Hosting and Discovery of Distributed Mobile Services in an XMPP Cloud

Philipp Grubitzsch and Daniel Schuster
Faculty of Computer Science
TU Dresden
Dresden, Germany
{*philipp.grubitzsch,daniel.schuster*}@*tu-dresden.de*

***Abstract*—While originating as an instant messaging infrastructure, the eXtensible Messaging and Presence Protocol (XMPP) has its applications as a lightweight middleware for real-time event distribution, e.g. for mobile computing or the Internet of Things. Services based on XMPP are traditionally realized as server components which does not allow for distributed mobile services. We present an alternative approach for distributed hosting of XMPP services. Service hosting nodes may be added and removed to the overall system at any time. The solution only uses standardized XMPP features and extensions like roster groups, entity capabilities, and inband registration. An evaluation shows how concrete service instances can be efficiently distributed on different XMPP-enabled cloud servers.**

***Keywords*-XMPP, mobile services, hosting, cloud, roster, security**

I. Introduction

Modern mobile applications and applications in the Internet of Things are often realized as a compound of different native apps for smartphones and tablets (Android, iOS), smart object software (e.g. running on Contiki), HTML5-based Web UIs as well as mobile services running in the cloud.

Developing such application compounds is a challenge regarding the protocol development and testing, different UI techniques on different platforms, heterogeneity of device and network capabilities, as well as discovery and hosting of the services. In this paper we focus on the last one of the mentioned challenges trying to support developers in building mobile services for mobile application compounds.

We were experiencing the problems mentioned above by ourselves when developing collaborative mobile applications with native clients on different platforms requiring real-time communication between the devices and the service as well as directly between the devices. Examples are a location-based group finder application, a location-based multi-player game to explore the public transport of a city, a collaborative robot control application as well as an application for connecting providers and consumers of power supply for electric cars [1]–[3].

We built these solutions on top of the eXtensible Messaging and Presence Protocol (XMPP) and its many extensions. XMPP is currently gaining momentum as low-cost middleware for any kind of real-time messaging where authentication and presence mechanisms are useful. Mobile services based on XMPP can be realized either as server-integrated components or as so-called XMPP bots connecting as an ordinary XMPP client.

The latter approach is more flexible as it enables the mobile service to run anywhere in the network. It doesn't matter if it is realized inside a dedicated VM, as a cloud service, on a developer machine or even a mobile device as long as it is able to connect to an XMPP server.

The drawback of this approach is that XMPP service discovery is only able to show XMPP server components like Multi-User Chat and its items (e.g., chat rooms). XMPP bots do not receive special attention by the XMPP server and thus have to make themselves known using other means.

This paper presents an approach how this can be done using XMPP's built-in presence mechanism in connection with roster groups, i.e., groups of contacts that can be managed by every XMPP client and are stored at the XMPP server. This idea can easily be extended to also manage the availability of hosts and services. Moreover, roster groups can be used for deployment access control, too. Thus developers are only able to modify the services they contributed themselves using roster groups again.

An overview of the approach is shown in Section IV while the roster-based concept is described in Section V. Using this approach we extended our XMPP-based service platform Mobilis as described in Section VI. An evaluation in Section VII shows the ability of the platform to efficiently add new service hosts at any time which only marginally affects the time to find and create service instances.

Before diving into the details of our approach we provide some background knowledge on XMPP as a middleware (Section II) and related work (Section III).

II. XMPP as a Middleware

When talking about XMPP, one often refers to the whole XMPP infrastructure consisting of different XMPP clients and federated servers running the core XMPP protocols as well as a number of XMPP Extension Protocols (XEPs).

In this XMPP world, each user is identified by a so-called JID (Jabber Identifier) which consists of three parts: *username@domain/resource*. The domain part identifies the XMPP server where the user is registered with his username and password. The user may connect an arbitrary number of devices (e.g., smart phone, tablet, notebook, smart TV) to the server using different resource identifiers for each device.

Exchange of information is done using small XML snippets called stanzas. When *user1@domain1.com* wants to send a message to *user2@domain2.com*, he can use the so-called Bare JID of the receiver without the resource part thus generating a message stanza like this:

Listing 1. Example XMPP message
```
<message from="user1@domain1.com/phone"
         to="user2@domain2.com">
 <event xmlns="mydomain:example">ExampleEvent</event>
</message>
```

The server then first forwards this message stanza to the XMPP server running at *domain2.com*. The target server then selects the appropriate resource of the receiver based on priorities and delivers the message to the target device. This just works in real-time as every device is connected to its server using an always-on TCP connection. If the target device is already known to the sender, he could also use the Full JID with resource ID in the *to=* attribute.

As can be seen in Listing 1, the message may contain any namespaced child element. The message stanzas implement a fire-and-forget semantics. There is also a reliable request-response mechanism called IQ (info/query) stanzas in XMPP. An IQ request always has to be answered by an IQ result or error. This mechanism is used to retrieve the buddy list of participants at startup (called roster). The online status of participants on the roster is transported via presence stanzas. These three basic mechanisms - presence, message and iq - build up a general purpose event middleware which can be used for instant messaging but many other use cases as well. An overview of XMPP technology can be found in [4].

If we want to use XMPP as a middleware for mobile app compounds as described above, the client apps need XMPP client libraries which are widely available for many platforms, e.g., Smack for Java and Android, Strophe.js for JavaScript or XMPPFramework for iOS. As already mentioned above, the service part may be realized either as an XMPP component (a server plug-in) or as an XMPP client (bot), while the latter approach is preferable due to its flexibility but does not offer built-in service discovery.

III. Related Work

Leveraging XMPP as a service middleware dates back to multi-agent systems [5]. Each agent connects to the system as an XMPP client and communicates its availability using XMPP presence. Basic services like discovery and management are realized as XMPP components. Kestrel [6] uses a similar approach for many-task computing where each machine gets its own user account at the server with each processor connecting as an own resource and publishing presence.

Later works extend this idea to XMPP-based service platforms and cloud computing. The i5Cloud [7] uses XMPP for inter-service communication in the cloud as well as communication between devices and the cloud service. Junction [8] uses a so-called Switchboard to offer XMPP-based services for peer-to-peer interactive mobile applications like a poker game. Mobilis [9] defines a service platform where mobile clients may access XMPP-based services running as XMPP clients. Its main focus is to support native multi-platform clients with a service description language offering client and server stubs for many platforms. We build upon Mobilis adding discovery of distributed hosting and discovery of services in this work. But the approach described in this paper could also be used in any other XMPP-based service environment.

The idea of XMPP-based service hosting was later extended to hosting services even on mobile devices. This was first realized for ad-hoc scenarios using the Serverless Messaging extension of XMPP [10]. Srirama et al. [11] are using XMPP to access mobile services on Android devices while discovery can be done ad-hoc (ZeroConf) or via a separate Peer-to-Peer network (JXTA). Our approach was thus designed to work for both scenarios, i.e., cloud service hosting as well as mobile service hosting without the need of external service discovery.

Other approaches build on traditional middleware solutions like MQTT, AMQP, or SOAP to realize the app-to-service communication described in this paper. But they lack the ability to federate with arbitrary servers running in different domains. Furthermore, the security and presence functionality is unique for XMPP. Thus, approaches based on these non-XMPP solutions are often extended with XMPP for federation [12], [13].

Our work adds important building blocks to existing XMPP-based service hosting: The concept of roster groups has not yet been leveraged by existing applications. Similar functionality for service discovery and access control can only be reached by proprietary mechanisms in the works described above. The big advantage of our work is its compatibility with standard XMPP functionality which thus can be implemented in any XMPP-based system.

IV. Overview

We aim to scale up the architecture for XMPP-based mobile services while making the service discovery on a distributed, inter-domain system transparent for the clients. We thus distinguish the roles *Broker* and *Runtime* for discovery servers and hosting servers, respectively. We expect Broker URIs to be known to their clients by external means (e.g. well-known ID or configuration).

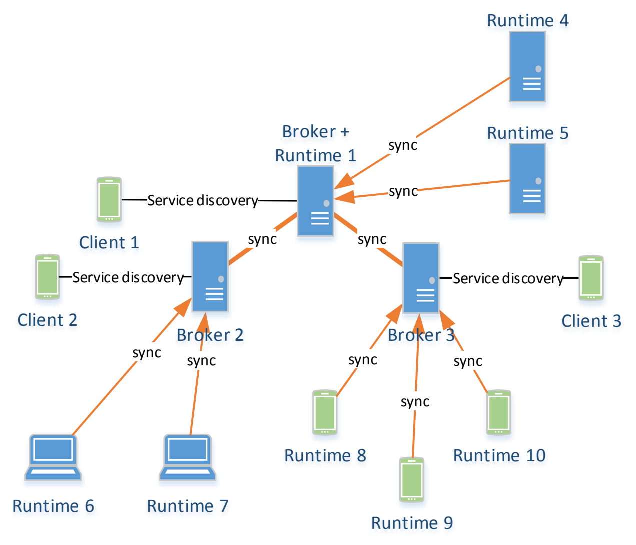


Figure 1. XMPP-based distributed service topology

As can be seen in the example topology in Figure 1, nodes can be Client, Broker, Runtime, or both Broker + Runtime.

As already mentioned above, we want the concept of service hosting to be flexible enough to run a service not only on cloud servers but also on client computers or even mobile devices. Runtimes register at a Broker to publish their availability and services. Clients use a Broker to discover available services and service instances. The Broker offers all service descriptions from the Runtimes registered at the Broker as well as service descriptions available at synchronized Brokers.

To interconnect with other Brokers, a Broker has to explicitly peer with another Broker which starts the synchronization process. In the pictured case, this leads to the fact, that Client 3 can just use the services of Broker 3 and Broker 1, while Client 1 can make use of all services on all Brokers. For the proposed distributed inter-domain service cloud, service discovery mechanisms and security issues have to be addressed.

All nodes in Figure 1 act as XMPP clients. Please note that we intentionally omitted XMPP servers. The placement of XMPP servers is out of the scope of our approach. For performance reasons, different cloud runtimes each should use their own XMPP server as well to separate the messaging traffic. But runtimes with low utilization may also share a common XMPP server. The same holds true for the Brokers. XMPP's inherent federation capabilities enable each node on the world-wide XMPP network to connect to each other and share presence information.

V. Roster-based Concept

Based on this basic architecture we now describe the details of our concept which entirely builds on XMPP's native roster management to realize a registry for distributed multi-instance services. In addition, we also show how to map general security features to further roster functions and realize access control for services at a Runtime. The following subsections all refer to Figure 2, which is showing the most simple setup of the architecture described above with just a Runtime synchronizing with a Broker + Runtime and a Client requesting the location of a service.

A. XMPP Accounts and JIDs

Each runtime uses its own XMPP account like *rt1@xmpp.com* and therefore has its own roster. As the roster of an XMPP account is stored on the XMPP server, all contained information is persistent, even if a Runtime is shut down.

We also set up an own XMPP account for each installed service of a Runtime. To ease the use of the system and to prevent naming conflicts, we propose the use of In-Band Registration [14] and let the runtimes create new accounts on installation of a service. For that, the following naming scheme should be used:
`[usernameOfServer].[servicename]@[domain]`

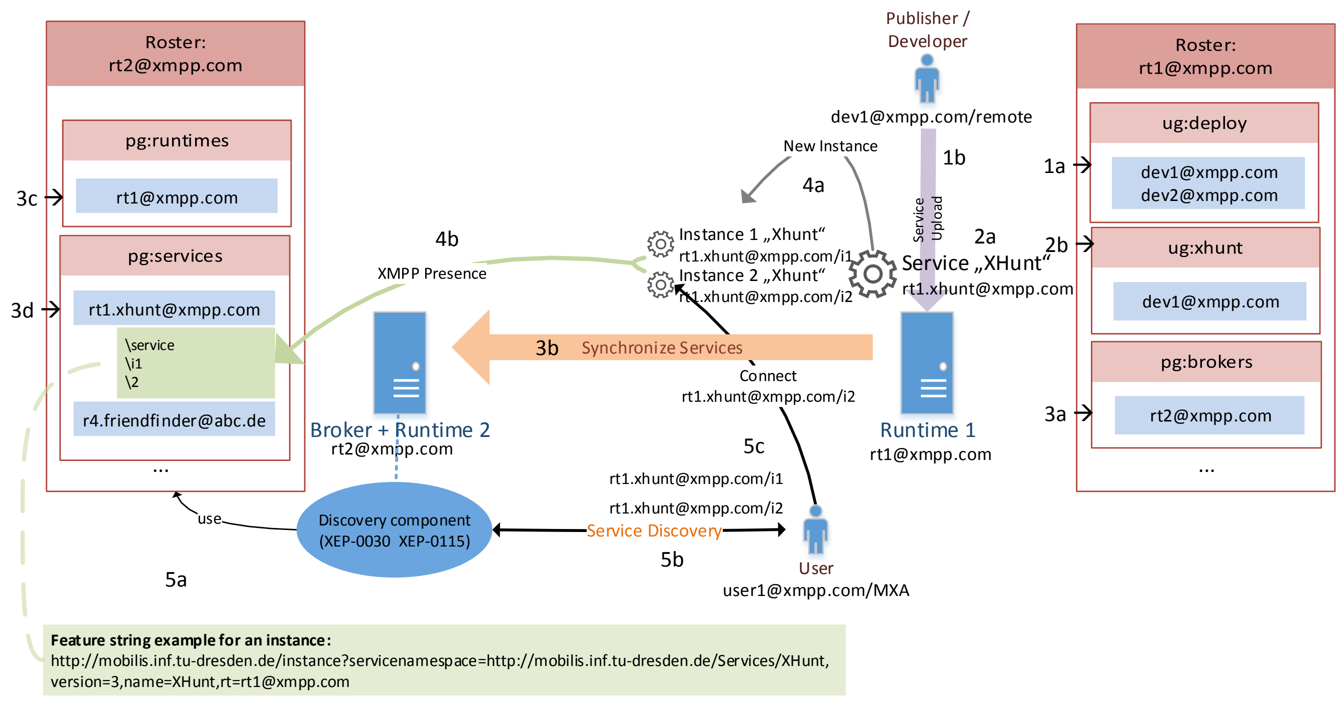
Every new instance of a service opens a separate XMPP connection with a new resource number. For an instance of a service XHunt in Figure 2 the result can be a JID like:
`rt1.xhunt@xmpp.com/i1`

Last but not least, client users and developers all also have their own XMPP accounts to assure their identity. Again, with the resource-enhanced JIDs in XMPP a client user can run multiple client devices (smartphone, tablet, notebook) with the same account.

B. Synchronization, Presence and Discovery

The main benefit of the roster in our approach is the propagation of service presence. Since every service on every runtime uses its own XMPP account, they all can be distinguished as separate entries in a roster. Now, if distributed Runtimes could share the JIDs of their installed services, they would not only be able to get notice of their availability, but also of new instances of a service. This is due to the fact that a new instance connects to the XMPP server as a new resource of its service XMPP account and propagates presence. This presence information is sent to all distributed Runtimes which have the particular XMPP account in their roster.

Two more issues need to get addressed. First, we need an extra resource for every service itself, to display service presence in case of no instance is active. We introduce a *service presence resource* which is getting active immediately after service installation:
`rt1.xhunt@xmpp.com/service`



Feature string example for an instance:
http://mobilis.inf.tu-dresden.de/instance?servicenamespace=http://mobilis.inf.tu-dresden.de/Services/XHunt,
version=3,name=XHunt,rt=rt1@xmpp.com

Figure 2. Roster for administration of service registry and access rights

Secondly, we need to get further service information from the roster entries when a runtime is processing a discovery request. To address this issue, we make use of XMPP Service Discovery [15] in combination with Entity Capabilities [16]. Each service and each instance has a *feature string* as described in [15]. This string contains all needed information about the service or the instance. We propose a feature string as follows (see example in Figure 2):

```
{myDomain}/{service|instance}?{pm1=value1,pm2=value2,...}
```

Service denotes a service presence resource while *instance* signalizes a concrete service instance running at a Runtime. The parameters are key-value pairs needed for service discovery. We used the keys *servicenamespace*, *version*, *name* and *rt* for our purposes, but this can be extended if needed.

To prevent unnecessary heavy network use and to speed up the processing of discovery requests, Entity Capabilities is utilized as it realizes a caching mechanism working together with XMPP presence. If a runtime needs the feature string of available resources in the roster, it usually has to send a disco#info request just for the first time. After that it caches the information locally, as long as the cached information doesn't change. Once it changes, the server sends out a new presence notification with the new hash value of the feature string. The client then has to issue the disco#info request again.

C. *Roster Groups*

Another useful feature of the XMPP roster is the ability to assign entries to roster groups. Our approach uses this mechanism for two different tasks: user access control and M2M access control.

1) User Access Control: It is essential that just certain people have access to special functions and resources of a Runtime. E.g., just authorized users should be able to install new services, reconfigure installed services or have access to the runtime configuration itself. Usually this can be realized by security groups. Each group gives its members access to a resource, which is represented by the group.

We map the described principle of security groups to the roster, by making use of *roster groups*. Every roster group is holding roster entries, which belong to particular JIDs. These JIDs can be checked against XMPP requests from certain users for a special resource.

To make these special roster groups distinguishable from other types of roster groups, we introduce a prefix tag *ug:* (user group). This and later tags can also be used for filtering roster groups in a GUI. We obtain the three following roster groups for access control:

- *ug:deploy* - User has the right to deploy new services.
- *ug:[serviceName]* - User is allowed to change an installed service with the given name.
- *ug:administrators* - User can change runtime configuration.

2) Peer Access Control and Discovery: From the machine-to-machine perspective we want to synchronize only peers which are trusting each other. We make use of roster groups for this issue again. If administrators want to allow their Runtime to synchronize, they put the JIDs of the Brokers they trust in a roster group *pg:brokers* where *pg* means *peer group*.

The other direction is represented by the group *pg:runtimes* which is used by a Broker to check if the Runtime is allowed to publish services. A Broker + Runtime has entries in both groups.

The last roster group represents all installed services to enable fast search of services during service discovery.

- *pg:runtimes* - List of Runtimes from which a Broker receives service updates.
- *pg:brokers* - List of Brokers to which a Runtime publishes service updates.
- *pg:services* - Installed services of all runtimes (local and remote) and their running instances (different resources).

D. Typical Scenario

A short scenario following the numbering in Figure 2 should make the described principles of the roster concept more comprehensible. In this scenario, a developer tries to publish a new service ”XHunt” on Runtime 1 which should be visible on Broker + Runtime 2 where a user tries to connect to the newly created service.

1) Service upload
    a) Check if uploading user is in ug:deploy.
    b) Service upload is allowed and executed by the developer.
2) Service installation
    a) Create XMPP account for the new service using In-Band Registration.
    b) Create roster group with scheme ug:[servicename] and add uploading user by default.
3) Runtime synchronization
    a) Get all trusted Brokers from roster group pg:brokers.
    b) Send JID of new service to trusted Brokers.
    c) Check if the JID of from attribute is a trusted Runtime in pg:runtimes.
    d) Add new service JID to roster group pg:services.
4) Starting new instances
    a) New instances connect as resources of their service XMPP account.
    b) Presence is sent by the service itself (service presence resource) and the instances to other Brokers to show their availability in real-time.
5) Discovery
    a) On a discovery request, the responsible Broker explores the roster group pg:services. XMPP Service Discovery and Entity Capabilities are used to get and process the service specific feature string for each resource of any entry from local cache or by a separate disco#info request from the JID itself.
    b) The user receives a response with JIDs of services matching his request and further information.
    c) The user can connect to a certain service instance directly using its JID.

VI. Case Study

To prove the conceptional approach, we implemented the proposed functions in our XMPP-based Service Environment Mobilis [9] as a case study. The code is open source and can be found on GitHub [17]. This is just an example of how the approach can be implemented in any XMPP-based system.

Figure 3 thus describes all involved protocols and the components on a conceptual level. It shows how the approach from Section V can be realized.

A. Discovery Protocol

The Discovery Protocol is responsible for the service discovery and the creation of new service instances. The Discovery Service handles all server-side communication of the coordinator protocol. We implemented three different algorithms for general service discovery, filtered discovery for a certain service, and to create a new service instance on an appropriate remote runtime that is offering the requested service.

All three algorithms have in common that they explore connected XMPP resources in the roster. They process all resources for each entry in the roster group pg:services. XMPP Service Discovery and Entity Capabilities are utilized to get the feature string of every resource. Then all algorithms decide their own way if either a resource matches their request to process or not.

B. Deployment Protocol

This protocol takes care of all service deployment features. Developers are able to install and update services, configure services and uninstall them. The Deployment Service uses the user groups to ensure that just authorized users can deploy and change services on a Runtime.

C. Runtime Protocol

The Runtime Protocol includes all communication between Runtimes and Brokers to synchronize their services. It covers two different cases:

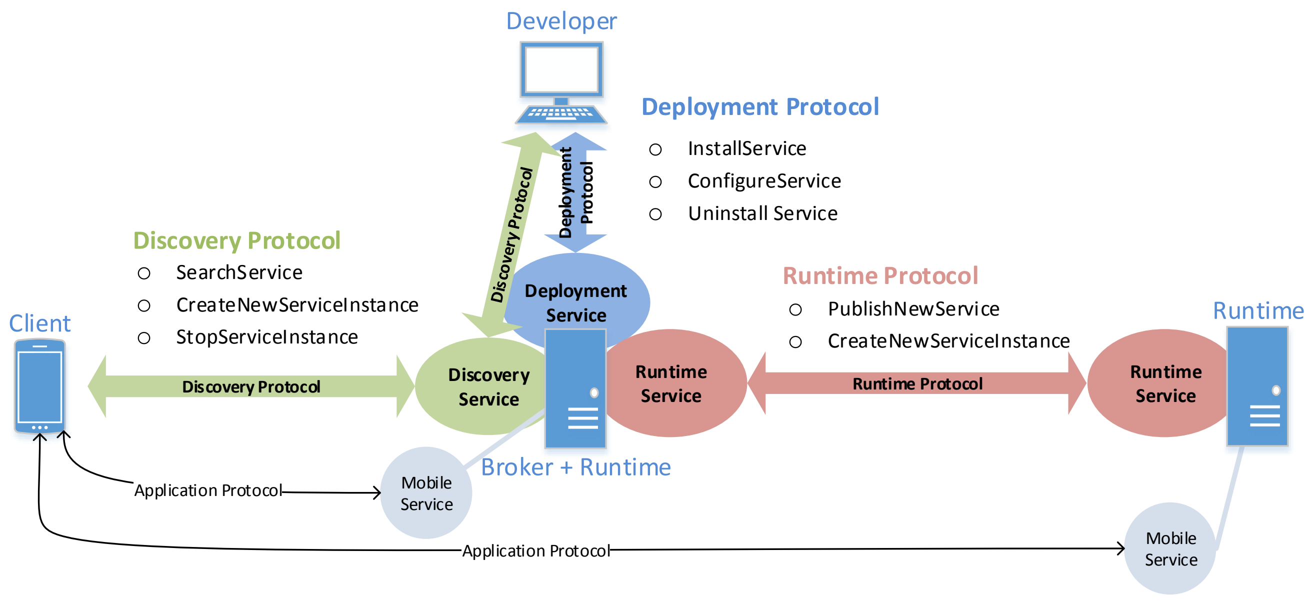


Figure 3. Involved protocols and components in our Mobilis-based case study

1) Publish a new Service: This case applies if a new service is installed on a Runtime. After the Runtime has got the list of all trusted Brokers from its roster it sends an XMPP IQ of the type set as shown in Listing 2.

Listing 2. Publish request of a new service

```
<iq id="m_32" to="rt1@xmpp.com/runtime"
    from="rt2@xmpp.com/runtime" type="set">
  <publishNewService
   xmlns="http://mobilis.inf.tu-dresden.de/runtime">
   <serviceJID>rt2.xhunt@xmpp.com</serviceJID>
  </publishNewService>
</iq>
```

As response it receives an empty IQ result for successfully publishing on one of the distributed runtimes or an IQ error otherwise.

2) Synchronize all Services: There are several cases making it necessary, that Runtimes and Brokers can synchronize all their services at once. This happens after the initial subscription handshake or after a shut down phase of any Runtime in the cloud. Hence, we simply extended the function of the IQ in Listing 2. For such a case, the request type is changed to get and can contain multiple serviceJIDs.

Listing 3. Sync. request of all services

```
<iq id="m_36" to="rt2@xmpp.com/deployment"
    from="rt1@xmpp.com/deployment" type="get">
  <publishNewService
   xmlns="http://mobilis.inf.tu-dresden.de/runtime">
   <serviceJID>rt1.geotwitter@xmpp.com</serviceJID>
   <serviceJID>rt1.xhunt@xmpp.com</serviceJID>
   <serviceJID>rt1.friendfinder@xmpp.com</serviceJID>
  </publishNewService>
</iq>
```

If the requested Broker is a Broker + Runtime the response also contains all services of it.

Listing 4. Sync response of all services

```
<iq id="m_36" to="rt1@xmpp.com/Deployment"
    from="rt2@xmpp.com/Deployment" type="result">
  <publishNewService
   xmlns="http://mobilis.inf.tu-dresden.de/runtime">
   <serviceJID>rt2.friendfinder@xmpp.com</serviceJID>
   <serviceJID>rt2.ninecards@xmpp.com</serviceJID>
  </publishNewService>
</iq>
```

The process is initialized by the Runtime that is coming online after a shut down phase or the certain Runtime that adds a Broker in case of subscription handshake.

VII. Evaluation

We obtained several benefits from building our distributed service environment up on the proposed XMPP roster concept. There is the ability to discover services in an inter-domain service cloud (by XMPP-Server federation), the possibility to scale-up the architecture in general and realizing load balancing in special.

But the proposed approach also results in additional communication and processing overhead. Especially because a runtime is using its XMPP roster instead of a local database, the key-value pairs describing an entity have to be retrieved directly from the entity using XMPP Service Discovery. This needs a lot of additional time in comparison to a local database query. Hence, we absolutely recommend the use of Entity Capabilities as well, because it caches an entity's feature string if it did not change since the last request.

To evaluate the performance costs of our approach, we set up three test cases as shown in Figure 4. Case A serves as indicator for the classical single-server architecture. It doesn't

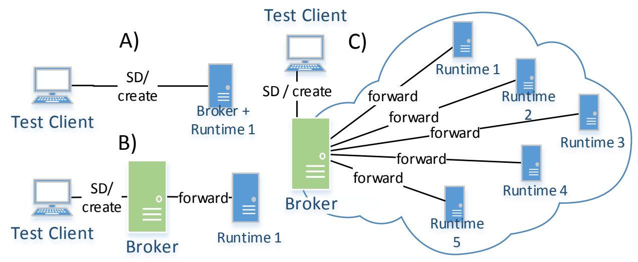


Figure 4. Test scenarios to measure the administrative costs of the approach

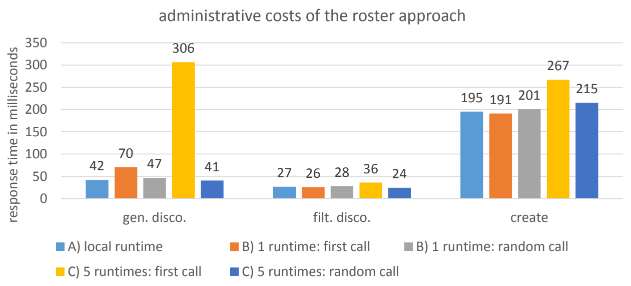


Figure 5. Administrative costs of the roster approach

make use of the proposed roster approach and was the only possible setup in the previous Mobilis architecture. Case B uses the roster approach with just one remote Runtime and a Broker that uses the own roster to process service discovery (SD) requests and to forward create instance requests. The last scenario in C is an upscaled architecture of case B with five remote Runtime servers.

In all cases, four services are installed on each Runtime. Three of the four services on every Runtime are running five instances at the beginning of the test. All these instances can get discovered as instance resources in the rosters of the Broker in case B and C. Additionally they have one *service presence resource* per service of a runtime. In summary that means $15+4 = 19$ resources for case B and $5*(15+4) = 95$ resources for case C to process in the Broker rosters.

The three algorithms, described in Section VI-A are called for the three possible client requests on the Broker. The responses for general service discovery contain all available services in the cloud. A filtered service discovery delivers all running instances for a requested service. The create instance request for a certain service is performed as a two step creation process. First, the manager retrieves the list of all Runtimes in the roster supporting the requested service. Secondly, it forwards the create request to a randomly chosen runtime (alternative load balancing: round robin or by load), that sends back the response containing the JID of the created instance.

We measured the response time at the test client for each request message. For the cases B and C we also logged the latency of the first request and a later random request. This was done to address the impact of Entity Capabilities.

Figure 5 shows the results for each request message in dependency to the driven test case setup. As can be seen, the latency for a general service discovery is much faster in case A (42 ms) in comparison to the first call in case B (70 ms), but especially in case C (306 ms). This is due to the fact, that for each connected resource in the roster group pg:services no feature string is cached yet. This impacts particularly on the latency of case C, where 95 separate Service Discovery requests have to be sent.

A later random call can reach the response time of A (B: 47 ms, C: 41 ms) because the needed information just has to be restored from the local cache of the Broker. The results for the filtered service discovery are almost the same pattern in the diagram, but have just a less average latency and a smaller deviation between all cases. This happens because the algorithm for the filtered discovery can skip entries that don't match the requested service namespace.

The test for a create message is showing nearly no difference between case A (195 ms) and B (191 ms / 201 ms). The Broker in B has to manage just one distributed runtime. So there is no much choice available to forward the request to. In contrast, the Broker in C has to process 95 resources and to choose from 5 possible runtimes. This is why more time is needed for both, first call (267 ms) and random call (215 ms).

The results are showing that the proposed approach needs more time for the first call of a resource's feature string, but can almost hit the speed of a local registry for a later random call.

VIII. Conclusions

Our approach shows how XMPP can be leveraged to realize flexible service hosting and discovery for mobile computing scenarios and/or the Internet of Things. Only standard XMPP extensions are used which are widely available as free server implementations as well as client libraries for many platforms. Our key idea is to represent client users, developers, hosts, and mobile services as XMPP users enabling presence exchange amongst all these entities. Roster groups are used as an efficient tool to manage access control and search. The evaluation shows that due to the use of the Entity Capabilities extension the performance is nearly as good as for the centralized setup.

In future work we want to further evaluate how the approach supports hosting of services on mobile devices. We already implemented a single service Runtime for Objective C which shows the feasibility of the approach. But further work has to be done in this field especially regarding the limited resources and occasional connection degradation or loss on mobile devices.

References

[1] R. Lübke, D. Schuster, and A. Schill, “Mobilisgroups: Location-based group formation in mobile social networks,” in *Second IEEE Workshop on Pervasive Collaboration and Social Networking (PerCol)*, Seattle, WA, USA, 2011.

[2] D. Schuster, D. Kiefner, R. Lübke, T. Springer, P. Bihler, and H. Mügge, “Step by step vs. catch me if you can - on the benefit of rounds in location-based games,” in *Third IEEE Workshop on Pervasive Collaboration and Social Networking (PerCol)*, Lugano, Switzerland, 2012.

[3] S. Bendel, T. Springer, D. Schuster, A. Schill, R. Ackermann, and M. Ameling, “A service infrastructure for the internet of things based on xmpp,” in *IEEE International Conference on Pervasive Computing and Communications (PerCom)*, San Diego, CA, USA, 2013.

[4] P. Saint-Andre, “XMPP: Lessons Learned from Ten Years of XML Messaging,” *IEEE Communications Magazine*, vol. 47, no. 4, pp. 92–96, 2009.

[5] M. E. Gregori, J. P. Cámara, and G. A. Bada, “A Jabber-based Multi-Agent System Platform,” in *Proc. of the 5th ACM Joint Conference on Autonomous Agents and Multi-Agent Systems*, 2006, pp. 1282–1284.

[6] L. Stout, M. A. Murphy, and S. Goasguen, “Kestrel: an XMPP-based Framework for Many Task Computing Applications,” in *Proc. of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS)*. ACM, 2009, pp. 11:1–11:6.

[7] D. Kovachev, Y. Cao, and R. Klamma, “Building mobile multimedia services: a hybrid cloud computing approach,” *Multimedia Tools and Applications*, pp. 1–29, 2012.

[8] B. Dodson, A. Cannon, T.-Y. Huang, and M. S. Lam, “The Junction Protocol for Ad Hoc Peer-to-Peer Mobile Applications,” [Online] http://mobisocial.stanford.edu/papers/junction.pdf, 2011.

[9] D. Schuster, R. Lübke, S. Bendel, T. Springer, and A. Schill, “Mobilis - comprehensive developer support for building pervasive social computing applications,” in *Networked Systems (NetSys)*, Stuttgart, Germany, 2013.

[10] G. Huerta-Canepa and D. Lee, “A Virtual Cloud Computing Provider for Mobile Devices,” in *Proc. of 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, 2010, pp. 1–6.

[11] S. N. Srirama, C. Paniagua, and J. Liivi, “Mobile web service provisioning and discovery in android days,” in *Proceedings of the 2013 IEEE Second International Conference on Mobile Services*. IEEE Computer Society, 2013, pp. 15–22.

[12] D. Bernstein, E. Ludvigson, K. Sankar, S. Diamond, and M. Morrow, “Blueprint for the Intercloud-Protocols and Formats for Cloud Computing Interoperability,” in *Proc. of the 4th IEEE Conference on Internet and Web Applications and Services (ICIW)*, 2009, pp. 328–336.

[13] E. Konieczny, R. Ashcraft, D. Cunningham, and S. Maripuri, “Establishing presence within the service-oriented environment,” in *Aerospace conference, 2009 IEEE*. IEEE, 2009, pp. 1–12.

[14] P. Saint-Andre, “XEP-0077: in-band registration,” http://xmpp.org/extensions/xep-0077.html, 2012.

[15] J. Hildebrand, P. Millard, R. Eatmon, and P. Saint-Andre, “XEP-0030: Service discovery,” http://xmpp.org/extensions/xep-0030.html, 2008.

[16] J. Hildebrand, P. Saint-Andre, R. Troncon, and J. Konieczny, “XEP-0115: Entity capabilites,” http://xmpp.org/extensions/xep-0115.html, 2008.

[17] TU Dresden, “Mobilis - pervasive social computing using XMPP,” https://github.com/mobilis, 2014.