well. While basic applications with thousands of users can be scaled easily, applications such as Facebook, Twitter and eBay have millions to hundreds of millions of simultaneous users and the functionality of the applications also allow a high degree of interaction between them.

This means that such distributed systems require a high level of scalability. While scalability of algorithms has always been a focus for research, even carefully designed system are often limited in practice when scaling to such large scale applications, often requiring the developing company to optimize the existing software for higher scalability, despite careful previous design.[4] Despite the efforts to produce scalable code, actual scalability of the complete system often ends up being more limited than expected. This is due to the fact that complex systems often contain easy to miss small parts that are not suitable for scaling beyond a certain limit. These parts end up as the bottlenecks for the whole system once this limit has been reached (cf. [2]).

Current systems primarily deal with large numbers of users using relatively simple services. In this type of application, scalability can be managed using simple architectures (client-server models, massively replicated server instances) that reduce the risk of introducing poorly scaling components. For example, while a large number of users interact with systems like Facebook, the number of services offered to the user is actually below 10^3 or lower. In such systems, service providers are usually directly included, e.g., as REST-URLs in the client-side software package. This approach works well on centralized systems where users interact with a limited set of server-based service and do not directly interact with each other. However, such an approach is not suitable for every type of application. Systems like smart homes and smart cities demand considerable autonomy between components. Enabling components and users to directly interact with each other without a central mediating component means that each part in the system has to offer services for use by other parts. In such a system, finding the right service for interaction can be a challenge to the components of the system. Of course, also other aspects like e.g. the trust of service providers are important for large open systems but are not discussed in this paper, see. e.g. [11].

This paper will specifically look at the practical implications of service discovery in systems with a large number of services, in particular in the range of 10^6 up to 10^9 simultaneous services providers. It can also be expected that the services appear and disappear dynamically at any time. Example application areas for these type of systems are as follows:

- Increased automation in homes (“smart home”) and cities (“smart cities”) result in large sets of smart components dealing with aspects of the automation. While a centralized approach may be feasible for homes, increased scales in case of cities as well as privacy concerns would suggest give advantages to a peer-to-peer-based approach.
- In some cases, decisions have to be made locally and communication being used opportunistically to enhance capabilities. For example, autonomous vehicles need to be able to act safely even if communication is interrupted but should be capable to leverage communication to enhance its capabilities by maintaining distance to other vehicles, avoiding heavy traffic and adjusting speed to match upcoming traffic lights.

[4] E.g., `http://highscalability.com/blog/2016/1/11/a-beginners-guide-to-scaling-to-11-million-users-on-amazons.html`

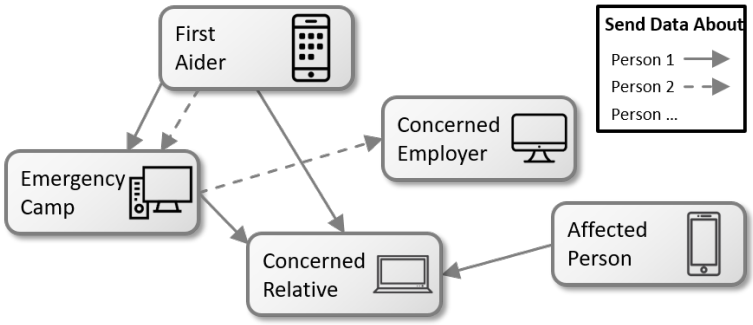

Fig. 1. Helpline application scenario

- Even in the case of existing centralized megascale applications, increasing application complexity encourages the use of an internal peer-to-peer model like microservices [14]. In contrast to more traditional architectures such as the three-tier architecture with presentation layer, business layer and data layer, the application consists of small, integrated services calling each other.
- Wearable devices become increasingly available and affordable. A typical use case consist of monitoring daily, sport, health states e.g. using smart watches. Currently, these devices act rather isolated and in many cases only communicate with the users smartphone. Future applications could benefit from enhanced interaction possibilities with other devices in the environment allowing the combination and processing of data. The described scenario involves very dynamic device and service (dis)appearance and could involve large number of devices e.g. in group events like festivals or fairs.
- Production systems in the Industry 4.0 era consist of huge numbers of (intelligent) workpieces and machines. The dimension of these scenarios is further increased if also interactions between multiple production sites are considered as e.g. proposed in [8].

In the following, we first identify requirements for service discovery in megascale distributed systems (Section 3). In Section 4, existing approaches to service discovery are analyzed and a new solution architecture is presented in Section 5. An empirical evaluation of the approach is illustrated in Section 6. The paper concludes with a summary and an outlook in Section 7.

2. Example Scenario

As a motivating example, we adapt the "Helpline" application scenario introduced in [6]. The helpline application can be seen as part of a "smart cities" environment and is a decentralized system for exchanging information about potentially affected people during large scale incidents, e.g. natural disasters such as earthquakes or floods. The main goal is to provide basic information about people's whereabouts to concerned parties such as relatives or employers.

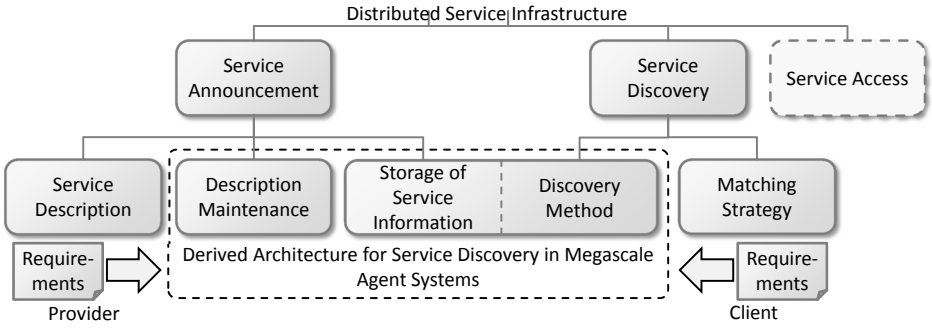

Fig. 2. Deriving requirements for a service discovery (upper part adapted from [21])

A sketch of the system architecture is shown in Fig. 1. In this architecture, a "Helpline App" may be installed on an arbitrarily large number of nodes. These nodes then use distributed lookup mechanisms to find other nodes that offer or request information about certain persons. For example, a victim (*Person 1*) might get treated by a *first aider*, which notes the name of the victim and maybe some medical data in its PDA. The victim gets moved to an *emergency camp*, which receives related information from the PDA of the first aider. In the meantime, a *concerned relative* requests information about the victim, which can be received directly from the first aiders PDA and later also from the emergency camp. Due to privacy issues, the relative will only receive general information, but no medical details. When the victim has recovered a little, he or she might also post information (*"Don't worry! I'm fine."*), which is automatically distributed to all parties that have stated their interest in this potentially *affected person*.

Regarding a single person, some of the nodes only act as information producers (e.g. first aiders or the person her/himself). Other nodes may only consume information about a specific person (concerned party) or consume and produce information (e.g. emergency camp). The architecture allows two parallel models of information exchange. 1) a *pull model*, i.e. consumers that search and query nodes offering information about a person and 2) a *push model*, with producers pro-actively contacting nodes that are interested in specific information. For distributed lookup, in both models the available nodes need to be matched against the names or descriptions of persons, for which they offer or request information.

Given that naturals disasters can affect thousands of people and that also many people might be concerned, even when relatives are not affected, it is not that hard to imagine a system like this to reach megascale.

3. Requirements

Any kind of distributed service infrastructure, like in the helpline scenario described above, requires mechanisms for the three fundamental steps: *announce*, *discover*, and *access* (cf. e.g. [21]). Whereas the last step - service access - is concerned with the technical details of service invocation, the focus of this paper is on the first two steps: announce-

ment and discovery. As shown in Fig. 2, [21][5] consider *service description*, *description maintenance* and *storage of service information* as relevant components of the service announcement part. Service discovery is subdivided into *discovery method* and *matching strategy*. In the following, the requirements for service discovery in large-scale distributed systems are derived by considering these components from a use-case-driven view.

The service description contains information about a service and is made available by the service provider. Matching strategy on the other hand is based on the information provided by the service client, i.e. information that is used to decide, if any given service description matches the criteria that are relevant to the client. We consider these two aspects as input to the service discovery system and thus as requirements for an appropriate architecture (cf. Fig. 2). Description maintenance (e.g. lease mechanisms), as well as the highly interdependent storage of service information and discovery method on the other hand are internal aspects of the discovery system, that are not directly perceived by either service provider or service client. Thus, these components are subject to architectural choices, which are based on the requirements derived from the service provider and service client side.

3.1. Provider-Side Requirements

Service providers want to be found by potential clients and delegate this responsibility to the discovery infrastructure. To support providers in a megascale system, the following requirements apply to the discovery infrastructure:

Many Providers In contrast to the state of the art, a megascale distributed service infrastructure should not only support millions of service clients, but also *millions of service providers* on (potentially) millions of nodes.

Auto Maintenance The service infrastructure should provide mechanisms for maintaining the consistency of descriptions of available services, even in case of failures (e.g. when service providers are unable to deregister their description due to network or node failures).

The helpline application e.g. has *many providers*, because all nodes need to act as service clients and service providers to support flexible push and pull information exchange. *Auto maintenance* is essential, as network disruptions are to be expected in case of natural disasters

3.2. Client-Side Requirements

Service clients access the discovery infrastructure for obtaining information about service providers that match specific needs of the client. Thus, the client-side requirements are derived from use cases with regard to service search specifications. To cover a broad range of use cases, the content of a search request is classified into different dimensions in the following. The first three dimensions are concerned with the expressivity of the search specification. Two other dimensions consider the life time of a search request, as well as quality expectations by the client:

[5] We use different terms compared to [21] in an attempt for more concise terminology.

Service Matching The search specification should allow referring to both *functional and non-functional properties* for expressing the needs of the client. Functional properties describe the general applicability of a service to a specific need, whereas non-functional properties allow capturing conditions describing how the service should be provided (e.g. costs, security constraints, or response times). *Exact matching* of properties can be based on known service type names or, e.g., tags. *Semantic matching* uses, e.g., logic-based reasoning techniques or statistical methods. As most practical use cases prefer exact matching for reliable system behavior, we do not explicitly consider semantic matching in the remainder of this paper.[6]

Result Size Common use cases can be distinguished with regard to requiring only *one* service (e.g. for buying a product) vs. looking for *all* services of a type (e.g. in a chat application). Given that a mega scale system can potentially have a huge number of services, clients may also want to *limit* the result size.

Result Ordering Service matching can be based on *boolean criteria* as well as *gradual measures*. Boolean criteria allow quickly selecting services based on, e.g., exact matching of functional properties like a service type name, whereas gradual measures, often applied to non-functional properties, allow ordering of search results based on user-specified fitness-functions (e.g. service provider with lowest utilization).

Query Persistence A service search can be *one-shot* or *continuous*. A one-shot search will only result in currently available services, whereas a continuous search will be stored as a persistent query in the infrastructure, such that the client will receive timely updates, whenever services matching the search specification appear or disappear.

Quality Levels Some clients may require to discover *all* services (or the globally *best* with regard to a given fitness function), whereas other clients just need *some* services. Specifying a quality level allows for optimizations inside the search system. Moreover, clients are usually interested in *result freshness*, which means that the search result should only contain services, for which a given max age has not been reached.

In the helpline scenario, *service matching* can be based on the names of persons (e.g. represented as tags in the service descriptions) and potentially also on a combination of more abstract properties, such as age, gender, height and ethnicity. A relevant non-functional property, useful for *result ordering*, is e.g. the freshness of the data, as people are generally interested in the most recent location of the person. But also technical properties should be considered, e.g. fetching data from the server of the emergency camp instead of overloading the first aider's PDA. Depending on the implemented push or pull models, *persistent queries* can be used by information consumers to get notified, when relevant data becomes first available at some other node.

3.3. Architecture Requirements

In addition to client and provider side requirements also the system architecture perspective has to be taken into account. Besides realizing the functional requirements mentioned in the last sections the system architecture has especially to consider non-functional aspects for a smooth operation of the registry. The following architecture requirements have been identified:

[6] Especially, in large-scale open environments semantic metaching becomes very important.

High scalability: The architecture has to support support massive amounts of service data from a high number of devices.

High performance: Service mediation is a fundamental functionality of each service based system, i.e. the corresponding actions - register, deregister, update and search services, have to very fast.

High availability and fault resilience: In the context of distributed systems failures can occur at any time in any component of the system including the registry. The registry layout should ensure that clients can make use of the registry at nearly any point in time but they should be able to keep functioning also in rare cases in which the registry cannot be reached. (recommendation, power rating for finding best registry)

Privacy: The registry should enable privacy of service data, i.e. it should not be possible to look up services that are intended for internal purposes or specific users only.

Zero Configuration: It should be possible to use a registry without tedious upfront configuration or deployment tasks.

4. Analysis of Service Discovery Approches

Service discovery mechanisms have originated in different research areas. Broadly one can distinguish hardware oriented service discovery, which is mainly motivated by the wish to find typical services in a local network, e.g. a print or scan service. This area has inspired discovery protocols with a focus on local area network technologies like IP/TCP multicast. Examples of this category are Jini [19]and UPnP [20]. These are not considered further, because they do not fit to the challenges of mega-scale and internetwork service environments. The second interesting area of research is centered around WSDL based web services. In this context service discovery played a central role and a lot of standards and proposals for service management emerged. Finally, practice paved the way towards the much simpler to use and build RESTful web services. In this field service discovery did not gain that much importance and even today many service providers are simply hardcoded in clients using the respective service URLs [1]. The novel trend of mircoservices changed this to some degree.

4.1. WSDL Web Service Solutions

From the beginnings UDDI (Universal Description, Discovery and Integration) [4] has been proposed as standard for service discovery as part of the so called SOA (Service Oriented Architecture) triangle [15]. The SOA triangle assumes provider register service at a central registry and user inquire the registry to get service descriptions including contact data to directly talk to the service. UDDI allows for storing services descriptions in WSDL together with meta information about these services including business as well as technical details. Registration and search is possible using white, yellow and green page service information via a dedicated UDDI API. In order to cope with large service amounts, publicly available enterprise UDDI registries had been set up by major players including Microsoft, IBM and SAP. Even though UDDI had been designed as open standard and major companies tries to push its usage both efforts failed to a large extent. The enterprise registries were silently taken down at end of 2005 and users searched for

alternatives to UDDI.[7] UDDI failure had several reasons, but one important aspect is the huge complexity of the standard that required much effort to setup and use a registry.

The problems with UDDI led to the development of alternatives like WS-Inspection [3] and WS-Discovery [16]. WS-Inspection defines an XML fomat for listing references to existing service descriptions typically in WSDL. Hence, WS-Inspection documents can be read and services can be contacted based on the deposited addresses. WS-Inpection can be seen in contrast to UDDI as a handy and simple but also quite limited solution because it does not emcompass service search functionalities. Both solutions reviewed so far concentrate on rather static service networks, i.e. it is assumed that the rate of leaving and arriving services is rather low. WS-Discovery has been devised to fill this gap. It is a hybrid mode protocol that can work with multicasts as well as registries (called discovery proxies).

4.2. Cloud-based Microservice Solutions

Microservices represent a specific interpretation of SOA in the sense that an application is seen as composition of rather small services [14]. In a microservice architecture each service may consist of the full vertical software stack from user interface to its own database making them independently developable. Business functionalities are realized by service interactions so that service discovery becomes a vital part also in these kinds of applications. Microservices are practice driven by major IT companies. Thus, in the following, architectures of registry software are evaluated.

Consul[8] is a cloud enabled service registry from Hachi Corp. Cloud-enabled means that functionalities facilitate the operation in cloud datacenters, e.g. support for different regions in queries. Its architecture is based on a strongly synchronized server cluster. Requests will be answered by a leader following the Raft [17] consensus protocol. Primary goal of the system is high fault tolerance, because server failures can be compensated to some degree by the used consensus protocol. As long as a quorum of servers answers to a request the system keeps functioning. The size of the cluster can be configured, but higher numbers of servers in the cluster negatively impact the response time due to increased overhead. The system offers a RESTful API for (de)registering and searching services. Search is based on arbitrary key value pairs that the system checks for presence in registered entries.

Eureka 2[9] is a service registry for microservices developed by Netflix that is also destined for use in cloud environments. Like Consul it provides first-level support for cloud properties within service specifications and queries. Eureka 2 has a more advanced architecture than its predecessor as well as most other registries available. The Eureka 2 architecture comprises two disjunctive server clusters: a write and a read cluster. As the names suggest, these clusters are responsible for handling service (de)registrations and query processing separately. The advantage of this separation of responsibilities is that the read cluster can be auto-scaled independently from the write cluster in accordance to the current query workload. The write cluster must be set up manually and is not auto-scaled. Eureka 2 is also the first system supporting persistent queries, i.e has a push model

[7] http://www.computerwoche.de/a/570059
[8] https://www.consul.io/
[9] https://github.com/Netflix/eureka/wiki/Eureka-2.0-Architecture-Overview

for service discovery. The queries itself are based on service properties described as key value pairs. In addition to checking for the presence of value simple operators are also supported.

Besides registry solutions also more simple value stores are frequently used for service lookup. One of them is Zookeeper [10] originally conceived as centralized service for maintaining configuration information, naming and providing distributed synchronization. It consists of a predefined number of servers that are completely meshed and each replicate the full data set. Zookeeper uses a Paxos [10] variant to achieve strong consistency among these nodes. The service is available as long as a quorum of servers can be contacted. Zookeeper offers a RESTful API for all client based interactions. Another well-known system is etcd [11], which has been designed as distributed, reliable key-value store for critical data of a distributed system. It uses the Raft [17] consensus mechanism and has been designed intentionally simple concerning the offered functionality but with a strong focus on non-functional criteria including performance and availability.

Both, Consul and Eureka contain problematic design decisions for megascale systems. First, both registries use health checks to periodically ping services, which is costly with many services, because many open connections are held. Second, query processing currently is not based on indexed data structures, which renders it necessary to check each registered service in each query. For Consul as well as Zookeeper and etcd, the data storage model is problematic, because strong consistency incurs unnecessary overhead. As potential failure is network immanent, services can appear, disappear or become invalid at any point in time. Thus, an *eventually consistent* data storage model is preferable (as also used by Eureka).

4.3. Requirements Evaluation

The approaches presented beforehand have been also systematically evaluated with respect to the requirements from Section 3. Table 1 shows the results of the evaluation and it becomes clear that none of the representatives is currently well suited for megascale systems. Regarding the challenge of handling many providers, UDDI, Consul and Eureka can at least deal with hundreds and even thousands of providers, but the latter two use persistent connections to all registered services limiting their overall capacities. Auto maintenance is only present in novel systems (Consul, Eureka 2, etcd) and realized using heart beat mechanisms based on the connection with a service. Service matching is kept simple in most approaches and based on service type names and properties in many cases. Zookeeper and etcd do not offer service matching capabilities besides a lookup mechanism. Eureka 2 has the most flexible solution, because it offers operators that can be used to build queries on service properties. None of the approaches has flexible means for controlling the result size of a query. In addition also none considers result ordering and quality levels. Support for persistent queries has been integrated in Consul and Eureka 2. In Consul, the concept has been realized as so called blocking queues, i.e. a call to service endpoint can be made to persist and changes on that endpoint will be automatically published back to the caller. Non-functional criteria have been considered by many novel approaches like Consul, Eureka 2, etcd and Zookeeper. These approaches

[10] `https://zookeeper.apache.org/`
[11] `https://coreos.com/etcd/`

	Provider-Side Requirements		Client –Side Requirements					System Architecture Requirements				
	Many Providers	Auto Maintenance	Service Matching	Result Size	Result Ordering	Query Persistence	Quality Levels	Scalability	Performance	Availability	Privacy	Zero Configuration
UDDI	=	-	+ (operators on properties)	-	-	-	-	+	=	+	-	-
WS-Inspection	-	-	- (no queries)	-	-	-	-	-	+	-	-	+
WS-Discovery	-	= (in ad-hoc mode)	= type name	-	-	-	-	-	+	+	-	-
Consul	=	+ (lease)	= (existence of properties)	-	-	=	-	+	+	+	+	-
Eureka 2	=	+ (lease)	+ (operators on properties)	-	-	+	-	+	+	+	=	-
Etcd	=	+ (lease)	- (key)	-	-	-	-	+	+	+	=	-
Zookeeper	=	+ (connection)	- (key)	-	-	-	-	+	=	=	=	-

Table 1. Existing service discovery approaches (support: + full, = some, - none)

are decentralized by design improving scalability, performance and availability. Privacy is also tackled by these systems at least in a limited way. They allow for securing the communication of clients with the registry and also offer client authentication. This makes it possible having registries not accessible by unknown clients. Security does not include authorization, i.e. defining which services a client can see. Zero configuration is not considered except when using WS inspection, which is so simple that nearly no configuration is necessary. The other systems require considerable set-up effort as most of them need to be configured as a cluster of machines. At least some systems like etcd and Consul offer Docker containers reducing the amount of individual configuration work. Nonetheless, Consul, Eureka 2 etcd and Zookeper need a predefined cluster of nodes to operate. They cannot dynamically extend or shrink the number of registry servers due to the underlying consensus mechanism using quorum decisions.

Some of the approaches offer interesting solutions to the requirements of service discovery in large-scale systems. E.g. many aspects of the realization of *Auto Maintenance* and *Service Matching* in *Eureka 2* are highly relevant to dynamic systems such as the helpline app. Yet, none of these approaches is able to cope with all of the requirements. For example, neither *Result Size* nor *Result Ordering* is present in any considered representative. Considering using only abstract properties like age or gender for matching missing persons, it becomes obvious that support of these features is essential for the helpline app to become feasible.

Summing up, existing solutions can be divided into pure service registry approaches and more generic distributed value stores. For simple scenarios the latter group of systems might be adequate, but in more complex scenarios they are not sufficient because they do not support advanced query definition and processing. Nonetheless, modern registry approaches like Eureka 2 and Consul make huge progress regarding the system architecture supporting an inherent distributed processing model. Despite the distributed set-up they do not support changing the cluster dynamically at runtime.

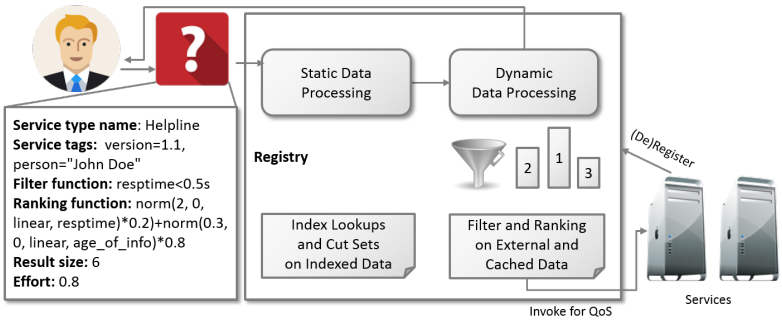

Fig. 3. Query processing model

5. Solution Architecture

The analysis from the last section has revealed, that current approaches lack support for the aspects *result size* and *result ordering*. The combination of both is a very useful feature because it allows to query the best set of services with regard to specific properties. Such behavior can be emulated in existing registries only with much effort by retrieving all matching services and ranking them on client side. In megascale systems this can be impractical given large result sets that would have to be transferred, processed and afterwards pruned nearly completely on client side.

5.1. Distributed Query Processing Model

Query processing is the key functionality of the registry and is handled in a specific way to ensure scalability. A query itself is defined by *service type name*, *service tags*, *a filter function*, *a ranking function*, *result size* and *effort* (all optional). This structure and example query is depicted in Fig. 3. The filter function defines a boolean function which must evaluate to true in order to include the current entry. In contrast, the ranking function maps an entry to a [0,1] interval with 1 being the best value. The example query requests the best six *Helpline* services of *version 1.1* that provide information about the *person "John Doe"* with a response time lower than *0.5 seconds* ranked by both *response time* and *age of information*.

The query processing is done in a very specific way that differs from other systems. It is based on the observation that service data can be subdivided into two different groups: static data that remains constant over time and dynamic data which is subject to frequent changes. Hence, the query processing system consists of two subsystems each respsonsible for one of those categories. The static data (type name and tags) is used by the static query processors to build up full indexes for all elements. During processing the engine looks up all static properties and compares the results set sizes. It then starts with the minimal set and creates the cut-set with all other result sets one by one. Starting with the smallest result set ensures that the biggest reduction of result elements is performed first.

After the static query data has been evaluated the dynamic processors are headed over the result set of services. The non-functional properties of the query can be both,

static and dynamic but are always treated using the dynamic processors. This is due to the fact that different operators can be used and a simple indexing is not possible any more.[12] If a property is static the query can be directly evaluated against the registered value of the service description. If not, it needs to be retrieved first. For this purpose the service description has to contain a REST-URL that can be queried for non-functional properties. The registry will evaluate the query against the fetched value and additionally store the value in a cache. This allows subsequent requests to reuse the fetched value during a freshness interval which can be defined in the service description. Requesting always fresh values can be enforced by explicitly setting the required maximum age to zero. The resulting services will then be ranked by the ranking function using static as well as dynamic properties. The ranking function consists of two phases. First, the non-functional result values will be normalized to the interval [0,1] and afterwards these values will be weighted and summed up. The overall weight values must be 1, i.e. each non-functional property can be set a proportionately importance relative to the other values. In the example query the ranking function first linearly maps the response time so that calls that last longer or equal than 2s will get zero points (2s=0). Faster calls get a value in direct correspondence to their distance from 0 (with 0s=1). A call that immediately returns is rewarded with full score of 1. A similar mapping is done for the costs. Afterwards the weights are applied, here favoring response time over costs, and the final result value is computed as sum. The results are sorted according to the overall ranking value and delivered to the client in that order. Depending on the required result size of the client, only a small fraction of the result set is finally transferred.

The registry also supports persistent queries via the same query definition, i.e. the query once submitted will be continuously be evaluated against all newly available services. In this case the ranking function is interpreted differently as the ordering of newly appearing services cannot be enforced when results are sent to the client right after discovery. Instead, the value of the ranking function is headed over to the client as mere quality value that it can use in comparison with formerly received services.

5.2. System Architecture

In Figure 4 an overview of the proposed system architecture is shown. Its basic layout is a layered model consisting of (at least) three levels. This design is inspired well-established Internet services like the network time protocol (NTP) architecture [12] and the domain name service (DNS) [13]. NTP is able to provide Internet-scale clock synchronization using multiple layers called strata. In NTP the stratum number determines the precision of the provided time service, i.e. allowing clients with different demands connecting at servers of different stata. In DNS the levels are used to divide responsibilities of domain management and queries are answered by running through a hierarchy od DNS servers forwarding the request according to the URL parts. Finally, a domain specific DNS server receives the request and can map the name to a corresponding IP. In this way DNS not only archieves separation of work (different requests reach different servers) but also allows for local management of domain data, i.e. organizations can administer their servers on their own.

[12] One could use B* trees in future versions to further enhance indexing for comparative operators.

The proposed architecture of the registry adapts this layer model for two main reasons. First, it enables keeping service registration data close to the clients and second it allows for data splitting according to visibility scopes. The following sections will elaborate these aspects in detail and also describe the tasks and protocols involved.

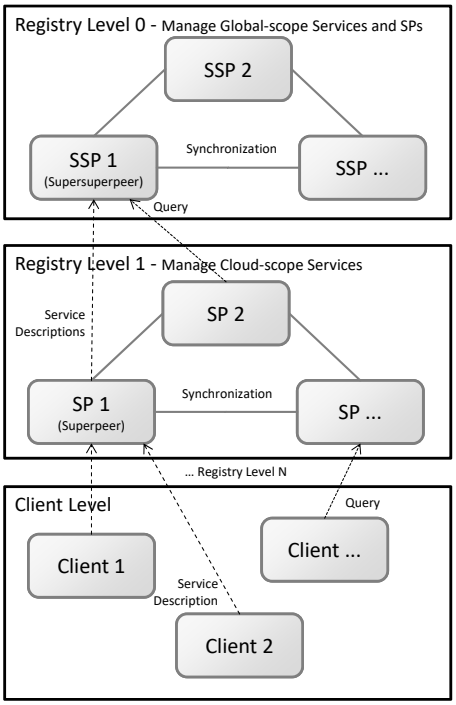

Fig. 4. Architecture Overview

High Scalability A direct consequence of the CAP theorem [7] in distributed systems is that typically partition tolerance must be considered. In these cases one cannot have both properties - consistency and availability - and needs to choose one in favor of the other. Even though existing solutions often try achieving strong consistency, we prefer a solution that guarantees high availability. The reason is that in a dynamic distributed systems services can appear, disappear and fail at any moment in time so that a consistent data set can become invalid also at any point in time. Thus, the gain of having strongly consistent service data is low and the efforts in terms of algorithmic complexity as well as loss of availability and runtime performance are substantial.

Having argued for a AP (available, partition-tolerant) architecture the second aspect is how data should be partitioned and stored. We argue for an approach that uses partitioning according to the visibility of services, i.e. when publishing a service description one has to specify which potential service users can see it. In this respect we introduce two scopes:

Algorithm 1 Replication algorithm

```
on_client_message(m):
 updateDependencies(m.clients);
 updateServiceData(m.addedServices, m.removedServices, m.modifiedServices);
 forwardEventToPartners(m);
 if(isRelevantForParent(m))
  forwardEventToParent(m);

on_partner_message(m):
 updateDependencies(m.clients);
 updateServiceData(m.addedServices, m.removedServices, m.modifiedServices);
```

global and organization scope. The first means that services can be found by all other services while the latter restricts the visibility to service users from the same organizational context. Of, course this model is open for extension by introducing further subscopes if needed. The visibility scopes are mapped 1:1 to the registry levels of the model, i.e. level 0 is responsible for managing globally visible services while level 1 handles the organizationally scoped services. Services registrations as well as requests are forwarded up to the layer that fits their scope. As each organization uses their own level 1 registry level, data is stored close to the clients that come from the same organization. In order to cope with a high number of clients, in each level multiple registries (called superpeers on level 1) can be made available. Data is fully replicated among registries belonging to the same level and organization so that a query can be processed in at most n steps, if n layers are used. In practice, many queries are scoped organizationally so that a single node is able to answer them.

High Performance Performance is important especially with respect to query processing, but also the read and write operations for service descriptions need to be fast. The former aspect has been discussed in the last Section 5.1 and its speed is based on extensive use of index structures. The latter is largely based on data distribution in the system. Typically, the CAP theorem requires to decide between an optimization for read or for write accesses. The first option is realized by replicating the full registry content to all superpeers so that client requests can be load-balanced between them. This also means that write requests are slow because the system has to wait until the change has reached all superpeers. On the other side, optimizing for write requests means that data sharding is used and data is stored only in a subset of all nodes. Consequently, read requests are slowed down because clients have to share certain superpeers on similar queries. In the proposed architecture fast read and write requests can be achieved, because the data consistency constraints are reduced. This allows to use full data replication without waiting for write requests to complete. Instead, it is only guaranteed that each node will eventually recover from data inconsistencies in case of network or node failures.

In the considered setting the replication algorithm does not need to follow complicated and performance reducing consensus mechanisms like Raft [17] or Paxos [10]. Instead the used replication algorithm can be designed so that no data conflicts occur. To achieve this the concept of responsibilities for clients is introduced, i.e. each client is managed by exactly one registry. The allocation is automatically performed by the client connecting a

registry. From that point in time the latest connected registry is responsible for the client. The implications are that superpeers forward service registration data only of those clients that are managed by them. Other data received from other superclients during sychronization is never distributed further. In case a client reconnects to another registry, the new registry will delete every existing data of that client and renew it. Moreover, the old registry will notice at some point that the client has disappeared; either because its lease time runs out or because it receives a synchronization request including data of that client (that is now managed by the other registry). In this case also the old registry will clean up its data leading to eventual consistency. [13] The core of the algorithm is depicted in Alg. 1. It is realized mainly by two methods: on_client_message(m) and on_partner_message(m). The first is called when a 'vertical' message occurs, i.e. a message from a lower layer to the layer above. In this case first the dependencies of the contacting client are updated and the new service data is included in the database. Afterwards the event is forwarded to all superpeers of the same level called partners here. If the event contains information that is relevant for higher levels according to the service scope it is also forwarded to the parent. When a partner message is received, first the partner dependencies are updated. Thereafter, the new data is included in the repository. Please note that updateDependencies() cleans old data of unmanaged clients - this is always done before new data is included.

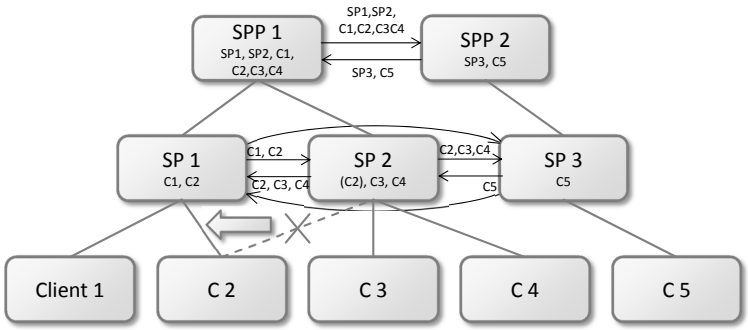

Fig. 5. Replication illustration

In Figure 5 the replication algorithm is illustrated with an example. Superpeer 2 (SP 2) initially manages data of Client 2 (C 2), C3 and C4. For some reason, C 2 reconnects to SP 1. SP 2 does not immediately notice this and thus sends SP 1 and SP 3 data of C2, C3, C4 although the C2 data is outdated. SP 1 knows that his C2 data is newer and discards the received info about C2. Of course C3 and C4 data is still relevant and included in its data base. SP 3 does not know that C2 data is not accurate any longer and includes it together with C3 and C4 data in its repository. The outdated entries are finally deleted with the

[13] The complete algorithm is a bit more complicated than described because it has to take into account also the different levels. This does not fundamentally change the behaviour but registries need to be enabled to manage also indirect clients, i.e. a superpeer keeps track of its direct clients (the superpeers of lower levels) as well of its indirect clients that are managed by the lower level superpeers.

next synchronization message of SP 1. SP 2 and SP 3 will remove their old entries and are eventually consistent with each other.

High Availability In order to provide high availability the architecture has been designed to work decentrally without single point of failure. This is achieved by replication in each layer so that breakdowns of single registry nodes can be tolerated by reconnecting to another one. This behavior is performed by clients as well as superpeers. Reconnecting requires the nodes being able to detect alternative superpeers which is achieved in the following way. Each client and superpeers is supplied with a list of level 0 superpeers as bootstrapping mechanism. In this way nodes can always find at least a level 0 superpeer even if no other discovery mechanism is successful. The level 0 superpeers keep track of any contacting nodes and save information about them including their specific node type (client, superpeer level N). During runtime level 0 superpeers inform their clients whenever other superpeers of a lower level become available, i.e. a client currently connected with a level 0 superpeer receives a message from this superpeer when suitable (same organization) level 1 superpeers have connected. Subsequently the client will automatically reconnect to a level 1 superpeer. The same mechanism is used to inform lower level superpeers about new (suitable) sibling superpeers, i.e. e.g. a superpeer of level 1 is informed when other (suitable) level 1 superpeers are known by a superpeer of level 0. This facilitates the synchronization of lower level superpeers. This superpeer based detection is complemented with additional decentralized awareness mechanisms. Each node uses several technically different discovery protocols like IP multicasting, DNS lookups and local host inspection to find superpeers of arbitrary levels. In case of success a clients prefers using a nearby superpeer and will always reconnect to a lower level in case it is connected to a level 0 superpeer reducing load substantially on level 0 superpeers.

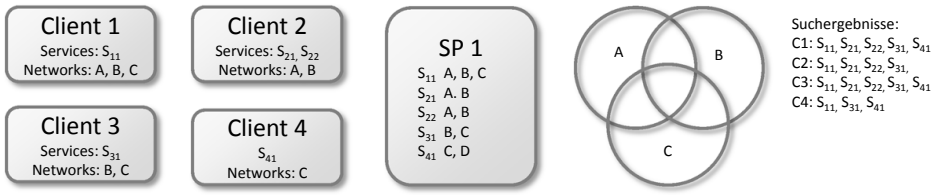

Fig. 6. Example scenario for security networks

Security Privacy of service data is a crucial aspect of a registry. While controlling access to services is the most important aspect of system security, a common problem of open systems like e.g. in UDDI registries is that leaking service information such as available services, their type and their interfaces can aid an attacker to discover weaknesses. More importantly, the information can help to accelerate attacks, increasing damage potential until the attack is discovered. It may also help the attacker to identify the most vital and vulnerable services of the system and increase denial of service attack efficiency by specifically targeting those services.

As a result, service information provided by the registries should be carefully controlled to prevent functionally unnecessary information leaks. The service information available to third parties can be restricted in two ways: First, the scope of registries and thus service data can reduced by avoiding exposure of registry addresses and restricting network access to registries e.g. to the organizations intranet. Second, the registry can use internal access control mechanisms to limit access to authorized service users. In the proposed solution both aspects have been combined. On the one hand lower level registries keep most of the service data so that these data remains near to the clients and on the other hand the registries use role based authentication.

Foundation for the role based authentication is the introduction of virtual security networks [5]. These networks consist of names and a trust anchor like e.g. a public key or a shared secret like a password. If services can provide authentication for security networks to each other, their communication is annotated with the network names. The service registries use this mechanism for service registration and query processing. When a service is newly registered, the associated network names of the service provider are stored along with the service data. Naturally, authentication secrets are neither transferred nor stored in the registry. Request processing uses the network names to filter the results and only delivers services for which at least one shared network is available. This is further illustrated in Fig. 6 with an example scenario. In this scenario four clients offer the same type of service and use the same registry SP 1. It can be seen that search requests from the clients yield different results ruling out those services for which no access exists, e.g. Client 2 does not get service S41 which is only available in network C not shared by the client.

Zero Configuration To minimize the upfront efforts for using the registry, the approach is meant to be largely self-configuring. Nodes use the already introduced awareness mechanisms to discover registries in their neighborhood. If no registries could be found, nodes can automatically activate/start a higher-level registry. Similarly, if the registries find out that too many registry instances are available they can shut themselves down. One main problem of this self-configuring approach is that cycling behavior has to be avoided, e.g. all nodes discover that no registries are available and start new ones leading to too many registries which subsequently shut down again a short time later.[14] For this reason a conflict resolution protocol needs to be applied. In this case the protocol should be as simple as possible and should not lead to too many messages. Hence, the proposed solution consists of a self-election mechanism in which a node broadcasts a 'promote-to-registry' message to all nodes in its neigborhood. Any of the nodes can subsequently send a veto message hindering the node becoming a registry during a small period of time. If no veto message is received the node will automatically promote itself as registry. Nodes use an individual quality value to compare each other. It is calculated as normalized sum function of several weighted factors including uptime, memory and CPU, i.e. $f_{power} = w_a * f(uptime) + w_b * f(memory) + w_c * f(CPU)$, $with\, w_a + w_b + w_c = 1$, $0 \leq fpower \leq 1$. The partial functions map values to the [0,1] interval using 1 as best possible value, e.g. in case of uptime 0 sec uptime = 0 and >1 h = 1. Intermediate

[14] Typically, if more than a handful of registries are connected to each other in the same level, the overhead for fully meshed synchronization becomes noticeable.

values are computed according to linear relationships. A similar mechanism is applied for determining if a registry shuts down.

5.3. Implementation and Discussion

The implementation of the registry is performed in two distinct phases. In the first phase the interaction protocols for inter-superpeer synchronization as well as client-superpeer interactions are realized. This part has already been finished and allows for preparing larger experimental testbeds on the infrastructure layer. In the second phase, the two-staged query processing is implemented. The functionalities for indexing, performing queries and ranking are in place but are currently performed on the same nodes, i.e. the separation of query stages including a suitable protcol has to be realized. The registry is implemented using the Jadex active components framework [18] because it facilitates the automatic superpeer discovery and synchronization considerably.

In megascale service-based systems, search requests often match hundreds of thousands of services. Thus, result size and result ordering are important criteria to fetch the right services for each client. Although we are targeting highly dynamic systems, we expect the load to be dominated by search requests and not service (de-)registrations. I.e., we assume that the number of search requests is usually much larger than the number of service (de-)registrations in the same amount of time. As a result, the following discussion focusses on matching complexity only.

A naive matching approach would lead to $f_{matchnaive} \in \mathcal{O}(n_p * n_c)$, i.e. a matching complexity proportional to the number of all service providers in the system n_p for each request, times the number of search requests, which is proportional to the number of clients n_c. Moreover, checking dynamic properties incurs a lot of communication overhead for each service, such that not only a huge number of checks would need to be performed, but also each check of a single service against a single request would consume a considerable amount of resources.

The proposed architecture contributes to handling this complexity in two important ways. First, scaling processing nodes can reduce n_c: While increasing processing nodes will increase service registration cost, the number of clients per processing node becomes constant if scaled linearly with the clients. As a result, query processing cost is reduced and depends only on the providers, i.e. $f_{matchscaled} \in \mathcal{O}(n_p)$. Second, the processing model uses indexing mechanisms reducing complexity with regard to n_p. Index lookups can be performed using, e.g., hashing mechanisms in constant time. As a result, only the smallest result set from all possible index lookups for a single search request needs to be searched linearily. We expect asymptotic behavior of the smallest-size result set being sublinear with service descriptions becoming more diverse with increasing numbers. Since the target scale is large but not infinite, it gives even slightly sublinear complexities large advantage in pessimistic cases.

In addition, practical issues have to be considered as well: A large part of the matching effort is offloaded to the processing nodes. This avoids the need for transferring of large data sets as well as reducing client workload, which can be a considerable advantage in e.g. mobile clients. Preselecting a result subset (e.g. 10 “best”) also helps in this goal. Finally, persistent queries reduce bandwidth and offer an easier programming model.

6. Evaluation

The empirical evaluation of the novel distributed registry system has been performed with respect to scalability and performance. Regarding availability and privacy additional scenarios have been used to verify the intended behavior.

6.1. Experiment Setup

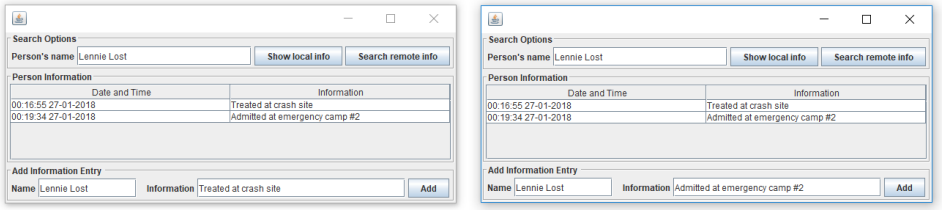

Fig. 7. Screenshot of two helpline nodes

For evaluating the proposed and realized architecture, an implementation of the helpline application has been created. In the resulting system, each helpline node has a user interface as shown in Fig. 7 and starts any number of simple services for each person of interest. Once the service is started, the user will get updated information from the network of other helpline nodes. Moreover, the user can query for existing information as well as post new information into the system. The connection between different helpline nodes and services is transparently and automatically handled by the registry architecture.

To further investigate the properties of the registry architecture and implementation, an evaluation scenario has been conceived that allows scaling the number helpline nodes and services in the system (cf. Fig. 8). In the scenario, there are three global SSPs (level 0) that serve as rendezvous nodes for discovery of all SPs (level 1). Each SP has a fully replicated copy of all registry data. For better scalability of the helpline application, we employ the multi-level characteristic of the registry architecture and introduce *regions* of SP groups, that are responsible for geographically related services. In the evaluation scenario we used 3 SPs for each region. In the client level, each helpline node will find an SP of the desired region[15] and uses this SP to query for services of other client nodes in the same region. For the evaluation, we scale the numbers of client nodes from 1 to 1000. Moreover, 1000 services (i.e. persons of interest) are created on each client node leading to a total of 1,000,000 services to be managed by the SPs in the registry level.

For comparison, we also executed the scenario runs with an alternative peer-to-peer (P2P) discovery mechanism. In the P2P scenario, there are no SSPs and no SPs. Instead the client nodes directly discover each other and for each service query, a node will send requests to all other known nodes.

[15] If the client node cannot discover an SP by itself, it uses a request to a global SSP to be relegated to a proper SP.

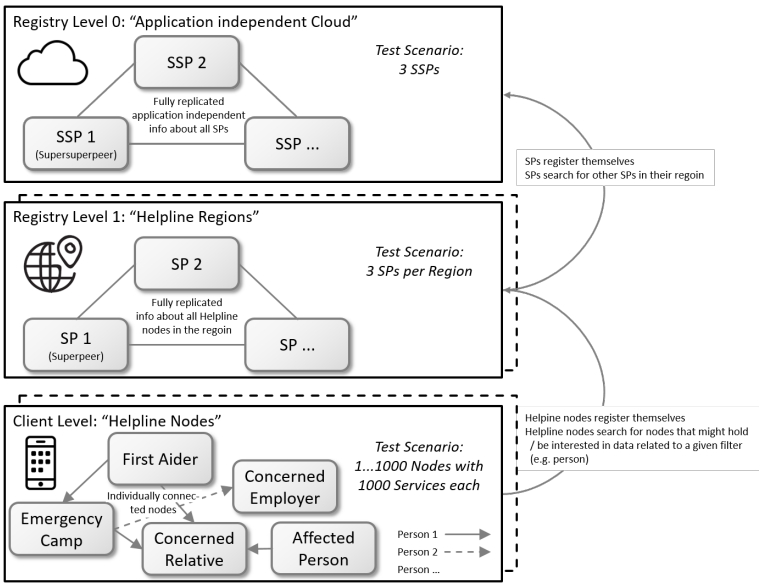

Fig. 8. Evaluation scenario using the helpline application

6.2. Results

Fig. 9 shows the results of the evaluation runs. We measured two aspects of the system: 1) the time to create a single node with 1000 services (*creation time*) and 2) the time to perform a single search through all services created up to this point (*search time*). With regard to the search time, we further differentiated two settings: 2a) searching for a service (i.e. person name) available on all client nodes, leading to result sizes from 1 to 1000 (*multi*) and 2b) searching for a service that is only present on a single other node, leading to a constant result size of 1 (*single*). Both settings (single and multi) were applied to the new superpeer architecture as well as to the alternative peer-to-peer model, thus giving a total of four settings (*SP-multi*, *P2P-multi*, *SP-single*, *P2P-single*).[16] As stated above, the number of client nodes was increased continuously from 1 to 1000 for each setting. Each value in the figure represents the median of 10 measurements (e.g. ten measurements for creating the first ten nodes and the first 10000 services).[17] All experiments were run with OpenJDK 1.8.0 (u151) on a single Google VM with 24 vCPUs (2.0 GHz Intel Skylake) and 156 GB memory. For the figure, all evaluation runs were stopped at 1,000,000 services, although the behavior in all settings seemed consistent well past this limit (not shown).

[16] The single vs. multi setting only affects the search time and not the creation time. Although we measured creation time for single as well as multi runs, only one of the (very similar) results of these runs is shown in the figure for improved readability.

[17] Using median instead of average values was chosen to factor out occasional outliers caused by Java garbage collection.

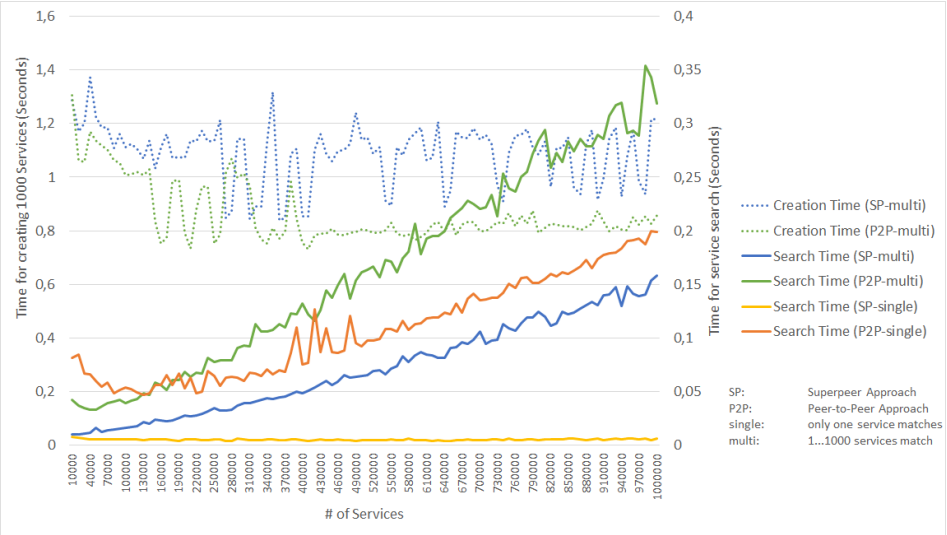

Fig. 9. Superpeer (SP) vs. peer-to-peer (P2P) discovery approach

The creation time, using the scale on the left hand side, shows constant behavior for both the SP (blue dotted line) and P2P settings (green dotted line).[18] The constant behavior is to be expected for the P2P approach, because services are initially only registered locally in each node. The creation time of the SP approach is higher due to the initial indexing of each created service. Yet, the SP approach also exhibits constant behavior although more and more services are added to the three available SPs. This means that in the SP approach, the indexing complexity only depends on the number of services added and not on the number of services already stored.

For the search time, the picture is quite different: The P2P search times both grow linearly (green and orange solid lines, scale on right hand side). This is due to the fact, that the searching node has to contact each other node. As the lines are quite similar, the size of the result set (1 for *single*, 1...1000 for *multi*) seems to be of little influence. The SP search times on the other hand (blue and yellow solid lines, scale on right hand side) show, that the effort is dominated by the result size. I.e. in the *single* setting (yellow), the constant result size leads to constant search times around 5 milliseconds. For the *multi* setting (blue), the search times increase proportionally with the linearily increasing result size (starting from 1 up to 1000 matches) and grow much slower than the P2P search times. This result is due to the fact that for the search only a single registry is queried, which can calculate the query result using a constant lookup.[19] The SP search is thus dominated by the communication effort of sending back the results from the SP to the client node, which is proportional to the result size.

[18] The quite big variance is due to the fact that a lot of memory needs to be allocated for node and service creation leading to frequent delays, when the Java garbage collector kicks in.

[19] Services are tagged by the person name, which causes the registries to automatically create an index for this tag.

6.3. Availability

Availability of the registry means that it keeps functioning in spite of failures occurring at different sites. Most critical is that a client detects that its superpeer does not respond (could be a connection or superpeer failure). We have tested this with the following scenario using a fragile setup of only two registry nodes: one SSP and one SP serving several thousand clients. After the clients have registered at the SP that SP has been killed. As consequence the clients now search for an alternative SP but there is none. Finally, they use their internal SSP list (level 0 SSPs) and connect to the available SSP. Starting two SPs level 1 lets the SSP inform the clients that two SPs are available. The clients disconnect from the SSP and reconnect to one of the SPs. Whenever clients disconnect from a (S)SP in this scenario their data is deleted on the (S)SPs. After a reconnect the data is then regained and distributed by the novel (S)SP of the client.

6.4. Privacy

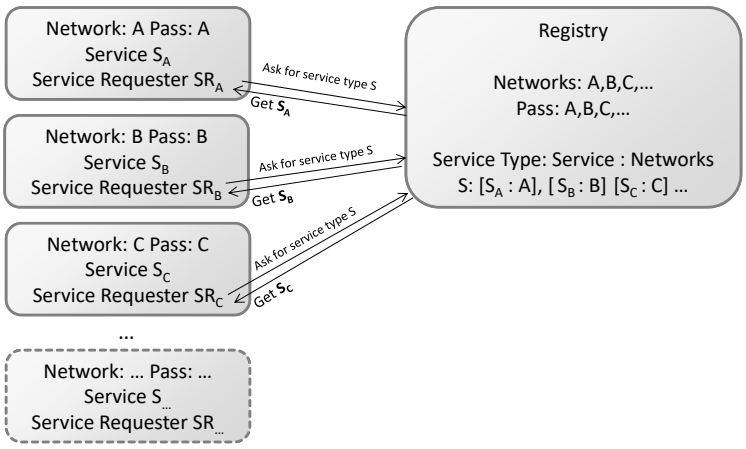

Fig. 10. Security example evaluation scenario

Privacy has been tested using an example scenario with one SP (level 1) and <*n*> services as well as <*n*> service requesters (we used n=1000). The set-up of the test scenario is illustrated in Fig. 10. Security settings have been set in the following way. For each pair of service and service requester a common security network name and password is assigned. The <*n*> pairs do not share any secret, i.e. they cannot communicate with each other. The SP has been configured to know the network names and passwords of all pairs, so that all pairs can communicate with the SP. Each service registers itself at the registry at startup while the requesters search for services over and over again in a delayed loop. It can be shown that in this setting each service requester does only find the associated service via the SP and no other service although all services are of the same type and will be found in case no security restrictions are in place.

7. Conclusion and Outook

This paper argues for the relevance of megascale systems, i.e. systems, in which 10^6 or more decentralized service providers exist and are used by potentially even larger numbers of clients. We expect such systems to appear in the near future due to ongoing trends such as smart cities or autonomous vehicles, where large numbers of existing devices are transformed into decentralized networks of service providers.

This paper tackles the problem of service discovery in such a system. Requirements with respect to the service provider and service client side are identified and existing approaches are analyzed with respect to their contribution to these requirements. Following this analysis, a solution architecture based on a scalable node structure and a request processing model is presented, that addresses open problems for a megascale service infrastructure. The request processing model allows distributing the matching load of service queries across different nodes in the network and thus allows handling static as well as dynamic service properties.The distributed service infrastructure is implemented using the *Jadex* framework. Practical and empirical evaluation using large numbers of service providers and requesters has shown that the proposed architecture is capable to fulfill the requirements regarding performance, scalability and high availability. As future work it is planned to expand the self-configuration and self-healing features of the distributed registry. Furthermore, we want to extend the security model allowing clients to connect to multiple superpeers belonging to different (security) networks. This will allow clients to use services provided by different organizational units.

References

1. Algermissen, J.: Using dns for rest web service discovery (2010), https://www.infoq.com/articles/rest-discovery-dns
2. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18-20, 1967, Spring Joint Computer Conference. pp. 483–485. AFIPS '67 (Spring), ACM, New York, NY, USA (1967)
3. Ballinger, K., Brittenham, P., Malhotra, A., Nagy, W.A., Pharies, S.: Web services inspection language (2002), https://svn.apache.org/repos/asf/webservices/archive/wsil4j/trunk/java/docs/wsinspection.html
4. Bellwood, T., Capell, S., Clement, L., Colgrave, J., Dovey, M.J., Feygin, D., Hately, A., Kochman, R., Macias, P., Novotny, M., Paolucci, M., von Riegen, C., Rogers, T., Sycara, K., Wenzel, P., Wu, Z.: Uddi version 3.0.2 (Oct 2004), http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm
5. Braubach, L., Jander, K., Pokahr, A.: A practical security infrastructure for open multi-agent systems. In: M. Klusch, M. Paprzycki, M.T. (ed.) Proceedings of Ninth German conference on Multi-Agent System TEchnologieS (MATES-2013). pp. 29–43. Springer (2013)
6. Braubach, L., Pokahr, A.: Addressing challenges of distributed systems using active components. In: Brazier, F., Nieuwenhuis, K., Pavlin, G., Warnier, M., Badica, C. (eds.) Intelligent Distributed Computing V - Proceedings of the 5th International Symposium on Intelligent Distributed Computing (IDC 2011). pp. 141–151. Springer (2011)
7. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33(2), 51–59 (Jun 2002)
8. Haubeck, C., Chakraborty, A., Ladiges, J., Pokahr, A., Lamersdorf, W., Fay, A.: Evolution of cyber-physical production systems supported by community-enabled experiences. In: IEEE 15th International Conference of Industrial Informatics INDIN 2017 (2017)

9. Jander, K., Pokahr, A., Braubach, L., Kalinowski, J.: Service discovery in megascale distributed systems. In: Proceedings of the 11th International Symposium on Intelligent Distributed Computing (IDC 2017). pp. 273–284. Springer (2017)
10. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. 16(2), 133–169 (May 1998)
11. Messina, F., Pappalardo, G., Rosaci, D., Santoro, C., Sarné, G.M.: A trust-aware, self-organizing system for large-scale federations of utility computing infrastructures. Future Generation Computer Systems 56, 77–94 (2016)
12. Mills, D., Martin, J., Burbank, J., Kasch, W.: Network time protocol version 4: Protocol and algorithms specification. RFC 5905 (Standard) (Jun 2010), https://tools.ietf.org/rfc/rfc5905.txt
13. Mockapetris, P.: Domain names - concepts and facilities. RFC 1034 (Standard) (Nov 1987), https://tools.ietf.org/rfc/rfc1034.txt
14. Newman, S.: Building Microservices - Designing Fine-Grained Systems. O'Reilly Media (2015)
15. OASIS: Reference Model for Service Oriented Architecture. Organization for the Advancement of Structured Information Standards (OASIS), version 1.0 edn. (2006)
16. OASIS: Web Services Dynamic Discovery (WS-Discovery). Organization for the Advancement of Structured Information Standards, version 1.1 edn. (2009)
17. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference. pp. 305–320. USENIX ATC'14, USENIX Association, Berkeley, CA, USA (2014)
18. Pokahr, A., Braubach, L.: The active components approach for distributed systems development. International Journal of Parallel, Emergent and Distributed Systems 28(4), 321–369 (2013)
19. Sun Microsystems: Jini Architecture Specification version 2 (2003)
20. UPnP Forum: UPnP device architecture version 1 (2000)
21. Zhu, J., Oliya, M., Pung, H.: Service Discovery for Mobile Computing - Classifications, Considerations, and Challenges. In: Handbook of Mobile Systems Applications and Services, chap. 2, pp. 45–90. Auerbach Publications (2012)

Prof. Dr. Lars Braubach is professor for engineering complex software systems at the University of Applied Sciences Bremen. His research interests focus on software concepts for developing grid and cloud applications. He is co-founder of the Actoron GmbH delivering solutions based of the self-developed Jadex platform.

Dr. Kai Jander is a postdoctoral researcher at the University of Hamburg as well as co-founder and chief operating officer of Actoron GmbH. The focus of his research are service-oriented architectures, business process management and distributed system security with a particular focus on cloud-based systems.

Dr. Alexander Pokahr is representing the Chair of Industrial Data Processing and Systems Analysis at the Helmut-Schmidt-University / University of the Bundeswehr Hamburg. Current research interests include development approaches for large-scale, intelligent cyber-physical systems as well as agent-oriented control of autonomous mobile robots. In addition, he is acting as CEO of Actoron GmbH, which he co-founded.

Received: January 31, 2018; Accepted: September 4, 2018.