A Concept for Interoperable IoT Intercloud Architectures

Philipp Grubitzsch, Thomas Springer, Tenshi Hara, Iris Braun and Alexander Schill
Chair of Computer Networks, School of Engineering Sciences
Technische Universität Dresden, Dresden, Germany

Keywords: IoT Cloud, Interoperability, Intercloud, Broker.

Abstract: Cloud platforms have evolved over the last years as means to provide value-added services for Internet of Things (IoT) infrastructures, particularly smart home applications. From different use cases the necessity arises to connect IoT cloud solutions of different vendors. While some established platforms support an integration of other vendors' systems into their own infrastructure, solutions to federate IoT cloud platforms can hardly be found. In this paper, we analyze existing IoT cloud platforms with respect to their similarities and derive a concept of an *Intercloud Broker (IB)* that enables the establishment of an IoT Intercloud to support interoperability of cloud-based IoT platforms from different vendors. To demonstrate the feasibility of our approach we evaluated the overhead introduced by the Intercloud Broker. As the results show, the IB can be implemented with minimal overhead in terms of throughput and delay even on commodity hardware.

1 INTRODUCTION

In recent years, cloud computing (Platform as a Service – PaaS) has been established as an important model to provide value-added services (data consumers) with data from devices located in smart homes (data producers) in a decoupled way. Those services are often the base of the "smartness" of things in this domain (Internet of Things – IoT). They collect and aggregate sensed information, make decisions based on this information and finally control one or more devices in the home. For instance, a proximity trigger recognizes a house owner coming home and a rule service unlocks the front door and switches on lights and air conditioning.

Interoperability between IoT clouds is motivated by different use cases, for instance if devices of a single user are managed by clouds of different device vendors, if multiple locations of a user are equipped with smart home systems of different vendors, or if different users cooperate, each with another smart home cloud provider.

If a service is developed just for one of the depending cloud platforms, it must be nonetheless possible to transparently access data from other clouds, where device data of the affiliated users are located. Efforts towards cloud interoperability are usually called Intercloud or Cloud Federation (Toosi et al., 2014). From the analysis of IoT cloud solutions it becomes apparent that cloud interoperability across platforms from different vendors is only rarely supported. Most of the existing platforms only foresee an integration of other vendors' systems and data into their own cloud. Support for an Intercloud setup that integrates IoT cloud systems in a peer-to-peer (P2P) manner can hardly be found.

As part of its cloud architecture NIST has defined a component called Cloud Broker (Liu et al., 2011). Its role is to provide data integration between cloud consumers and multiple cloud providers. Referring to the Intercloud, we propose a concept of an IoT Intercloud Broker (IB) to enable interoperability between IoT cloud infrastructures from different vendors. IoT cloud infrastructures are mainly PaaS solutions providing abstractions for data consumption, device control, and management. With the focus on data consumption and device control the proposed IoT IB is responsible for adapting vendor-dependent APIs.

Contributions of this paper are threefold. First, we analyze a representative set of IoT PaaS clouds to show that existing PaaS cloud solutions share sufficient similarities to allow a mapping between different solutions with respect to communication APIs and device model (cf. section 2). Second, we introduce a conceptual architecture for an IoT IB and specify the interfaces of the IoT IB that allow for an Intercloud setup of IoT cloud infrastructures from different vendors (cf. section 3). Third, we demonstrate with a performance evaluation that the proposed IB can be implemented on commodity hardware with minimal

Grubitzsch, P., Springer, T., Hara, T., Braun, I. and Schill, A.
A Concept for Interoperable IoT Intercloud Architectures.
DOI: 10.5220/0006375107040713
In *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*, pages 676-685
ISBN: 978-989-758-243-1
Copyright © 2017 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

overhead in terms of throughput and delay introduced to Intercloud data exchange (cf. sections 4 and 5).

2 SERVICE APIS OF IoT CLOUDS

In the following section we present the results of an analysis of smart home and IoT clouds with respect to: *communication* for requesting device data and controlling devices in real-time, *device data model* to understand and express the semantics of device states, mechanism to provide *3rd-parties access* to the device data and control options, and support for *cloud federation*. Our goal is to demonstrate that PaaS clouds adopt comparable concepts and technologies that can be mapped to each other. In addition, we derive information to guide the design of the envisioned IoT IB.

The selection of considered cloud platforms is not exhaustive, but we picked representative examples from different categories. The selection was made with respect to their importance to the market and the target audience, and their intended field of use respectively. Thus, this analysis includes popular proprietary solutions like Google Nest[1], Samsung Artik[2] Cloud, and AWS IoT[3], which mainly targets end-user products, but also open source solutions like DeviceHive[4] for do-it-yourself IoT enthusiasts. The results of our analysis are presented in Table 1. In the following, all criteria will be briefly introduced. We are going to discuss the results at the end of this section.

For an in-depth comparison, we separated *communication* into three high-level abstractions for our analysis: *Pull* to retrieve full views of device states, *Push* to keep track of state changes in real time, and *Control* to change cloud-controlled device states.

2.1 Get Data with Pull Semantics

The current state of devices can be retrieved by clients when they initiate a corresponding *Pull* request. It is essentially useful for updating full views with large data sets for a particular point in time or if updates are required in a low frequency. A periodic Pull for updated data is also known as *Polling*. To overcome high protocol overhead, pending long-lived requests (*Long-Polling*) have emerged. The server can respond with new data to the open request, or with a timeout response if the life-time of the request expires. Because clients have to send new requests after each response, both Polling mechanisms are no real Push technologies (cf. next subsection). In case of frequently updated data, even Long-Polling becomes inefficient. Pull-based data requests differ in a set of aspects that we consider relevant to achieve interoperability. They are discussed in the following.

Different ***protocols*** *(Prot)*[5] can be used to implement pull-based data request. For the analyzed platforms the dominant solutions are RESTful APIs based on HTTP and the Websocket protocol[6] (WS). Both have become standards for the request-response pattern in the web. While all platforms support REST/HTTP, the Google Nest and Kiwigrid[7] platforms additionally offer Websocket support.

Filters *(Filt)* are the most important parameter for querying data, as they specify the information of interest. Filter are specified based on a device model (cf. subsection 2.4) and a query language. All clouds provide slightly different query methods to filter device lists, including parameters for ids, types, owners, single affiliated data point keys or values of devices that match an arbitrary condition. The majority of the investigated clouds apply key-value filters only to a single target object (e.g., list with IDs, owner or types of devices). This is equal to the Select-From-Where SQL statements of relational DBs. For each target object an own request is provided by the API (e.g., GetDevicesByOwner, GetDeviceByType). This results in *Path*-like browsing for detailed device data.

Contrary to this filter concept, the Kiwigrid cloud supports full key-value semantics (KVS) to filter for arbitrary devices without any starting object. This API provides an own query language which has to be used for the filter definition. A related concept is known from schema-less NoSQL databases and yield a very powerful filter engine.

Projections *(Pj)* further reduce filtered results. Related to a database table, a filter reduces the row set, while a projection reduces the column set. It is useful, if only a subset of information (e.g., GUID or device type) is required.

Paging *(Pg)* is the method to retrieve just a subset of the full result. It works with a *limit* (how many devices to respond) and an *offset* (start device for the next devices till limit) parameter. Nonetheless, some clouds just support limit as the only parameter, and thus no real paging.

[1] https://developers.nest.com/documentation/cloud/concepts

[2] https://developer.artik.cloud/documentation/api-reference/

[3] http://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html

[4] http://devicehive.com/restful/

[5] The abbreviations introduced in brackets for all criteria are the keys to Table 1 were these abbreviations are used to name the columns.

[6] https://tools.ietf.org/html/rfc6455

[7] http://developers.kiwigrid.com/wiki/EM-SDK

Table 1: Features of the investigated Cloud-APIs.

Cloud	Communication													Device Model	3rd Party		Fed.
	Pull API							Push API				Control API					
	Prot	Filt	Pj	Pg	H	S	Rl	Prot	Filt	PU	RA	Prot	Mth		Mth	U2U	
Google Nest[1]	WS, HTTP	Path	√	○	×	√	√	WS	TB	×	○	HTTP	PDU	Single JSON Doc, static D.	OA2	×	√
Samsung Artik[2]	HTTP	Path	√	√	√	√	√	WS, HTTP	TB	×	○	HTTP	RPC	JSON, NS + D.-Types	OA2	√	√
AWS IoT[3]	HTTP, WS, MQTT	Topic, Path	×	×	×	×	√	HTTP, WS, MQTT	TB	×	×	HTTP, WS, MQTT	PDU	JSON, NS + inheritable Thing-Types	UC	√	√
MS Azure IoT Suite[13]	n/a	Topic, Path	n/a	n/a	n/a	n/a	n/a	MQTT, AMQP, HTTP, WS	TB	n/a	n/a	MQTT, AMQP, HTTP, WS	PDU, RPC	JSON schema for Device Twins	n/a	n/a	n/a
Kiwigrid[7]	WS, HTTP	KVS	√	√	○	√	√	WS	CB	○	√	WS, HTTP	PDU	YAML, NS + inheritable D.-Classes	OA2	√	√
Digi/Etherios[14]	HTTP	Path	×	√	√	√	○	TCP, HTTP	TB	√	○	prop.	RPC	XML/JSON schema	UC	○	×
Device-Hive[4]	HTTP	Path	×	○	√	√	√	WS	TB	×	○	WS, HTTP	RPC	JSON D.-Classes	OA2	○	×
Eurotech Everyware[8]	MQTT, HTTP	Path	√	√	○	√	√	MQTT	CB	×	√	MQTT, other	RPC	XML/JSON schema	UC	○	×

History *(H)* introduces two parameters *startDate* and *endDate*, and is an additional filter for the resulting device states (based on concrete values of data points) over time. Without given history parameters the result usually includes just the current device state. The Kiwigrid and Eurotech[8] clouds support historic data only by an API extension. Google Nest does not provide access to historic data at all.

Sorting *(S)* the result set based on a specific row requires parameters for the *key* (e.g., device type), a *method* (e.g., alphabetic order), and a *direction* (ascending, descending).

Result List *(Rl)* is the main parameter of the query response. Based on several successive Pull-based queries it describes, if it is finally possible to retrieve a full view of the devices states. This is important, as a request for device data with our proposed IB API needs to be possibly mapped to cloud-internal methods, and the depending response should deliver the same results.

2.2 Get Data with Push Semantics

Based on the interaction schema of Publish-Subscribe clients can use push semantics to get informed about frequent but small changes of device states in real-time. After an initial *Subscribe* request, clients do not need to send subsequent requests to get new *Notifications* about changed data. If a client doesn't want to receive notifications any longer, it sends an *Unsubscribe* request. The *Notification* phase between both requests is equal to data streaming controlled by the server. *Push-based APIs usually work on top of a message bus, rsp. queue or a stream where data is not persisted before being processed.* Databases of Pull APIs may persist data from those streams.

Protocols based on REST can fully map Subscribe and Unsubscribe requests. Nonetheless, during the Notification phase the servers cannot operate stateless anymore, as they have to keep clients subscription information. Hence, HTTP-based communication with former client-server roles doesn't work. For Web-based Push transport between server and client, the *Websocket* (WS) protocol has recently become the state-of-the-art. Basically, a WS is a long-lived full-duplex TCP-based connection initiated via an Upgrade request on an existing HTTP connection. It does not come with a Publish-Subscribe, but often is the base for a higher level application layer protocol supporting this pattern (e.g., HTML5 server-sent-events[9]). The main advantage of WS is, servers are enabled to send messages without a previous client request. In the IoT and Messaging domain, protocols[10] like MQTT, CoAP, XMPP and others have been used for Push-based communication. Some clouds also support HTTP, but the client has to host its own webserver and provide a *Callback URL* to the cloud. The cloud then sends HTTP-POST requests to notify the

[8] http://everywarecloud.eurotech.com/doc/ECDev-Guide/latest/default.asp

[9] https://www.w3.org/TR/eventsource/

[10] https://iotprotocols.wordpress.com/

client about new data.

Filters work similar to the Pull-based API. For Push-based Publish-Subscribe systems, two types of filters can be distinguished. *Topic-based* (TB) filtering uses statically named (addresses) data channels and allows filtering of key-values within a subset (defined by the topic) of all possible key-values. Topic names are usually defined by a userID, device GUID or type. Some systems hierarchically group topics based on wildcards for the topic name, which allows filtering on parent topics. The most advanced approach is *Content-based* (CB) filtering. It works on an arbitrary combination of all possible key-value pairs. It can also include Boolean operators to compare value conditions. Content-based Publish-Subscribe is still a huge research field and is addressed mainly by *DSMS* and *CEP* systems (Cugola and Margara, 2012).

Periodic Updates *(PU)* are synchronous notifications to a client. The main parameter is the *rate*, describing static time intervals between two new notifications. An optional parameter is an *aggregation* method to aggregate on simple (e.g., last value) or complex functions (e.g., average, median).

Reply Addressing is feature to distinguish different queries on client side. A client submits an *address* parameter, where it expects to fetch the notifications for a specific query. Most clouds just support server-controlled addressing. In both cases the response from the server can optionally include the exact address as parameter. For HTTP-based Push, the client parameter is the Callback URL.

2.3 Control-ability of Devices

A basic requirement of IoT and Smart Home clouds lies in controllability. This is in contrast to mere monitoring systems. A device control interface offers methods to update a device in its virtual representation at the cloud. The cloud is then responsible to resolve this change into device specific commands and to forward them to the physical device. Like Pull-based data requests, it is generally realized by the request-response pattern. A control request must include an *identifier* of the device to be updated and information about what is intended to change at the physical device.

Protocols – The control interfaces can be easily implemented as RESTful API (often HTTP or WS), as it comes with *POST* and *PUT* methods for creating and updating a certain resource. Therefore, the URL acts as device ID, while another parameter is the *method* of the control request described bellow. Nonetheless, there are also no drawbacks when using alternative approaches and protocols.

Methods (Mth) to control devices are very different. A classic approach are *remote procedure calls (RPC)* or their object-oriented counterparts, remote method invocations, respectively. A client has to add the *method* to be called and the *method specific parameters* to its request. The most cloud APIs follow this approach. The main drawback is, that the service has to know about each device specific method. For a unified usage an additional service description needs be provided to the clients. A sophisticated method by some clouds is to use the data model itself to control a device, namely *partial device representation update (PDU)*. It is very similar to changing an HTML DOM representation through a partial update[11]. Advantageously, the service can fetch the device model representation when reading its state and let the cloud platform do its job to translate changed data to the appropriate device command (abstracting device infrastructure is actually the main goal of PaaS IoT clouds).

2.4 Device Data Model

Device types as well as related sensor data and control commands that are maintained by an IoT cloud platform are represented by a device model. It usually includes information like ID, name, model, types of sensed values and the values itself, actions, states, read-write access, units, descriptions, and possibly many more. Equal to the class-instance relationship in object oriented programming, concrete device instances are often derived from templates, named *device type* or *device class*. Similar to this concept, some device data models also support (multiple) inheritance. That enables building of generic devices with basic parameters (e.g., GUID, manufacturer, model), more specific types like a “dishwasher”, or more concrete “dishwasher model Ω by vendor X”. Every reviewed cloud API provides its own device data model and format. While they compare in basic structure, the concrete device models largely vary. While the Google Nest device model only supports three concrete device types, relying on a simple JSON representation and does not support inheritance, the Kiwigrid device model has an underlying schema, supports inheritance, and is multi-hierarchical.

2.5 3rd-party Access

Value-added services are not naturally allowed to retrieve data of users’ devices. They are generally a

[11] https://cmatskas.com/update-an-mvc-partial-view-with-ajax/

third party (3rd-party) with respect to the user-to-cloud relationship of storing users' device data. Users must be able to grant fine-grained access for their devices. This can be device data (e.g., single key-values) and/or control abilities.

Methods to grant or retrieve access based on *user credentials* (UC) with login and password, or access token-based concepts like *Oauth 2.0*[12] (OA2). Nowadays, OA2 is the state of the art for granting 3rd-party access to own resources. It avoids exposing credentials while combining authentication and access control into a single operation.

User to User (u2u) access is another possible 3rd-party scenario. Imagine a neighborhood where people on vacation want to give other neighbors temporary access to devices (e.g., door lock or CCTV). Hence, we investigated if clouds allow sharing device resources across users. We distinguish no (e.g., forbidden), full (across all users in the cloud), and partial sharing (e.g., tenant or account based).

2.6 Cloud-federation Support

Some of the reviewed clouds' API support cloud federation (Fed.). Because our intention is to build a P2P-Intercloud, we were interested in the aspirations of cloud providers to interconnect with other clouds. It confirms our assumption of a general need for this idea. The main difference to our approach is that 3rd-party cloud providers have integrated their solution in each of those ecosystems.

2.7 Discussion of the Results

The Microsoft Azure IoT Suite[13] is more like a PaaS Framework than a ready solution. It provides building blocks with different implementations to compose a solution. There are neither reasonable constraints nor a complete data model nor defined interfaces. Thus, most of our evaluation criteria are not applicable (n/a). We included it for sake of completeness.

The **Pull API** only shows minor differences among all clouds. To the best of our knowledge, only the Digi/Etherios[14] cloud is not capable to deliver a full device state in the result list, while only utilizing the Pull API. For the request parameters (Pj, Pg, H, S), adaptation strategies have to be found for each cloud, if they support a certain parameter only partially, or not at all. E.g., the Nest cloud would need a solution for history parameters. As only the last state is supported, a possible strategy could be, to only deliver this last state, if the requested time interval includes the present time, otherwise nothing. Clouds missing projection capabilities could respond the full device projection, while their results need to be processed either in the adapter of the IB or at service client side.

The same applies to parameters (PU, RA) of the *Push API*. Here, the IB may add missing logic for aggregation of periodic updates and also introduce addresses instead of handler objects, which the most clouds only support. This can be achieved by utilizing CEP and IoT messaging protocols. For the Notification phase of the Push-based communication, all clouds provide communication on base of full-duplex protocols. Notably, with MQTT the Eurotech Everyware cloud already has in use a protocol designed for IoT Publish-Subscribe scenarios.

The most important differences of the clouds on both, Pull and Push API, are related to the *filters* and the *device model*. As they work together, a unified powerful solution needs to be selected for the IB, covering all vendor-dependent scopes. The proceed model for filtering are equal for *Path* and *TB*, rsp. *KVS* and *CB*. The firsts needs a structure to apply key-value filters (table/object, topic) on, while seconds work on all contents (stream, NoSQL DB).

The Kiwigrid Cloud is able to filter key-values with the same filter engine for Pull and Push. The Eurotech Everyware cloud uses a CEP Engine for Push data. Others' filters are limited to a predefined set of query operations rsp. filter on predefined topics. Because filtering with key-values is more flexible, our API should support it. The adaptation for Pull-based requests between NoSQL and relational databases is possible. Following the Entity-Attribute-Value (EAV) model, key-value queries can be mapped on schema-based databases. Mainly, with a well-defined EAV temporary DB, the required property mapping and relationship mapping are feasible. The reversal is also possible. Mapping between topic to content-based filters can be achieved by connecting to all topics, and then filtering the result with a CEP engine.

Interoperability of *device models* is of special importance since some of the communication aspects directly depend on a uniform/compatible interpretation of device data. In particular, the complex filters passed to the *Pull* or *Push* API and the *Control* API base on the device model. The heterogeneity of device models makes it fairly impossible to simply link APIs of different cloud providers. Complex mapping based on ontologies and data adapters would be necessary to establish a common base. In future, development can be facilitated, if a common high-level standard for

[12] https://tools.ietf.org/html/rfc6749

[13] https://docs.microsoft.com/en-gb/azure/iot-suite/iot-suite-what-is-azure-iot

[14] http://www.digi.com/resources/documentation/digidocs/90002008/default.htm

an IoT device model would be established by all stakeholders. Nonetheless, solving this issue is out of scope of this paper.

The previously discussed *Control API* method *PDU* is suggested to be used for the IB. State changes of a device can be mapped to certain RPC methods in the low level device abstraction. Even the two clouds supporting PDU natively, have yet done so.

For *3rd-party access*, a unified permission object, that can hold both, OA2 access tokens and UC could resolve differences related to the method. Further considerations are needed to address users among different clouds. This might be achieved by introducing domains to user names (e.g., alice@cloudB), like in other federated communication systems (e.g., e-mail, XMPP). Therefore, no additional user management is necessary on Intercloud layer.

As commercial cloud providers like Google and Samsung already support *cloud federation* (however, with selected partners only), the need for Intercloud communication in the IoT domain becomes evident.

3 IoT INTERCLOUD ARCHITECTURE

The architecture for the IB depicted in Figure 1 refines our prior work (Grubitzsch, 2015), to achieve an Intercloud communication to exchange device data rsp. control devices among cloud providers. Applied to XaaS definitions, the brokers task can be concluded as Device as a Service.

We envision an IB per IoT cloud. Our broker architecture is designed with a set of common components. These are mainly the *Client Connector*, the *Intercloud Gateway/Proxy* and the before mentioned *Cloud Adapter* working on top of an uniform Intercloud Data Model. They are glued together by three interfaces (IFace 1, 2, 3).

The ***Client Connector*** is responsible to delegate internal or external service client requests through *IFace 1* to the internal components of the Intercloud Proxy/Gateway. Figure 2 shows the supported messages of *IFace 1* with related parameters/objects. It is derived from our analysis in the previous section. We introduce a general *Permission* object for OA2 access tokens or UC to the base message. *Sampling* encapsulates periodic updates and the related aggregation functions. For both, Pull and Push based *Filters*, we suggest to utilize a NoSQL-like query language to support full key-value semantics. The *PartialDeviceMap* has device identifiers as keys and a partial representation of each device to support the discussed PDU. IFaces 2 and 3 are derived from IFace 1.

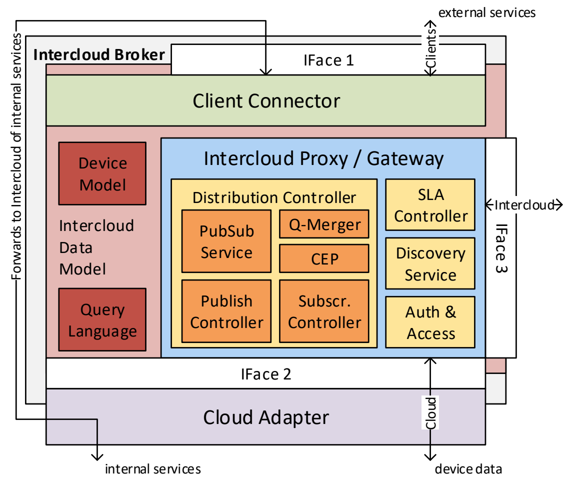


Figure 1: Architecture of the proposed Intercloud Broker.

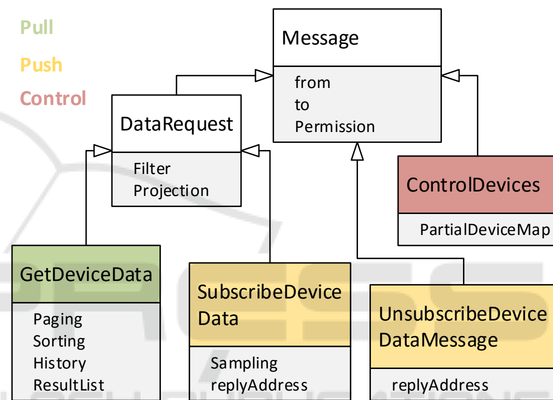


Figure 2: Messages with parameters processed by IFace1.

IFace 2 is an enabling interface to encourage developers to adapt cloud provider API behind this methods. Hence, a ***Cloud Adapter*** needs to implement this interface. It is marked by SERVE methods of IFace 1, as *serveGetDeviceData(...)*. Our analysis from section 2 shows that this should be feasible for most IoT clouds. *IFace 3* enriches messages with information necessary for Intercloud communication (e.g., initial requester, address of the messaging service receiving real time device events) and expects the *cloud address* in the *to*-field, where a message is supposed to be forwarded to. It is marked by FORWARD methods as forwardGetDeviceData(...).

The ***Intercloud Proxy/Gateway*** can directly forward an external request to the related local cloud via *IFace 2*, which has to be implemented by the discussed cloud adapter. Cloud-internal service calls would not be forwarded to the adapter of its own cloud again. But the main task of the Broker is to forward requests to other clouds. Therefore, the Intercloud Proxy/Gateway communicates with Brokers of other clouds via *IFace 3*. For that, it utilizes se-

veral components and services. Two of them shall be briefly discussed, namely the Discovery Service and the Distribution Controller.

The *Discovery Service* responses lists of clouds for each request to be forwarded. This can be dependent from queried devices in each request, to avoid unnecessary forwards to other clouds, but requires the discovery service to work with a partially replicated directory. Such a directory possibly stores information about devices, device types and access rights to be synchronized between trusted cloud partners. We plan to experiment with index-based search engines like Apache Lucene.

The *Distribution Controller* handles Push-based communication in the Intercloud. It utilizes a federated Publish-Subscribe service (pubsub) to deliver device event notifications to service clients via Publish-Subscribe. The Subscription Controller subcomponent administrates all client subscriptions forwarded to the Intercloud. The Publish Controller subcomponent administrates several publishers which republish device notifications received from the local cloud (via the cloud adapter) to the Intercloud.

Figure 3 shows the procedure from a client subscription propagated to the Intercloud till the delivery of the notification through all related base components. (1.) The client subscription is propagated to the Subscription Controller, which creates a subscription id, virtually representing pubsub topics (t1,t2,t3,...), and chooses a pubsub instance. The client will subscribe to this topic on the given PubSub Service instance. (2.) The Subscription is enriched by the pubsub address and the topic id. Then the Subscription Controller calls the Proxy to forward the subscription request. (3.) The Proxy uses information from the Discovery Service (DS) to forward the subscription to a local cloud adapter and/or proxies in the Intercloud, which will also forward it to their cloud adapters. The Adapter is calling a local cloud operation to register a new device subscription and (4.) link the proprietary notification handler provided by the cloud with the Publish Controller. For load-balancing reasons it uses multiple publishers to push data to a PubSub Service given in the forwarded subscription. (5.) Publishers will now start to send notifications to the Client via the PubSub Service (6.).

For our final architecture as shown in Figure 1, we hope to overcome the described redundancy issue on similar service requests (Grubitzsch, 2015) by introducing query merging. A Query-Merger (Q-Merger) work on filter semantics of different client subscriptions and is able to recognize covering filters as described by (Mühl et al., 2006). Instead of directly forwarding client subscriptions, the proxy will forward own

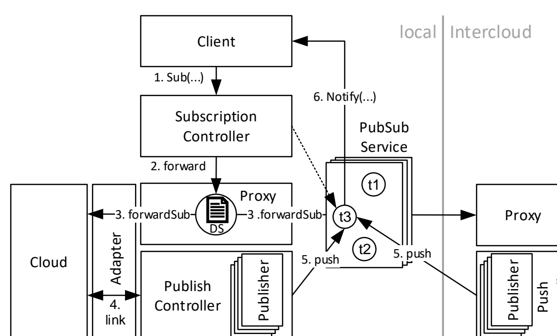


Figure 3: Push-Subscription in the Intercloud.

subscriptions with merged queries derived from the detected covering filters to the Intercloud. Thus, the remote clouds have to handle less subscriptions and will not redundantly transmit device notifications. We propose to utilize a CEP engine to retrieve back notifications for each client from the merged data stream, which will be republished to the clients by the broker.

4 IMPLEMENTATION DETAILS

We have built a first prototype of the IB for the Kiwigrid cloud. It is written in Java on top of the event driven framework Vert.x[15]. Vert.x utilizes the (multi-)reactor pattern and allows clustering of actors. Hence, it is well suited for development of distributed high performance cloud platforms. Each component could be realized as an actor, but our implementation only distinguishes between actors for the Publish Controller, the “link”-part of the Adapter, and all other components. The communication between the broker’s internal components utilizes the distributed event-bus of Vert.x. For the external communication through IFace 1 and 3 (cf. Figure 2), we choose XMPP with Openfire for the Broker, as it comes with all desired communication patterns such as authentication, entity addressing, and support for federated communication among several message brokers.

Since experiments with different XMPP-Brokers uncovered a limitation of the throughput for push notifications of device events, we decided to rebuild the pub-sub part on top of ZeroMQ’s XSub/XPub-Proxy[16] pattern written in Java[17]. (Happ et al., 2017) have shown that ZeroMQ is very suitable for huge IoT scenarios. Each actor of a Publish Controller has its own ZeroMQ Publisher. For now, the Intercloud data

[15]http://vertx.io/

[16]http://zguide.zeromq.org/page:all#The-Dynamic-Discovery-Problem

[17]https://github.com/zeromq/jeromq

model is the Kiwigrid device model and query language (similar to MongoDB[18]), which results in a powerful filter engine. The main goal of this first prototype is to keep details of the Discovery Service and Distribution Controller as simple as possible, even if this leads to communication redundancy. Hence, the current Discovery Service is realized as a simple list of trusted clouds. Each client request is simply forwarded to all known clouds in the Intercloud. The current Distribution Service comes without the proposed Query-Merger and CEP-Engine.

5 EVALUATION

To evaluate the feasibility of the introduced IB concept, we analyzed the overhead introduced by the IB. We especially examined the performance scaling of our broker in push scenarios, which cause significant load to IoT Cloud systems. The evaluation is based on the prototype implementation introduced earlier.

Figure 4 shows the testbed setup and the data flow. The hard- and software specifications of the three computer systems are provided in Table 2. All computers are connected via Gigabit Ethernet.

Senders are a set of simulated IoT devices which can be controlled by their total number as well as the notifications per second each device sends as messages to the IoT Cloud. As device type we only use a combined thermometer for inside and outdoor temperatures. The notification messages are encoded in JSON and have an average length of 465 Bytes. This includes a timestamp when the notification was generated. The receivers are simulated services, which can fully utilize the Broker's IFace 1 (cf. Figure 2). In our experiments they only use the PUSH part of the interface with different filters (device ID, owner, etc.) to only receive a subset of all devices in the Intercloud. Both, senders and receivers ran on the same machine.

The Kiwigrid IoT cloud was deployed as single node on the most powerful machine, due to high load produced during our experiments.

The broker was also running on its own machine to separate all performance related factors we are interested in from other processes. We measured the *CPU* utilization, memory allocation and network throughput every second for all broker involved processes (main component, Openfire, ZeroMQ-Proxy). Immediately after receiving a new message from the cloud, the broker adds an additional timestamp to the message. This way, we can later calculate the *delay* introduced by the broker. At the Publish Controller we

[18]https://www.mongodb.com

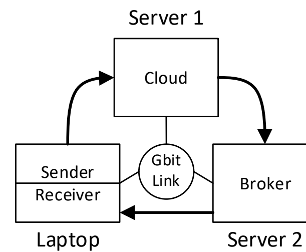


Figure 4: Data flow in the evaluation testbed.

Table 2: System specs used for evaluation.

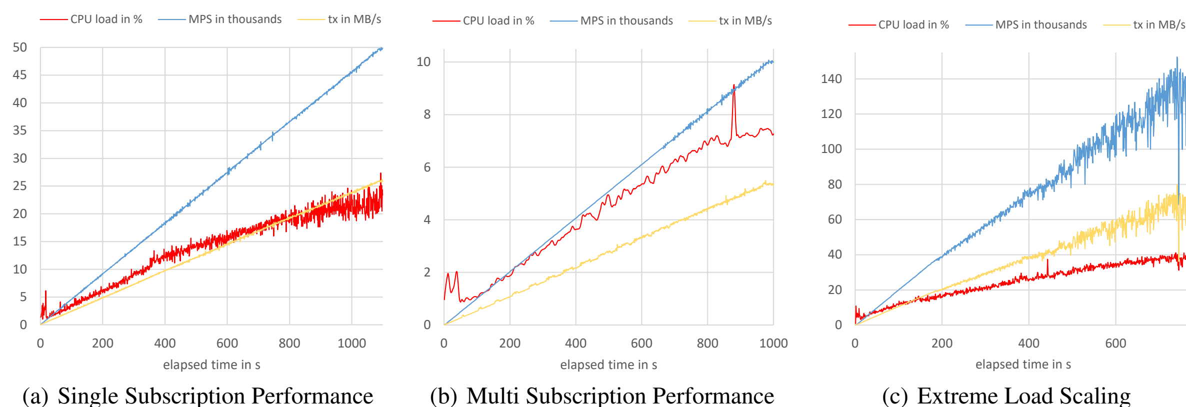
Laptop	Server 1	Server 2
Intel i7-4702HQ, 16 GB RAM, Windows 10 Build 14393, Oracle JDK 1.8.0_92	Intel i7-6700T, 16 GB RAM, Ubuntu 16.04, OpenJDK 1.8.0_121	Intel i7-4765T, 16 GB RAM, Ubuntu 16.04, OpenJDK 1.8.0_121

measured every second the message throughput (messages per second (MPS)) of the broker.

Even if this testbed is a single Cloud/Broker setup, its performance results are the same as if we would run a 2-Cloud/Broker setup, with devices sending their messages to cloud 1 and receivers listening on broker of cloud 2. The only difference is, that we would have to measure performance impacts of Publish Controller and pubsub on two different machines (cf. Figure 3). Due to the inevitable rise in complexity of our testbed setup to a minimum of five involved computer systems without any gain in knowledge, we decided to use the described reduced setup.

We ran three experiments. In our first experiment we analyzed how a single subscription scales for a variable number of message per second (MPS). Therefore, a single client has been subscribed to the broker with an empty filter, which matches all notifications from the cloud. During the experiment about every second a new IoT device was spawned, with each generating 50 MPS. The experiment was stopped after spawning 1k devices and a total load of 50k MPS. As Figure 5(a) shows, the CPU load appears to be linear or better. The delay introduced by the broker is below 1 ms in average. A maximum data rate of about 25 MB/s was transmitted to the receivers. The maximum memory allocation across all broker processes was about 6.8 GB. Because all components are written in Java, the memory allocation is dependent of the given maximum heap size. It was set to the maximum available RAM of the machine. Hence the JVM used much more memory than actually required in order to reduce garbage collection.

With our second experiment we tried to find out,



(a) Single Subscription Performance (b) Multi Subscription Performance (c) Extreme Load Scaling

Figure 5: Evaluation Results.

how the number of subscriptions impacts performance. Because the cloud is not able to process the same load as in our first experiment, we reduced the number of MPS per device. We spawned 1,000 devices before the experiment, each with an unique owner and generating 10 MPS. During the experiment, we spawned new subscriptions every second. Every subscription was exactly mapped to one device by a filter matching a single owner. The total message throughput achieved was 10k. Figure 5(b) shows very similar load behavior to our first experiment. We assume that the broker either scales at least linearly with the number of subscriptions, or the scalability of the broker is only dependent on the overall processed MPS. Again, the introduced delay was below an average of 1 ms, and allocated memory was ca. 6.4 GB.

Our last experiment targeted the highest message throughput we could achieve with our available cloud. We spawned 100 IoT devices, each generating a single message per second before the start of the experiment. During the experiment, we spawned 2 subscriptions per second (filter matched all devices) until the message delay through the whole system (Sender-Cloud-Broker-Receiver) exceeded 100 ms. As Figure 5(c) shows, the maximum CPU load at the Broker was still below 40%, while processing up to 140k MPS with a network throughput of up to 70 MB/s which is fairly $^2/_3$ of Gigabit Ethernet. There is also a jitter starting for the message throughput, due to the high CPU load in the cloud. The delay overhead introduced at the broker remained below 1 ms, while memory allocation was at a maximum of 6.9 GB.

6 RELATED WORK

There has been initial research on the need and the requirements of Intercloud computing (Toosi et al., 2014). Thus, the most basic components of our Intercloud approach pre-exist (Kliem, 2015; Aazam et al., 2016). However, the main concepts differ in some important details, or address non-IoT cloud domains with deviating architectural requirements like virtualization on IaaS layer.

(Kliem, 2015) presented a unified bottom-up concept for a federated IoT device cloud. The concept requires a centralized Root Domain Operator, which is responsible for providing global device knowledge and managing user accounts of other principal entities. Interoperability is achieved by developing stakeholders using the same platform-dependent API. We consider the bottom-up driven design unrealistic, as cloud platforms on the market would need to be completely rebuild. In addition, resulting from the proposed centralized "Root Domain Operator", all cloud providers would get dependent on this gatekeeper.

The media Intercloud by Aazam and Huh share the same idea, to let application service clients to either use the PaaS cloud internal API or the broker directly to find and use resources in the Intercloud (Aazam et al., 2016).

In contrast to the discussed related work, our concept does not introduce a root operator as super principal to the architecture. Furthermore, our conceptual design addresses the existing cloud solutions (no bottom-up design) and how cloud providers actually cooperate (P2P, trust based).

(Botta et al., 2016) also conducted an analysis of IoT clouds. They consider the mere existence of an application API while we analyzed APIs in detail. Also, the platforms included in their study differ from the ones we analyzed.

(Laukkarinen et al., 2012) presented a survey

on *infrastructure abstractions* in wireless sensor networks, a sub-domain of the IoT. They also stated that a versatile infrastructure abstraction (including query language, data model/format) is required to ease development on top of heterogeneous infrastructure (i.e., devices). Their survey and our analysis of IoT cloud APIs complement each other. The survey includes basic API considerations for data access, too. The results are equivalent to our Pull (historical query), Push (stream/event based Publish-Subscribe) and Control model, but were not investigated in depth.

They focus on data interoperability; i.e., data format, ontology/homogenized model, meta data and others, which we only addressed superficially. They concluded the open research as a result of lacking abstractions standardization. We share this assessment, also demanding a unified Intercloud data model.

7 CONCLUSION AND FUTURE WORK

We believe the Intercloud Broker is a viable approach to achieve interoperability between IoT cloud platforms. IoT cloud providers strive to not only compete in price, but try to differentiate in further features[19]. Thus, establishing standards for API and device data model will fail. By analyzing a representative subset of IoT cloud platforms we demonstrated that existing platforms share major commonalities that allow interconnecting them by the proposed Intercloud approach. The communication APIs share sufficient similarities allowing mapping between these APIs. Nonetheless, the mapping will not be trivial due to larger differences with respect to device model and federation support. Since the device model is strongly related to the communication APIs, further investigation is required to allow a mapping.

The evaluation of our prototype implementation shows that the IB can be implemented with minimal overhead on average hardware. This mainly addresses the expected high throughput while introducing insignificant additional delay for push notifications which is a main requirement for real-time notifications in large IoT scenarios.

Our future work will focus on sophisticated concepts and implementations for (1) the Discovery Service to work with a device index, and (2) integration of Query-Merger and CEP into the Distribution Controller. The goal for the next prototype is to optimize communication between clouds, while keeping the necessary hardware footprint as low as possible.

Our main research goal is to find the optimal trade off in broker design between communication redundancy and performance overhead for avoiding this redundancy. Thus, we plan to compare the results presented with the performance of the sophisticated architecture. We also intent to run performance tests for larger Intercloud scenarios with multiple clouds.

[19] http://www.computerworld.com/article/2508726/cloud-computing/cloud-interoperability–problems-and-best-practices.html

REFERENCES

Aazam, M., Huh, E. N., and St-Hilaire, M. (2016). Towards Media Inter-cloud Standardization Evaluating Impact of Cloud Storage Heterogeneity. *Journal of Grid Comp.*, pages 1–19.

Botta, A., De Donato, W., Persico, V., and Pescapé, A. (2016). Integration of Cloud computing and Internet of Things: A survey. *Future Generation Computer Systems*, 56:684–700.

Cugola, G. and Margara, A. (2012). Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15.

Grubitzsch, P. (2015). Intercloud communication for value-added smart home and smart grid services. In *On the Move to Meaningful Internet Systems: OTM 2015 Workshops*, pages 10–19. Springer.

Happ, D., Karowski, N., Menzel, T., Handziski, V., and Wolisz, A. (2017). Meeting iot platform requirements with open pub/sub solutions. *Annals of Telecommunications*, 72(1):41–52.

Kliem, A. (2015). *Cooperative Device Cloud - Provisioning Embedded Devices in Ubiquitous Environments*. Dissertation, Technische Universität Berlin.

Laukkarinen, T., Suhonen, J., and Hännikäinen, M. (2012). A survey of wireless sensor network abstraction for application development. *International Journal of Distributed Sensor Networks*, 2012.

Liu, F., Tong, J., Mao, J., Bohn, R., Messina, J., Badger, L., and Leaf, D. (2011). Nist cloud computing reference architecture. *NIST special publication*, 500(2011):292.

Mühl, G., Fiege, L., and Pietzuch, P. (2006). *Distributed event-based systems*. Springer Science & Business Media.

Toosi, A. N., Calheiros, R. N., and Buyya, R. (2014). Interconnected Cloud Computing Environments. *ACM Computing Surveys*, 47(1):1–47.

All URLs were last successfully accessed on 11/02/2016.