A Microservice based Architecture for a Presence Service in the Cloud

Suryaveer Singh Chauhan

A Thesis

in

The Department

of

Computer Science & Software Engineering

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Computer Science at

Concordia University

Montréal, Québec, Canada

_____ 2017

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: Suryaveer Singh Chauhan

Entitled: "A Microservice based Architecture for a Presence Service in the Cloud"

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

____ Chair
Dr. R. Jayakumar

____ Examiner
Dr. J. Rilling

____ Examiner
Dr. E. Shihab

____ Supervisor
Dr. R. Glitho

Approved by ____
Chair of Department or Graduate Program Director

____
Dean of Faculty

Date ____________________ 2017

ABSTRACT

A Microservice based Architecture for a Presence Service in the Cloud

Suryaveer Singh Chauhan

Presence service enables sharing of, and a subscription to the end users presence (online or offline) status. Primarily used for instant messaging applications, the presence service now finds its way into innovative solutions for domains such as wireless sensor networks and Internet of Things. The growth in users of instant messaging applications is ever increasing since the advent of social media networks. Presence service needs to be highly scalable to handle growing load of the users. Moreover, the user activity is inherently dynamic in nature which requires the presence service to be highly elastic to utilise resources efficiently. Traditional presence services are built as monoliths. Monolithic architectures by design are difficult to scale, lacks elasticity and are resource inefficient. Moreover, overprovisioning of resources to handle unanticipated loads further adds to resource inefficiency. Cloud computing and microservices are emerging paradigms that can help tackling the challenges above. Cloud computing with three key facets: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS) enable rapid provisioning and release of resources (e.g. storage, compute, network) on demand. Microservices is an approach of developing applications as a set of smaller, independent, and individually scalable services which communicates with each other using lightweight protocols. The on-demand nature of cloud computing provides a platform to achieve elastic scalability whereas microservices increase the scalability of the architecture. This thesis presents a microservice architecture for a presence service in the cloud. The architecture is based on a state of the art business model. The proposed architecture has three main components: A stateless front-end, a repository and a cache. The front end is built as a set of microservices exposed as SaaS. The front end, to remain technology agnostic, communicates with the repository using the Representational State Transfer (REST) interface. The cache provides fast data access to the front end. The front end microservices use message queues to communicate with each other. Besides, to check the feasibility of the architecture, a proof of concept prototype is implemented for a Session Initiation Protocol for Instant Messaging and Presence (SIMPLE) based presence service. Performance measurements have been made for the proposed and traditional architectures. Also, a comparative analysis of the results is done. The analysis of the results shows that the proposed architecture provides the desired scalability and elasticity to the presence service. Moreover, the proposed architecture provides lower response time and higher throughput in comparison to the traditional architecture.

Acknowledgments

First and foremost, I would like to offer my sincere gratitude to my thesis supervisor, Dr. Roch H. Glitho, who has supported me patiently and guided me throughout the thesis. I would also like to thank Dr. Sami Yangui and Dr. Jagruti Sahoo for their continuous help, guidance, and advice on different parts of this thesis.

I would also like to thank Dr. J. Rilling and Dr. E. Shihab for serving as members of my thesis committee and providing valuable feedback. I also thank Dr. R Jayakumar for serving as the chair at my thesis defence.

It was a pleasure to work with my colleagues at Telecommunications Service Engineering (TSE) lab. I would like to thank Mohammed Abu-Lebdeh and Abbas Soltanian for their valuable advice.

Finally, I must express my deepest gratitude to my loving wife Pankhuri Gupta for her sacrifices and to my parents, brother, sister in law, sisters and lot of other people whose name I didn't mention for their continuous support and encouragement throughout my years of study.

Table of Contents

List of Figures

List of Tables

Acronyms and Abbreviations

ACID	Atomicity, Consistency, Isolation, and Durability
ANSI	American National Standards Institute
API	Application Programming Interface
ASN	Abstract Syntax Notation
BER	Basic Encoding Rules
CIPID	Contact Information for the Presence Information Data
CPU	Central Processing Unit
CRUD	Create Read Update Delete
CSV	Comma Separated Variable
DIT	Directory Information Tree
EE	Enterprise Edition
EPA	Event Publication Agent
FE	Front End
FEaaS	Frontend-as-a-Service
GB	Giga Byte
GPS	Global Positioning System
GSNP	Generic Subscribe Notify and Publish
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure-as-a-Service
IBM	International Business Machine
IDEAL	Isolated state, Distribution, Elasticity, Automated management and Loose coupling
IETF	Internet Engineering Task Force
IMS	IP Multimedia Subsystem
IoT	Internet of Things

IRC	Internet Relay Chat
JSON	JavaScript Object Notation
LAMP	Linux, Apache, MySQL, PHP/Python/Perl
LDAP	Lightweight Directory Access Protocol
LRU	Least Recently used
NGN	Next Generation Network
NIST	National Institute of Standards & Technology
OAS	OPTIONS Aggregation Server
OMA	Open Mobile Alliance
PA	Presence Agent
PaaS	Platform-as-a-Service
PC	Personal Computer
PIDF	Presence Information Data Format
PSAP	Public-Saftey Answering Point
PSaaS	Presence Server-as-a-Service
PSS	Presence Service Substrate
PUA	Presence User Agent
PVS	Presence Virtualisation Service
RAM	Random-Access Memory
RDBMS	Relational Database Management System
RaaS	Repository-as-a-Service
REST	Representational State Transfer
RFC	Request For Comment
RLS	Resource Location Server
RMI	Remote Method Invocation
ROA	Resource Oriented Architecture
RPID	Rich Presence Information Data
RTP	Real-Time Transport Protocol

RTSP	Real Time Streaming Protocol
SaaS	Software-as-a-Service
SASL	Simple Authentication and Security Layer
SCTP	Stream Control Transmission Protocol
SDP	Session Description Protocol
SIMPLE	SIP for Instant Messaging and Presence Leveraging Extensions
SIP	Session Initiation Protocol
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TSDB	Time Series Database
UAC	User Agent Client
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
VIM	Virtual Infrastructure Manager
VM	Virtual Machine
VPS	Virtual Presence Service
WSN	Wireless Sensor Network
XCAP	XML Configuration Access Protocol
XDMS	XML Document Management Server
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol
XSLT	Extensible Stylesheet Language Transformation

Chapter 1

Introduction

This chapter first provides an overview of the key concepts related to the research domain. Then the motivation and problem statement are discussed. A summary of contributions made in this thesis is also presented. The last section gives the outline of the thesis organisation.

1.1 Definitions

1.1.1 Presence Service

Presence service is a service that allows users to share information such as their availability, willingness to communicate, device preference and more [1]. This information collectively is known as presence information. Presence service allows the participating entities to make informed decisions based on the available presence information. Presence service is popularly used in instant messaging applications such as Skype[1] and Google Hangouts[2]. Gaming, conferencing, wireless sensor networks (WSN), and Internet of Things (IoT) are among other application domains that utilise presence service. Essentially, any application which requires interaction between entities can use presence service.

[1] "Skype" [Online]. Available: https://www.skype.com/en/ [Accessed: 01-Aug-2017]
[2] "Google hangouts." [Online]. Available: https://hangouts.google.com [Accessed: 01-Aug-2017]

1.1.2 Cloud Computing

Cloud computing refers the delivery of computing resources such as storage, applications, hardware, networking and so on over the internet [2]. The resources are available on-demand in a pay-as-you-go manner [3]. Cloud computing defines three service models: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS). IaaS provides the network, storage and compute resources through virtualized hardware. PaaS manages the underlying resources on IaaS layer in a transparent manner. Additionally, it provides the platform to develop and execute applications. The applications deployed on PaaS are offered as SaaS to various users.

1.1.3 Microservices

A monolithic application bundle all its functionality into a single executable unit. Scaling such application cause wastage of computing resources as it requires scaling application as whole rather than a part of it. Moreover, the monolithic application is difficult to manage. A change in one part of application requires the whole application to be rebuilt and deployed. Microservices is an architectural style to build an application as a set of small, independently deployable services [4]. Microservices allows scaling only the required part of the application which in this case is an individual service. Also, maintaining and managing the smaller services becomes easier.

1.2 Motivation and Problem Statement

Presence service is an integral part of instant messaging and social networks. However, it can be used to build numerous innovative applications such as Smart Homes, Intelligent Transport Network, applications for WSN or IoT and more. With the proliferation of social networks and smart devices, increasingly more users are now using presence service and their inherent dynamic online activity behaviour demands a highly scalable and elastic presence service. Traditional presence service architectures are monolithic, meaning, all application logic and data is bundled together in a single unit. This approach has its advantage as it enables fast data access. However, the architecture does not scale well. Scaling the monolithic presence service means scaling the application in its entirety, no matter which component required scaling. Such scaling causes the wastage of computing resources. Moreover, data in each scaled instance may not be in sync with each other which makes scaling down difficult causing resource wastage. Additionally, to handle the unanticipated load, the servers need to be overprovisioned which remains underutilized if the demand does not match the anticipation. Above challenges show that the monolithic architecture lacks elastic scalability and can be highly resource inefficient.

A presence service that can gracefully handle high load and dynamic user behaviour need a sound architecture. This thesis aims at proposing a highly elastic and scalable architecture for presence service which also utilizes resources efficiently. However, it is also appropriate to highlight that elastic scalability and high resource efficiency is not desired in every application area. Applications dealing with light load and less fluctuation might as well able to provide acceptable service and resource efficiency using monolithic architecture. The architecture is based on cloud computing and microservices.

1.3 Thesis Contribution

This thesis proposes a highly elastic and scalable architecture for presence service in the cloud.

The main contributions of this thesis are as follows:

- A set of requirements for a presence service in the cloud.

- Review of state of the art relevant to our work.
- A set of architectural principles.
- A business model identifying the actors involved.
- Identify the protocols used by various actors.
- The overall system architecture for the presence service in the cloud.
- Implementation architecture, a proof of concept prototype, performance evaluation and analysis of the results.

1.4 Thesis Organization

The rest of the thesis is organised as follows:

Chapter 2 discusses the key concept related to the research domain in more detail.

Chapter 3 defines the requirements for a presence service. The state of the art is also evaluated against those requirements.

Chapter 4 presents the business model and architectural principles. Additionally, it presents the proposed architecture and communication interfaces.

Chapter 5 describes the implementation architecture and technologies used for proof-of-concept prototype. It also presents the performance measurement results and their analysis.

Chapter 6 concludes the thesis by summarizing the contributions and possible future research directions.

Chapter 2

Background on Presence, Cloud Computing, and Microservice architecture

This chapter discusses topics related to the research domain. The first section introduces the presence service and various protocols followed by a discussion on cloud computing. The last section explains microservices architecture.

2.1 Presence

This section first provides a brief introduction of presence followed by a sub-section with details of *a model for presence.* Later sub-sections discuss protocol requirements for presence and describe few presence protocols. The last sub-section provides examples of few presence applications.

2.1.1 A Brief Introduction to Presence

Presence is a technology that enables the user to share their presence information and subscribe to, and get notified about the presence information of other users. At the basic level, the presence information can provide the availability status of the user which can be online, offline or busy. The presence information can provide various other details about the user such as the status message, type of device, location, mood, calendar details, and more. Internet Engineering Task Force (IETF) defines a model of presence.

2.1.2 A model for Presence

IETF provides a model for presence in the Request for Comment (RFC) 2778 [1]. The model as shown in figure 2.1 defines the presence service with two distinct set of clients known as Presentities and Watchers. The presentities provide presence information to the presence service while the watchers receive this information. The function of presence service is to store and distribute the presence information received from the presentities.

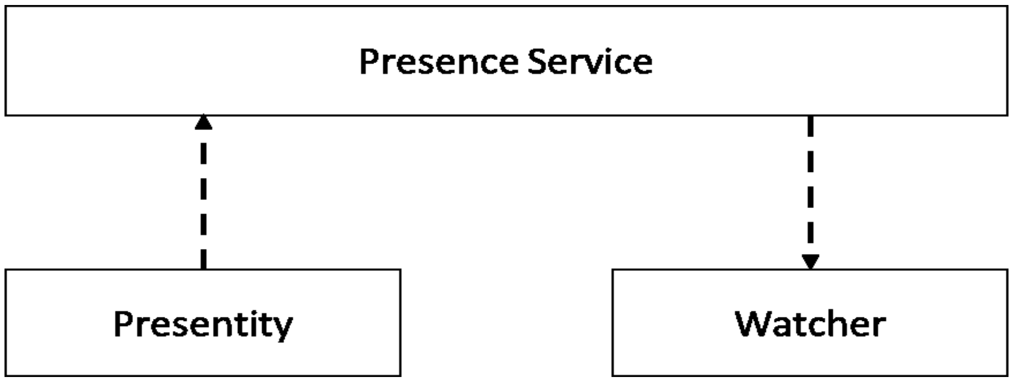


Figure 2.1 A model for Presence

The watchers are categorised into two kind — Fetchers and Subscribers. A Fetcher only requests a presentities current presence information as and when required whereas; a Subscriber requests notifications for future updates of the presentities presence information. To receive the future updates of the presentities changed presence information a subscriber creates a subscription to the presence service. There is a special kind of fetcher, known as a Poller. A Poller fetches the presence information at regular intervals. Figure 2.2 shows different kinds of Watchers.

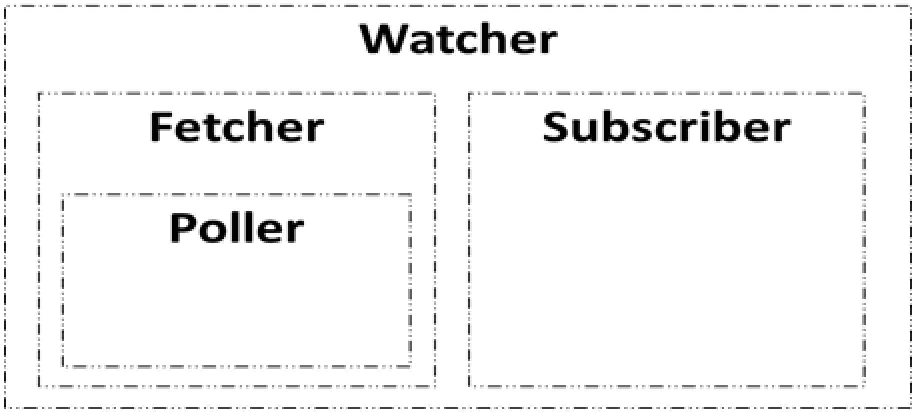


Figure 2.2 Types of Watchers

The presence model includes few other elements known as Principal and Agent. The principal is a real world entity that can be a person, group or software. A principal interacts with the system via one of its respective agents. An agent acts as a coupling between the principal and the core entity of the system. As shown in figure 2.3, the Presence User Agent (PUA) couples a presentity and principal while a Watcher User Agent couples a principal with the watcher.

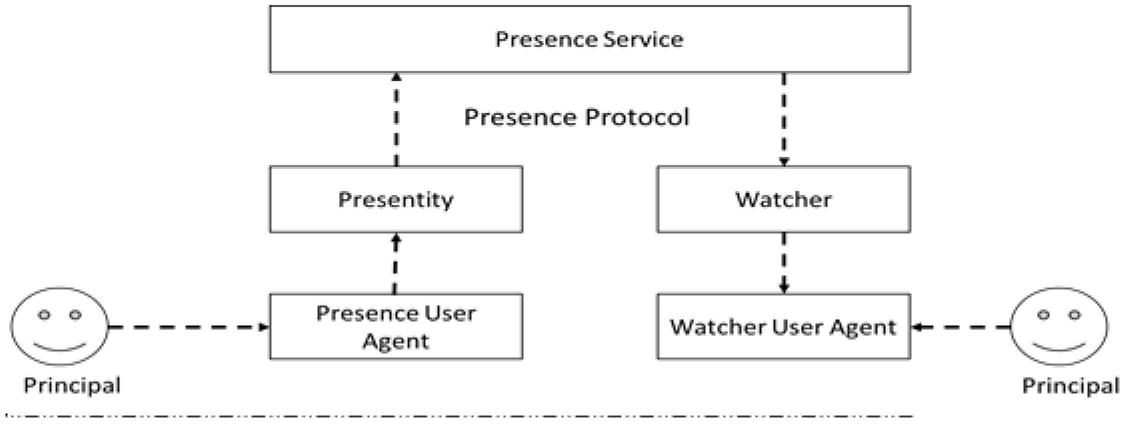


Figure 2.3 A presence system

The presence service does the distribution of the presence information by sending the notifications to the watchers. The distribution is done based on the access rules set by the presentities. Access rules enable privacy features for presentities presence information. A presentity may wish to hide some information from certain watchers; it can do so by defining the access rules. A watcher is interested in the presence information of the presentities likewise, a presentity may also be interested in knowing who are the subscribers to its presence information. The presence service maintains this information known as watcher information and may present this information to the presentity.

2.1.3 Presence Protocol Requirements

There existed several vendors who developed protocols for the presence and instant messaging. The problem was that those protocols were non-standard and non-interoperable. Therefore, IETF formed a working group to standardise the protocol so that the several independently developed applications can interoperate across the Internet. The Instant Messaging / Presence Protocol Requirements document — RFC 2779 defines the minimal set of requirements that a protocol must meet. The requirements define an administrator that has authority over the principal's device, network or firewall. Furthermore, the RFC defines several shared security and non-security requirements for presence and instant messages. However, only requirements related to presence are discussed here. The shared non-security requirements cover issues related to the namespace and administration, scalability, network topology, access control, and message encryption and authentication. Additional requirements for presence information deals with Common Presence Format, performance, presence lookup and notification, and presence caching and replication. The security related requirements deal with Subscriptions and Notifications.

2.1.4 Presence Protocols

The two open standard protocols discussed in this section are SIMPLE and XMPP. As the thesis is focused on SIMPLE based presence, this protocol is discussed in greater detail.

2.1.4.1 SIP for Instant Messaging and Presence Leveraging Extensions (SIMPLE)

SIMPLE [5] is a Session Initiation Protocol (SIP) [6] based application layer protocol suite for instant messaging and presence. This subsection first provides a brief introduction to SIP protocol and later discuss the SIMPLE protocol and its functioning.

SIP is an application layer text-based signaling protocol used for creating, modifying and terminating multimedia sessions that work independently of underlying transport layer protocols. SIP itself does not provide any services, rather, it works in conjunction with several other protocols that carry real-time multimedia session data such as voice, video, and text messages. The example of such protocols are but not limited to Session Description Protocol (SDP), Real-time Transport Protocol (RTP), and Real-time Streaming Protocol (RTSP). SIP also defines extensions to provide an extensible framework by which SIP nodes can request notifications from remote nodes [7] and for publishing event state [8]. The SIP-specific event notification [7] formed the basis to use SIP as a presence protocol. Several specifications were produced to utilise SIP for Presence and Instant Messaging. Collectively, these specifications are known as SIP for Instant Messaging and Presence Leveraging Extensions (SIMPLE) defined in RFC6914 [5].

SIMPLE is a relatively complex set of specifications expanding to several documents. Therefore, for comprehensibility the specification documents can be categorised into following:

- **Core protocol machinery**: defines the actual SIP extensions for publications, subscriptions and notifications.
- **Presence documents**: defines the XML documents that represent the presence information.
- **Privacy and Policy**: provides a way to express a preference about what presence information is visible to different users.

- **Provisioning**: describes how users can manage their privacy preferences, buddy list, and other information.
- **Optimisations**: defines the improvement in the core protocol machinery, defined to improve the performance of SIMPLE.

The specification is covered in detail below:

- **Core protocol machinery**

This set of specifications deals with the extensions to the SIP protocol describes the process of publications, subscriptions, and notifications. A Presence Event Package for Session Initiation Protocol [9] proposes usage of SIP as a presence protocol. This event package introduces a new logical entity called *presence agent (PA)*. A PA is capable of accepting subscriptions, storing subscription state and generating notifications. As a PA is a logical entity, it is co-located with another entity in the presence system. SIMPLE works in a publish-subscribe system, where a user publish its presence information and other users subscribe to the published information which is then notified by the system of the updates in the information. The specification for subscription notifications and publications are defined in the SIP events framework documents [7] and [8] respectively. The following three SIP methods proposed in the events framework are used for exchanging the presence information:

1. **SUBSCRIBE:** This method is used by the user agent requesting the presence information.
2. **NOTIFY:** This method is used by a user agent to notify other user agents about change in the presence information.
3. **PUBLISH:** This method is used by a user agent to publish the event state (presence information).

Given below are few definitions that are key to this presence protocol:

Subscriber: A subscriber is a user agent which generates a SUBSCRIBE requests and receives the NOTIFY request from the notifiers. Henceforth,

Subscription: A subscription is an application state created when a SUBSCRIBE request is generated. A subscription exists in both a subscriber and a notifier.

Notifier: A notifier is a user agent that generates the NOTIFY requests with a purpose to notify a subscriber of the event state.

Notification: Notification is the act of sending the NOTIFY by the notifier.

Presence Agent (PA): A presence agent is capable of accepting subscriptions, storing subscription state and generating notifications.

Presence User Agent (PUA): A Presence User Agent manipulates presence information for a presentity. There could be multiple PUAs per presentity. PUAs push data to the server and do not receive NOTIFY or send SUBSCRIBE request.

Presence Server: A presence server is a physical entity that can act as a presence agent for SUBSCRIBE requests.

Event State: State information for a resource. For simplicity, referred as presence information in this document.

Event Publication Agent (EPA): The User Agent Client (UAC) that issues PUBLISH request to publish event state. Henceforth, EPA, publisher and presentity are used interchangeably.

Publication: The act of sending a PUBLISH request.

A presentity generates a PUBLISH request with its event state to publish the presence information. The event state has a defined lifetime. Once published, the presentity can modify, refresh or remove the event state until expired. The subscriber sends a SUBSCRIBE request to the presence server to express its interest in receiving notifications for event state changes of some presentity. Like the publication, each subscription has a defined lifetime. A subscription can be refreshed or removed until expired. The presence server sends a 200 response for both the PUBLISH and SUBSCRIBE request. Whenever the presence server receives a SUBSCRIBE request, it generates a NOTIFY request with the event state of the requested presentity. If there's no event state available for the presentity, a NOTIFY request is sent with an empty body. The subscriber responds to the NOTIFY request with a 200 response message. A simple message flow is shown in figure 2.4. The presence information is represented as XML documents which are carried by SIP. There are various types of presence documents which are explained in the next section.

```
                              Subscriber          Notifier          Presentity
Request state subscription        | 1. SUBSCRIBE       |                   |
                                  |------------------->|                   |
Acknowledge subscription          | 2. 200             |                   |
                                  |<-------------------|                   |
Return blank state information    | 3. NOTIFY          |                   |
                                  |<-------------------|                   |
Acknowledge notification          | 4. 200             |                   |
                                  |<-------------------|                   |
                                  |                    | 5. PUBLISH        |  Publish state information.
                                  |                    |<------------------|
                                  |                    | 6. 200            |  Acknowledge publication.
                                  |                    |------------------>|
Return new state information      | 7. NOTIFY          |                   |
                                  |<-------------------|                   |
Acknowledge notification          | 8. 200             |                   |
                                  |<-------------------|                   |
```

Figure 2.4 Presence message flow

The subscribers usually subscribe to multiple publishers by sending a SUBSCRIBE request as well as subscription refreshes for each publisher. Doing so will generate a substantial message traffic which can be problematic for limited bandwidth environments. Therefore, the SIMPLE specification provides another way for subscribers to request a subscription using resource lists [10]. The resource list is a list of all the publishers that a subscriber is interested in. With the resource lists in place, the subscriber can now subscribe to the list and will be notified whenever status of any publisher changes in the list. A Resource List Server (RLS) acts as a notifier for the list and acts on behalf of the subscribers. The subscribers send their subscription request to the RLS which in turn will send SUBSCRIBE request to the presence server for each item in the list and will later notify the subscribers for status changes of the publishers in the list. Subscribers can create the subscriptions to the list by sending the resource list in the request body as described in the RFC5367 [11].

- **Presence documents**

The presence information is represented in the form of XML documents. Presence Information Data Format (PIDF) [12] defines the base presence format to represent basic presence information and provides an extensibility framework to enrich the presence information. Rich Presence Information Data Format (RPID) [13] is one such extension that enriches the presence information with data such as *user mood, status-icon, place-type* and more. Other extensions to PIDF are Contact Information for the Presence Information Data Format (CIPID) [14] and Timed Presence Extensions to PIDF [15] which further enrich presence information. PIDF provides a way to represent presence information. However, it does not specify how to model the presence information and how to map real-world communications system into presence document. *A Data Model for Presence* [16] addresses the issues above in the PIDF. The purpose of the data model is to take into account how the real-world situations and the devices, and services might affect the user's willingness or ability to communicate. As an example, a user may not be willing to communicate if he is in a meeting or on another call or may be the service or device he is using is not suitable for the calls. Such information can be part of the presence information. Example, a service might report the user is busy on another call with status "on call" and also can report the device as "mobile phone". Therefore, this data model as shown in figure 2.5 defines three components— device, service and person which are described below:

- **Person-** A person is the end user characterised by states relevant to the presence system, such as "busy" or "happy".
- **Service-** A communication service that can be used to interact with the user, such as instant messaging or telephony.
- **Device-** A physical communication device with which a person interacts to use a service to make communication, such as phone, or PC.

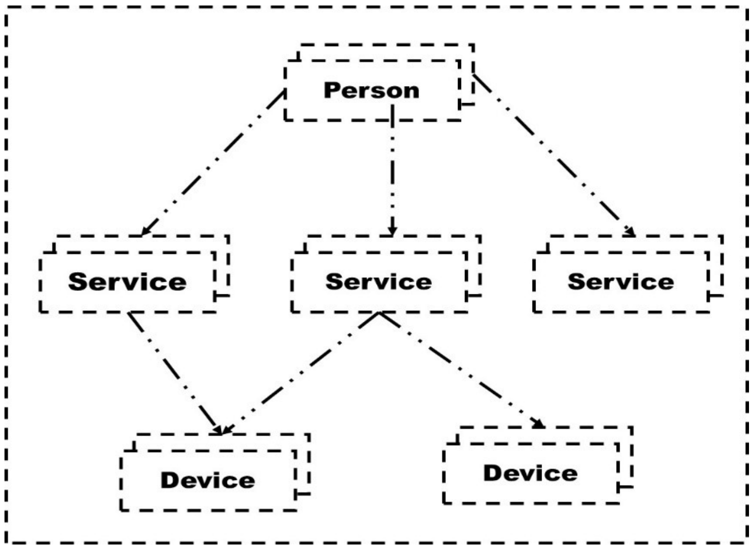


Figure 2.5 A Data model for presence

- **Privacy and policy**

The presence information is sensitive information. The presentity can decide to what information to share to certain subscribers. The presentity can do so by setting the presence authorization rules [17]. The notifiers apply the policy to determine whether the particular subscriber is authorised for certain set of events. The notifiers also do the authentication of the SUBSCRIBE requests. The user might also want to know who all are the subscribers to its presence information and make a decision whom to authorise to view its information. The information about all the subscribers for a user is known as watcher information to which the user can subscribe [18].

- **Provisioning**

  The presence system needs to store and access several pieces of user information which are managed by the end-users. Buddy list, authorization rules, and privacy policies are few example of such information. An XML Configuration Access Protocol (XCAP) server [19] is used to store such documents which can be accessed by the user agent using the XCAP protocol. The SIMPLE specification also provides a way to store the offline presence document which can be used by the presence server when the user has not published its status [20].

- **Optimizations**

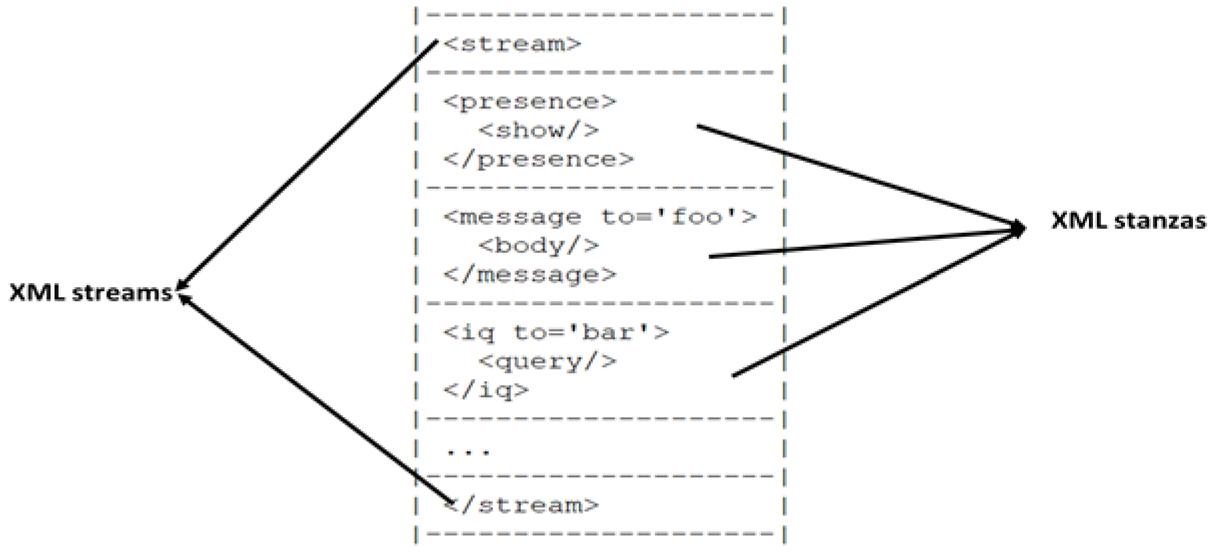
  Sending presence information can be very expensive. The notifications contain the full presence information and often being sent when the subscriber does not desire them. To address above problems, the SIMPLE specification provides a way for subscribers to describe filters for the presence information according to their needs [21][22]. Also, the PIDF format inherently needs to carry all the presence information a user has, PIDF extension for Partial Presence [23] provides a specification to send notifications using partial presence information. [24] and [25] provides a specification to receive partial notifications and to publish presence status using partial PIDF document respectively.

2.1.4.2 eXtensible Messaging and Presence Protocol (XMPP)

XMPP [26] is a protocol for streaming XML elements between two network endpoints in near real time. However, it is mainly used for instant messaging and presence. The protocol initially was developed by Jabber open-source community, but later development was done to make it suitable as an IETF instant messaging and presence protocol. The core features of XMPP are defined in the document XMPP-CORE [26] while the document XMPP-IM [27] defines the extensions required to provide instant messaging and presence functionality as defined in RFC2779 [28].

Although XMPP is not devoted to any network architecture, it is usually deployed via a client-server architecture. The client-server and server-server communication happens over TCP. Apart from client and server, there is another entity involved in the XMPP architecture, Gateway. A Gateway is a server-side service used to enable communication with non-XMPP clients. Gateways translate XMPP to the protocol used by non-XMPP clients and vice-versa. Example gateways are to email, Internet Relay Chat (IRC), SIMPLE, and legacy IM services such as ICQ, and Yahoo! Messenger.

Two fundamental concepts of XMPP are XML streams and XML stanzas. XML stream is a container for the exchange of XML elements while the XML stanza is a discrete semantic unit of structured information that is sent from one entity to another over an XML stream. Three kinds of XML stanzas are defined: <message/>, <presence/>, and <iq/>. To start the communication, the client opens a stream with the server by sending an opening XML <stream> tag. To close the stream a closing XML </stream> tag is sent. During the lifetime of the stream, the entity can send an unbounded number of XML elements over the stream. Figure 2.6 gives a representation of XML stanza and XML streams.



```
|--------------------|
|<stream>            |
|--------------------|
| <presence>         |
|   <show/>          |
| </presence>        |
|--------------------|
| <message to='foo'> |
|   <body/>          |
| </message>         |
|--------------------|
| <iq to='bar'>      |
|   <query/>         |
| </iq>              |
|--------------------|
| ...                |
|--------------------|
| </stream>          |
|--------------------|
```

Figure 2.6 Representation of XML stanza and XML streams

XMPP also defines several security considerations such as high security for mutual authentication and integrity checking, certificate validation, firewalls. XMPP also mandates the use of Transport Layer Security (TLS) for encryption of streams and Simple Authentication and Security Layer (SASL) protocol for authentication of XML streams for both the client-server and server-server communication.

2.1.5 Presence Applications

Presence can be used to develop various innovative applications apart from its common application i.e. Instant Messaging. Given below are few examples of presence-based applications:

1. **Presence Aware Location-Based Service:** A solution is proposed by Singh et al. [29] to combine the user presence information and location to build an integrated communication environment, which can be used to create various domain specific services such as vehicle status monitoring, and more. The researchers also constructed a prototype which offers to

improve the communication capabilities of the mobile workforce working in the field as well as improve the manageability of the vehicles used by the workforce. With this solution in place, the location information collected from the on-vehicle device is integrated with the user's presence information gathered from the user device such as cell-phone. The integrated information allows the supervisor to know the status of all the crew without communication. The supervisor now knows whether the crew is working on customer site or moving towards different customer site. The supervisor may also defer the decision to call the crew is the status of the crew is "driving".

2. **Wireless Healthcare Application:** Barachi et al. [30] propose a health monitoring application using the mix of Wireless Sensor Networks (WSN), IMS and presence server. In the proposed solution, the wireless sensors monitor the user's health and publish events to the presence server when it senses a critical situation. The notifications are sent to the health monitoring application. The application then creates a conference between the Public-Saftey Answering Point (PSAP) and the patient. Also, the application queries the presence server for the nearest ambulance and joins it in the conference. The application keeps track of the ambulance and keeps it updated with the patient information until the ambulance reaches the patient.

2.2 Cloud Computing

This section first provides the definition of cloud computing and its characteristics. Later sub-sections discuss various cloud service models and deployment models.

2.2.1 Definition of Cloud Computing

Cloud computing is an emerging model to provision the computational resources over the network. Several attempts are made to define cloud computing. Vaquero *et.al* [31] says "*Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs.*" Armbrust et al. define cloud computing as *"data center hardware and software that provide services."* A detailed definition is provided by The National Institute of Standards and Technology (NIST). According to NIST [32] "*Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*"

2.2.2 Characteristics of Cloud Computing

According to NIST, cloud model is composed of five essential characteristics [32]:

- **On-demand self-service:** A consumer can provision computing services automatically as needed without any human intervention.

- **Broad network access:** Cloud computing resources are available over the network, supporting heterogeneous thin or thick client platforms such as mobile, tablet, and workstations.
- **Resource pooling:** Service multiple customers using a multi-tenant model providing a sense of location independence to each customer. However, the customer might be served from a same physical resource which is logically separated securely.
- **Rapid elasticity:** Resources are provisioned and released on-demand and/or automatically making sure that the application has exactly the required capacity at any point in time.
- **Measured service:** Resource usage is monitored, measured, and billed transparently based on usage creating a pay-per-use model.

2.2.3 Cloud Service Models

Cloud computing has three service models [32], namely: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS). Each service model provides a different level of abstraction. Figure 2.7 shows the layered view of cloud service models.

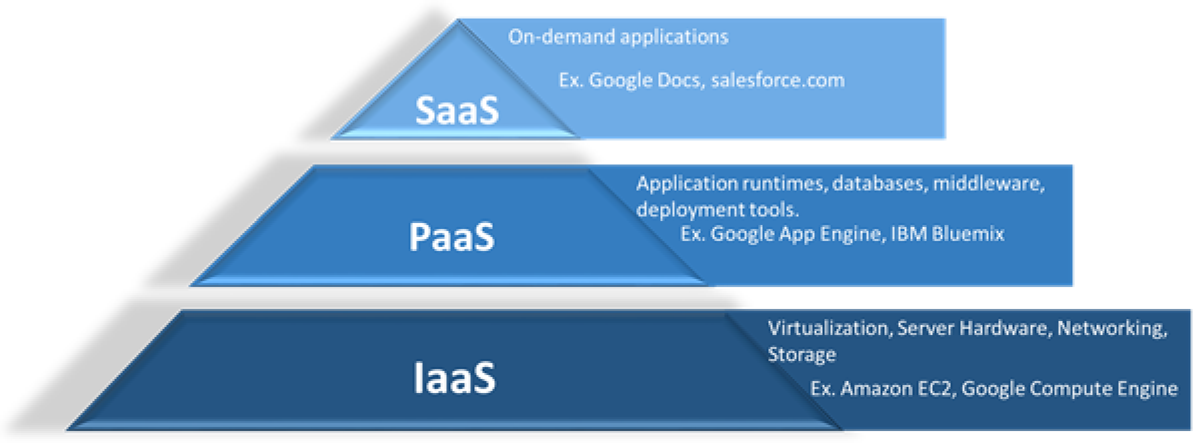

SaaS
On-demand applications
Ex. Google Docs, salesforce.com
PaaS
Application runtimes, databases, middleware, deployment tools.
Ex. Google App Engine, IBM Bluemix
IaaS
Virtualization, Server Hardware, Networking, Storage
Ex. Amazon EC2, Google Compute Engine

Figure 2.7 Cloud Service Models

2.2.3.1 Infrastructure-as-a-Service

Infrastructure-as-a-Service is the lowest layer of the cloud computing systems. The IaaS enables on-demand provisioning of computational resources (e.g., Compute, storage, and network) in the form of Virtual Machines (VMs) [33]. NIST defines IaaS as "*The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications*" [32]. IaaS layer manages the underlying hardware and provides a unified view of the heterogeneous resources to the consumer. Virtualization plays a major role in the IaaS due to which IaaS can serve in a multi-tenant environment and can service multiple consumers from a same physical resource, thus achieving cost and resource efficiency. Consumers do not have any control over the underlying infrastructure. However, it does control the VM and everything inside it.

The cloud infrastructure needs to manage the physical and virtual resources, the lifecycle of the virtual machines, create networks dynamically and so on. The software toolkit that provides this capability is a Virtual Infrastructure Manager (VIM) [33]. The VIM provides management APIs as well as a front end interface through which the consumers can request the virtual machines and other resources. Virtual machines are offered in various configurations of CPU and memory, commonly referred as flavour. Openstack and VMware vSphere are examples of VIM.

IaaS provides several advantages to the consumer. It provides cost savings as the consumer does not need to invest in the physical resource. Moreover, the consumer now pays only for the resources used. Provisioning virtual resources instead of physical resources also improve the time to market as resource provisioning is almost instantaneous instead of days. Also, the businesses can now completely focus on core competency instead of maintaining the private physical infrastructure. Above all, IaaS offer virtually limitless on-demand scalability. There are various public IaaS providers such as Amazon EC2, Google Compute Engine whereas VMWare vSphere and Openstack offers to build private IaaS solutions.

2.2.3.2 Platform-as-a-Service

Platform-as-a-Service provides an abstraction of underlying infrastructure and the software and application resources. NIST defines PaaS as "*The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider.*" PaaS provides a platform to build, deploy, execute and manage the software applications. However, the consumer does not manage the underlying infrastructure such as operating system, network or servers, but has control over the deployed application and its hosting environment [32]. Notable examples of PaaS platforms are IBM Bluemix, Microsoft Azure, and Google App Engine.

2.2.3.3 Software-as-a-Service

The application deployed on the PaaS are offered to the consumers over the network as Software-as-a-Service. SaaS has the highest level of abstraction. SaaS consumers have no control over the underlying infrastructure and the application capabilities [32]. The application can be consumed either by the end-users directly via the clients such as a browser or by third-party applications via APIs. SaaS offers an alternative to the applications installed on the local computers [34]. Examples of SaaS include salesforce.com, Google Docs and more.

2.2.4 Cloud Deployment Models

There are mainly four deployment models for cloud computing as described by NIST. Each deployment model is defined below:

2.2.4.1 Private Cloud

Private cloud is a cloud infrastructure that is not available to general public and is exclusive to an organisation. A private cloud may exist on or off premises and can be owned, managed and operated by the organisation, a third party or any combination of both. The motivation for private cloud is to utilise the in-house resources and security concerns such as data privacy and trust [35]. Companies like Redhat and VMWare are among the private cloud vendors.

2.2.4.2 Public Cloud

The cloud infrastructure available to the general public is referred to as a public cloud. The cloud service provider has the control over the infrastructure. Each public cloud provider can have their set of policies and charging model. Public cloud benefits include savings in the upfront cost of the hardware, on-demand scalability, easy setup to use and more. Pivotal, Amazon EC2 are some examples of the public cloud.

2.2.4.3 Community Cloud

Several organisations which share same concerns can provision a cloud infrastructure exclusively for their use. The cloud infrastructure may be owned or managed by one or more organisations or a third-party.

2.2.4.4 Hybrid Cloud

Hybrid cloud is a combination of two or more distinct cloud infrastructures (public, private or community). Entities, however, remain unique and are bound together by standardised or proprietary technology that enables data and application portability (e.g., cloud bursting for load balancing between clouds) [32].

2.3 Microservice Architecture

This section starts with an introduction to the microservice architecture. Next discussed are microservice characteristics and advantages. In the end, an example of microservice architecture design is provided.

2.3.1 Introduction to Microservice Architecture & Microservices

Microservices is an emerging software architecture style to build distributed systems. Microservices emphasise on building highly scalable and maintainable software. Several definitions of microservices and microservices architecture have been suggested. [36] puts microservice as *"a minimal independent process interacting via messages"* and microservice architecture as *"A microservice architecture is a distributed application where all its modules are microservices."* James Lewis and Martin Fowler [37] define microservice architecture as *"an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.*" Unlike monolithic architectures where all the functionality is interwoven into a single executable unit, microservice architecture's focus is to build the application as a set of fine-grained services with independent functionality. These fine-grained services are called microservices. Thones [38] put microservices as "*a small application that can be deployed independently, scaled independently, and tested independently and that has a single responsibility*".

Microservice architecture is still in its infancy and lacks consensus on what microservice actually are [36]. There is much debate surrounding microservices and Service-Oriented Architecture (SOA). Microservices proponents see microservices as a new architectural style whereas others see it as a form of Service-Oriented Architecture (SOA). [39] considers microservices as an implementation approach to SOA. Gabbrielli et al. [40] see microservices as following the principles of component-based software engineering. Sam Newman considers microservices as one way of doing SOA(right) [39] – *"The microservices approach has emerged from real-world use, taking our better understanding of systems and architecture to do SOA well."*

2.3.2 Tenets of Microservice

Following are the common tenets of a microservices compiled by Olaf Zimmermann [39] from various studies:

- Fine-grained interfaces: Single responsibility units that contain processing logic are exposed via RESTful resources or message queues. These units can be deployed, changed, scaled independently.
- Business-driven development: development of services is business driven and based on domain-driven design.

- Cloud-native application: cloud native application design principles are followed. Some of the various principles are summarized in IDEAL (isolated state, distribution, elasticity, automated management and loose coupling) or the twelve app factors in Heroku's method.
- Multiple computing paradigms: the services are developed in polyglot programming which contains multiple programming languages and storage paradigms.
- Lightweight containers: the services are deployed using lightweight container technologies such as Docker.
- Decentralized continuous delivery: this practice followed during the service development promotes a high degree of automation and autonomy.
- DevOps Lean: automated approaches to configuration, performance and fault management are employed.

2.3.3 Advantages of Microservice architecture

The services in the microservice architecture based application are inherently modular which makes them easier to understand, develop, test, and manage. Monolithic applications, on the other hand, grow huge in size over a period making them difficult to manage and understand. Change in single functionality requires the rebuilding and deployment of the whole application. Moreover, all the available functionality bundled in the application may not be used together yet whole application is scaled even if the only subset of functionality required scaling causing wastage of computational resources. There are several other benefits provided by microservices architecture as described in [41]:

- **Technology Heterogeneity:** As the microservices are independently deployable units it allows to choose different technology stack for various microservices depending on the service needs.

- **Resilience:** Microservices have defined service boundaries which make it easier to identify and isolate the problem in the case of failures allowing to degrade functionality accordingly unlike monolithic where a failure can cause whole application to shutdown.
- **Scaling:** Microservices allows scaling only the required services instead of whole application as in the monoliths.
- **Ease of Deployment:** A change needed in functionality requires modification and deployment of only the concerned service while a monolithic application requires the whole application to be deployed which could be high risk and high impact deployment.
- **Organisational Alignment:** Monolithic application development requires large teams handling huge codebase. Microservices allows to reduce the team size and code base handled by them.
- **Composability:** Independent microservices allows reusability. The functionality exposed by the services can be consumed in a different application.
- **Optimising for Replaceability:** The monolithic applications grows huge and complex over a period making them risky and difficult to replace. However, with fine-grained services, it is easy to replace or remove service.

2.3.4 Microservice architecture design

This section provides an example of how to design the microservice architecture for a software application.

M Villamizar et al. [42] demonstrates how a web application is designed according to monolithic and microservices architecture. A simple web application is intended to generate and query payment plans for loans provided to the customers. The payment generation and query services are referred as S1 and S2 respectively henceforth. The monolithic design of the application is shown in figure 2.8. Unsurprisingly, the monolithic application has a single code base encapsulating the services S1, and S2 and is developed using a single technology stack. The microservices architecture based application shown in figure 2.9, on the other hand, is developed as two different microservices independent from each other. Each microservice can be developed using a different technological stack and can be deployed independently from each other. The microservice architecture introduces a gateway that consumes the microservices on behalf of the client.

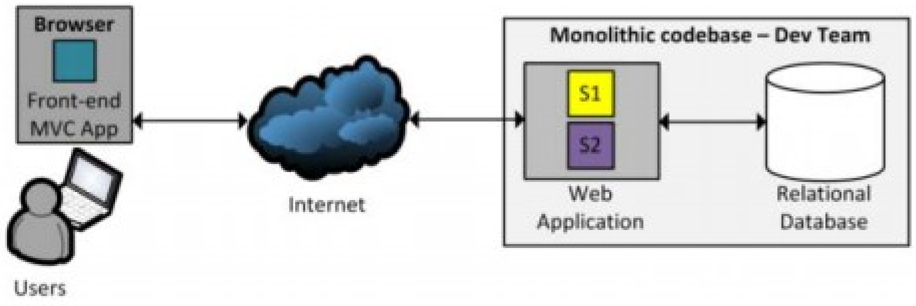


Figure 2.8 Monolithic Architecture [42]

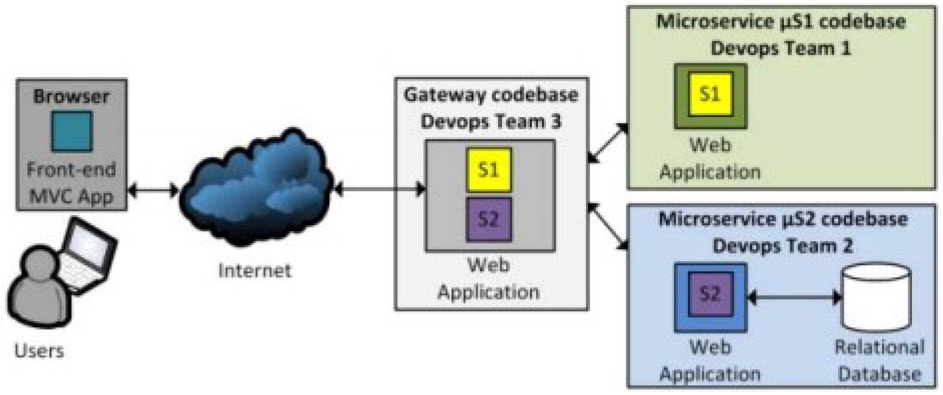


Figure 2.9 Microservice Architecture [42]

From the architecture design, it is evident that the microservices architecture eases the maintenance and enhances the manageability of microservices due to their small codebase. Also, it enables the scaling of only the desired services and also allows to choose a technology stack that is suitable for a particular kind of service. In the presence service, the three described messages i.e. PUBLISH, SUBSCRIBE and NOTIFY perform independent processing. Moreover, the NOTIFY processing is most time and resource consuming which means that when NOTIFY is being processed other requests will not get required resources and processing time. Effectively, NOTIFY processing will directly impact the performance of the other two messages. Therefore, splitting the presence service frontend into multiple microservices is proposed, whereby each message is handled in its own process without interfering in the processing of other messages. Additionally, each individual service can scale independently depending on the load.

2.4 Chapter Summary

This chapter discussed the concepts related to the research domain. First, it discussed the presence technology, its related protocols such as SIMPLE and XMPP, and few applications based on presence technology. Then the Cloud Computing, its characteristics and its various service and deployment models were discussed. The last section discussed the microservice architecture and its benefits.

Chapter 3

Requirements and State of the Art Evaluation

This chapter is divided into four sections. The first section presents few motivating scenarios from which requirements are derived. The second section presents a set of requirements for a presence service in the cloud. Third section reviews and evaluates state of the art based on the requirements and the last section presents the summary of the chapter.

3.1 Scenarios

This section presents two motivating scenarios taken from the literature. These scenarios help derive the requirements for the presence service in the cloud.

3.1.1 Smart Home

A smart home enables the interaction of the homeowner with the various appliances installed at home. Moreover, it also allows to monitor the status and control the functionality of the appliance from any remote location. [43] provides an innovative concept of the ubiquitous home control system. The concept is built upon the integration of the KNX[3] bus and Next Generation Network (NGN) core to provide the smart home automation system. The KNX is an open standard home automation bus, and IMS is the NGN. The IMS has presence server that is used by this concept to

[3] "KNX" [Online]. Available: https://www.knx.org/knx-en/index.php [Accessed: 01-Aug-2017]

provide the remote control ability to the homeowners. The figure 3.1 below shows the system design.

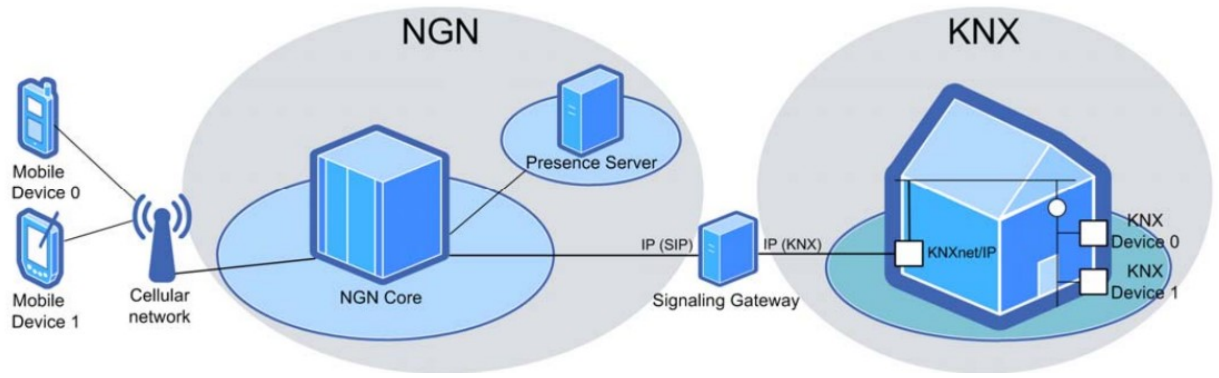


Figure 3.1 Detailed system design of Smart Home Automation [43]

Each home automation appliance in this system is a user and gets a uniquely identifiable SIP address. The homeowner and the appliances are registered to the presence server. In this case, the homeowner acts as the subscriber, and the home appliances are the publishers. Each appliance can generate a status update that is sent to the presence server through a gateway. The presence server then notifies the registered owner with the device status. For example, a light sensor can report a "light on" status or a temperature sensor can report home temperature as 20 degrees. The owner can then send a message to switch off the lights or decrease the temperature as she wishes. Figure 3.2 below shows a sample message interaction. This type of system can suffer from scalability issues as new or old appliances are added or removed. Moreover, a system can also be put in place where the appliances do not publish at night or when the owner is at home. Such a situation will reduce the load on the presence server requiring it to reduce the amount of resources it is consuming elastically.

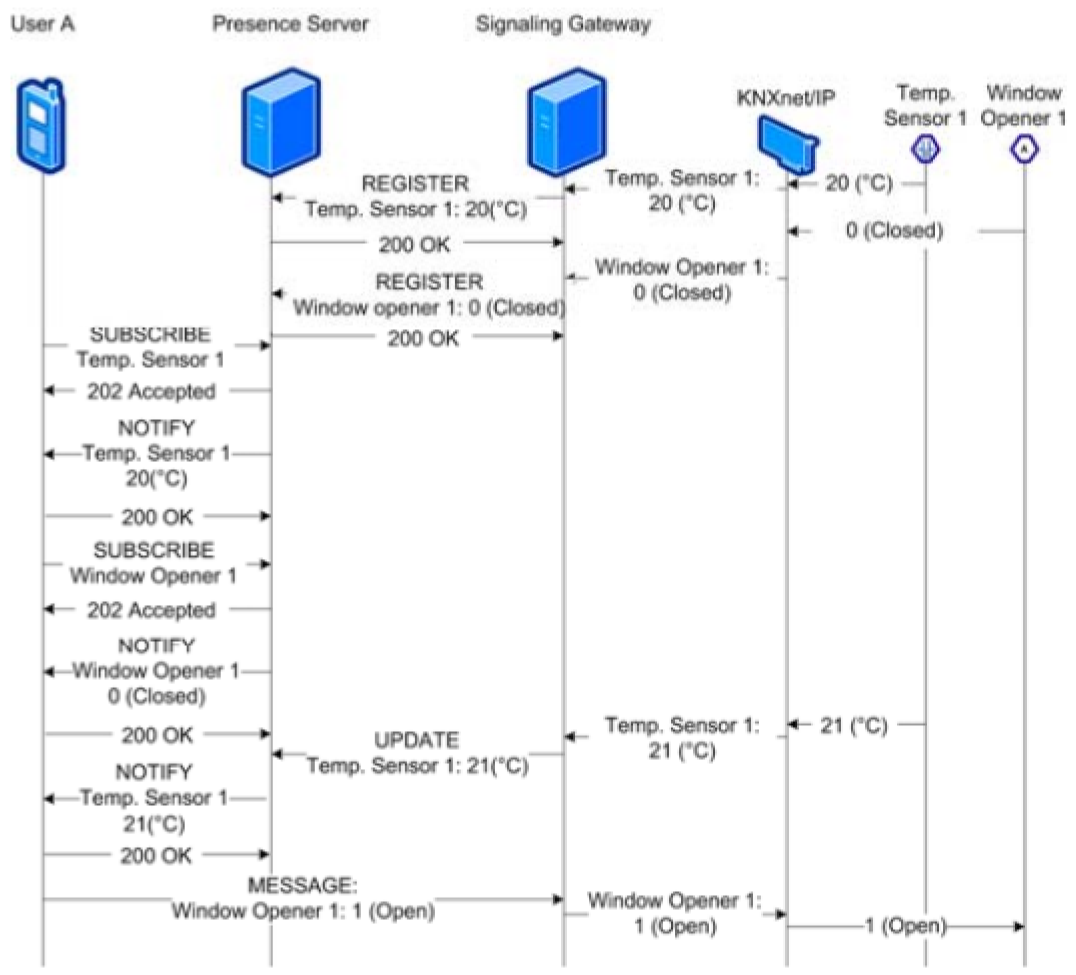

User A
Presence Server
Signaling Gateway
KNXnet/IP
Temp. Sensor 1
Window Opener 1
REGISTER Temp. Sensor 1: 20(°C)
200 OK
REGISTER Window opener 1: 0 (Closed)
200 OK
Temp. Sensor 1: 20 (°C)
20 (°C)
0 (Closed)
Window Opener 1: 0 (Closed)
SUBSCRIBE Temp. Sensor 1
202 Accepted
NOTIFY Temp. Sensor 1 20(°C)
200 OK
SUBSCRIBE Window Opener 1
202 Accepted
NOTIFY Window Opener 1 0 (Closed)
200 OK
UPDATE Temp. Sensor 1: 21(°C)
Temp. Sensor 1: 21 (°C)
21 (°C)
NOTIFY Temp. Sensor 1 21(°C)
200 OK
MESSAGE: Window Opener 1: 1 (Open)
Window Opener 1: 1 (Open)
1 (Open)

Figure 3.2 Sample message flow [43].

3.1.2 Location Aware service for elderly

P. A. Moreno et. al. [44] proposes a solution to monitor the person with mild cognitive impairment (PwMCI). A PwMCI is at risk of spatial disorientation, meaning the person may get lost. Such risk compels the person to remain indoors. The solution proposes to employ a context-aware service to track the user movement. The proposed solution uses the presence server to achieve the required functionality. With such service in place, the person has more freedom and the caregivers can monitor the location and manage the situation. The service transparently monitors person's location based on the Cell Id and GPS. The service automatically subscribes to the patient's presence information. On the NOTIFY message received from the patient, the service applies context rules to identify if the person is lost, moving, accompanied or require emergency attention. Based on the situation the service alerts the registered contacts, or a call is established with the nearest contact. The high-level architecture of the system is shown in figure 3.3. The emergency situation requires quick dissemination of information. Therefore, services dealing with emergency situation requires low latency and high consistency.

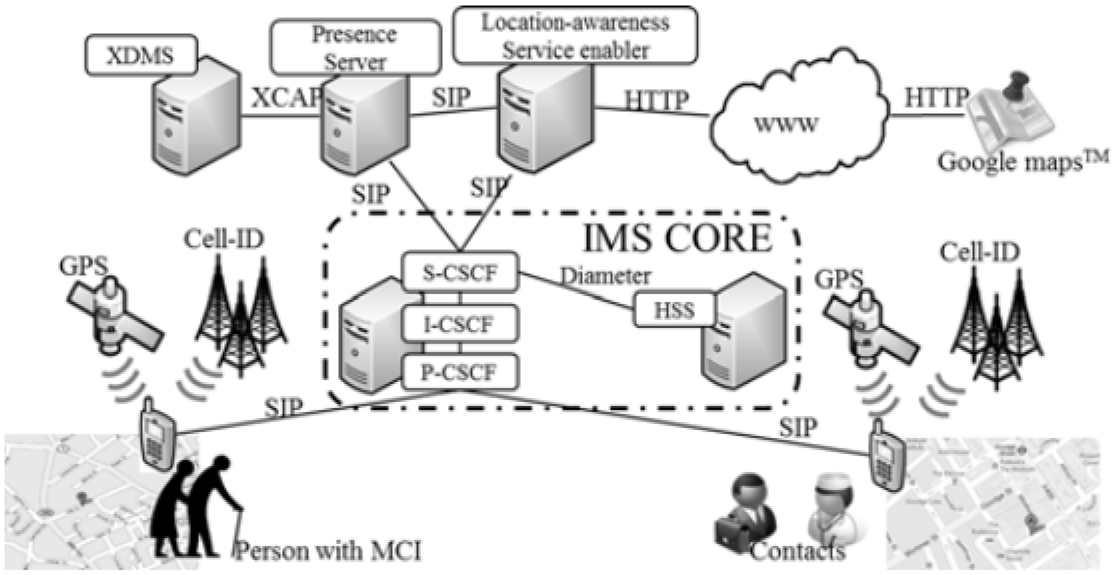


Figure 3.3 Location aware service enabler architecture [44]

3.2 Requirements

This section provides the four requirements for a presence service in the cloud.

1. **Elastic Scalability:** The presence service should be able to scale in response to the fluctuating load by provisioning and de-provisioning resources in an autonomic manner.

Several presence service applications experience high user load and dynamic user activity causing fluctuation in resource demand. To achieve resource efficiency, the presence service should be able to scale elastically so as to match the available resources to the current demand. Online social networks are an example of such applications where the user load is ever increasing, and the user activity is highly dynamic.

2. **Latency:** Presence service can be used in wide variety of applications where each application has different latency requirements. For example, an emergency service as described in [45] may have stringent latency requirements while instant messaging applications might tolerate high latency. Latency can also be affected by the increasing user load such as in online social networking applications. Therefore, the architecture should be able to satisfy various latency requirements.
3. **Consistency:** As with the latency, different applications may have varying consistency requirements. An emergency service as described in [45] may have strict consistency requirements while applications such as instant messaging may tolerate low consistency. Indeed, a different presence status of a person as seen by their contacts may not be an as critical issue as the different status of a fire sensor visible to other monitoring sensors.
4. **Technology Independent Repository:** The storage of presence information should not be tied to a database technology. There are several database technologies available, each having some advantages over others. The architecture must be flexible enough to absorb the changes and the evolutions if, in the future, the repository is changed to a different technology.

The requirements described above need not fit every presence application. The actual requirements depends upon the specific application areas. For instance the Smart home scenario presented in motivation above might require elastic scalability as there might be hundred thousands of sensors publishing their presence information during day time and possibly they may not do so during night time. Latency is also not critical in this scenario. However, in the location aware service for elderly motivational scenario, latency and consistency are critical requirement whereas elastic scalability might not be required as the traffic generation in this case is not highly fluctuation prone.

3.3 State of the Art Evaluation

The design of traditional presence service architecture received significant attention from the research community while there is only handful of work done for presence service on a cloud platform. This section is divided into two sub-sections. The first sub-section reviews existing work done on traditional presence service architectures while the second section reviews cloud-based presence service.

3.3.1 Traditional Presence Service

This section reviews the work done on improving the scalability and performance of the traditional presence service.

Chen et al. [46] propose a weakly consistent scheme that focuses on reducing the network traffic by reducing the number of NOTIFY messages generated from the presence server. The presence server generates a NOTIFY message for each publication. The weakly consistent scheme called as *delayed update* avoids sending NOTIFY message for each presence update thereby reducing the notification traffic. Under this scheme, whenever the presence server receives an update, a timer is started for some period T instead of sending the notification. This period is referred to as *delayed* *threshold*. Any presence update received within this *delayed threshold* period replaces the previously received update. A notification is only sent when the timer expires. Once the timer expires, the last received update is used to notify the watchers. Therefore, saving the notifications for updates received within the time T. Figure 3.4 shows the whole process with a timing diagram. The watcher accesses the presence information at t0, t3, and t8. The presence information is updated at t1, t4, t5, t6, and t9. At t2 and t7, the presence server notifies the watcher of modified presence information. While this certainly reduces the network traffic, the scalability aspect of the presence server is not discussed.

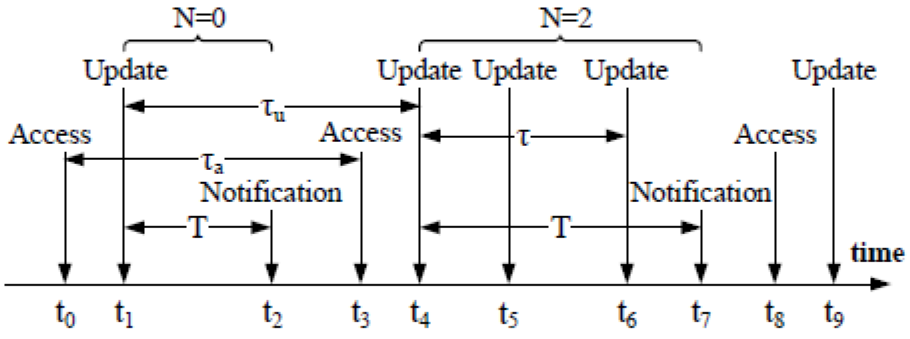


Figure 3.4 Timing diagram for delayed update [46]

Lee et al. [47] propose a polling architecture for the presence server. The presence server provides the presence updates to the watchers in near real-time as soon as it has new presence information consuming much network resources. The proposed architecture aims to tackle the problem above. The architecture proposes to use SIP OPTIONS message instead of SUBSCRIBE messages to get the presence information. The SIP OPTIONS message is used to query server capabilities, and it does not create a subscription on the server. With this approach, whenever the presence information is required, the subscriber will query it using the OPTIONS message instead of the server generating NOTIFY at every presence update. Furthermore, the Presentities do not use SIP PUBLISH method to advertise the presence information. Instead, the Presentities sends XCAP PUT message to store the presence document in the XDMS server.

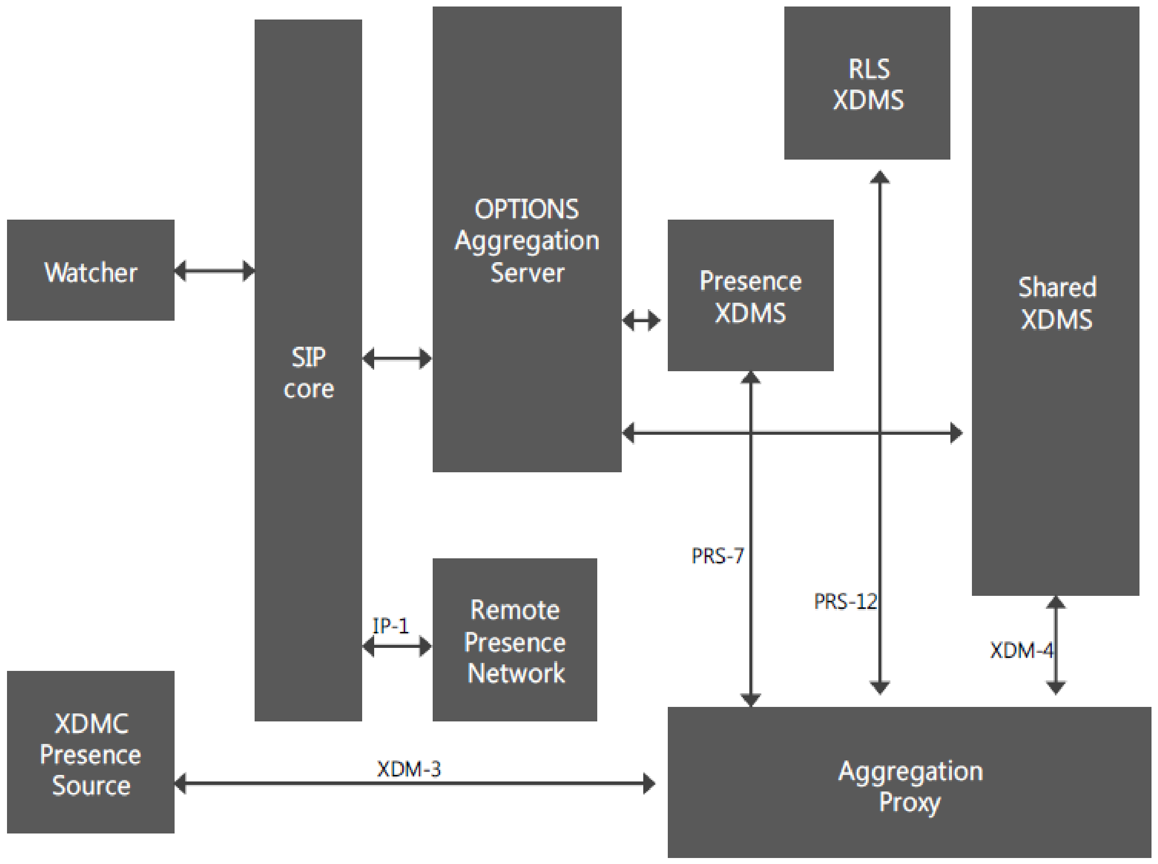


Figure 3.5 SIP OPTIONS based architecture for OMA presence [47].

The architecture further modifies the OMA presence architecture by introducing a OPTIONS aggregation server (OAS). The purpose of OPTIONS aggregation server is to aggregate the presence information of various resources into one response message. The use of OAS is to prevent subscribers from requesting presence information for each resource individually. The use of Figure 3.5 depicts the modified architecture. The OPTIONS aggregation server can receive a single OPTIONS message containing the list of resources whose presence information is desired by the subscriber instead of *n* number of individual OPTIONS messages. OAS extract the individual resources from the message and requests presence information for each resource from the XDMS using XCAP GET message. Once the OAS receives presence information of all of the

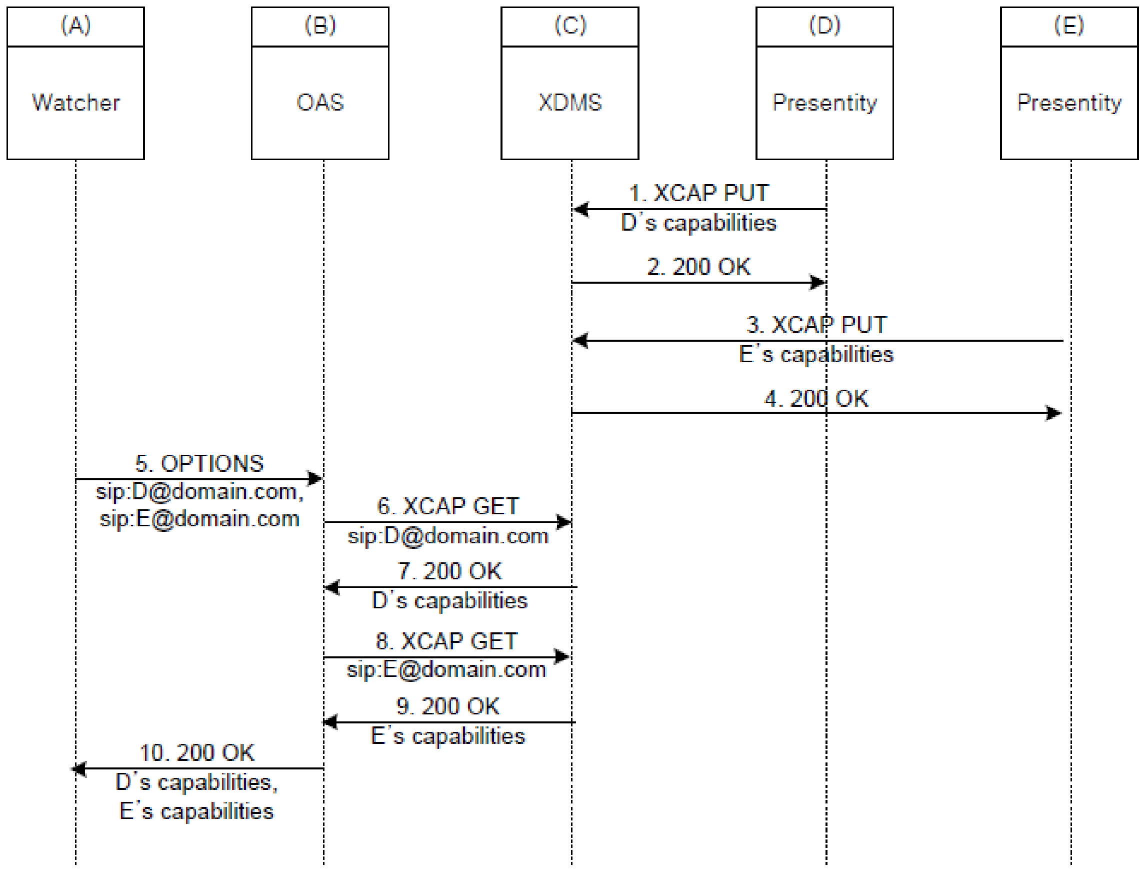


Figure 3.6 Call flow of SIP OPTIONS architecture [47].

n resources, it then aggregates information into a single response instead of *n* individual responses. Figure 3.6 depicts the message flow. This approach reduces the load on presence server while making it only suitable for cases where real-time updates are not required.

Peternal et al. [48] proposes an architecture to enable effective communication among the coworkers in enterprise environments. The proposed architecture aggregates presence information from various sources and presents it to the consumers. The proposed architecture is depicted in figure 3.7.

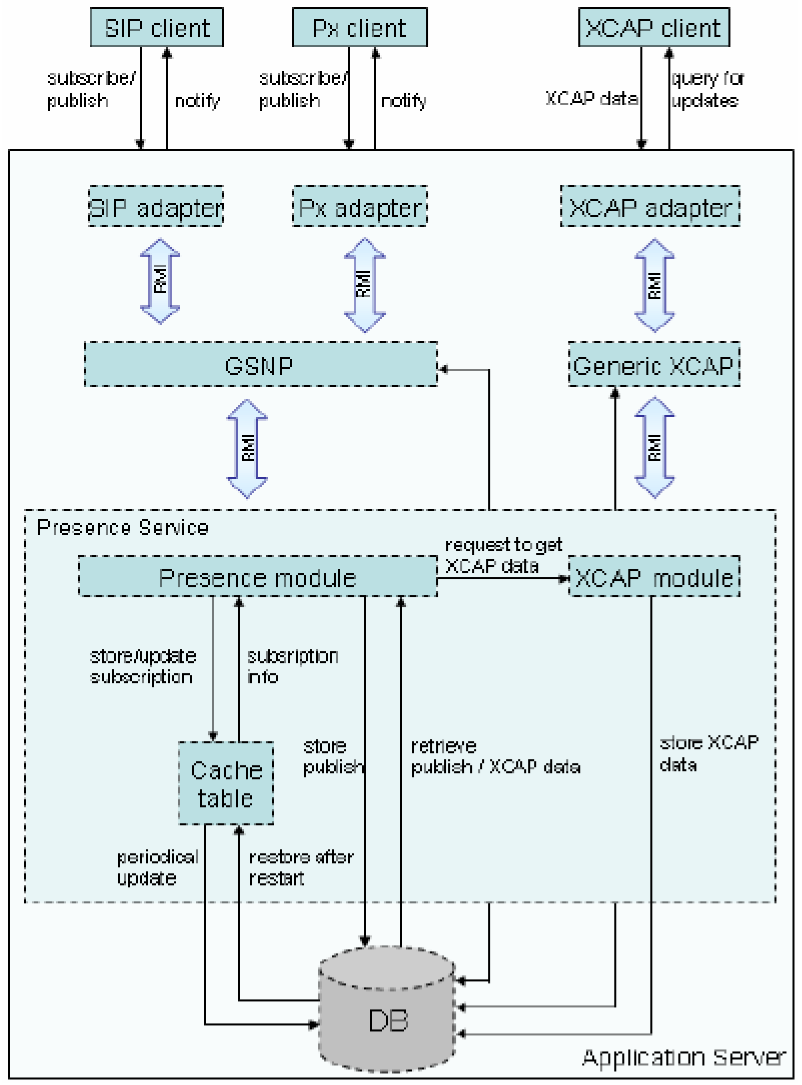


Figure 3.7 Architecture for Presence Service [48]

The proposed architecture interfaces with different presence sources using adapters. The Generic Subscribe Notify and Publish (GSNP) module acts as a bridge between the adapters and the presence server and uses RMI as the communication interface. The server maintains the user database in sync with the organization-wide user database. Also, the server uses rule-based logic to infer presence state of the user if incomplete presence information is available. These rules are stored in the XDMS server along with the user preferences. As evident from the figure 3.3 the architecture combines the presence server, database, and various other custom logic in a single server. The servers are deployed in a cluster. From the cluster of these servers, one can become a load balancer, another server as master database while rest can act as a presence server. While the architecture scales by adding new nodes but it lacks efficiency as various functions are bundled together which are scaled together.

3.3.2 Cloud-based Presence Service

This section provides a review of the literature on the cloud-based presence service.

Quan et al.[49] built a testbed for a presence service in the cloud. The approach followed to build the testbed is to deploy the traditional presence service onto the virtual machines which are running on the Infrastructure-as-a-Service layer. Figure 3.8 depicts the architecture for the test bed.

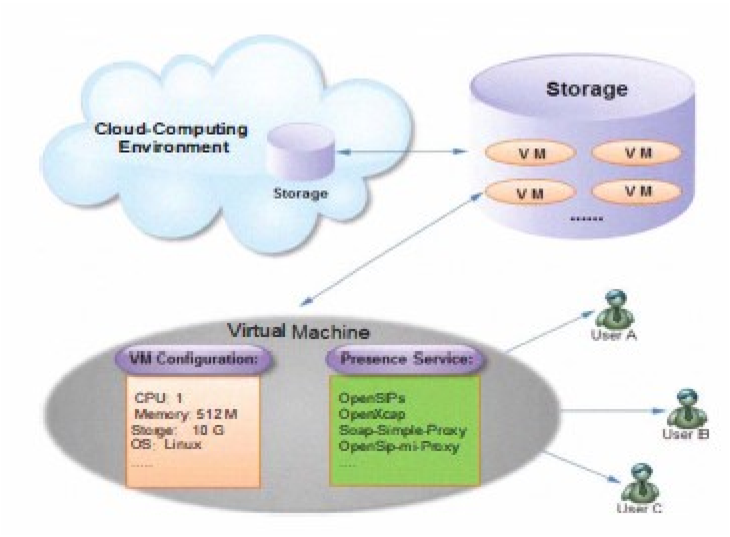


Figure 3.8 Architecture of the testbed [49]

The testbed implementation uses Eucalyptus[4] platform for the infrastructure running two virtual machines. One of the VM houses the presence server, database and other entities. Few tests were performed to check the functionality and performance of the presence server and other components in the setup. The author provides the performance metrics of individual entities like the presence server, database and XCAP server while the benefits achieved regarding scalability and elasticity are not discussed. While this approach provides the auto-scaling capabilities yet it is not the efficient way to scale as the application logic and the data layer are being scaled regardless of what needs to be scaled.

Belqasmi et al. [50] propose an architecture for virtualized presence service for future internet. The architecture aim at the rapid development of virtual presence services and scalability through reusable presence service substrates. The architecture consists of three layers and two planes. The three layers are: Virtual Presence Service (VPS), Presence Virtualisation Service (PVS), and Presence Service Substrate (PSS) and the two planes are presence service plane, and virtualization control and management plane. The VPS layer provides presence service to the consumers through some presence protocol (such as SIMPLE, and XMPP), The PSS layer provides the presence protocol substrate and a PSS provisioning platform to the VPS layer. The PVS layer act as a mediator between the VPS and PSS and also provides a mapping between VPS presence protocol and presence protocol substrate. The control plane on each layer assists in creating the presence service and managing the required resources. For example, the VPS layer can request for a presence service instantiation with a message describing the data model and a number of requests/minute; the PVS layer may translate this message into terms of CPU and memory and

[4] "Eucalyptus" [Online]. Available: http://www.eucalyptus.com/ [Accessed: 01-Aug-2017]

pass it onto the PSS layer that instantiates the presence service. Figure 3.9 depicts the proposed architecture.

The architecture enables the use of single PSS by multiple VPS and of multiple PSS by single VPS. Scalability in the architecture is ensured using the PSS, but the level of granularity of substrate is not discussed. It is, therefore, hard to assess whether the architecture could scale in a fine-grained manner.

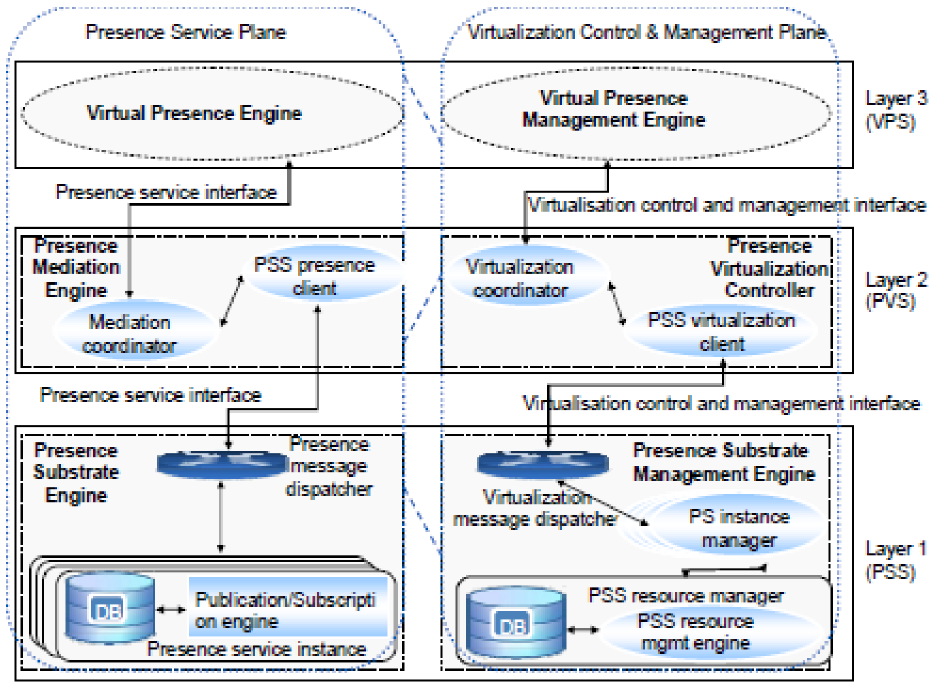

Presence Service Plane
Virtualization Control & Management Plane
Virtual Presence Engine
Virtual Presence Management Engine
Layer 3 (VPS)
Presence service interface
Virtualisation control and management interface
Presence Mediation Engine
PSS presence client
Mediation coordinator
Virtualization coordinator
Presence Virtualization Controller
PSS virtualization client
Layer 2 (PVS)
Presence service interface
Virtualisation control and management interface
Presence Substrate Engine
Presence message dispatcher
Presence Substrate Management Engine
Virtualization message dispatcher
PS instance manager
Layer 1 (PSS)
DB
Publication/Subscription engine
Presence service instance
PSS resource manager
PSS resource mgmt engine
DB

Figure 3.9 Virtualized Presence Service Architecture

Arup Acharya et al. [51] propose a virtual aggregation layer over the presence server to aggregate presence information from several domains and provide query-based presence information to users. The virtualization layer exists over the existing presence server and allows presence-enabled applications to consume and compose presence information from various sources. The presence-

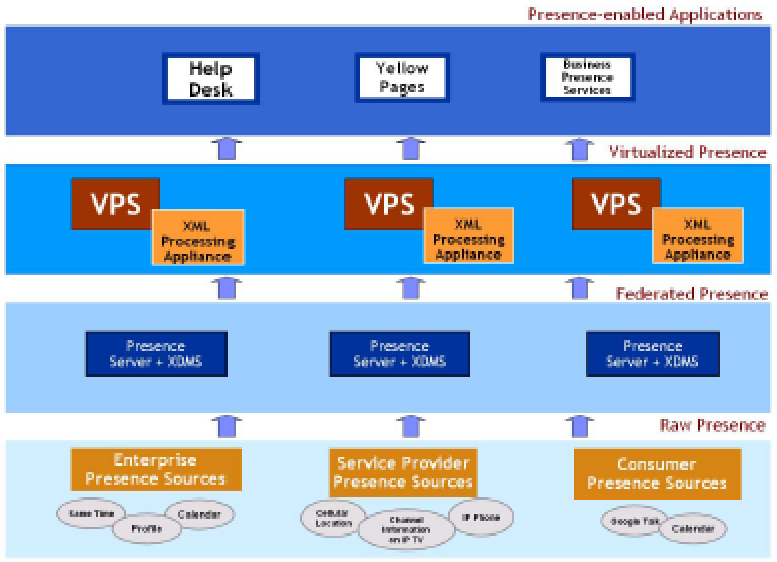
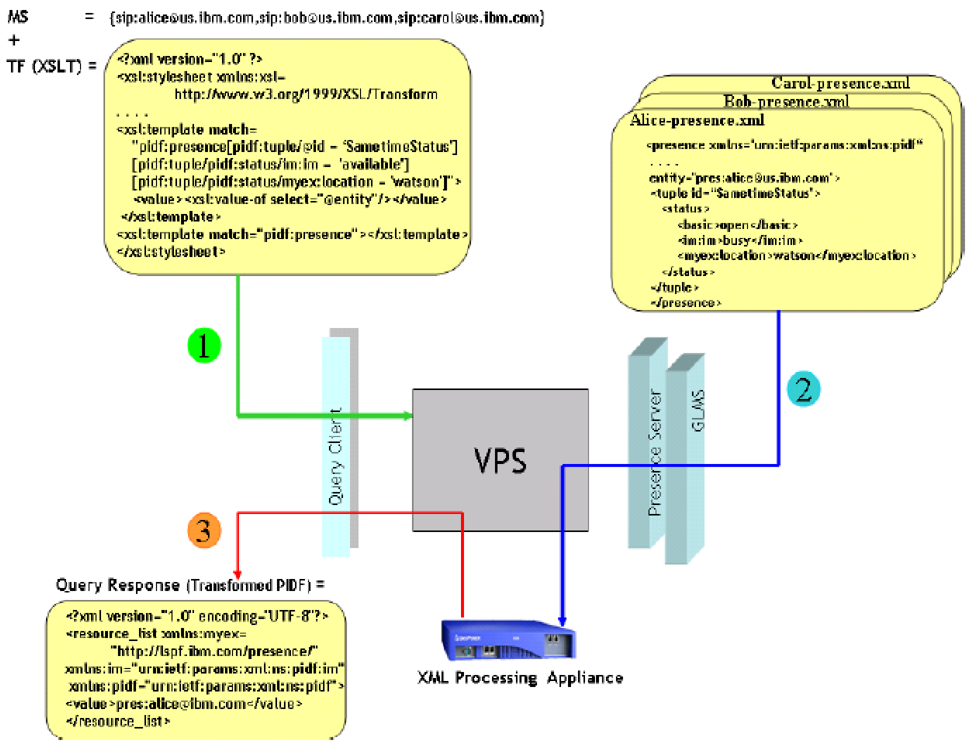


Figure 3.10 Presence Virtualization Architecture [51]

enabled applications can request presence information in the form of queries (such as all the users in Montreal having status as "online"). The queries are sent as subscription requests in the payload of SIP SUBSCRIBE message. The queries are sent to the virtualization layer known as Virtual Presence Servers (VPS) which is responsible for extracting and processing these queries. The VPS, based on the query, also sends a subscription to various presence servers in the related domains and apply the transformation function on the various presence information received to build the aggregate response. As the presence information is XML based, the presence queries and the transform function are XSLT based. For the purpose of scalability, the XML processing is offloaded to XML processing appliances. Figure 3.11 depicts a sample query request and response. The author claims to achieve scalability in the architecture by reducing the number of messages exchanged and offloading the XML processing to separate appliance. However, there's no discussion to improve the scalability and elasticity of the underlying presence servers.



```
MS        = {sip:alice@us.ibm.com,sip:bob@us.ibm.com,sip:carol@us.ibm.com}
+
TF (XSLT) = <?xml version="1.0" ?>
            <xsl:stylesheet xmlns:xsl=
                        http://www.w3.org/1999/XSL/Transform
            . . . .
            <xsl:template match=
               "pidf:presence[pidf:tuple/@id = 'SametimeStatus']
               [pidf:tuple/pidf:status/im:im = 'available']
               [pidf:tuple/pidf:status/myex:location = 'watson']">
               <value><xsl:value-of select="@entity"/></value>
             </xsl:template>
            <xsl:template match="pidf:presence"></xsl:template>
            </xsl:stylesheet>
```

```
Carol-presence.xml
Bob-presence.xml
Alice-presence.xml
   <presence xmlns="urn:ietf:params:xml:ns:pidf"
   . . . .
   entity="pres:alice@us.ibm.com">
   <tuple id="SametimeStatus">
      <status>
         <basic>open</basic>
         <im:im>busy</im:im>
         <myex:location>watson</myex:location>
      </status>
   </tuple>
   </presence>
```

```
Query Response (Transformed PIDF) =
<?xml version="1.0" encoding="UTF-8"?>
<resource_list xmlns:myex=
        "http://lspf.ibm.com/presence/"
xmlns:im="urn:ietf:params:xml:ns:pidf:im"
 xmlns:pidf="urn:ietf:params:xml:ns:pidf">
<value>pres:alice@ibm.com</value>
</resource_list>
```

Figure 3.11 XSLT-Based Query Specification for Presence Virtualization [51]

The table below summarises the evaluation of the related works. The column value "Not discussed" means the related work does not deal with the requirement whereas "No" means the related work deals with the requirement but does not meet it. "Yes" means the requirement is fully satisfied by the related work. "Partially" indicates that the related work deals with and satisfies only parts of the requirement.

Table 3.1 Evaluation Summary

Requirements / Related Work	Elastic Scalability	Latency	Consistency	Technology Independent Repository
[46]	NO	Not Discussed	NO	Not Discussed
[47]	NO	Not Discussed	Not Discussed	Not Discussed
[48]	NO	Not Discussed	Not Discussed	Not Discussed
[49]	Partially	Not Discussed	Not Discussed	Not Discussed
[50]	Partially	Not Discussed	Not Discussed	Not Discussed
[51]	Partially	Not Discussed	Not Discussed	Not Discussed

3.4 Chapter Summary

The chapter first discusses few scenarios for the presence service application to derive the set of requirements. Then it discusses various requirements for a presence service in the cloud and later evaluates the various state of the art based on the requirements. Finally, we come to the conclusion that none of the existing works satisfies all of our requirements.

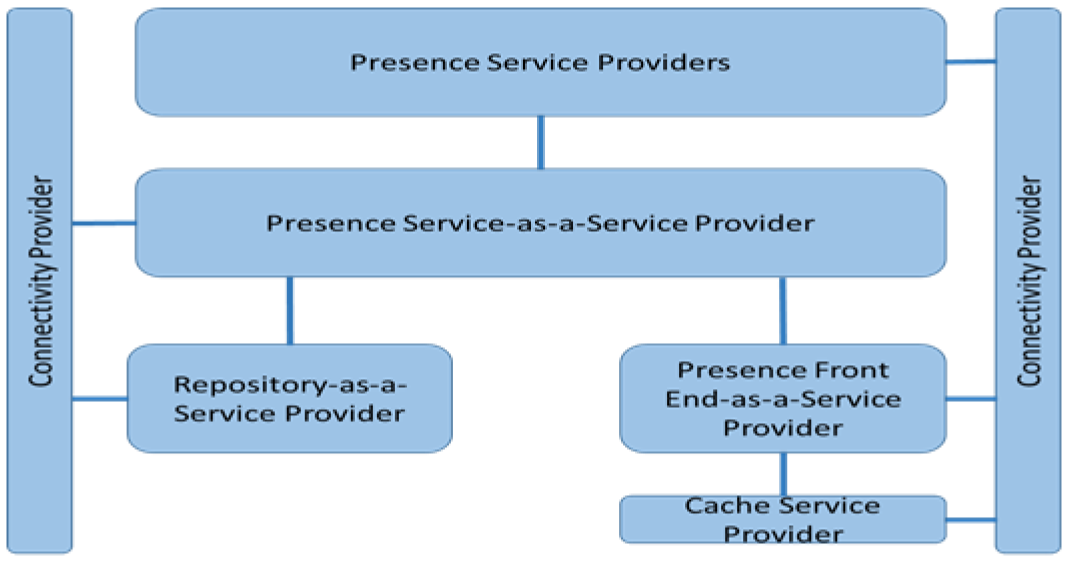
Chapter 4

Proposed Architecture

The previous chapter set the requirements for a presence service in the cloud. This chapter presents an architecture based on those requirements. The first section discusses the proposed business model. The second section discusses the various communication interfaces, provides the overall architecture and the REST resources. The third section provides two illustrative scenarios that show the interaction between the entities. The last section concludes the chapter with a summary.

4.1 Business Model

The proposed architecture relies on an open business model which includes following actors: Presence-Service-as-a-Service (PSaaS) provider, Frontend-as-a-Service (FEaaS) provider, Repository-as-a-Service (RaaS) provider, Cache service provider, Presence service provider and connectivity provider. Figure 4.1 represents the business model.



Presence Service Providers
Connectivity Provider
Presence Service-as-a-Service Provider
Connectivity Provider
Repository-as-a-Service Provider
Presence Front End-as-a-Service Provider
Cache Service Provider

Figure 4.1 Business Model

Each business actor is described below:

- **Frontend-as-a-Service Provider (FEaaS):** The FEaaS provider exposes the presence functionality to be consumed by other providers.
- **Repository-as-a-Service Provider (RaaS):** The RaaS provider is responsible for providing the storage function to the presence service. The RaaS exposes lightweight APIs to communicate with the repository.
- **Cache Service Provider:** Cache is used to provide quick data access to the frontends. Usually, the cache-as-a-service is provided by the PaaS marketplace (e.g. Redis from CloudFoundry[5]). However, the FEaaS can choose to deploy its caching solution in the PaaS.
- **Presence-Service-as-a-Service Provider (PSaaS):** The role of PSaaS provider is to act as a broker between the parties interested in consuming the presence service and the FEaaS, and RaaS providers. The PSaaS offers its service by composing the presence service using the

[5] "Redis labs pivotal marketplace" [Online]. Available: console.run.pivotal.io/marketplace/services/99b6e48d-f319-4a72-b854-a6e43eea9c3c [Accessed: 01-Aug-2017]

FEaaS and RaaS. Hence, PSaaS also provides the abstract view of the presence service to the consumers.

- **Presence Service Provider:** The presence service providers are the actors those provide the presence service to the end users.
- **Connectivity Provider:** The connectivity providers provide the basic connectivity to the various business actors.

4.2 Overall Architecture

This section discusses the architectural principles, the proposed architecture and various interfaces between the actors.

4.2.1 Architectural Principles

The first principle of the architecture is to split the monolithic presence server into multiple independent entities. The splitting also includes separation of the application logic (henceforth referred as application frontend or FE) from the embedded data component (the repository). The splitting leads to better scalability as each entity can scale independently as the need be. Also, splitting leads to the efficient resource usage as the resources are now consumed only by the scaled entity. The splitting induces communication overhead and latency. Therefore, to minimise the effect of splitting, a cache can be introduced between entities where required, such as between the FE and repository. A cache can provide fast data access thereby reducing the communication overhead and latency. Additionally, the cache also helps to conserve the network bandwidth.

The second principle is that the interface between the FE and the repository be payload agnostic. The repository can be implemented to accept and send different data formats by different providers; therefore, the interface between the FE and repository must be able to support different data formats; otherwise, the change in the data format by the repository will compel the FE to also adapt to the change.

4.2.2 Interfaces

This section discusses various interfaces among the functional entities and among the service providers. The interfaces are categorised as those enables data exchange and those allow to request or modify services. The former type is called data interface and the later type is called control interface. First sub-section discusses the data interfaces while second subsection discusses the control interfaces.

4.2.2.1 Data Interface

The data interface carries the data among the various functional entities and users. The data interfaces are among the functional entities. Discussed below are the various data interfaces used by the functional entities to exchange data.

The splitting of the presence server induces the communication overhead. Therefore, the interface between these entities must be light weight and in accordance with the architectural principles.

i. The interface between frontend and cache: As cache is used for fast data access, it is imperative to keep the interface lightweight. Therefore, the interface between the frontend and the cache should be as the native interface offered by the caching solution employed.

ii. The interface between the frontend and the repository: for this interface, different interfaces were considered and compared based on the following requirements:

- **Low Overhead (Lightweight):** The interface needs to be lightweight so as to minimise the overhead caused due to the network communication.
- **Cacheable:** The interface should be able to cache the results so as to reduce the network calls to the repository. This also conforms to the first architectural principle.
- **Data Agnostic:** This requirement conforms to the second architectural principle. The interface should be able to carry various data formats.

The interfaces compared are REpresentational State Transfer (REST), SOAP and Lightweight Directory Access Protocol (LDAP). Below is the brief description of each interface:

- **REST:** REST is an acronym for REpresentational State Transfer. REST is a style to build and deliver the web services over the internet. RESTful web services are those following the REST principles. REST follows the client-server architecture of the web and is not tied to any specific protocol and data representation [52]. However, as HTTP is the dominant protocol on the web, REST is mostly associated with the HTTP.

  REST is not an architecture but a set of design criteria to build web services. The main entity which is of concern in RESTful web services is a Resource. A resource is anything that can be stored on a computer and can be referenced. Resources are identified by a URI which is the name and address of the resource.

  RESTful web services have following features: addressability, statelessness, connectedness and uniform interface [52]. Addressability is a feature to identify and address each resource which is done through a URI. Statelessness means that each request happens in isolation meaning that each request has enough information for the server to process that request. The server does not rely on information from previous requests. Statelessness leads to better scalability and performance. Connectedness means that resources can be reached from the representation of the other resources via links. Uniform interface means that the REST resources can be accessed and manipulated in a standard way.

Resource Oriented Architecture (ROA) is an architecture that is built on RESTful web services. ROA supports a wide range of data formats (e.g. JSON and XML) and uses HTTP as a communication protocol. In ROA architecture resources can be accessed and manipulated using HTTP methods such as GET, POST, PUT and DELETE. These methods roughly map to the Create, Read, Update and Delete (CRUD) operations.

- **SOAP:** SOAP is a protocol to exchange XML messages over the internet. Fundamentally, SOAP is one-way communication model but can be combined with other messages to implement request/response patterns. SOAP specification defines a messaging framework which is platform independent and programming language neutral [53]. SOAP can use several protocols for transport however HTTP is most commonly used protocol. SOAP-based architecture defines three entities: service requester, service provider and service registry. The service provider publishes the service to the service registry using a service description language. The service requestor queries the registry to get the service and create the bindings. In SOAP-based web services, there’s no concept of resources, and they hardly use HTTP’s features which mean they are not addressable, cacheable or respect uniform interface [53]. When using HTTP, SOAP mostly uses POST method.
- **LDAP:** LDAP is a Lightweight Directory Access Protocol; based on X.500 standards for accessing directory services over the network. LDAP is a fairly complex protocol. However, LDAP is lightweight compared to its predecessor X.500 [54]. A directory service allows users

to find information about users, network devices and so on which are stored in a searchable repository [54]. LDAP follows the client-server model and LDAP directories are arranged hierarchically. The hierarchy tree structure is called Directory Information Tree (DIT). The protocol is described using Abstract Syntax Notation 1 (ASN.1) and encoded using Basic Encoding Rules (BER). Since directories are used to store data which changes infrequently, the data repository is highly optimised for reads than writes.

Based on the study of the various interfaces, discussed below is the choice of the interface and the motivation to choose that interface:

The LDAP is a binary interface which makes it the most lightweight among the three, but it fails in the other two requirements. SOAP can carry only XML payloads which fail it in the third requirement. Moreover, XML payloads are also larger in size and have much overhead. RESTful interfaces are text based and can carry multiple data formats. Also, the RESTful interface supports caching. Moreover, RESTful interfaces are most accepted and dominant in the cloud computing environment. Among the three discussed interfaces RESTful interfaces satisfies all of the requirements. Therefore, the RESTful interface was selected as the choice for the communication interface between the frontend and the repository. Table 4.1 below summarises the comparison of the interfaces.

Table 4.1 FE and Repository interface comparison

Criteria ▼ Interface ▶	RESTful	SOAP	LDAP
Low protocol overhead	✓	✗	✓
Data agnostic	✓	✗	✗
Cacheable	✓	✗	✗

4.2.2.2 Control Interface

The control interface allows the service consumers to request for new services and modify existing services. The control interface exists between the service providers. Discussed below are the various choices for control interface and the motivation for the chosen interface.

To request for new services or manage existing services, the service providers expose APIs usable by the service requestors. For the control interface, two alternatives were considered, that is, REST and SIP. REST is considered because it is understood to be lightweight interface when compared to various other interfaces as established in the section 4.2.2.1 above. Moreover, RESTful interfaces can utilise HTTP which is natively available on every device. Therefore, RESTful web services have greater acceptance and widely used in cloud computing environment. SIP is considered because the core protocol for the architecture is SIP. Therefore, the use of SIP can be extended to provide control features too without introducing a new protocol. However, unlike HTTP, SIP is not natively available and need specialised devices which can understand the SIP communication. If chosen as a control interface the service providers and the consumers will have to build infrastructure to handle SIP communication which includes setting up new servers and getting specialised clients for SIP.

RESTful interface apart from being lightweight does not require additional setup on the service consumers and service provider's part. Therefore, the RESTful interfaces are the proposed choice for the control interfaces.

Figure 4.2 below illustrates the interaction between the entities and the interface they use for the interaction.

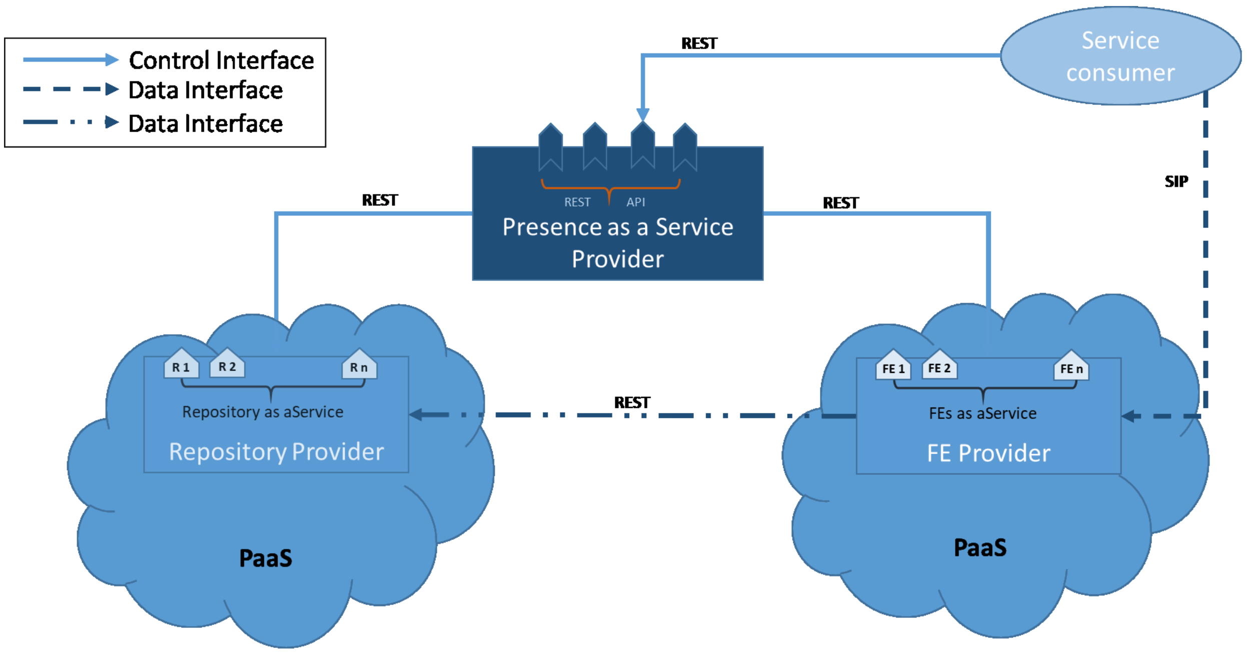


Figure 4.2 Entities and interfaces

4.2.3 Architecture

This subsection provides the details of the proposed architecture.

4.2.3.1 Preliminary Architecture

The preliminary architecture splits the traditional presence service into three independent components. The three components are as follows: a stateless application front end, a repository and a cache. These components are deployed on a PaaS platform. The preliminary architecture follows the business model as proposed in section 4.1. A case study [55] was conducted to study the impact of splitting on the performance of the presence server and evaluate the preliminary architecture for the elastic scalability. The preliminary architecture from the case study is shown in figure 4.3.

The preliminary architecture followed the architecture principles described earlier. The traditional presence service was split into three components following the first principle. The REST interface between the frontend and repository is in accordance with the second principle. The first principle states the use of cache for providing faster data access to the frontends. However, there are various ways for the cache placement such as to have a common cache for all the instances of the frontends in a cloud or have an individual cache for each frontend instance. Each approach offers some benefits and drawbacks which are discussed in a greater detail in the next section. Below discussed is the functionality of the various components in the proposed preliminary architecture:

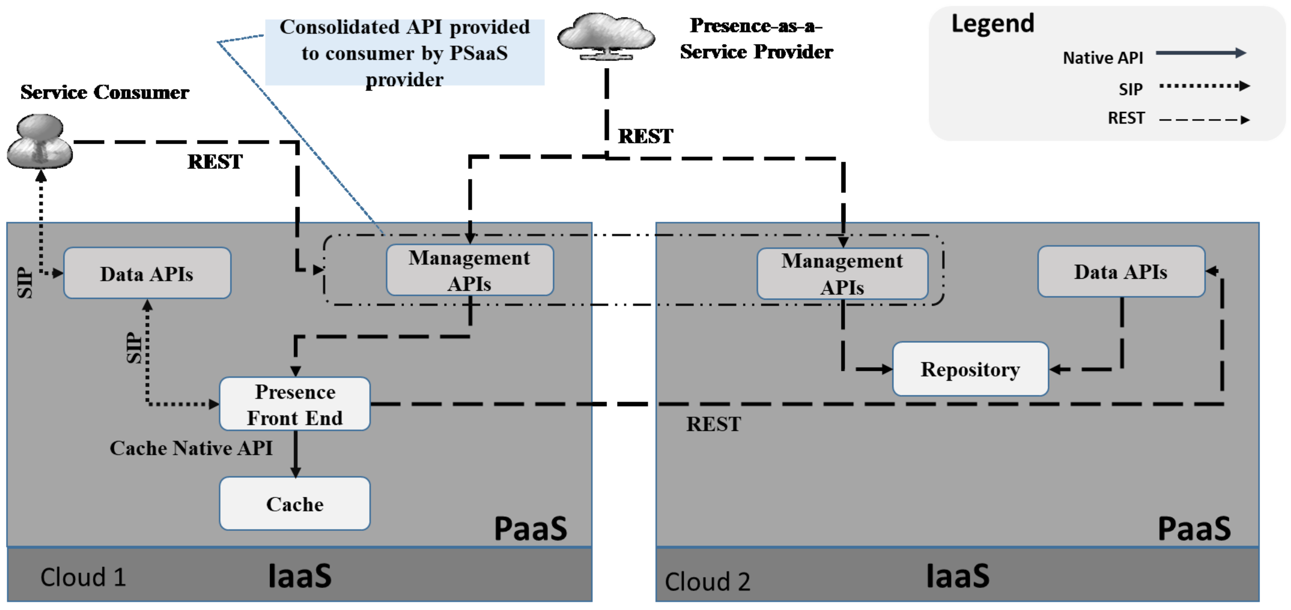


Figure 4.3 Preliminary Architecture

- **IaaS:** IaaS layer provides the infrastructure service to the hosted PaaS.
- **PaaS:** PaaS layer provides the environment to develop and execute the applications.
- **Presence frontend:** The FEaaS provider provides presence frontend. FEaaS also provides the management APIs to request and manage the exposed services. The presence frontend accepts requests from the consumers through the data interface. The consumers can request to provide

the presence information (publication request) or request the presence information of other users (subscription request). Upon receiving the publish or subscribe request, the presence frontend inserts or updates the data in the repository as well as in the cache.

- **Repository:** The RaaS provider provides the repository. RaaS provider also provides the control interfaces for requesting and managing new services. The repository provides the data access through the RESTful APIs. The repository can be in the different cloud than the presence front end.
- **Cache:** The PaaS provider provides cache, or the frontend can deploy a custom caching solution. The purpose of the cache is to provide quick data access to the frontend and save unnecessary network calls to the repository.
- **Presence Service-as-a-Service:** The Presence-Service-as-a-Service composes the complete presence service solution from one of the various presence frontends and repositories. The choice of frontend or repository may be influenced by the user requirements such as latency or throughput.

The evaluation of the preliminary architecture provided the positive outcomes. The results showed improved performance, better throughput, lower latency and higher elastic scalability than the traditional architecture. However, this architecture bundles together the handling of all the presence messages. When handling all the messages in a single entity, processing of one type of message impedes the processing of other messages, thereby affecting the performance. Therefore, considering how publication, subscription and notifications are handled, we believed that the splitting of the application frontend may further improve the elastic scalability of the system and can also achieve better performance and higher throughput. This proposed splitting is the extension of the preliminary architecture presented as part of the final architecture discussed later.

4.2.3.2 The Cache Issue

Having individual cache for each instance can provide some performance benefits as each cache instance will service requests only from one frontend instance but there is a bigger problem to tackle using individual cache approach. Having an individual cache has a high potential to provide stale and inconsistent presence information to the subscribers. Details of such a scenario are provided next.

A frontend instance will update its servicing cache only and update the repository upon receiving the presence information from a presentity. If the subscribe request for the presentity is received on the same frontend instance as receiving the publish request, the subscriber will be notified using the latest information from the cache. However, there may be a case where a presentity updates its presence information which may be processed by some other frontend instance. In such a case different cache instances will have inconsistent presence information of the presentity. Subscribers will receive inconsistent and stale presence information in this case. To avoid such a scenario, there are few alternatives: i) Ensure all publish from a presentity goes to the same instance. ii) Let the publish request go to any instance but synchronise the cache after every insert or update. In the cloud environment, to minimise the resource wastage, the instances are created and destroyed on a need basis. Using the first option will keep the instances from being destroyed, therefore, becoming highly resource inefficient and not realising the potential of cloud computing. Option two can quickly turn into network traffic storm with each increasing instance. Also, the cache will be busy synchronising with each other instead of servicing the clients. Figure 4.4 below illustrates the scenario of subscriber getting stale and inconsistent presence information due to individual cache for the presence frontends. However, having a single cache instance for all the instances of frontend on a cloud platform does not suffer from any of the above-mentioned issues. Considering various factors, it is proposed to use a single cache for all the frontend instances on a cloud platform.

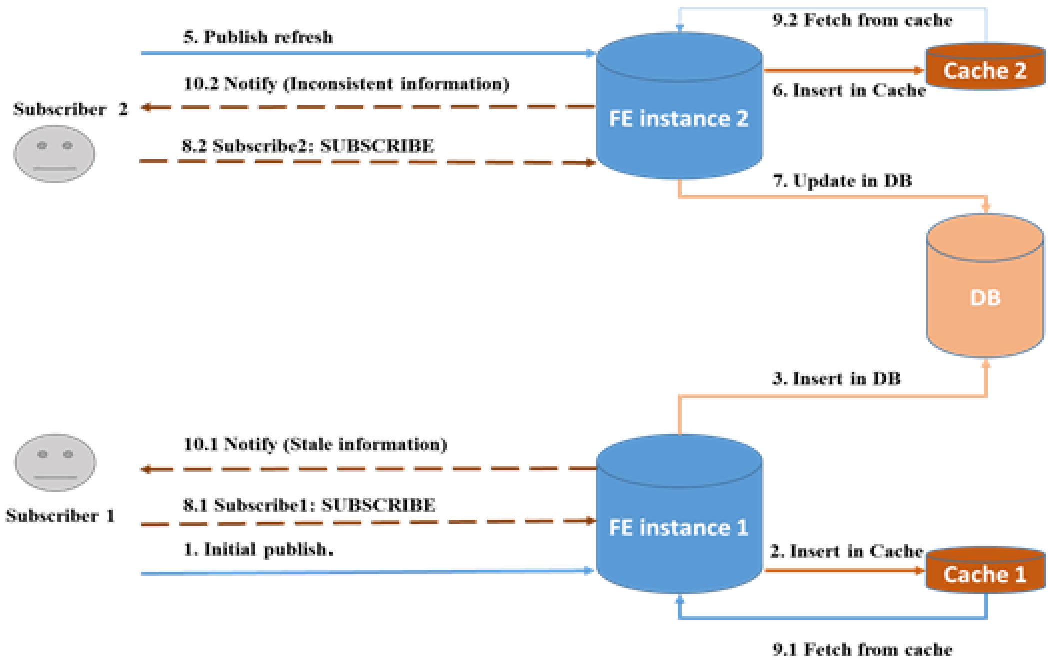


Figure 4.4 Individual cache problem scenario

4.2.3.3 Final Architecture

The motivation to propose a new architecture for a presence server in the cloud was to achieve elastic scalability and improve the resource efficiency of the presence server in the cloud environments. By splitting the monolithic presence server into an application frontend, a database and a cache, the preliminary architecture was able to provide the elastic scalability and increase the resource efficiency of the presence server in the cloud. However, the application frontend in the preliminary architecture still bundles lots of functionality. Managing publications, subscriptions and notifications being the major three functions. In the extended architecture, we propose to disintegrate the application frontend into separate smaller services each of which will handle a major function. The proposed splitting decision is based on the microservice architecture style. Handling publications and subscriptions are disparate functionality. Therefore, these functionalities can be easily split into two independent components. The notifications function is coupled to the publication and subscription functions as it is triggered by each publication and subscription activity. However, the notification functionality is the most complex among the three. To send a notification the server has to merge several presence documents, apply presentity rules and then apply subscriber rules per subscriber. All the processing is done on the XML documents. XML processing can be time-consuming and is also resource intensive operation. More the time and resources spent on the notification process means less the time available for serving publication and subscription request. Therefore, we propose to split the notification function too into a separate independent service. The application frontend now is disintegrated into three independent services. The three services handle publication (PUBLISH), subscription (SUBSCRIBE) and notification (NOTIFY) processing respectively. A message queue is used to enable the communication between the services. The message queue ensures the in-order delivery of messages. The publication and subscription services will publish to the queue, and the notification service subscribes to the queue. Figure 4.5 shows the extended architecture. The difference from the preliminary architecture is that the frontend is designed in microservice architectural style. The rest of the architecture remains as in the preliminary architecture. Each service connects to the cache to store and retrieve data for fast access and connects to the remote central database using the REST API. The cache is shared among all the service instances within the datacenter. The perceived benefit of this design is that each service is solely responsible for handling its respective message and does not affect the processing of other messages.

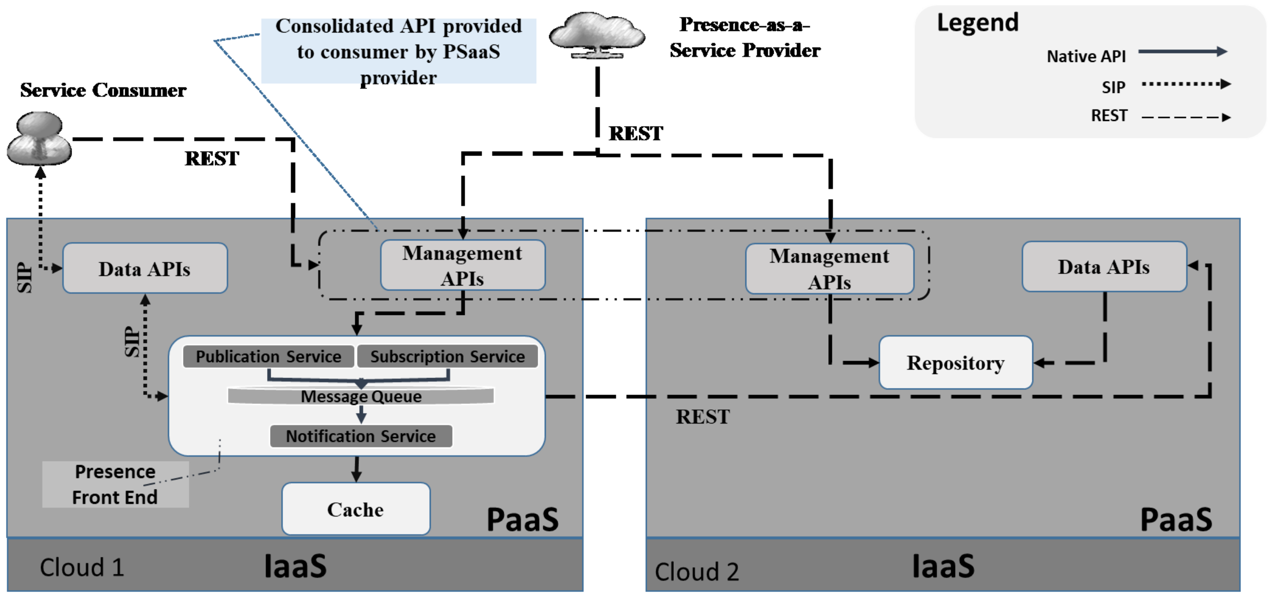


Figure 4.5 Proposed Architecture

4.2.4 REST resources

The proposed architecture relies on the REST interfaces for communication with the database. A resource can be uniquely identified by a Uniform Resource Identifier (URI). Table 4.2 summarises the proposed resources, their operations and the URIs. The presence server can create, retrieve, update and delete the resources from the repository using the REST APIS. Following are the three identified resources:

i. **Presentity:** the presentity resource represents the publisher which publishes it presence documents.

ii. **Watcher:** the watcher resources identifies the subscriber that requests the subscription for status updates for a presentity.

iii. **Subscription:** the subscription resource is created whenever a watcher request for a new subscription for any presentity.

Table 4.2 Presence resources

Resource	Operation	HTTP action
Presentity	Create: new presentity	POST:/presentity/
	Update: existing presentity	PUT:/presentity/{id}
	Fetch: existing presentity	GET:/presentity/{id}
	Delete: existing presentity	DELETE:/presentity/{id}
	Check: for the existence of presentity.	HEAD:/presentity/{id}
Watcher	Create: new watcher	POST:/watcher/
	Update: existing watcher	PUT:/watcher/{id}
	Fetch: existing watcher	GET:/watcher/{id}
	Delete: existing watcher	DELETE:/watcher/{id}
	Check: for the existence of watcher.	HEAD:/watcher/{id}
Subscription	Create: new subscription	POST:/subscription/
	Update: existing subscription. Keys identify the subscription and filter define the specific fields to update.	PUT:/subscription/?keys={}&filter={}
	Fetch: existing subscription. Keys identify the subscription and filter define the specific fields to update.	GET:/subscription/?keys={}&filter={}
	Delete: existing subscription. Keys identify the subscription and filter define the specific fields to update.	DELETE:/subscription/?keys={}
	Check: existence of existing subscription	HEAD:/subscription/?keys={}

4.3 Illustrative Scenarios

This section presents two basic scenarios to show how the presence service works in the proposed architecture. The first scenario shows the steps involved in the publication process and the second scenario shows the initial subscription process. The scenarios are presented to show the interaction between various entities in the proposed architecture therefore complex details, and steps are omitted from the discussion.

4.3.1 Scenario: Initial publication request by the presentity

This section illustrates the process that takes place when a presentity publishes its presence status for the first time. The overall scenario is depicted in figure 4.6.

The scenario starts with the sending of the initial SIP PUBLISH message by the presentity. The publication service receives the message and sends the SIP 200 response to the presentity (steps 1 & 2). Once the response is sent to the presentity, the publication service inserts the presentity details and the presence document into the cache and also inserts the details into the database using the REST API. The HTTP method used to insert into the database is POST. The database responds with HTTP 201 created response for the successful creation of the presentity. The publication service then publishes a message in the message queue with all the required details (steps 3-7). The notification service upon receiving the message from the queue identifies the presentity and fetches all the subscriptions for that presentity from the database using the REST API call. Upon receiving the subscriptions, the notification service sends SIP NOTIFY message with the presence status of the presentity to each watcher found in the subscriptions fetched from the database. Each watcher responds with a SIP 200 response to the notification service upon receiving the NOTIFY message.

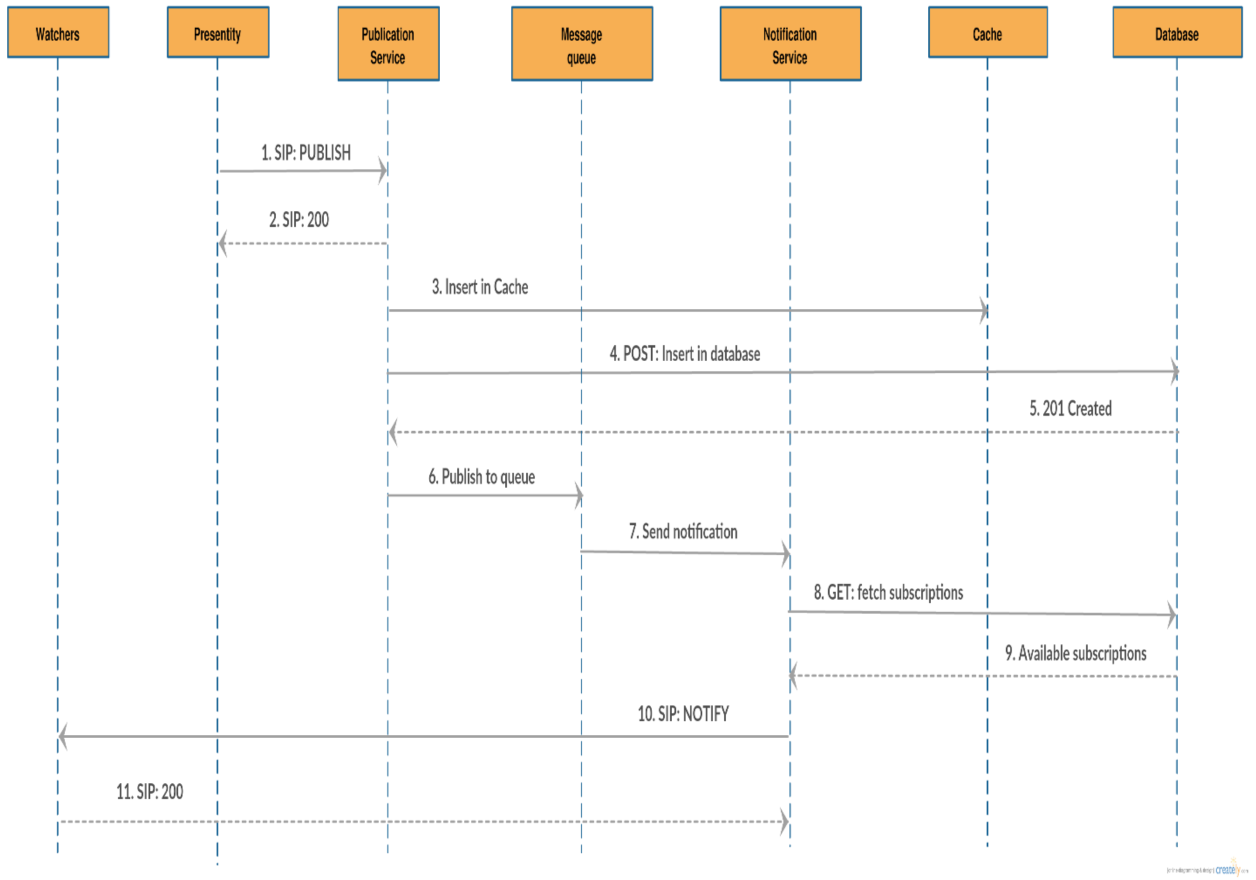


Figure 4.6 Publish scenario

4.3.2 Scenario: Initial subscription request by the watcher

This section presents the scenario of a watcher sending the subscription request. It is assumed that this is the first subscription request from the watcher and the subscribed presentity has already published its presence information. Also, in the scenario, everyone is allowed to see others presence information. The scenario is depicted in figure 4.7.

To start the scenario, the watcher sends a SIP SUBSCRIBE request to the subscription service. The subscription service responds with a SIP 200 message (steps 1 & 2). After sending the response, the subscription service inserts the subscription into the cache and through REST APIs HTTP POST method into the database. The database returns HTTP 201 created response upon successful completion of the request (steps 3-5). Once the subscription service finishes with the processing of the SUBSCRIBE request, it publishes the subscription message into the message queue with all the necessary information (steps 6-7). The notification service which has subscribed to the message queue picks up the message just posted and retrieves the presentity id from the message for which the subscription has been made. Once the presentity id has been retrieved, the notification service fetches the presentity’s presence documents from the cache. Once all the presence documents for the presentity have been fetched, the notification service processes the documents to create single presence documents (steps 8-11). Once the final presence document has been created, the notification service sends a SIP NOTIFY message with the presence status of the subscribed presentity to the watcher. The watcher, upon receiving the NOTIFY message responds with a SIP 200 response (steps 12-13).

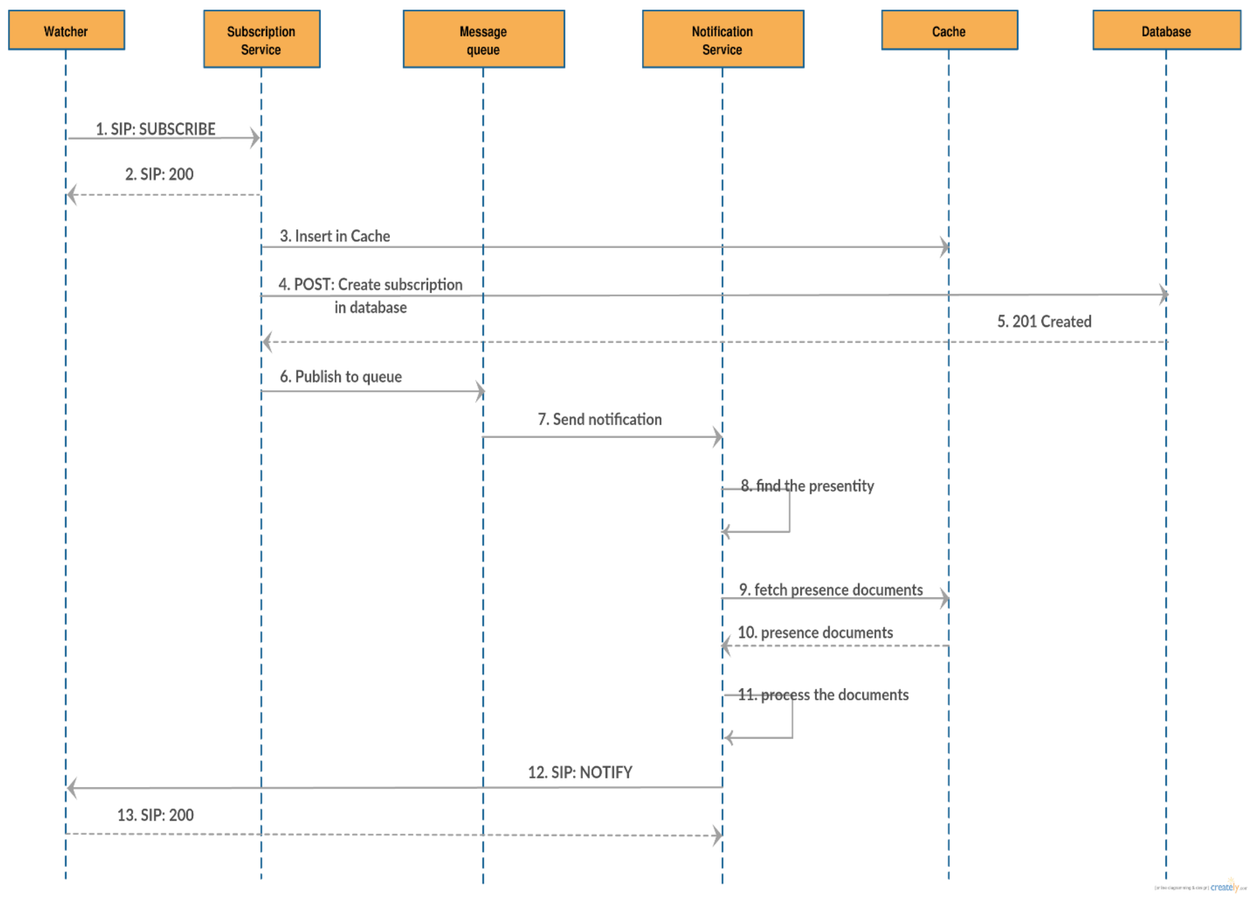


Figure 4.7 Subscription scenario

4.4 Summary

This chapter first describes the business model and its various actors followed by the overall architectural details. Architectural principles that were followed during the design were also discussed. Discussed later on was the choice of communication interfaces between the various functional entities and motivation for the choice. The preliminary architecture and the final architecture was discussed followed by the discussion on the REST resources and illustrative scenarios showing how various components of the proposed architecture communicate with each other.

Chapter 5

Prototype Implementation and Evaluation

The previous chapter discussed the proposed architecture for the presence service in the cloud. This chapter focuses on the software architecture, prototype implementation and evaluation of the results. First sub-section provides details of the software architecture of the implemented prototype. After that, details of the implemented prototype are provided. The last sub-section compares and discusses the performance measurement results of the proposed and the traditional presence server architecture.

5.1 Software Architecture

Figure 5.1 shows the overall software architecture for the proposed presence service in the cloud. The architecture consists of three layers: the SIP server layer, the presence layer and the communication layer. The SIP server layer consists of the components that provide the core SIP server functions such as protocol parsing, network management, memory management and more. The presence layer is responsible to provide the presence functionality. It consists components such as publication service component, subscription service component and notification service component that performs various presence functions. These presence components receive SIP messages from the SIP server layer and perform the intended function. Furthermore, these components interact with several external components such as a cache component to store data for fast access, to the API server component for communication with the repository and with the messaging component to communicate with each other. The communication layer provides the required connectors to communicate with the external components such as the API server, messaging and cache components. The API server uses the repository provided an interface to interact with the repository.

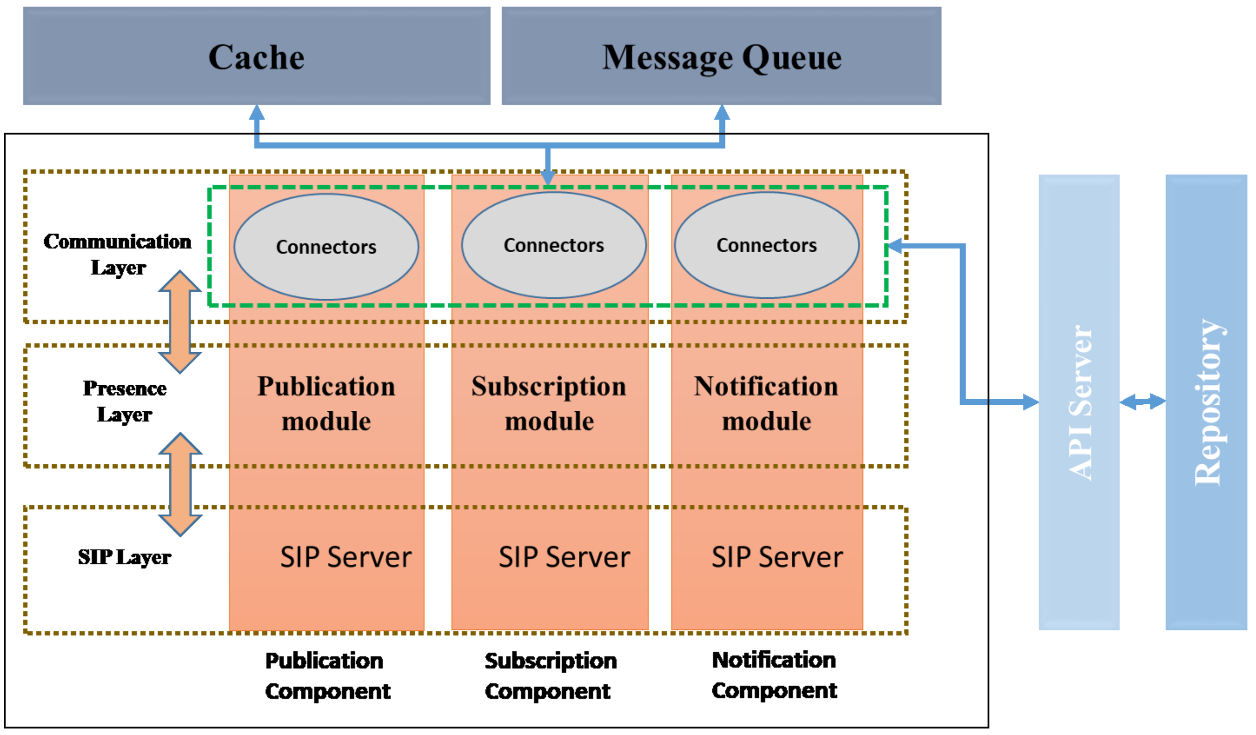


Figure 5.1 Proposed Software Architecture

5.1.1 Operational Procedures

Based on the proposed software architecture presented below are the operational procedures for publish and subscribe requests.

Figure 5.2 shows the interaction of the software components during the processing of publish message request assuming nobody has subscribed yet. The presentity sends a PUBLISH message to the presence server. The request is received by the publication service component which acknowledges the presentity by sending the SIP 200 message and then inserts the message in the cache component. After that, it inserts the message into the repository component using the REST API. Simultaneously, it checks with the repository for existing subscriptions for the current presentity. The API server responds with a '404 not found' response, meaning no subscriptions exists. In such a case, the publication service component refrains from posting a message to the messaging component as there would be nothing to be processed by the notification service component.

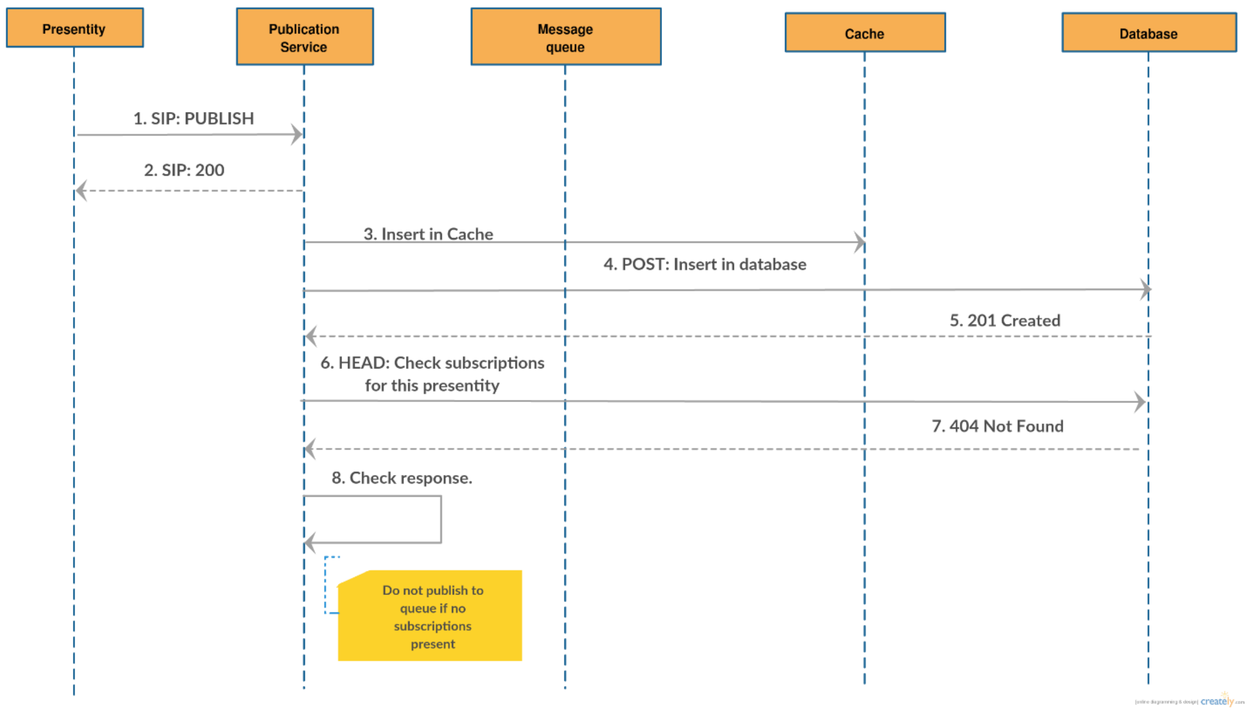


Figure 5.2 Publish message processing in the absence of subscriptions.

Figure 5.3 shows the interaction of the software components during the processing of subscribe message request assuming that the presentity for which the subscription request is made has not yet published. The watcher sends a SUBSCRIBE request to the presence server which is received by the subscription service component. The subscription service component acknowledges the request with a SIP 200 response. After that, the subscription service component inserts the request details in the cache component and request the API server to create the subscription resource in the repository. Assuming the successful creation of the resource, subscription service component receives a ‘201 created’ response and publishes the message to the messaging component to be picked by the notification service component. The notification service component extracts the presentity details from the message and checks with the cache component for the availability of the presentity. Unable to find in the cache (as per the assumption), the notification service component requests the API server component to search the presentity in the repository. The API server component responds with a ‘404 not found’ message. In such a case, the notification service component sends a dummy message to the watcher. The watcher acknowledges the message with SIP 200 response.

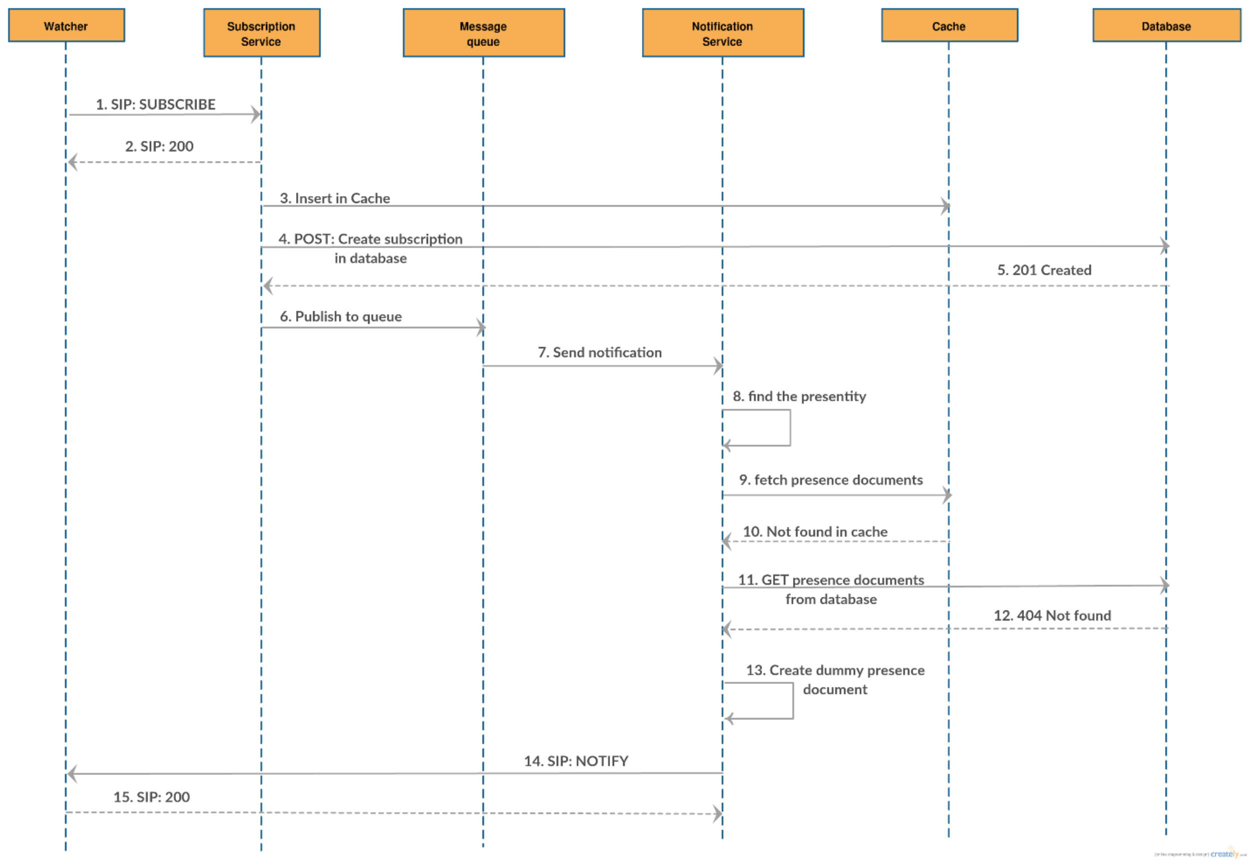


Figure 5.3 Subscribe message processing in the absence of publication.

5.2 Prototype Implementation

This section first describes the scenarios considered for the test. Then it describes the prototype architecture and execution environment setup. The last sub-section provides the description of software tools used for the implementation of the prototype and prototype setup.

5.2.1 Implemented Scenarios

To measure the performance of the proposed architecture six scenarios were implemented. Based on these six scenarios the traditional and proposed architecture was evaluated. The six scenarios are divided into three groups. Each group is divided based on the load it puts on the server. The six scenarios are built upon the two simple scenarios. First, a PUBLISH-NOTIFY scenario, where a presentity publishes its presence status and a notification is sent to the subscribers if any present. Second, a SUBSCRIBE-NOTIFY scenario, where a watcher sends a subscription request and a notification with the presence status of the requested presentity is sent to the watcher. Each group has these two basic scenarios discussed before; the only difference lies in the number of presence documents published by a presentity and number of watchers subscribed to a presentity in SUBSCRIBE-NOTIFY and PUBLISH-NOTIFY scenarios respectively. Following are the three groups of scenarios:

- **Group 1**

  1 **Scenario 1: Publish, no Notify (No subscriptions)**

  In this scenario, there are no subscriptions present which means there will be no NOTIFY messages generated. This scenario measures the PUBLISH message handling capabilities of the presence server under no other load. This scenario offers the lowest load among the three publish-notify scenarios.

  2 **Scenario 2: Subscribe-Notify (No publications)**

  This scenario measures the SUBSCRIBE message handling capabilities of the presence server under a minimum load of Presentities. In this scenario, there exists no presence data of any Presentities subscribed by the watchers. Therefore, in this scenario, the server, for each subscribe request, first looks for respective Presentities presence status in the cache where it does not find any, then the server checks in the repository where also it does not find any records. Finally, the server prepares a blank NOTIFY message for the presentity.

Once the message is ready, the server sends the NOTIFY message to the watcher. This scenario offers the lowest load among the three subscribe-notify scenarios.

- **Group 2**

3 **Scenario 3: Publish-Notify (1 subscription/presentity)**

This scenario tests the performance of presence server under minimal notification load. The scenario offers a single subscriber per presentity, meaning, a single NOTIFY message will be generated for each PUBLISH message received. The scenario offers slightly higher load than the scenario 1 but tests the performance of handling publications along with generation of notifications.

4 **Scenario 4: Subscribe-Notify (1 publication/presentity)**

The previous subscribe-notify scenario i.e. scenario 2 tests the server when no Presentities were available which meant there is no presence XML to handle by the server. This scenario puts to test the performance of the server when there's a presentity available for each subscriber. It is assumed that each presentity has already published its presence status. As it is assumed that presentity has already published, the presence document will be fetched from the cache. Therefore, in this scenario, for each SUBSCRIBE request the server fetches the presence document of the concerned presentity and creates a NOTIFY message using the fetched presence XML document. Once the NOTIFY message is generated, the server sends a notification to the concerned watcher. This scenario tests the performance of server when handling subscribe and notify messages along with XML processing of the presence documents.

- **Group 3**

5 **Scenario 5: Publish-Notify (10 subscriptions/presentity)**

This scenario is similar to the publish-notify scenario 3, but instead of a single subscriber per presentity, this scenario handles 10 subscribers per presentity. Under this scenario, each received PUBLISH message will generate 10 NOTIFY messages. This scenario measures the performance of the presence server under highest load among the three publish-notify scenarios.

6 Scenario 6: Subscribe-Notify (5 publication/presentity)

A presentity can use multiple devices such as a laptop, office PC, mobile, tablet, and more. It is possible that each device is capable of sharing presence status of the user which means that a user can have multiple presence status documents available at any instance. The server has to derive correct presence information from all these documents and has to present a single presence status document to the watchers. Therefore, this scenario measures the performance of the presence server under the conditions where each watcher has subscribed to a single presentity, and each presentity has five presence documents published. Hence, for each SUBSCRIBE request, the server has to fetch five presence documents and merge them into a single document to prepare the NOTIFY message. This scenario puts the highest load on the server among the three subscribe-notify scenarios. Also, this scenario measures the performance under heavy XML processing along with subscription and notification handling.

5.2.2 Software Tools

This section briefly describes the software tools used for the implementation of the prototype.

5.2.2.1 Opensips

Opensips [56] is an open source software implementation of SIP server. Opensips unifies voice, video, IM and presence. This software not only provides SIP proxy and router but also provides various SIP functions such as registrar server, application server, IM server, a presence agent, load-balancer, Dispatcher and much more. For the prototype, opensips is used as a presence server and a dispatcher. The dispatcher is used to route PUBLISH and SUBSCRIBE messages to the appropriate service i.e. publication service and subscription service respectively whereas the opensips presence server is modified to build publication, subscription and notification microservices. The three services combined are referred to as the presence server.

5.2.2.2 Redis

Redis [57] is an open source in-memory data structure store. Redis can be used as a database, cache and message broker. Redis supports various data structures such as strings, hashes, sets, sorted sets, lists and more. It is a feature rich NoSQL [58] database that has built-in support for replication, transactions and LRU eviction. Redis ensures high performance by working on the in-memory dataset which can be persisted to disk once in a while. Persistence can be turned off if Redis is used only as an in-memory cache. In the prototype implementation, Redis is used as in-memory cache component.

5.2.2.3 MySQL

MySQL [59] is an open source relational database management system (RDBMS) that uses Structured Query Language. MySQL can be utilised for a variety of applications mainly those which require transactions support and ACID compliance. MySQL is famously part of the LAMP stack that is used for web application development. The prototype uses the MySQL to create the repository component.

5.2.2.4 NATS

In the developed prototype, the NATS server is used as a messaging component. NATS [60] is an open source messaging system. It is very simple and high-performance messaging system written in GO language. NATS was originally developed as messaging plane for the Cloud Foundry PaaS which currently supports various messaging models such as publish-subscribe, request reply and queueing. The basic NATS server is a fire and forget system which means if the client is not listening when the message is sent, the client will not receive the message later. NATS provide guaranteed delivery using the NATS streaming server. To enable high scalability, NATS automatically cuts off the clients who are unresponsive.

5.2.2.5 Apache Tomcat

Apache [61] Tomcat, generally referred as Tomcat server is an open-source implementation of Java Servlet Container. Tomcat offers a pure Java HTTP web server environment and also implements several Java EE specifications such as Java Server Pages, Java Servlet, WebSocket and Java EL (Unified Expression Language). The API server in the prototype is built on the Tomcat server.

5.2.2.6 SIPp

SIPp [62] is an open source traffic generator tool for SIP protocol. SIPp can be used to test various real SIP equipment like SIP proxies, SIP media servers and more. It can emulate thousands of SIP users calling a SIP system. SIPp provides few basic scenario out of the box, but custom scenario can be created using the XML files. SIPp works with both TCP and UDP protocols over multiple sockets or single socket. Some of the other SIPp features are support for IPV6, TLS, SCTP, SIP authentication, UDP retransmissions and many more. SIPp also displays current test statistics which are updated dynamically on screen and can also be exported to the CSV files. In the prototype, two instances of SIPp were used to generate higher traffic.

5.2.2.7 Kubernetes

Kubernetes [63] is an open source container management and orchestration system developed by Google. It provides a platform for automating deployment, scaling and operations of application containers across a cluster of hosts. Kubernetes can scale the applications automatically based on resource usage, or the user can choose to scale it through the user interface. Kubernetes deploys the containers automatically on hosts which fit the container requirements and also restarts the containers if they fail or reschedule them if the node dies. Among other features, Kubernetes provide automatic service discovery and load balancing. Kubernetes follows a loosely coupled architecture and its various system communicate using the REST APIs. In the prototype implementation, Kubernetes is deployed in one master, three child node configuration.

5.2.2.8 Docker

Docker [64] is an open source software that provides container platform. A container packages the software application along with its dependencies, runtime, code and settings into a lightweight, standalone executable. A containerized application will run the same regardless of the environment. Docker makes it possible to bundle the software application into containers. Containers run on same host operating system but in a separate process. As containers are lightweight, they have very minimal start-up time which allows the system to scale-up rapidly. In the prototype implementation, various presence server services and dispatcher are running as containers.

5.2.2.9 Openstack

Openstack [65] is a popular open source software that is used to build and manage public or private clouds. It lies at the Infrastructure-as-a-Service layer of the cloud computing stack. Openstack is built as a collection of open source tools that handle the core cloud computing services such as compute, storage, and networking. Other than these, openstack also provides image service to store OS images and identity service for authentication. Various openstack services can be availed and managed through openstack APIs or a dashboard. The prototype implementation uses Openstack as the infrastructure over which the Kubernetes virtual machines and various other virtual machines runs.

5.2.2.10 Additional Software Tools

Java Servlets were used to develop the RESTful API to access the repository. cJSON [66] is a JSON parser for ANSI C language which is used in the modified version of opensips presence server. Hiredis [67] is a minimalistic C client to access the Redis database is also utilised in the modified opensips presence server. Heapster [68] which is a cluster-wide aggregator of monitoring and event data is used to collect information about resource usage by containers running on Kubernetes. InfluxDB [69] is an opens source time series database (TSDB). Heapster uses InfluxDB as data sink in the implemented prototype. Grafana [70] is an open source software to visualise metrics. Grafana is used to generate graphs for the resource usage from the data collected in the InfluxDB. The Kubernetes cluster is deployed on the CoreOS operating system. CoreOS [71] is an open source lightweight operating system based on Linux kernel. Ubuntu which is another open source operating system is used to run MySQL, Redis, SIPp, NATS and Tomcat server.

5.2.3 Prototype Architecture

The prototype involves various systems such as a presence server, cache, repository, API server, message queue, a load generator and a dispatcher. The prototype architecture is shown in figure 5.4. The dispatcher and the three presence microservices that are: notification service, publication service, and subscription service are built using the opensource Opensips server and are deployed on the Docker containers. The containers are managed by the Kubernetes container orchestration engine which is deployed in a master child configuration having one master node and three child nodes. Redis is used as the caching solution while MySQL is used for the repository. The messaging system utilises the service of NATS system. The REST APIs are hosted by Apache

Tomcat web server. To generate the traffic SIPp load generator is used. Barring the presence microservices and dispatcher, all the other systems are deployed on high capacity virtual machines. The Openstack serves as the Infrastructure provider to host the virtual machines.

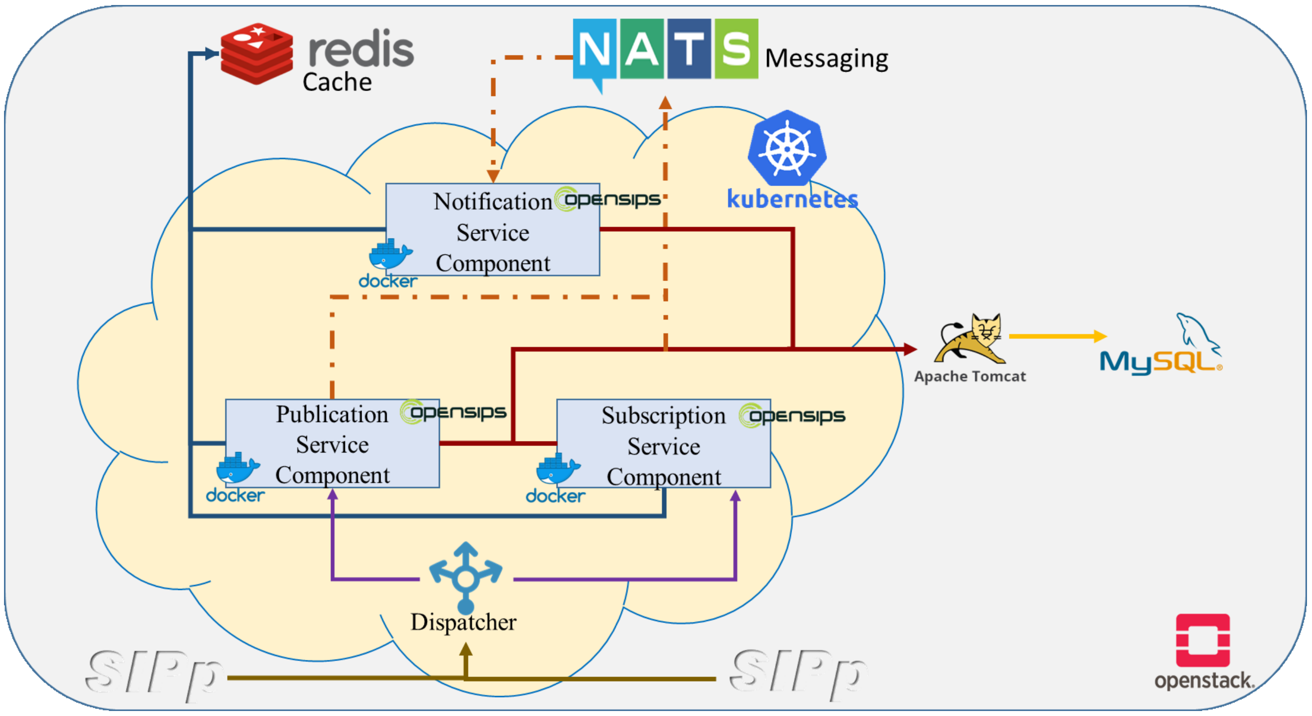


Figure 5.4 Prototype Architecture

5.2.4 Prototype Setup

An Openstack public cloud is used as a testbed for the prototype deployment. The Kubernetes is deployed in a single master node and three child node configuration. Each node of Kubernetes is deployed on a VM. The master node VM has 4 vCPUs and 4 GB RAM while the three nodes each has 16 vCPUs and 32 GB RAM. Each node runs CoreOS operating system. VMs for MySQL, Redis, NATS, Tomcat and SIPp each has 16 vCPUs and 32 GB RAM. To generate high traffic two instances of SIPp were used, each running on its own VM.

The tests were performed till the peak call rate of 2000 calls per second (cps), starting from 20 cps and increasing by 20 cps at regular intervals. The number of calls generated in the tests is much higher than what is done in the related work.

5.3 Validation and Performance Evaluation

This subsection first provide the metrics used for performance measurement, and it later provides the performance measurement results and analysis.

5.3.1.1 Performance Metrics

Following are the three metrics considered for the evaluation of the proposed prototype.

i. **CPU Utilisation:** CPU capacity required to complete a test scenario. It is measured in terms of the number of containers required.
ii. **Memory Utilisation:** This metrics measures total memory utilised to complete a test scenario. It is measured as the cumulative amount of memory required by all the containers.
iii. **Response Time:** This measures the delay in receiving the SIP 200 response message from the server after sending the PUBLISH and SUBSCRIBE message and NOTIFY request message after receiving the SIP 200 by the watchers.

5.3.1.2 Performance Measurement and Analysis

This subsection provides the performance measurement results of the proposed final architecture and analysed them for each metrics.

5.3.1.2.1 CPU Utilisation

Figure 5.5 shows the CPU utilisation for the scenarios in group 1. Figure 5.6 and 5.7 shows the results for scenarios in group 3 & group 4. It is noticeable in figure 5.5 that the proposed architecture (henceforth referred to as Microservice architecture) consumes a higher amount of CPU for both the scenarios in group 1. Group 2 scenarios slightly increase the load on the server which can be noticed by the increased CPU utilisation for both the architectures. Group 2 results also reflect that the microservice architecture consumes more CPU resources than the traditional architecture. However, group 3 results show that the microservice architecture CPU requirements closely matches the CPU usage of the traditional architecture. In group 3, the scenario 5 i.e. the publish-notify scenario requires less CPU in microservice architecture whereas in the scenario 6 i.e. the subscribe-notify scenario microservice architecture consume slightly higher CPU resources. It must also be noted that the group 3 scenario put far more load on the server than the group 1 and group 2 scenarios. To summarise the results, traditional architecture is highly CPU-efficient in scenarios where the server load is on the lighter side. The reason for such behaviour is that the microservice architecture performs several network calls for a single request as compared to the traditional architecture. The traditional architecture uses in-memory built-in cache while microservice architecture uses the external cache. Apart from the cache, the microservice architecture makes network calls to the API server and the message bus. Moreover, the server in microservice architecture needs to maintain TCP connections with cache, API server and message bus. Also, while performing network calls the data needs to go through serialisation and deserialization steps. Maintaining TCP connections and data serialisation consumes CPU resources. Additionally, in microservice architecture, the notification service needs to recreate several data structures to accept the request and process the data. The effect of CPU consuming activities is evident in lighter load scenarios for microservice architecture, but at higher loads, the traditional architecture consumes a similar amount of CPU resources. The reason for traditional architecture consuming more CPU resources can be attributed to the fact that the traditional server builds the in-memory cache at the start-up by pulling a subset of data from the repository which consumes a good amount of CPU resources and the effect adds up to every new instance created.

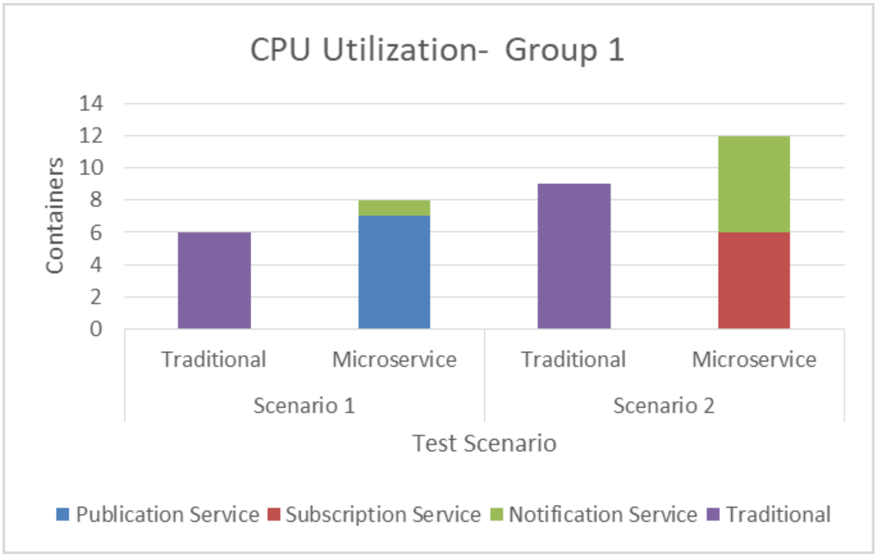


Figure 5.5 CPU Utilisation: Group 1

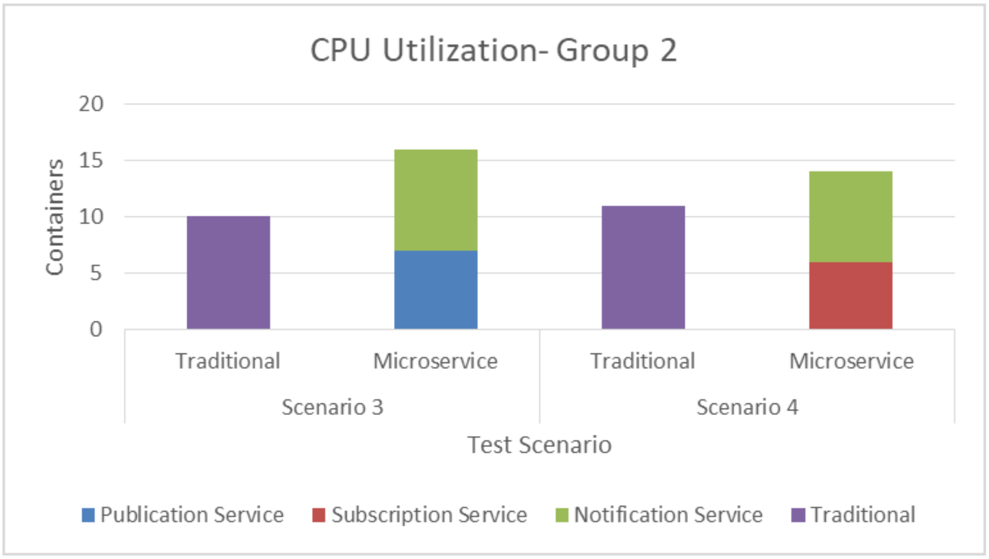


Figure 5.6 CPU Utilisation: Group 2

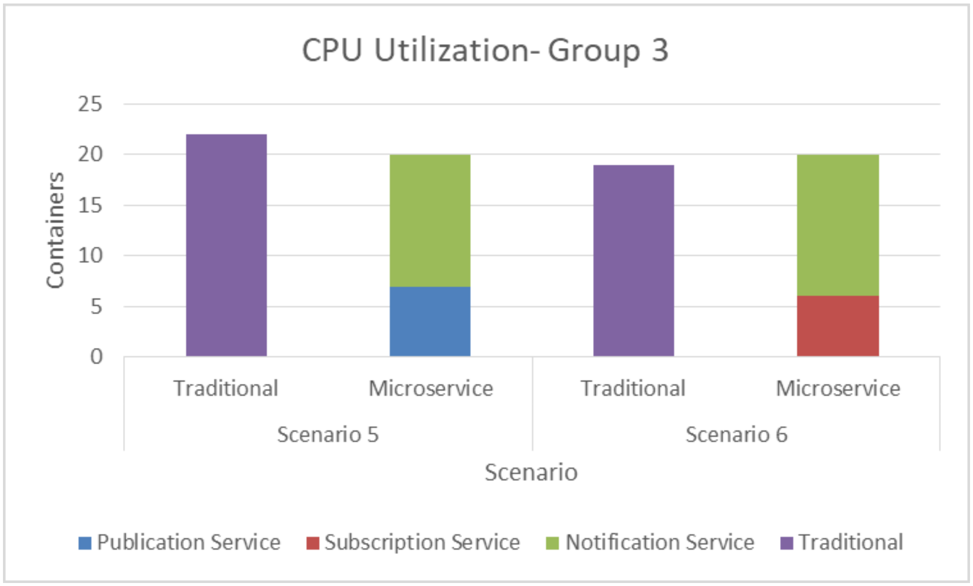


Figure 5.7 CPU Utilisation: Group 3

5.3.1.2.2 Memory Utilisation

Results for the second metric i.e. memory utilisation are shown in Figures 5.8, 5.9 and 5.10 for group 1, 2, and 3 respectively. From the results, it is evident that traditional architecture consumes the significantly larger amount of memory as compared to the microservice architecture. The behaviour is noticeable in every scenario of every group, and as the load increases, memory consumption also increases significantly in traditional architecture. The reason for such behaviour in the traditional architecture is the in-memory cache that is built-in on the server. The server fills the cache at start-up by pulling data from the repository. As the number of instances grows due increased number of calls which leads to scaling, the memory usage grows because each instance builds its cache. The higher the data in the repository more will be the memory utilisation; this can be noticed in figure 5.10. With each group, the memory utilisation also increases significantly as each group has more amount of data than the previous group; this also means that the there's much duplicate information in each cache instance. Moreover, cache across the instances are not in sync; this increases the chance of either the cache miss or old cache data; making the cache fairly useless and wasting the memory resource. Whereas, the microservice architecture uses a central cache that is common to all the instances, which eliminates the duplication of data and wastage of memory; this also eliminates the problem of old data as the data is read from and updated to a common entity. The stateless instances also eradicate the possibility of cache misses and enable better elastic scalability as any instance can serve the request and also can be destroyed when no longer required.

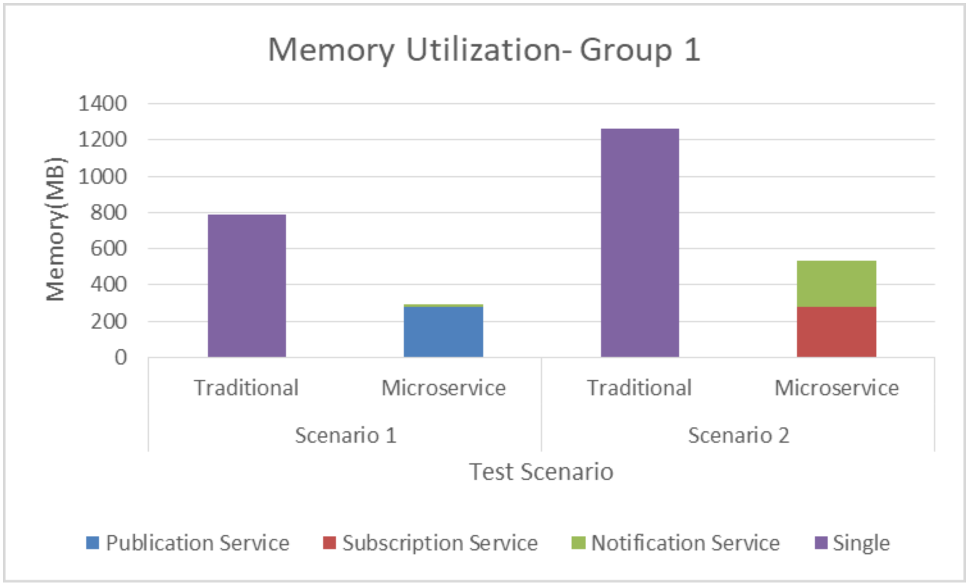


Figure 5.8 Memory Utilisation: Group 1

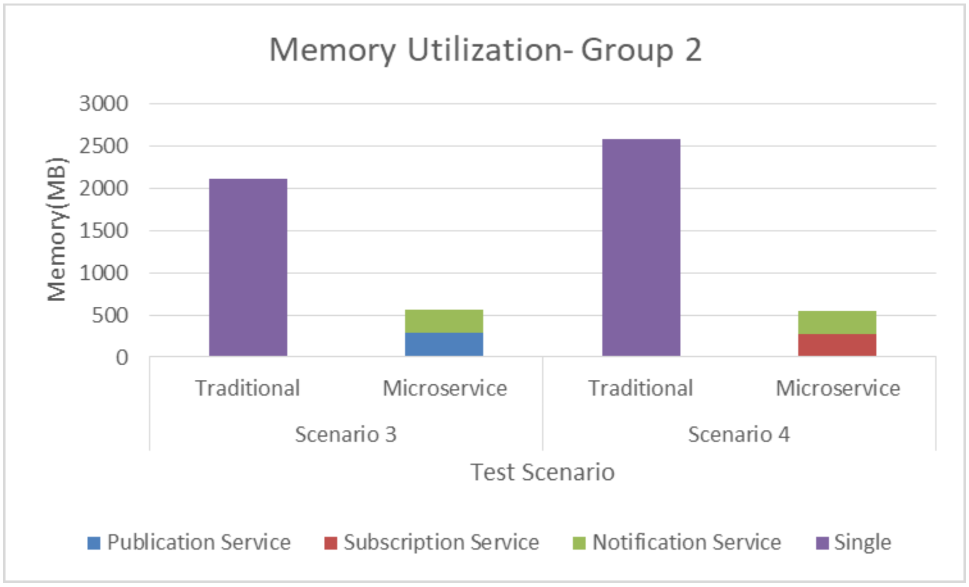


Figure 5.9 Memory Utilisation: Group 2

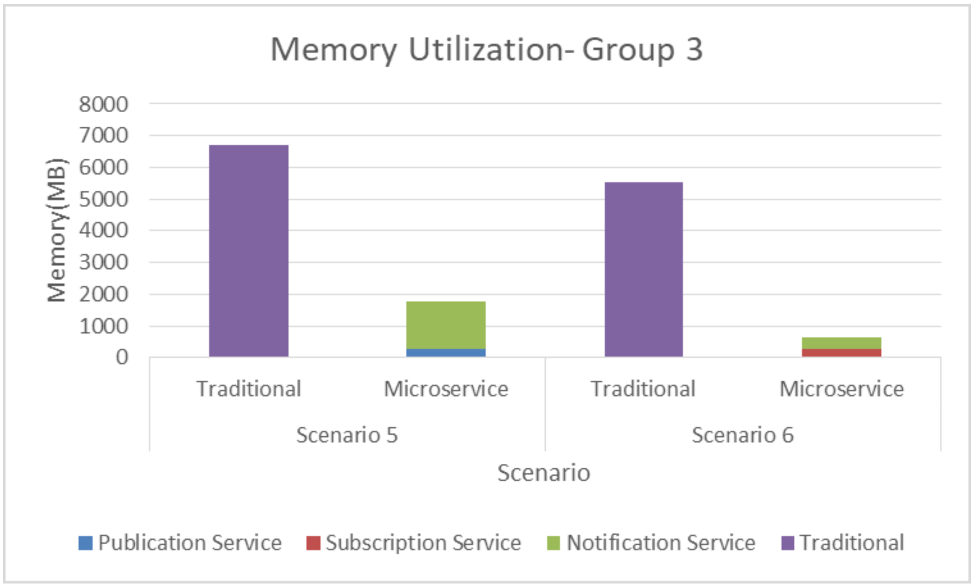


Figure 5.10 Memory Utilisation: Group 3

5.3.1.2.3 Response Time

Finally, the results of third metrics i.e. response time are presented in Figures 5.11, 5.12 and 5.13 for the groups 1, 2, and 3 respectively. The response times are measured for an acknowledgement for a publish request in publication scenario and for the subscription scenario the measurements has two parts – first is the time to receive the message acknowledgement and second is the time to receive the notification after the acknowledgement is received. For group 1, the graph in figure 5.11A suggests that the response time for publish acknowledgement is mostly similar for both the architectures while for the subscription acknowledgement, microservice architecture response time is significantly better than the traditional architecture (figure 5.11B). Whereas, the traditional architecture gives better response time for the notification requests (for approximately 20% calls) as seen in figure 5.11C. However, it is also noticed that the traditional architecture suffers from the 10%-20% failed calls in subscribe-notify scenario (figure 5.11B & 5.11C). In group 2, the results as seen in the figure 5.12 suggest that the microservice architecture performs better than the traditional architecture in every scenario. Moreover, the traditional architecture suffers from the significant amount of failed calls. Results of group 3 are shown in figure 5.13. Group 3 publication results (figure 5.13A) give a similar impression as of group 2 where microservices perform better than the traditional architecture, but the striking point is that while the performance of microservices remains mostly similar as in group 2, there's a significant decline in performance of the traditional architecture (figure 5.13A). Similarly, for subscribe, acknowledgements the microservice architecture performs better than the traditional architecture (figure 5.13B). Comparing the subscribe acknowledgement results with the group 2 (figure 5.12B & 5.13B) shows that the microservices performance remains virtually unchanged while there's a significant decline in the performance of traditional architecture. Additionally, the traditional architecture result shows failed calls too. The results of publish and subscribe acknowledgements demonstrate the benefit of microservices where the load on notification process does not affect the performance of publication and subscriptions handling. The results for notification response as illustrated in figure 5.13C shows traditional architecture performing better for approximately 35% calls, but it also has the approximately 20% failed calls. However, the microservice architecture has only a few failed calls and provides overall better response time than the traditional architecture. To summarise the results of third metrics, overall the proposed microservice architecture has shown better response time across all the scenarios whereas the traditional approach shows better performance for few percent calls at very lower call rate, but at higher loads it has the largest number of failed calls.

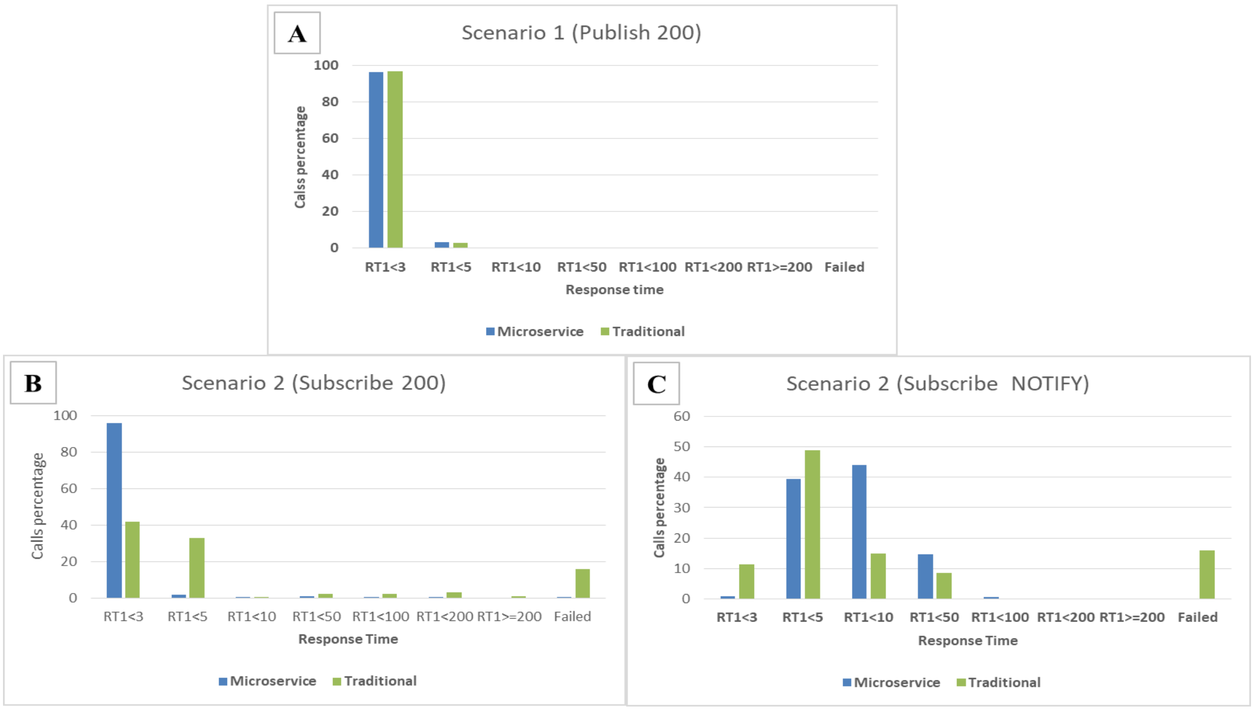


Figure 5.11 Response Time: Group 1

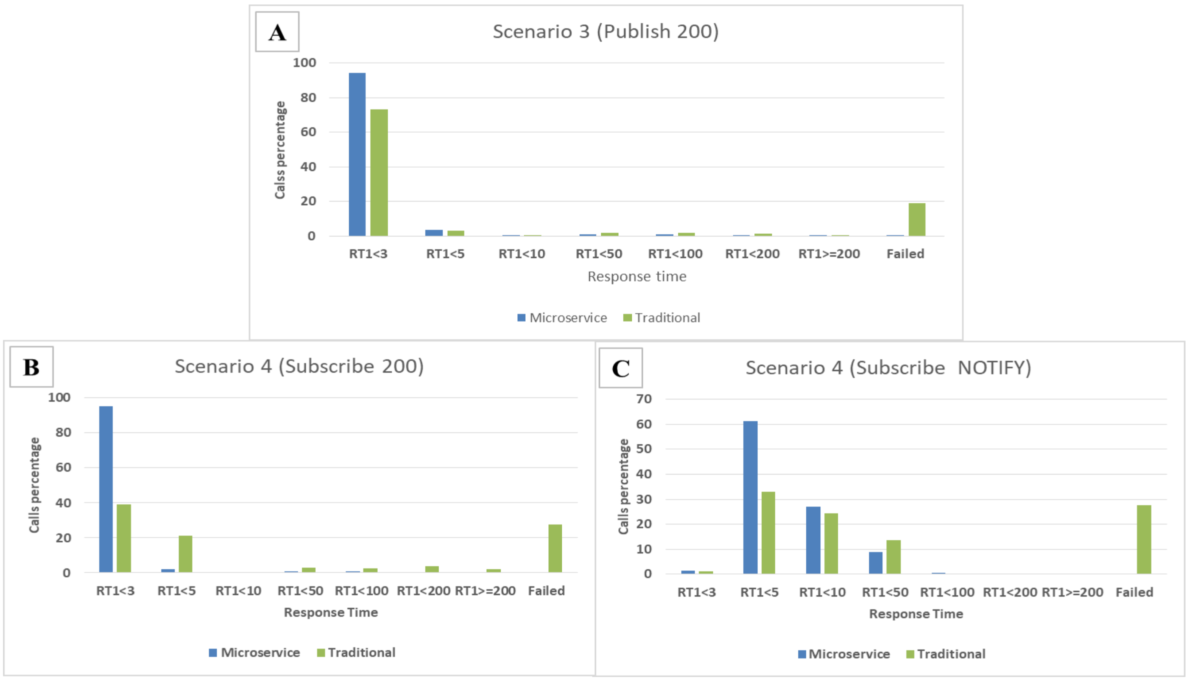


Figure 5.12 Response Time: Group 2

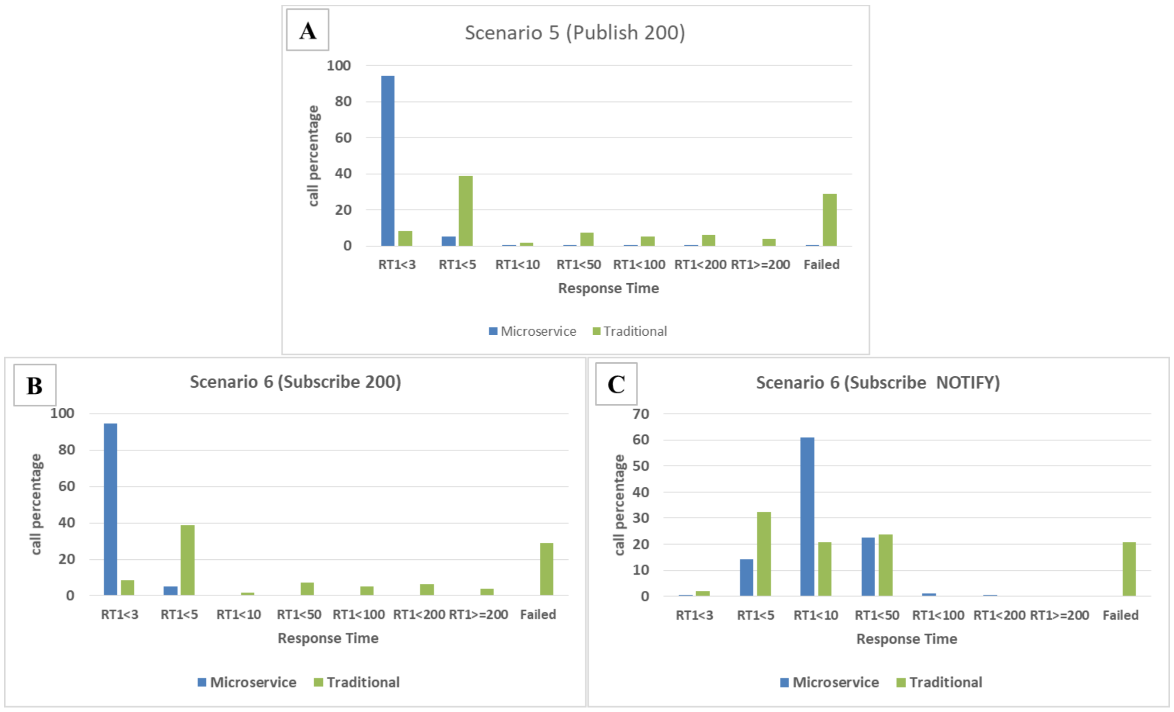


Figure 5.13 Response Time: Group 3

Analysing the publish message acknowledgement performance of the three architectures across the groups shows that whatever the load is; the microservice architecture gives similar results while the performance of traditional architecture degrades with increasing amount of load. Same is the case with the subscribe message acknowledgements. The reason behind the behaviour is that in the microservice architecture, the publication and subscription service are performing a single task of accepting the requests and sending the acknowledgement respectively while the notification task is delegated to the notification service whereas the traditional architecture performs all tasks in the same process. As a result, the time spent in the notification processing does not affect the performance of the publication and subscription microservices while it directly affects the performance of the server in the traditional architecture. Same is the case with the notification processing. For the microservice architecture, the notification service is only performing the notification task. Therefore, its performance is not affected by the processing of publication or subscription requests, nor it affects their performance. Whereas the server in the traditional architecture handles all the requests in the same process which influence the performance of all three requests i.e. publish, subscribe and notify. Therefore in the notification results (figure 5.11C, 5.12C and 5.13C), it is noticed that the performance of traditional architecture degrades with each group where each group puts more stress on the server than the previous group. Further analysis of the notification results reveal that the traditional architecture mostly provides a better response time for few percent calls, but overall performance lags behind significantly in comparison to the microservice architecture. There are few reasons for such behaviour in the traditional architecture that are:

i) The traditional architecture does not interact with external cache or message bus, nor it has any REST overhead, therefore at lower call rates and lighter load, therefore it can process the calls very quickly in comparison to the microservice architecture.

ii) The traditional architecture builds the cache by loading data from the repository into the memory at server start-up. The larger the data in the repository, more the time spent in pulling the data. The process happens for every new instance created. The time spent in building the cache causes in the increased response time and failed calls.

iii) The notification process can take a longer to complete depending on the number of publications or subscriptions per presentity. For instance, in scenario 5 each presentity had 10 subscriptions that required the server to send 10 notifications before servicing next request. Similarly, in scenario 6 each presentity had five publications which means the server had to merge five documents into a single document to send the notification. It implies that more the time is spent on the notification higher the delay in processing other requests.

The first reason results in the better performance of traditional architecture initially at lower traffic and loads, but the other two reasons degrade the performance significantly. In contrast to traditional architecture, the microservice architecture has to communicate with the external cache and the API server, microservices additionally communicates with the message queue. All this communication consumes some time that reflects in the total response time. At lower call rates and lighter server load, the communication with external entities seems counter-productive but as the call rate and load on the server increases, the effect of the communication overhead start to diminish. Instead, it now works in favour of the microservice architecture. Moreover, as described earlier, in microservice architecture each service is independent. Hence, they do not interfere with each other's processing. Therefore, microservices notification response time is not affected by the number of publication or subscriptions being processed at that instant or vice-versa. Additionally, microservices does not use the in-memory caching thus eliminating the step to warm up the cache at server start, hence eliminating the primary cause of delay and failed calls.

The analysis shows that separating the data component from the application logic can improve the performance. Also, by splitting a service into microservices such that each service performs an independent function can further improve the overall performance of the system.

5.4 Summary

This chapter first presented the proposed software architecture for the presence service in the cloud. Then, two scenarios are presented showing the interaction between the software components. After that, a brief introduction of the software tools used during implementation is given and the prototype architecture is defined. Finally, performance measurement results are given, and results are compared and analysed for both, the proposed and the traditional Presence server architecture.

Chapter 6

Conclusion and Future Work

This chapter first highlights the contribution of the thesis and then provide future research directions.

6.1 Summary of the Contributions

Presence service finds its usage in various applications from instant messaging to innovative solutions which include WSN and IoT devices. The proliferation of IoT devices and social media users require the presence service to be highly scalable, elastic and resource efficient. Traditional implementations of presence service are monolithic in nature which limits the scalability, lacks elasticity and are resource inefficient. This thesis proposes an architecture for a presence service in the cloud.

This work identifies the requirements for the presence service in the cloud and reviews most relevant related works. On the evaluation of the related works based on the requirements, it is observed that none of the works fulfils all the requirements.

Additionally, architectural principles for the proposal were identified, and a business model is proposed. The business model consists of following entities: Presence-Service-as-a-Service (PSaaS) provider, Frontend-as-a-Service (FEaaS) provider, Repository-as-a-Service (RaaS) provider, Cache service provider, Presence service provider and connectivity provider. A preliminary architecture was proposed which was later extended to the final architecture based on the microservice architecture. The proposed architecture is based on the earlier proposed business model and fulfils the requirements that were set at early stages. The architecture ensures high elastic scalability by splitting the monolithic presence service into multiple microservices and a repository accessible through the REST APIs. Cache is also used to provide fast data access to the microservices. The REST interfaces, software architecture and operational procedures are also discussed.

A proof of concept prototype is implemented to validate the architecture. The implementation is based on the Opensips presence server. The prototype runs in the Docker containers which are managed by Kubernetes container orchestration engine. The Kubernetes runs on the Infrastructure-as-a-Service platform managed by Openstack. The performance evaluation of the architecture is done to validate the prototype. The results of the assessment show that the proposed architecture provides better elastic scalability, better response time, higher throughput and lower memory utilisation than the traditional architecture.

6.2 Future Work

The current architecture consists of several microservices, external cache component and repository. All the components have statically defined boundaries. The results showed that at lighter load scenarios the proposed architecture consumed higher CPU. It would be interesting to have an algorithm that can provide optimal splitting of the presence server dynamically based on the service requirements and the acceptable trade-offs. For example, a service might find response time more critical than the resources consumed other might be conservative about the resources and response time is not much critical. Based on such requirements the algorithm might be able to decide whether the monolithic presence service is more suitable for the requirements or the microservice architecture or maybe some architecture that lies between these two architectures like the preliminary architecture in the proposal. Another research direction could be to design a framework that works on the optimal splitting algorithm and can build the presence service on demand. Such type of framework might accept the service requirements and runs the optimal splitting algorithm which may produce a template document. The template might contain the details of the split components, and the framework should be able to build the presence service based on the produced template. The architecture presented here is the basic architecture on which various existing presence and database optimization techniques can be applied. As part of future work it would be interesting to see the effect on the various performance metrics after applying state-of-the-art optimization techniques.

Bibliography

[1] M. Day, J. Rosenberg, H. Sugano. A Model for Presence and Instant Messaging (RFC 2778), February 2000

[2] Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I. and Zaharia, M., 2010. A view of cloud computing. *Communications of the ACM*, 53(4), pp.50-58.

[3] Geelan, J., 2009. Twenty one experts define cloud computing. *Cloud Computing Journal*, 4, pp.1-5.

[4] N. Alshuqayran, N. Ali and R. Evans, "A Systematic Mapping Study in Microservice Architecture," *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, Macau, 2016, pp. 44-51.

[5] Rosenberg, Jonathan. "SIMPLE made simple: An overview of the IETF specifications for instant messaging and presence using the session initiation protocol (SIP)." (2013).

[6] Rosenberg, J., H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. "IETF RFC 3261-SIP: Session Initiation Protocol." The Internet Society (2002).

[7] Roach, A. B. "Session Initiation Protocol (SIP)-Specific Event Notification", RFC 3265." (2002).

[8] Niemi, A. SIP extension for event state publication. RFC 3903, 2004.

[9] Rosenberg, J. "A Presence Event Package for the Session Initiation Protocol (SIP)(RFC 3856). IETF." (2004).

[10] Racoh, A. B. "A session initiation protocol (SIP) event notification extension for resource lists." IETF RFC 4662 (2006).

[11] Camarillo, Gonzalo, Adam Roach, and Orit Levin. "Subscriptions to request-contained resource lists in the session initiation protocol (SIP)." (2008).

[12] Sugano, H., S. Fujimoto, G. Klyne, A. Bateman, W. Carr, and J. Peterson. "RFC 3863: presence information data format (PIDF)." Internet Engineering Task Force (2004): 206.

[13] Schulzrinne, H., V. Gurbani, P. Kyzivat, and J. Rosenberg. "RPID: Rich presence extensions to the presence information data format (pidf)(rfc 4480)." IETF (2006).

[14] Schulzrinne, Henning. "CIPID: Contact Information for the Presence Information Data Format." (2006).

[15] Schulzrinne, Henning. "Timed Presence Extensions to the Presence Information Data Format (PIDF) to Indicate Status Information for Past and Future Time Intervals." (2006).

[16] Rosenberg, J. "RFC 4479." A Data Model for Presence, IETF.

[17] Rosenberg, Jonathan. "Presence authorization rules." (2007).

[18] Rosenberg, Jonathan. "An Extensible Markup Language (XML) based format for watcher information." (2004).

[19] Rosenberg, J. "RFC 4825-The Extensible Markup Language Configuration Access Protocol." (2007).

[20] Isomaki, Markus, and Eva Leppanen. "An Extensible Markup Language (XML) Configuration Access Protocol (XCAP) Usage for Manipulating Presence Document Contents." (2007).

[21] Lonnfors, Mikko, Jose Costa-Requena, Eva Leppanen, and Hisham Khartabil. "Functional description of event notification filtering." (2006).

[22] Khartabil, H., E. Leppanen, M. Lonnfors, and J. Costa-Requena. An Extensible Markup Language (XML)-Based Format for Event Notification Filtering. No. RFC 4661. 2006.

[23] Lonnfors, Mikko, Jari Urpalainen, Eva Leppanen, and Hisham Khartabil. "Presence information data format (PIDF) extension for partial presence." (2008).

[24] Lonnfors, Mikko, Jose Costa-Requena, Eva Leppanen, and Hisham Khartabil. "Session initiation protocol (SIP) extension for partial notification of presence information." (2008).

[25] Niemi, A., M. Lonnfors, and E. Leppanen. "RFC 5264." Publication of Partial Presence Information (2008).

[26] Saint-Andre, P. "RFC 3920: Extensible messaging and presence protocol (XMPP)." Instant Messaging and Presence, IETFproposed Standard (2004).

[27] Saint-Andre, P. "RFC 3921: Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence, Oct. 2004." Status: PROPOSED STANDARD.

[28] Day, Mark, Sonu Aggarwal, Gordon Mohr, and Jesse Vincent. Instant messaging/presence protocol requirements. No. RFC 2779. 2000.

[29] Singh, Vishal Kumar, Henning Schulzrinne, Piotr Boni, Boris Elman, and David Kenneson. "Presence aware location-based service for managing mobile communications." In *Consumer Communications and Networking Conference, 2007. CCNC 2007*. 4th IEEE, pp. 514-519. IEEE, 2007.

[30] M. El Barachi, A. Kadiwal, R. Glitho, F. Khendek and R. Dssouli, "An Architecture for the Provision of Context-Aware Emergency Services in the IP Multimedia Subsystem," *VTC Spring 2008 - IEEE Vehicular Technology Conference,* Singapore, 2008, pp. 2784-2788.

[31] Vaquero, Luis M., Luis Rodero-Merino, Juan Caceres, and Maik Lindner. "A break in the clouds: towards a cloud definition." *ACM SIGCOMM Computer Communication* Review 39, no. 1 (2008): 50-55.

[32] Mell, Peter M., and Timothy Grance. "Sp 800-145. the nist definition of cloud computing." (2011).

[33] B. Sotomayor, R. S. Montero, I. M. Llorente and I. Foster, "Virtual Infrastructure Management in Private and Hybrid Clouds," in *IEEE Internet Computing*, vol. 13, no. 5, pp. 14-22, Sept.-Oct. 2009.

[34] Voorsluys, William, James Broberg, and Rajkumar Buyya. "Introduction to cloud computing." Cloud computing: Principles and paradigms (2011): 1-41.

[35] T. Dillon, C. Wu and E. Chang, "Cloud Computing: Issues and Challenges," *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, Perth, WA, 2010, pp. 27-33.

[36] Dragoni, Nicola, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. "Microservices: yesterday, today, and tomorrow." *arXiv preprint arXiv:1606.04036* (2016).

[37] Pautasso, Cesare, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai Josuttis. "Microservices in Practice, Part 1: Reality Check and Service Design." *IEEE Software* 34, no. 1 (2017): 91-98.

[38] J. Thönes, "Microservices," in *IEEE Software, vol. 32, no. 1*, pp. 116-116, Jan.-Feb. 2015.

[39] O. Zimmermann, “Microservices Tenets: Agile Approach to Service Development and Deployment,” *Computer Science—Research and Development*, 2016

[40] Gabbrielli, Maurizio, Saverio Giallorenzo, Claudio Guidi, Jacopo Mauro, and Fabrizio Montesi. "Self-reconfiguring microservices." *In Theory and Practice of Formal Methods, pp. 194-210. Springer International Publishing*, 2016.

[41] Newman, Sam. Building microservices: designing fine-grained systems. " O'Reilly Media, Inc.", 2015.

[42] M. Villamizar et al., "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," *2015 10th Computing Colombian Conference (10CCC),* Bogota, 2015, pp. 583-590.

[43] M. Massoth, R. Acker, N. Buchmann, T. Fugmann, C. Knoell, M. Porzelt, “Ubiquitous Smart Grid Control Solution based on a Next Generation Network as Integration Platform”, *ENERGY 2011, The First International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, pp. 173-178,2011

[44] P. A. Moreno, M. E. Hernando and E. J. Gómez, "Design and Technical Evaluation of an Enhanced Location-Awareness Service Enabler for Spatial Disorientation Management of Elderly With Mild Cognitive Impairment," in *IEEE Journal of Biomedical and Health Informatics,* vol. 19, no. 1, pp. 37-43, Jan. 2015.

[45] M. El Barachi, A. Kadiwal, R. Glitho, F. Khendek, R. Dssouli, "A Presence-Based Architecture for the Integration of the Sensing Capabilities of Wireless Sensor Networks in the IP Multimedia Subsystem," in *IEEE Wireless Communications and Networking Conference*, pp.3116-3121, 2008

[46] W. E. Chen, Y. B. Lin and R. H. Liou, "A weakly consistent scheme for IMS presence service," in *IEEE Transactions on Wireless Communications*, vol. 8, no. 7, pp. 3815-3821, July 2009.

[47] Dongcheul Lee. "Reducing Multimedia Presence Information Traffic between Mobile Applications", *International Journal of Energy, Information and Communications*, Vol. 2, Issue 3, August 2011

[48] K. Peternel, L. Zebec, A. Kos, "Using presence information for an effective collaboration," in *Communication Systems, Networks and Digital Signal Processing*, pp.119-123, July 2008

[49] Wei Quan, Jun Wu, Xiaosu Zhan, Xiaohong Huang and Yan Ma, "Research of presence service testbed on cloud-computing environment," *2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)*, Beijing, 2010, pp. 865-869.

[50] F. Belqasmi, C. Fu, M. Shashvelayati, H. Khlifi, R. Glitho, "A Peer to Peer Architecture for Enabling a Universal Presence Service," *2010 Fourth International Conference on Next Generation Mobile Applications, Services and Technologies,* Amman, pp. 90-95, 2010.

[51] A. Acharya et al., "Programmable presence virtualization for next-generation context-based applications," *2009 IEEE International Conference on Pervasive Computing and Communications,* Galveston, TX, 2009, pp. 1-10.

[52] Richardson, Leonard, and Sam Ruby. RESTful web services. " O'Reilly Media, Inc.", 2008.

[53] Newcomer, Eric. Understanding Web Services: XML, Wsdl, Soap, and UDDI. Addison-Wesley Professional, 2002.

[54] Johner, Heinz, Larry Brown, Franz-Stefan Hinner, Wolfgang Reis, and Johan Westman. Understanding ldap. Vol. 6. IBM, 1998.

[55] S. S. Chauhan, S. Yangui, R. H. Glitho and C. Wette, "A case study for a presence service in the cloud," *2016 7th International Conference on the Network of the Future (NOF)*, Buzios, 2016, pp. 1-7.

[56] "Opensips." [Online]. Available: http://www.opensips.org/ [Accessed: 01-Aug-2017]

[57] "Redis." [Online]. Available: https://redis.io/[Accessed: 01-Aug-2017]

[58] R. Hecht and S. Jablonski, "NoSQL evaluation: A use case oriented survey," *2011 International Conference on Cloud and Service Computing, Hong Kong*, 2011, pp. 336-341.

[59] “MySQL Database.” [Online]. Available: https://www.mysql.com/ [Accessed: 01-Aug-2017]

[60] “NATS.” [Online]. Available: https://nats.io/ [Accessed: 01-Aug-2017]

[61] “Tomcat Servlet Container.” [Online]. Available: https://tomcat.apache.org/ [Accessed: 01-Aug-2017]

[62] “SIPp traffic generator.” [Online]. Available: http://sipp.sourceforge.net/ [Accessed: 01-Aug-2017]

[63] “Kubernetes.” [Online]. Available: https://kubernetes.io/ [Accessed: 01-Aug-2017]

[64] “Docker.” [Online]. Available: https://www.docker.com/ [Accessed: 01-Aug-2017]

[65] “Openstack.” [Online]. Available: https://www.openstack.org/ [Accessed: 01-Aug-2017]

[66] “cJSON- JSON parser in ANSI C.” [Online]. Available: https://github.com/DaveGamble/cJSON [Accessed: 01-Aug-2017]

[67] “Hiredis- minimalist C client for redis.” [Online]. Available: https://github.com/redis/hiredis [Accessed: 01-Aug-2017]

[68] “Heapster.” [Online]. Available: https://github.com/kubernetes/heapster [Accessed: 01-Aug-2017]

[69] “InfluxDB.” [Online]. Available: https://github.com/influxdata/influxdb [Accessed: 01-Aug-2017]

[70] “Grafana.” [Online]. Available: https://github.com/grafana/grafana [Accessed: 01-Aug-2017]

[71] “CoreOS.” [Online]. Available: https://coreos.com/ [Accessed: 01-Aug-2017]