AN ACTOR MODEL BASED PLATFORM FOR DEVELOPING CONTEXT-AWARE APPLICATIONS

ORKUT KARAÇALIK

DECEMBER 2018

AN ACTOR MODEL BASED PLATFORM FOR DEVELOPING CONTEXT-AWARE APPLICATIONS

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF

NATURAL AND APPLIED SCIENCES OF

IZMIR UNIVERSITY OF ECONOMICS

BY

ORKUT KARAÇALIK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

IN THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

DECEMBER 2018

M.S. THESIS EXAMINATION RESULT FORM

Approval of the Graduate School of Natural and Applied Sciences

[illegible]

Prof. Dr. Abbas Kenan Çiftçi
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

[illegible]

Assoc. Prof. Dr. Süleyman Kondakçı
Head of Department

We have read the thesis entitled **"An Actor Model based Platform for Developing Context-Aware Applications"** completed by **ORKUT KARAÇALIK** under supervision of **Asst. Prof. Dr. Ufuk Çelikkan** and **Asst. Prof. Dr. Kaan Kurtel** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]	[illegible]
Asst. Prof. Dr. Kaan Kurtel Co-Supervisor	Asst. Prof. Dr. Ufuk Çelikkan Supervisor

Examining Committee Members Date: 24/12/2018

Asst. Prof. Dr. Ufuk Çelikkan Dept. of Software Engineering, IUE	[illegible]
Prof. Dr. Cem Evrendilek Dept. of Computer Engineering, IUE	[illegible]
Assoc. Prof. Dr. Murat Erten Dept. of Computer Engineering, IZTECH	[illegible]
Assoc. Prof. Dr. Adil Alpkoçak Dept. of Computer Engineering, DEU	[illegible]
Asst. Prof. Dr. Kaan Kurtel Dept. of Software Engineering, IUE	[illegible]

ABSTRACT

AN ACTOR MODEL BASED PLATFORM FOR DEVELOPING CONTEXT-AWARE APPLICATIONS

ORKUT KARAÇALIK
M.S. in Computer Engineering
Graduate School of Natural and Applied Sciences
Supervisor: Asst. Prof. Dr. Ufuk Çelikkan
Co-Supervisor: Asst. Prof. Dr. Kaan Kurtel
December 2018

Applications using Information and Communication Technologies are collecting and processing a diverse range of data using networks of machines connected to each other through communication networks. This phenomenon is captured in the term Internet of Things. In an open, dynamic and continuously changing environment, generated data must be interpreted by the applications on a contextual basis. A piece of information is considered context data only if it is interpreted; otherwise, it is simply information belonging to an environment. The goal of this thesis is to present the design and implementation of an infrastructure platform to facilitate application developers' rapid and easy development of context-aware applications for various domains. The platform is inspired from an operating system and modeled using a layered architecture. The platform relieves the application developers from doing tasks such as data collection, storage and management. Actor Model is chosen as the computational model to implement platform services, and a viable alternative to meet the needs of demanding modern systems. It defines how the system's components should behave and interact with each other. The platform offers its functions as services implemented using Actors. The core services are Security and Privacy, Rule, Data Management and, Alarm and Notification. The platform provides two interfaces to applications, and data providers to communicate with the platform. Applications can use either a RESTful interface or an application programming library when interacting with the platform. Authentication is provided via JSON Web Tokens and for authorization, a simple Role based access control is used. The platform is furnished with a web interface for administration tasks such as registering users, applications and data providers.

Keywords: context-aware computing, software platform, middleware, actor model, REST interface.

ÖZ

DURUM FARKINDA UYGULAMALAR GELİŞTİRMEK İÇİN AKTÖR MODEL TABANLI YAZILIM PLATFORMU GERÇEKLEŞTİRİLMESİ

ORKUT KARAÇALIK
Bilgisayar Mühendisliği, Yüksek Lisans
Fen Bilimleri Enstitüsü
Tez Danışmanı: Dr. Öğr. Üyesi Ufuk Çelikkan
İkinci Tez Danışmanı: Dr. Öğr. Üyesi Kaan Kurtel
Aralık 2018

Bilgi ve iletişim teknolojilerini kullanan uygulamalar, haberleşme ağları ile birbirlerine bağlı makineler yardımıyla farklı türden verileri toplayıp bunları işlemektedir. Bu olgu, Nesnelerin İnterneti olarak da adlandırılmaktadır. Üretilen veriler, sürekli değisen ve açık çevresel şartlar içerisinde durumsal temeldeki uygulamalar tarafından işlenmektedir. Bir bilgi parçası, eğer yorumlanırsa durumsal veri olarak kabul edilir, aksi takdirde sadece çevre hakkında bir veridir. Bu tezin amacı uygulama geliştiricilerin hızlı ve kolay bir şekilde durum farkında uygulamalar yapmaları için bir altyapı platformunun tasarımını ve gerçekleştirilmesini sağlamaktır. Platform, işletim sisteminden esinlenilerek katmanlı mimari olarak tasarlanmıştır. Platform uygulama geliştiricilere veri toplanması, saklanması ve yönetimi gibi hizmetler sunarak kolaylık sağlamaktadır. Platformun servislerini gerçekleştirmek üzere Aktör modeli işlemsel model olarak seçilmiştir. Aktör modeli, sistem bileşenlerinin nasıl davranmaları ve birbirleri arasında etkileşime girmeleri gerektiğini net bir şekilde tanımlamakta, ve modern bir sistemin ihtiyaçlarını karşılayabilecek düzeyde imkanlar sunmaktadır. Platform, işlevlerini aktör olarak tanımlanan servisleri sayesinde sağlar. Temel olarak, Güvenlik ve Gizlilik, Kural, Veri Yönetimi, Uyarı ve Bildirim servislerinden oluşur. Platform, uygulamalara ve veri sağlayıcalara platform ile iletişime geçebilmeleri için iki adet arayüz sağlar. Uygulamalar platform ile etkileşime girmek için RESTful arayüzü veya hazır kütüphaneyi kullanabilir. Kimlik denetimi JSON Web Tokens aracılığı ile sağlanır ve yetkilendirme rol tabanlı erişim kontrolü prensipleriyle sağlanır. Platform, kullanıcı, veri sağlayıcı ve uygulamaların kaydı gibi yönetimsel işler için kullanılacak web arayüzü ile birlikte sunulmaktadır.

Anahtar Kelimeler: durum farkında bilişim, yazılım platformu, aktör modeli, ara katman yazılımı, REST arayüzü.

ACKNOWLEDGEMENT

I wish to acknowledge the help provided by Asst. Prof. Dr. Ufuk Çelikkan. He provided me with valuable advice, and support while writing this thesis. He guided me about how to approach problems, and finding ways to resolve them.

I also wish to acknowledge Asst. Prof. Dr. Kaan Kurtel who shared his experience and support.

This work is supported by Scientific and Technological Research Council of Turkey (TÜBİTAK) Grant No: 114E938. Therefore, I would like to thank TÜBİTAK.

Lastly, I am thankful to my family who always supported me. Their love and help encouraged me to do new things in my life. Their presence, whenever I need, always made me feel strong against any challenge. They helped me to become the person who I am. Thus, I am always grateful to them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

List of Abbreviations

AJAX	Asynchronous JavaScript and XML
ANS	Alarm and Notification Service
API	Application Programming Interface
BNF	Backus-Naur Form
CPU	Central Processing Unit
DMS	Data Management Service
FIFO	First in First out
GPS	Global Positioning System
GUI	Graphical User Interface
HTML	Hyper-Text Markup Language
HTTP	Hypertext Transfer Protocol
ICS	Interoperability and Communication Service
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
JWT	JSON Web Token
MVC	Model View Controller
OS	Operationg System
PCAD	Platform for Context-Aware Application Development
POSIX	Portable Operating System Interface for Unix
RBAC	Role Based Access Control
REST	Representational State Transfer
RFID	Radio Frequency Identification
RPS	Reporting Service
RS	Rule Service
SOAP	Simple Object Access Protocol
SPS	Security and Privacy Service
SQL	Structured Query Language

STM	Software Transactional Memory
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URI	Universal Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language

Chapter 1

Introduction

The 21st century is named as Knowledge, Information or Digital age. The most important enabler of digital age is the proliferation of computing devices and their applications. Applications using information and communication technologies collect and process a diverse range of data using machines connected through communication networks. This phenomenon is captured in the term the Internet of Things. The Internet of Things is a complex interconnection of heterogeneous devices that include sensors, cameras, micro chips and RFID based products, which generate large amount of data obtained from various domains. Internet of Things brings a new era in computation, which has a potential to transform many fields, such as agriculture, shopping, industry, transportation etc. According to a survey in [1], the total number of devices in the world exceeded world population in year 2011, and is expected to reach 24 billion by 2020. Sensors play an important role among those devices, since they sense our environment and supply data about it such as temperature, humidity, pollution, and congestion. A great number of applications such as the one that makes a self-driving car possible are developed to make use of the sensor data. In the future, the impact of Internet of Things-related applications will be greater in certain areas than others. According to [2] 41% of the applications will be in health care services, and manufacturing will take 33% share of all applications by 2025.

Platforms and architectures must be designed in anticipation of an increase in the number of devices, and must accommodate varying demands of users and applications in different contexts. In order for a platform to be successful, it is imperative that it recognizes the context in which users and applications operate, and enable service customization for a particular user. The creation of smart applications and environments then becomes possible through context-aware computing, which encompasses acquisition, analysis, and interpretation of relevant context information, and responds to contextual changes. Examples of context information are temperature, humidity, traffic congestion, road conditions, sea pollution, and river level. Such context information can be used alone or in combination within context-aware applications to provide custom services in various domains, such as transportation, health and medical systems, tracking and control of environment, energy, agriculture, industry, sport events, and tourism. The effective use of this context information requires its efficient and effective acquisition, storage, processing and reasoning. In this way, productivity, economic output and quality of life can be increased.

The primary goal of this thesis is to present a novel, service-based application development platform called PCAD (Platform for Context Aware Development) based on the notion of context-awareness. In an open, dynamic and continuously changing environment, context data must be acquired, managed and eventually offered to applications which interpret data according to the situation. The PCAD platform described in this thesis simplifies the development of context-aware applications by relieving applications from complex data management issues by separating context acquisition from application code and handling many context data management issues on behalf of the applications. The ease of context acquisition and context use by the applications creates a positive usability experience. The platform basically follows a middleware approach, which draws on techniques taken from the operating system design. Scalability and reliability are indispensable features of system. The system is likely to operate under high load, since numerous data providers and applications are expected to work with the platform. The platform must be responsive and stable, no matter how many data providers and applications use the platform. The fundamental design force behind the implementation is that it is agile, robust, and capable of reacting to new requirements without a need for fundamental and substantial changes. An easy-to-configure, simple to understand, scalable and reliable system is the primary goal.

The thesis is logically organized in two parts. The first part which consists of Chapters 1 through 4 explains motivation, design and architectural aspects of the platform. The second part of the thesis which consists of Chapters 5, 6 and 7 explains implementation details, testing of the PCAD platform, and conclusion. Chapter 2 discusses context aware computing, and surveys other context-aware infrastructure platforms. Chapter 3 gives a brief introduction of Actor model principles, and its implementation AKKA system. Actor model and AKKA lays the theoretical foundation of the platform. Chapter 4 presents the architecture used in the design and building of the platform. Core features of the platform is also discussed here. The platform decouples context data generation and storage from its consumption. It is based on a service oriented paradigm.

Implementation details of the PCAD platform is explained in Chapter 5. The implementation includes a core to manage services, request/response mechanisms to interact with the platform, database operations and provides a rule language and a simple rule engine to ease data management for application developers. It includes API libraries for applications to use when they bind to the platform. Test and performance results are discussed in Chapter 6. The thesis ends in Chapter 7 with the conclusion and future work. The Appendix gives programming and user interfaces, several sample programs demonstrating how to use the system and installation instructions.

A significant portion of our TÜBİTAK project is embodied in this thesis. The thesis lays down the theoretical foundations of the platform, and discusses architectural and implementation decisions to make the platform a reality. The resulting implementation has met the requirements set forth in the project proposal.

Chapter 2

Related Work

Context-aware systems are adaptive systems that can react to changes in their environment without user interventions. In its broadest sense, the term context refers to any information used to characterize the situation of a person, data, or object relevant to the interaction between a user and an application [3]. Various architectures are suggested in the literature to use in the implementation of context-aware systems. According to a study given in [4], context-aware system architectures are classified using one of the following two criteria:

1. how context is acquired, and
2. how context and components are managed

2.1 Context Acquisition

Context acquisition is one of the most important aspects of Context-aware architectures. There are two main approaches used in context acquisition:

1. Direct sensor access,
2. Middleware infrastructure.

In architectures that employ direct sensor access, applications and sensors are tightly coupled to each other. Since applications communicate directly with the sensor in such architectures, they need to know how to interact with the sensor, thus sensor-specific software becomes part of the application. Despite being relatively easy and fast to implement, such architectures are rather inflexible and difficult to maintain, therefore leading to higher development costs in the long run. They are also unsuitable for distributed systems, as the coordination of concurrent sensor access is difficult due to a lack of a component responsible for the coordination.

Middleware infrastructures separate context acquisition from context use. Applications and data providers are only loosely coupled with each other. Such decoupling is achieved by hiding lower layer transaction details from the upper layers using an in-between layer. Therefore, context acquisition, storage, management and user business logic processes are separated from each other. Middleware based architectures exhibit well-known advantages of layered architectures. By hiding the lower layer context acquisition functionality from applications, middleware architectures promote reusability and extensibility. Context acquisition functions can be reused by multiple applications, and a change in context acquisition software does not affect the applications (Baldauf et al., 2007) [4]. Extending a middleware architecture by incorporating a *context server* allows distributed operations as the context server coordinates simultaneous sensor access by the applications.

A layered infrastructure for contex-aware computing typically consists of 5 layers, as it was conceptually proposed by Ailisto [5] and illustrated in Figure 2.1. These 5 layers are physical, data acquisition, inference, storage/management and application. The use of a layered architecture makes it easier to extend the system and reuse modules. For those familiar with Model-view-Controller paradigm, it is not difficult to easily map these 5 layers to the MVC layers. An MVC-based Context-aware architecture has been proposed in [6].

2.2 Context Management

The other criterion used in classifying context-aware architectures is based on context and component management. There exists three major approaches in managing context [7]:

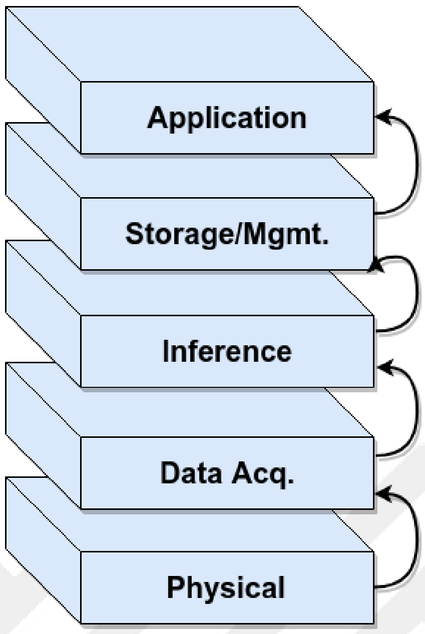


Figure 2.1: Conceptual Context-Aware Architecture Layers

1. Widgets
2. Networked Services
3. Blackboard

Widgets: Widgets are software components that communicate with each other by sending messages. Widgets encapsulate details about the component, therefore, applications use the data sent by the widgets without knowing the details. Due to encapsulation capability, widgets allow substitution of one widget with another as long as it handles the same kind of sensor data. A widget manager is responsible for managing the widgets. A slight variation of this approach is used in networked services architectures. Widgets are prone to failures and not robust since they are tightly coupled; however they boost efficiency.

Networked Services: The Networked services architecture suggests that components are on the network and providing services. There is no global widget manager-like scheme; instead, a form of global discovery mechanism is used to locate the components on the network. The Widgets model is more efficient than networked services, but it is not as configurable as networked services.

Blackboard: The blackboard architecture is built on a publish-subscribe system where subscription works based on events. When a registered event happens, interested subscribers are alerted. Components send sensor data to registered components via a filtering mechanism after the receiving components register themselves with the sensor data that they are interested in.

It is difficult to make a direct comparison between these architectures. Each architecture exhibits certain advantages and disadvantages in regard to four points: efficiency, configurability, robustness and simplicity. The Efficiency refers to the speed and space usage; the configurability refers to the ease of adding new devices on the system; the robustness refers to proper handling of the errors and peaceful termination; and the simplicity refers to how easy the system is to understand.

The Networked services model is easy to configure and offers robustness as there is no single point of failure. However, it is inefficient when compared to the widgets architectures, as there are a number of network components. The Blackboard architecture makes it much easier to add new context data providers to the system, hence it has a high degree of configurability. However, since the subscription and filtering processes add an extra step, the data transfer time is increased, which degrades the efficiency. It is also very easy to understand the blackboard model [4].

Based on the architecture discussion above, the implementation of the platform discussed in this thesis uses a combination of the blackboard and middleware models for context management and context acquisition respectively. The middleware approach decouples context acquisition from context use and allow reusability, therefore allows more freedom in developing individual components. The Blackboard model provides flexibility, and provides a plug-and-play behavior in order to incorporate more features for further improvements. Context providers and context consumers can be easily added or removed from the system. The blackboard model also uses an event-based notification. Both middleware and blackboard models are advantageous in terms of configurability, robustness and simplicity. Both models lend themselves to loosely coupled structures, so it is easier to understand and modify them, since internal components are weakly attached to each other.

Since middleware model is very flexible, we can take advantage of cloud computing in the implementation of it. Cloud provides a virtual computation environment for different purposes, and its main advantage is its universality and accessibility [2]. Cloud computing provides a reliable, consistent, scalable, cost effective and collaborative environment using a massive pool of computers. A middleware platform running on cloud allows clients to communicate with the platform to receive data from different contexts or send data with lower cost and higher mobility.

Various applications have been developed based on the architectures described above. Applications include home, smart classroom [8], hospital [9], tourism [10], decision support systems, communication systems [11], laundry [12, 13, 14, 15]. A comprehensive survey of practices and architectures in this area is given in [11]. The following section gives a review of several middleware platforms for context-aware computing.

2.3 Existing Systems

This section takes a look at several systems similar to the PCAD platform but take different perspectives to issues. Each one takes a different approach to accomplish its goals, but they are all considered context-aware computing applications. Systems are briefly summarized for a general overview of their functionality. In total, 19 different projects and 2 industrial implementation are mentioned in this section.

Context Toolkit [3] provides facility to develop context-aware applications. It has a centralized architecture to fetch data from context sources, then present them to applications. Any type of context sources can be utilized to generate raw data. It uses interpreters to process raw data. It permits data access through security and privacy transactions. Also, it stores data in its history feature, and data is made available to applications via services. It does not use the middleware approach.

CARISMA [16] is a context-aware middleware that focuses on handling policy based context changes in the presence of conflicting policies. The system targets mobile applications and resolves conflicts at execution time. Unlike other systems mentioned in this section, context acquisition, storage and modeling are not the primary concerns in *CARISMA*.

Gaia [17] is basically an operating system inspired middleware system for context-aware applications, so that it is referred as Gaia Operating System. Via its context manager, it can work with various physical sensors to gather context related data. Sensors connect to Gaia and provide data based on their configuration with Gaia OS. Gaia OS gets input, processes and stores data within the system, and 3rd party applications communicate with Gaia OS to receive context related information. It has functions commonly found in an OS, such as program execution, input-output operations, file-system manipulation, communications, error detection and resource allocation. Gaia provides a set of basic services used by the applications. *Event Manager Service* provides reliable event-based message transfer via channels between decoupled information suppliers and consumers. *Context File System* stores context related configurations and rules for providing meaningful information to applications. *Presence Service* keeps track of existence of entities; for example, when a sensor stops working, it informs the related application that a sensor stopped. *Context Service* allows applications to query current context, and it also consists of a rule based mechanism to provide information.

CASS [18] is a middleware platform that binds context sources to applications. It is an event-based and extensible platform which supports a wide range of context sources, context history records, context interpretation, context-awareness rules and behaviors separation. The middleware consists of four major components. *Sensor Listener* receives data from context sources. *Rule Engine* processes rules and manages behaviors. *Context Retriever* fetches context based data from storage. *Interpreter* abstracts away low level data into high level, for instance by converting a raw value to Celsius or Fahrenheit. It has an SQL database to store context data and a knowledge base containing rules which are used by its inference engine to solve problems. For example, if the goal is to decide whether to perform indoor activities or not, the rule in the knowledge base would use context data for brightness, temperature and rain to reach the goal. Basically, the rule takes conditions or parameters, and finds a match in the knowledge base. The knowledge base is established by users of the platform, and can be changed based on user needs.

SOCAM [19] is a distributed middleware that works as a client-server model.

The context provider collects data, and context interpreter takes and converts it into high level information. Data is stored in context databases, and context information is provided to context-aware mobile services and service-locating-service for third party applications to use.

Another distributed middleware is *COSMOS* [20] which consists of three layers responsible for context collecting, processing and adaptation. The context collector acquires sensor data, the context processor refines raw data, and the context adapter provides access to the previously processed context data. In this distributed architecture, the context information nodes are organized as a hierarchical structure that operate as individual modules within the system. The system supports fetching data from various context sources.

Context Broker Architecture (*CoBrA*) [21] is a centralized middleware architecture that can work with many context brokers to get context data. CoBrA processes and extracts raw data and provides them to consumers. It has a policy management module that controls data access.

CASP is a context gathering framework for context aware mobile solutions [22]. It consists of several components that perform context sensing, modeling, association, storage and retrieval tasks. It provides client-side and server-side sensory API libraries so that sensor and application software developers amortize their development efforts.

Octopus [23] is also a middleware that focuses on home/office domain based context-aware computing. It aims developers to create sensor or application software with minimal knowledge. It has an extensible architecture to add new modules.

Hydrogen [24] is a layered architecture with three layers: adaptation, management and application. The adaptation layer is assigned the task of collecting sensor data, and the application layer conveys the data to applications. The middle layer, management, is equipped with a context server for the purpose of binding the adaptation and application layers to one another.

Hydra [25] is another middleware based system that focuses on IoT, wireless sensors, and devices. It provides a context aware framework that consists of data acquisition and context manager components. In addition to its low level data acquisition and storage services, it has a rule engine that performs context reasoning and access control mechanisms.

Ubiware [26] is a decentralized system that works with heterogeneous sources. One of its goals is to provide scalability and reliable communication among context agents. It also features context discovery and sophisticated configurations.

TinyRest [27] is used in homes and offices and uses IoT infrastructure. It utilizes wireless sensor networks to access various actuators and sensors in available domains.

Context-Aware Control Platform [28] is a middleware based cloud integrated IoT framework that integrates context-aware computing and cloud technologies. It is a three-tier architecture that consists of context sensing module, context information management, and context aware services management and user interface tiers. The *Context Sensing* module is assigned to handle sensor related tasks such as networking, and interacting with environment. The *Context Information Management* manages tasks such as context mapping, reasoning and storing. The *Context-aware Services Management* provides services to users via a user interface. Since the platform is a cloud integrated IoT framework, access mechanisms are introduced to map the components of the middleware into different cloud levels of abstraction appropriate for the type of service they perform. The *Cloud Access Model* consists of four layers, namely, sensor, infrastructure, platform and user, and each layer has different tasks. The sensor layer introduces an interface for accessing hardware through the sensor network access interface, and it works as software as a service(SaaS). The Infrastructure layer offers cloud facilities such as storage, deployment, and it works as infrastructure as a service(IaaS). The platform layer is where context-aware middleware software components are placed to provide library tools for application deployment. It works as platform as a service(PaaS). The user layer contains the management software for the platform and applications and also interfaces for the humans, and it works software as a service(SaaS).

There exists other platforms developed to address the needs of different domains such as health and navigation. *CUPUS* is a middleware platform for sensors that focuses on Cloud-based publish-subscribe pattern. One of its components -‘Mobile

Broker'- runs on a mobile device and acts as a gateway between the cloud and the sensing device. It acquires, filters and transmits data to subscribers. *CUPUS* offers other functionalities such as location management of sensors, exploring sensors, sensor data management and quality of services management [29].

The system described in [30] is a web based platform intended for people with less technical skills to configure their mobile applications based on context-data. The context data is made of location, time and date information. Platform has a rule engine to match context data in the rule against the one configured in the platform and sends the specified data to the mobile application if a match occurs. The platform lets users to receive data if their location can be determined via GPS; therefore, its applicability to indoors is very limited.

Freeband [31] project uses web services to rapidly develop mobile applications. It uses smart-phone location and speech inputs as context data. The context data is matched to the related context services and then the data is provided for mobile applications. The platform is modeled for navigation purposes, for example, it presents sightseeing information to users based on location. The *Freeband* study does not allow using other types of sensor data, it only uses location data like the previous web based platform and exhibits similar problems.

ERMHAN's goal is to provide a system for health-care services using patient data [32]. Patients have their own sensors for measuring health-related parameters such as temperature and heart rate. Those context-data are sent to a centralized service, then it is presented to health-care workers via a graphical user interface (GUI). There are two context managers defined in *ERMHAN*, the patient context manager and the central context manager. The first one handles data acquisition and data sending tasks, the second one works as a middleware and distributes data to related services. Moreover, the platform is capable of notifying of certain conditions and it also has a web interface for health-care workers, thus patients can be remotely monitored. For an example scenario, a patient is monitored continuously via wearable sensors, and the doctors or other health-care workers can see the patient's status using the platform's front-end interface. This platform offers an effective solution for special problems but it is not general enough.

The majority of the systems discussed so far are experimental and academic in nature. There exists industrial products in order to bring IoT to public use.

We are going to briefly review two of them, *Weeronline*[1] and *Waylay*[2]. *Scala* [33] programming language and *AKKA* model is used in developing and managing these two platforms. For this reason, these platforms exhibit similarities to PCAD. *Weeronline* weather forecast platform collects data from multiple providers all over the world. It presents weather related information such as temperature and wind speed, and it has a points system(1-10) for outside activities, e.g "To what extent the weather is good for playing football?" would be answered using a point scale, in which ten means very good. Both Weeronline and Waylay use multiple client and server technologies to bring services to life. Between the two *Waylay*, is a more general purpose platform that targets IoT applications in various fields. Waylay is a platform as a service (PaaS) that binds hardware level applications and end-user applications. The platform consists of sensor and application level interfaces for communication. Collected data from sensors is processed and tailored for the use of high-end applications. It works as a middleware between sensors and applications.

[1]`https://www.weeronline.nl/`
[2]`https://www.waylay.io/`

Chapter 3

Actor Model and AKKA

Parallel programs consist of a collection of interacting computational processes which execute simultaneously, affecting each-other's work by exchanging information. Several libraries exist to provide the necessary abstractions in addition to operating system resources to simplify concurrent programming. The two fundamental attributes of such a library are first, that it provides a computation unit that could be executed in parallel (concurrency primitives) and second, a means of communication between these computation units. Traditional implementations of these libraries use threads as the smallest execution unit and shared memory, message passing or signals are used for communication. A common criticism of using threads is that they are heavyweight, not scalable and lead to client code being error prone. The Actor model was introduced by Carl Hewitt, Peter Bishop, and Richard Steiger in 1973 [34] in response to these concerns and at the same time to meet the needs of concurrent computation in distributed systems. It defines some general rules for how the system components should behave and interact with each other, avoiding the issues caused by threads and locks found in concurrent computation. In the actor model, the smallest executable unit is an actor - a concurrency primitive- that does not share any resources with other actors. Communication among actors is implemented sending each other messages.

3.1 Actor

An actor is a computational entity that encapsulates a state and a thread of control that manipulates this state independent of other actors. Actors do not share state. Therefore, the only way that an actor can have an impact on another actor is by sending a message. Actors have unique addresses and communicate with other actors by sending a message to the address of another actor in the system. Each actor has a mailbox (ordered message queue) that buffers messages before it processes. Sent messages are not necessarily received in the order they are sent, and multiple messages cannot be handled at the same time. Messages are processed when received and preserved until they are processed. Message delivery is not guaranteed, and messages are delivered at most once. In other words, when actor sends a message to another actor, the message may not arrive to target, and it arrives only once. However, various implementations of Actor model extended message delivery property by offering other features such as at-least-once delivery instead of at-most-once. This is because a message must be delivered at least once since lost messages in real-world industrial applications may cause incorrect operations, as messages can carry critical content. Therefore, some Actor implementations provided useful additions to the theoretical model.

The Actor Model is a pure *Asynchronous Message Passing* model. When an actor receives a message from another actor, it performs one of the following actions:

- alter its current state, possibly changing its future behavior,
- send messages to other actors asynchronously,
- create a new actor with a specified behavior,
- migrate to another computing host.

Actors can be distributed among other hosts. This makes actor based computation a very good fit for distributed programming. Since inter-actor communication is based on message passing rather than shared memory, actor based systems are highly configurable. Figure 3.1 shows the structure of an Actor and message passing among actors. Actors can be used to model functional, procedural, or object

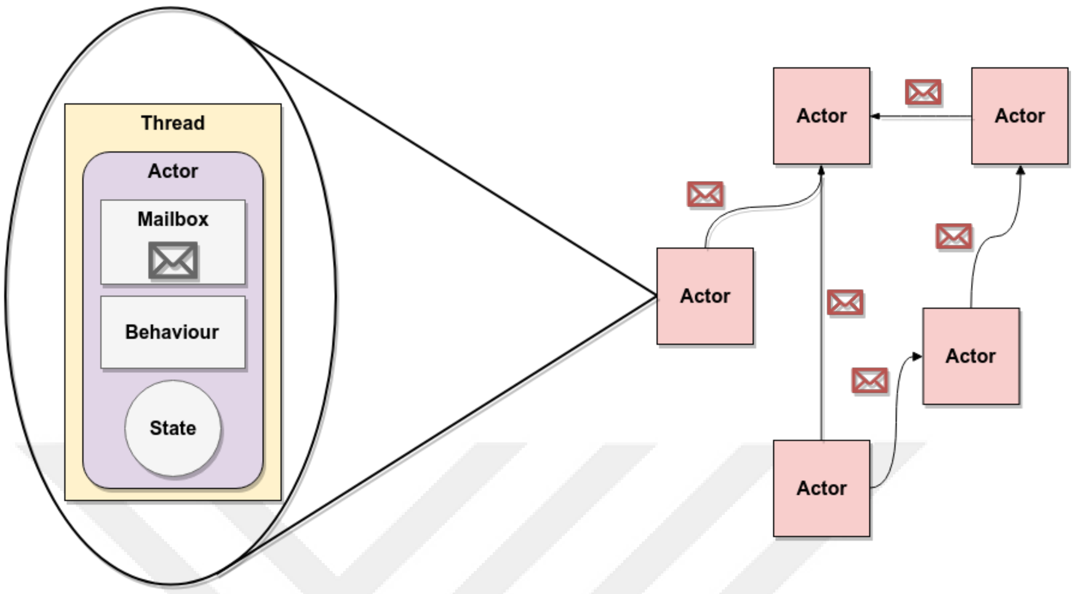


Figure 3.1: Actor Structure and Message Passing

oriented systems. In an Object-Oriented paradigm objects are base elements for computation where method calls are synchronous and calls on an instance triggers a computation which results in a state change of the instance. This is in contrast to the Actor model where actors are base elements and computation by actors are triggered by asynchronous message processing. Each actor is a unit that is assigned for a specific computation.

In order to explain the basics of the Actor model, we illustrate its principles on a calculator example. This example portrays essential features of the model, and tries to paint the whole picture with concrete actions. A calculator is modeled through a *CalculatorActor* actor which performs one of the four basic arithmetic operations when it receives a message. When the *CalculatorActor* receives an *add* message, it creates an *AdderActor* to calculate the sum of the given numbers and when it receives a *divide* message, it creates an *DividerActor* to divide the given numbers. This means a new actor is created by another actor and the newly created actor is assigned to perform a certain computation such as addition and multiplication. Creating actors this way forms a hierarchical actor structure in which each single actor is responsible for completing a simple task [35].

In Figure 3.2, multiple *Users* sends messages to *CalculatorActor*, which creates a child actor for each calculation request, and assigns the task to the related child actor to perform four basic arithmetic operations. *CalculatorActor* is the parent, and *SubtractorActor, AdderActor, MultiplierActor, DividerActor* are the child actors. The Actor model restricts the number of child actors, allowing creation of a fixed number of them based on configuration. Parent actors have supervision over child actors that are present in a hierarchy. Advantage of having an actor hierarchy is that tasks are broken up into little parts and assigned to child actors. New tasks can be added easily this way. If we want to add logical or bit wise operations, it is sufficient to add a new type of child actor to extend main actor's - i.e. *CalculatorActor* - functions. Therefore, the system can be extended elegantly. On top of the Actor hierarchy sits the Actor system itself as the root. The receiver of a message can send a brand new message or forward the received message to other actors. In our example, *CalculatorActor* forwards the messages to child actors, then gets the results back, and lastly, it transmits the final result to *User*. One point to note is that it is not certain which result will be conveyed first to the user in the presence of multiple messages Since message passing is not a synchronous process, this is an expected behavior. Actors can have a state that they are able to modify. Returning to our example, *CalculatorActor* can have a state which may hold an intermediate result. It performs operations using the intermediate result, and when the *AllClear* message is received, it sends the result to the user, and resets it to zero.

3.2 Actor Model Properties

There are four key semantic properties of the Actor model [36].

- **Encapsulation:** Encapsulation suggests that multiple actors do not share state. Therefore, the race condition, in which two or more entities, such as threads, try to change the same state is impossible in Actor model. Messages are processed in an atomic manner, and the queued messages are handled sequentially. When there is an attempt to mutate the state, firstly the current message is taken into account.
- **Atomicity:** Atomicity ensures that a method is executed atomically, in response to a message.
- **Fairness:** Fairness means each actor is alive if any computation needs to be

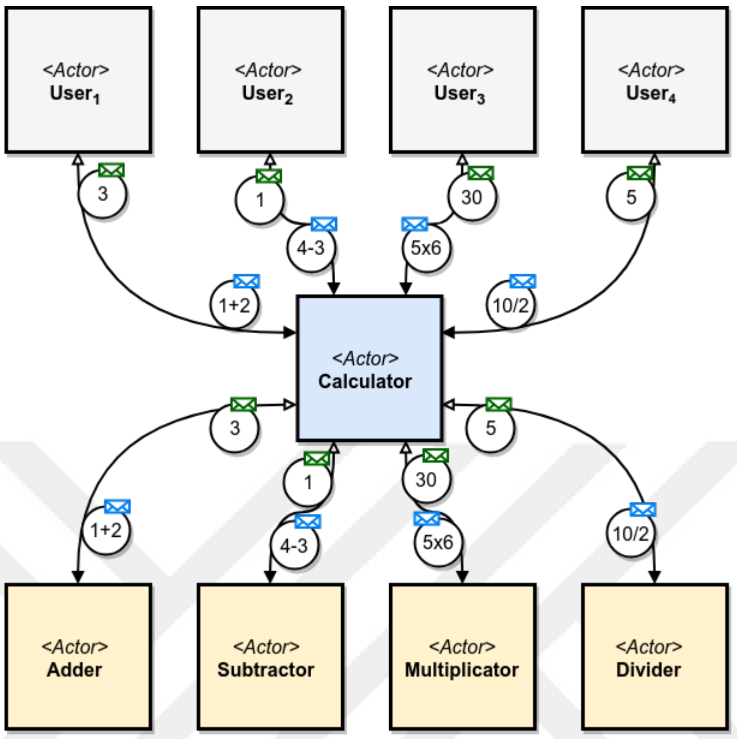


Figure 3.2: Actor Messaging

completed and each sent message will arrive at the receiver actor. This means that messages will be handled, but it is indeterminate when it finishes i.e. the message could be processed at any time. At most once delivery feature that is mentioned before is related to this model property.

One exception occurs when the target actor is terminated for good, or taken out of the system, in which case the message never arrives at its destination.

- **Location Transparency:** location transparency implies that the location of an actor does not cause any issues in execution. Actor have an address which specify the location of the actor, therefore it can run on a single machine as well as on different machines. Mobility is provided using those addresses which allow components to move across machines. This brings physical scalability to the system.

The three primary concurrency problems, divergence, deadlock and mutual exclusion are conveniently addressed in the Actor model. Let us briefly explain these problems. *Divergence* occurs when a program runs in an infinite loop, and cannot be reached because it does not accept any interventions due to an infinitely running process. The solution to divergence is the Actor itself. When an Actor is involved in an infinite process, and receives a stop message, the infinitely running process would be eventually terminated, because the stop message will be eventually processed. Infinite processes are always available for interactions. *Deadlock* is a resource-sharing problem that commonly occurs in distributed systems. When two threads attempt to access to a shared resource that is held by the other one, both are blocked, therefore leading to a deadlock situation. The *Dining professors* [37] problem is a classical example of deadlock. Actors are independent entities that communicate with each other through message passing. Detecting a deadlock is possible by the actors themselves because actors can query the state of other actors by sending messages. The *Mutual exclusion* problem happens when a shared resource is accessed simultaneously by multiple processes. The solution to the mutual exclusion problem is the Actor messaging principle itself. All messages are collected in a mailbox, and they are processed one at a time, so this is impossible. The actor simply buffers messages requesting a resource until the resource becomes free, thus ensuring mutual exclusion [38].

3.3 Actor Model Implementations

There are several implementations of the Actor model. To name few, Erlang, Scala, Java, and .Net languages have support for the Actor model. The Actor model is inherent in Erlang and is built into the language. Everything in Erlang [39] is a process, and behaves exactly like actors, and processes interact with each other through message passing. Scala [33] has an actors library but has been deprecated to promote the Akka toolkit which is created using Scala. Akka is a special library which implements the Actor model and has extra features that help to design concurrent and distributed systems. It has been ported to Java. It has limited availability on .Net platform [40]. Java has also Kilim [41], a library for implementing the Actor model.

Akka has been chosen as the underlying model for the platform's implementation as it works on JVM platform and has features such as streaming, clustering, sharding, and event-sourcing

3.4 AKKA

Akka is an implementation of the Actor model as described in 3.1 [34]. It is a toolkit for building concurrent, and distributed applications on the JVM using the Actor model. It has some deviations from the pure actor model of Hewitt to make it practical and usable. It provides a developer with a well-defined API to develop large concurrent systems and allows for easy scaling out of a single machine [42].

When using Akka, the developer must have the mindset that everything is an actor, similar to the mindset that everything is an object in Object Oriented Programming. This mindset makes a system easier to understand. Every created actor belongs to an actor system in the Akka model. Messages can be passed to reference the created actor.

Actors are designed to be small coherent computation units that are specialized to perform a single task. When a task becomes too big for an actor to handle, it is broken down into smaller pieces which are assigned to sub-actors, leading to an actor hierarchy. The depth of the hierarchy is determined by the complexity of the problem. The parent actor supervises and manages the life cycle of the child actor. If the child actor is unable to handle the message, it asks the supervisor actor to handle it. The message propagates all the way to the root supervisor if not handled on its way by an actor. Every actor in Akka has one and only one supervisor. This actor hierarchy forms the basis of the Akka's "Let It Crash" fault-tolerance model

Akka does not have to run on a single machine. It has a remoting layer which supports preconfigured parts of the system to work on remote hosts. For example, an Actor running on host A and a second Actor running on host B can work seamlessly because, built-in remoting support provides ready-to-use interface. Basically, actor references are used to send messages to one another, remoting only requires remote machine configuration. Thus, multiple devices located in different machines enable scaling out an application. Remote actors can be reached by specifying a protocol and location by which the corresponding actor is reachable, following the path in actor hierarchy. There exists a supervision link between a child and a parent down towards to the root of the Actor system. The following example uses TCP protocol to reach a remote actor.

`akka.tcp://my-sys@host.example.com:5678/user/service-b`
For UDP akka.udp protocol will be used. For local actors only akka keyword is needed.
`akka://my-sys/user/service-a/worker1`

Every actor uses the reference of a recipient actor to send a message. Handling of messages by the receiver Actor is done in a FIFO fashion. Furthermore, Akka implements a publish-subscribe mechanism through the Event Bus. This allows peer-to-many communication where one Actor publishes an event and all the other Actors subscribed to that event are notified. This mode is in contrast to peer-to-peer communication, in which an actor directly sends a message to another Actor's message box. Actors in Akka sit idle unless there is a message in the mailbox. When the mailbox becomes nonempty, a thread picks up the Actor with a message and performs some computation in response to the message. Actors perform tasks completely in parallel and messages can be reordered, which makes the system entirely concurrent. Concurrency is the core feature of Akka that provides great advantages in high traffic systems. Performance is a big concern in scalable systems. According to Akka documentation, an actor instance consumes nearly 300 bytes of memory, so one can have 3 millions actors per gigabyte [43]. The low memory requirement of Akka actors is an important factor in developing scalable applications. This allows a great number of parallel threads on a large number of CPUs in the same system.

The Akka framework provides the following features:

- **Concurrency:** Akka Actor Model abstracts concurrency handling and allows programmer to focus on business logic.
- **Scalability:** Akka Actor Model's asynchronous message passing allows applications to scale up on multi-core servers.
- **Fault tolerance:** Akka borrows the concepts and techniques from Erlang to build a "Let It Crash" fault-tolerance model using supervisor hierarchies to allow applications to fail fast and recover from the failure as soon as possible.
- **Event-driven architecture:** Asynchronous messaging makes Akka a perfect platform for building event-driven architectures.

- **Transaction support:** Akka implements transactors that combine actors and software transactional memory (STM) into transactional actors. This allows composition of atomic message flows with automatic retry and rollback.

- **Location transparency:** Akka treats remote and local process actors the same, providing a unified programming model for multi core and distributed computing needs.

- **Scala/Java APIs:** Akka supports both Java and Scala APIs for building applications.

Chapter 4

Architecture and Design of the Platform

The platform implemented in the thesis is based on a proposal given in "A Platform for Context-Aware Application Development: PCAD", which characterizes the platform as *"a novel software platform based on the notion of context-awareness which allows rapid and easy development of context aware applications."* [44]. The design of PCAD was inspired by operating systems with the purpose of responding to context-aware computation needs. Data sources which are responsible for collecting context-data from environment, such as city traffic, can easily be integrated to PCAD due to the platform's plug-and-play feature analogous to connecting a USB device to a computer with an OS running. Similarly, applications can use the services of PCAD by binding themselves to the platform using an API library. This resembles the user level application programs running on a UNIX operating system and linked to POSIX application interface libraries. In the following sections, features and design principles behind the platform are explained.

4.1 PCAD

PCAD architecture and design was inspired by operating system design. An operating system provides an environment and services to programs and users. These operating system services allow programs, programmers and users to perform tasks

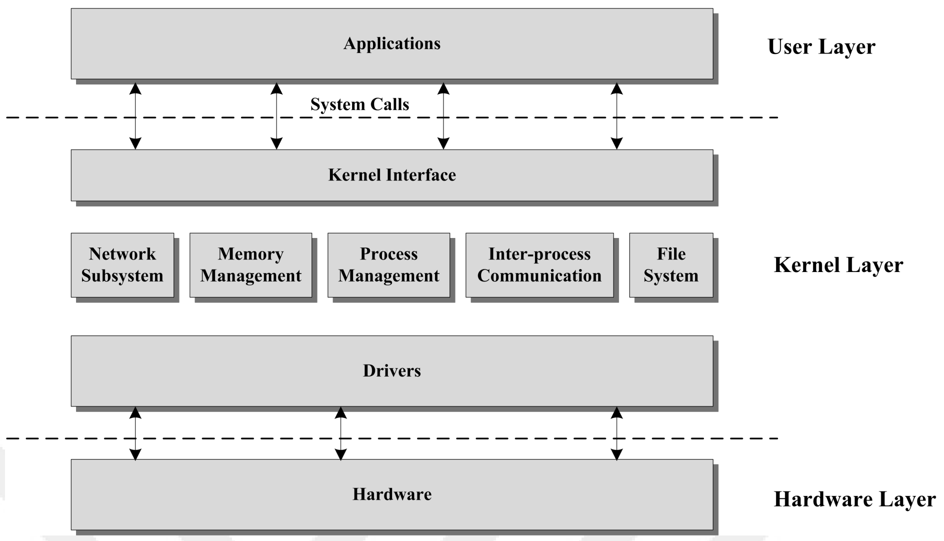


Figure 4.1: Operating System Layers

easily and conveniently. Programs and users get access to the OS services through a public interface. Some examples of these services are I/O operations, file system, protection and security, memory management, intra- and inter computer communication. A majority of the modern operating systems implement these services in terms of loadable modules [37]. Such a design provides for extendibility and dynamic behavior. Another important design approach used in the implementation of the modern operating systems is the layering of the functions. The layers found in a typical operating system are shown in Figure 4.1. Layered architectures help eliminate the tight coupling of the functions, thus paving the way for reusability and maintainability. It also simplifies the implementation and debugging of the operating system code. Even though an operating system has multiple layers, it can be conceptually viewed as comprised of two levels: user level and kernel level. Obviously, each level contains several sub-layers in them.

PCAD uses a middleware based layered architecture augmented with the Blackboard model to connect the context providers to the context consumers. An overview of PCAD is shown in Figure 4.2.

Using the OS analogy, one can view the platform as a layer providing kernel-like services and the applications providing user level services. Physical sensors and their software act like devices and device drivers respectively, providing data to applications. PCAD functionality is made available to applications through services. Each service is responsible for exercising a particular function, and when needed

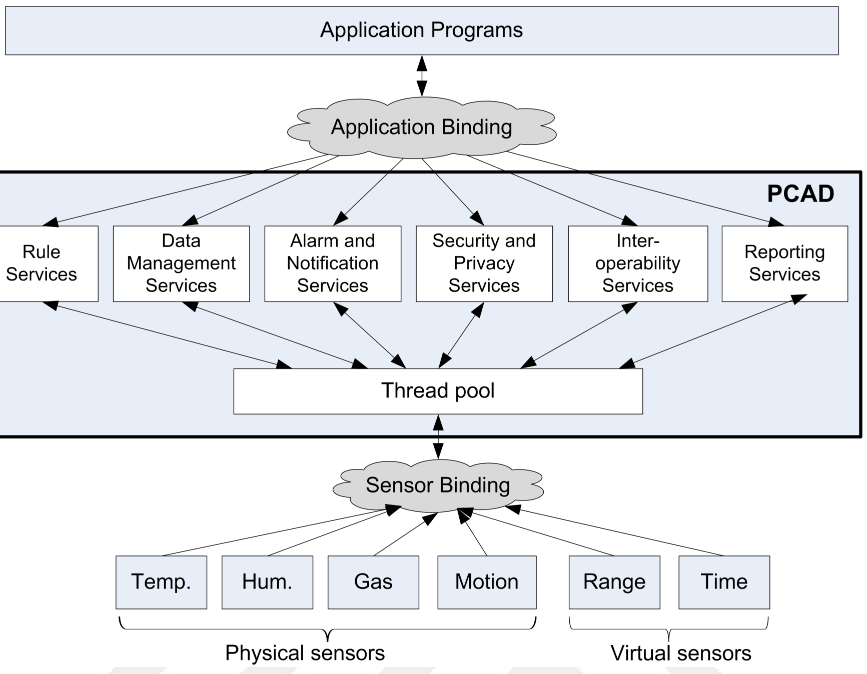


Figure 4.2: PCAD Architecture Overview

multiple services interact with each other to complete an application request. For instance data management and security and privacy services work together to ensure data is made available only to those allowed to access.

AKKA and the Actor model is chosen as the programming model for the implementation of the platform for the reason that the Actor model is a viable alternative when implementing concurrent and distributed systems. Actors are used to model services. This allows the platform to scale and let the services to execute on another computer, hence distributing the load. Our implementation has not used this feature; however, extending the system to distribute work across other computers can easily be done after minor modifications to the system, since the underlying computation model is the Actor model.

4.1.1 Services

PCAD offers its functions as services to applications. When requesting a function from PCAD, based on the nature of the request, the platform determines which service or services will participate in fulfilling the corresponding request. Each service is implemented as an actor. The concept of service based computing coupled with the Actor model provides the advantage of distributing the load across other machines over the network. Service orientation enables loose coupling between transacting entities, making the system flexible and extensible.

PCAD consists of six services listed in Table 4.1, and explained in Sections 4.1.1.1 through 4.1.1.6.

Table 4.1: PCAD Services

Service	**Task**
Rule Service (RS)	Processes context based on rules.
Data Management Service (DMS)	Collects context-data and gives them whenever a new request received via its interface. It is transparent within storage mechanism.
Alarm and Notification Service (ANS)	Informs related clients using context-data and filters them if necessary.
Reporting Service (RPS)	Creates reports on context-data generators with details such as data and status of related device.
Security and Privacy Service (SPS)	Protects context-data using access control mechanism and only authorized clients can reach data.
Interoperability and Communication Service (ICS)	Receives and delivers data among context-aware platforms using standard communication protocols and message formats.

4.1.1.1 Rule Service

The Rule Service, in essence, is responsible for “context processing”, according to the user-supplied rules. It responds to application requests that require special processing, such as aggregation of data acquired from different sensors. The Rule Service performs this task according to the rules specified in a separate rule file sent by the application, or according to the rules embedded in the JSON request sent by users.

An application can request complex actions using this service. Details of the syntax and semantics of the rules are given in Section 5.1.1.2. The rules are simple if-then structures interpreted by the rule service. More advanced rules are created by using “and” or “or” logical connectives. For example, `"if (windSpeed > 10 and humidity > 50) then notifyRain"` rule specifies that the application is going to be notified when the wind and humidity sensors report values greater than 10 and 50 respectively.

4.1.1.2 Data Management Service

The Data Management Service is assigned the task of storing context related data received from a variety of sources. Besides sensor data, other types of data, such as user profiles, privacy settings, and sensor meta data are also stored and managed by this service.

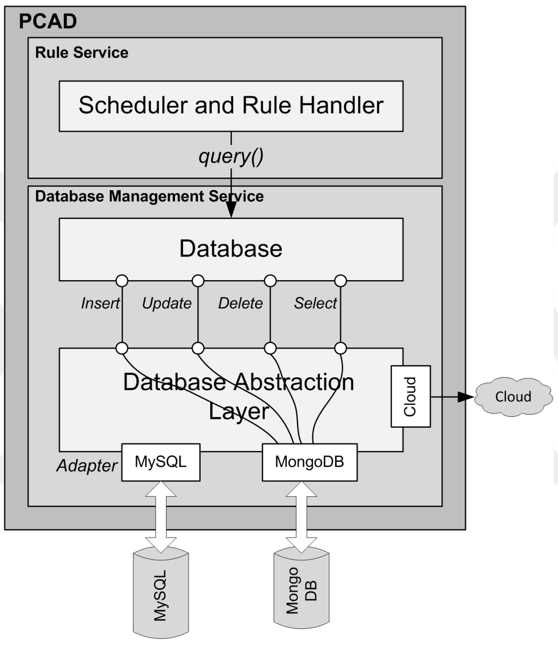


Figure 4.3: Database Abstraction

Data is stored in a database. The two alternatives for the database are relational (SQL) and document (noSQL). Relational databases keep data in tables organized as rows and columns. Creating a new field in the relational database involves adding a new column to a table. Manipulation of the database is done using the SQL database language. NoSQL databases such as MongoDB, store data in the form of key-value pairs providing faster query, update, addition, and deletion compared to relational databases. Since there is no concept of tables, rows and columns, a new field is created simply by adding a key-value pair directly. The Data Management Service is architected to support both types of database and incorporate various types SQL database options such as PostgreSQL. However, the implementation described in the thesis is done only for the relational database and supports only MySQL. Figure 4.3 shows the planned future design to support both databases. The design provides a unified interface for database operations such as insert, update and filter and uses the Adapter design pattern principle to adapt the unified interface to the underlying database specific operations.

4.1.1.3 Alarm and Notification Service

The Alarm and Notification Service (ANS) follows the Observer design pattern paradigm [45]. Applications attach themselves to a sensor of interest to obtain sensor data and status information. When the data becomes available from a sensor, applications are notified. Two modes of operation can be employed to acquire the sensor data: Asynchronous and Synchronous. In the asynchronous mode, ANS (i.e. platform) periodically queries the sensor data, while in the synchronous mode, sensor software sends an interrupt to ANS causing the platform to initiate a sensor data read. In the asynchronous mode, the platform initiates data retrieval from the sensor and stores it in the database. This data then becomes available for interested applications and is served to pending data requests. The asynchronous mode requires sensor software support to keep the sensor software listening and ready to respond to requests from ANS. The asynchronous mode is not the usual way for clients to interact with a sensor, since asynchronous mode is usually not supported by sensor software. Therefore, the synchronous mode of operation is used in the interaction between a sensor and the platform. ANS transfers the sensor data to applications using push method. In the push model, ANS sends sensor data when constraints are satisfied using a filtering mechanism. The filtering mechanism allows a client to control how and when data is sent, i.e. instant versus delay tolerant data delivery. Further details on filtering will be explained in the next chapter.

4.1.1.4 Reporting Service

The Reporting Service is intended to generate reports about sensor information using various visualization tools such as graphs, tables, charts. Among the information reported are sensor meta data, status and actual sensor data. Open source reporting tools can be integrated into this service making it more familiar and beneficial for platform users. This service is included conceptually in this version and no implementation has been provided mainly because it is much less relevant to the general functions of the platform. Several off-the-shelf third party software tools exist to help visualize data and create reports. Once the platform stores data in its database, these tools can produce sophisticated reports by accessing the database directly. Moreover, user requirements may radically differ on the type of the report and data visualization tool. Therefore, it was decided to leave the reporting service as a future add-on, currently using the services of the third party tools independent of the platform.

4.1.1.5 Security and Privacy Service

From an application perspective, the platform is the owner and the applications are the users of the data. Therefore, it is the platform's responsibility to ensure the security and privacy of data. The Security and Privacy Service is responsible for protecting the security and privacy of sensor data and deciding which applications are authorized to access data. A very basic and simplified Role Based Access Control (RBAC) [46] mechanism is used to determine which sensor data is accessible by which application, with the help of a set of roles. Our use of RBAC resembles the Mandatory Access Control [47] mechanism, in which the sensor data privacy level is checked against the clearance level of the application. The roles are defined by the platform and based on a policy configuration. For example, a weather application does not need access to the transportation data; on the other hand, a transport application is allowed to access the weather information.

RBAC model involves three components: subject, object and access right. Regarding the platform, the subjects are applications, the objects are sensors. The subjects are assigned roles to determine the access rights on objects. When an application requests data, the request is granted if the application has the right role assigned to it.

4.1.1.6 Interoperability and Communication Service

This service's intention is to provide a capability to communicate with PCAD-like platforms. Inter-platform data transfer allows data provisioning from third party data providers. Other platforms can ask data from the PCAD platform as if they are ordinary platform users. PCAD has provided an Application-Platform

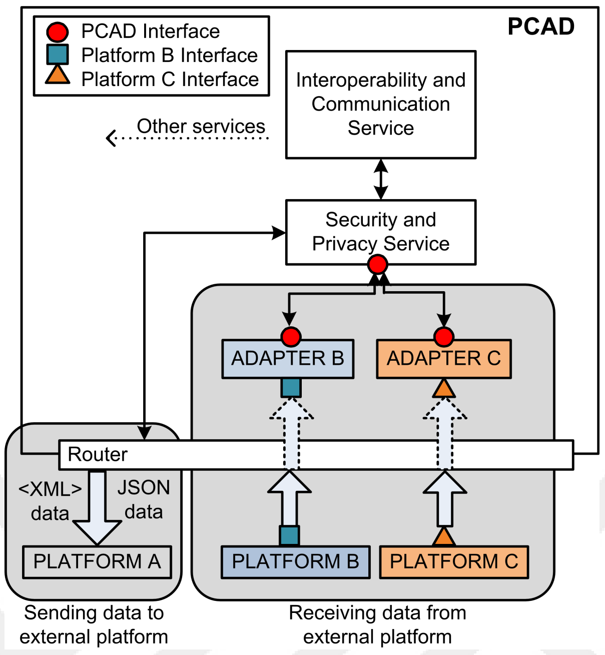


Figure 4.4: Interoperability and Communication Service Adapters

Interface for general use. This interface is built using the HTTP Request/Response mechanism, therefore it provides a standardized way to communicate with the platform. Other platforms can thus use HTTP request/response to send and receive portable JSON formatted data. Third party platforms can additionally use a language specific API library similar to an application linking to a library.

In order for PCAD to receive data from other platforms, the interface of the corresponding platform must be known. A plug-and-play system that uses the adapter design pattern [45] is still under consideration to allow transparent access to other platform's functionality transparently. The design of this structure is shown in Figure 4.4 However, this function is not included in the current version of the platform; the design alone is presented in the thesis.

4.1.2 Real-Time Support

Some applications are sensitive to prompt data delivery, and hence impose constraints on the data delivery time. For instance, an application that is using two different sensors to analyze driving behavior has two different data delivery requirements. One sensor provides the GPS coordinates of the car and another sensor - a motion sensor (i.e. G-sensor or gravity sensor) - provides data such as linear acceleration. Using the motion sensor one can analyze various driving behavior patterns such as sudden acceleration, sudden brakes and sudden turns. Location data should be made available to the application immediately, since the application would dynamically instruct the driver to turn either left or right. On the other hand, the data collected by the motion sensor is delay tolerant, as this data can be processed and interpreted later on, in conjunction with other data (e.g. weather). Therefore, this kind of data impose more lenient constraints on data delivery.

As per the discussion in Section 2.1, the two main approaches in acquiring context are the direct sensor access and the middleware infrastructure. The applications and sensors are tightly coupled to each other in the direct sensor access architectures. Therefore, the applications acquire context directly from the sensor without any intervening components, thus direct sensor access is better suited to fulfilling hard real time requirements. On the other hand, the middleware architectures decouple the components, thus offering many advantages in software development, but at the same incurring time penalty as data moves between the layers. For this reason, the middleware architectures are more suited for delay tolerant data delivery requirements. Systems that are more tolerant to time latency continue to operate even when some of the time constraints are violated.

PCAD as a middleware based architecture, supports immediate data delivery requirement by creating a direct communication pipe between a sensor and an application, and assigning an actor to it. When the data becomes available from the sensor, it is delivered to the application through this pipe, while the same data goes through its usual processing within the platform simultaneously. This is illustrated in Figure 4.5. When received from a sensor, the data is provided to the application through a pipe managed by Thread 1. The same data is simultaneously given to Thread 2, which takes the data through the services for its usual processing, which is eventually written to a database.

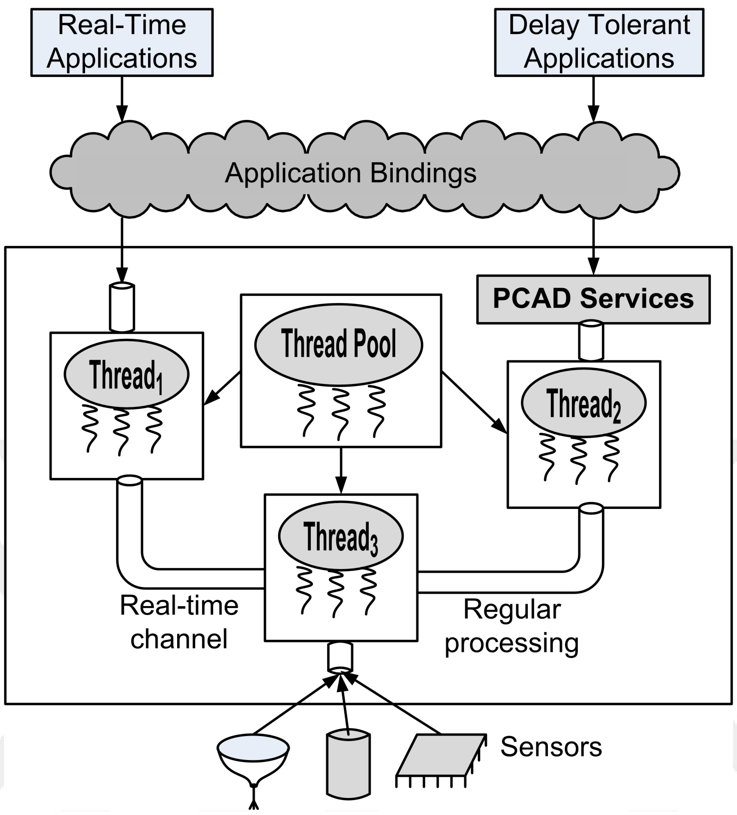


Figure 4.5: Real-Time Support

4.1.3 Sensor-Platform Interface

Sensors are designed to generate data. Physical sensors are data sources for PCAD, which generate data by interacting with the physical environment and virtual sensors provide data to PCAD using software applications or services such as e-mail, weather service, mouse movement etc. The data generated by physical and virtual sensors needs to be sent to PCAD in a certain format. For this purpose, PCAD has a specific layer to provide the needed interface for physical and virtual sensors to send data. This layer is basically a communication library, therefore sensor software primarily focuses on measuring and collecting environmental data, delegating other tasks, such as sending data and storing it to PCAD. It has a simple, intuitive and easy-to-use work mechanism. The sensor-platform interface simplifies the task of delivering data from sensor to PCAD. Data providers use only the library functions to send data to the platform, while the library communicates with the platform using the REST technology. RESTful based communication is gaining momentum in standard web technologies for the primary reason that it is an HTTP based protocol, and is much simpler to use than SOAP based methods.

HTTP is the de-facto internet standard, and it is universally supported by the operating systems and programming languages. Note that since the platform is ready to respond to the REST requests, data providers may opt to use RESTful API directly bypassing the provided library.

4.1.4 Application-Platform Interface

Applications are consumers of the platform which use sensor generated data. Applications need to access to the data and sensors seamlessly and simultaneously. A binding layer between the application and the platform is instituted to address these needs. An interface is created to exchange data between the application and the platform. It is very similar to the sensor-platform binding layer described above but specifically designed for applications. It works using the same principle as explained in the sensor-platform interface. The interface consists of a set of functions which send requests to carry out the desired actions to the platform services. In response, the platform processes the request and sends the result back to the application. Applications can use library functions or RESTful API. WebSockets are used to implement real-time communication [48] with PCAD. WebSocket is a well-known, easy to understand and widely used protocol for the data exchange. The WebSocket protocol remains open until the connection is closed and unlike the HTTP protocol, does not end. The provided interface allows applications to make rule-based requests. Rules enable the construction of more complex requests, and it paves the way for more sophisticated applications. Interface layers are shown in Figures 4.6 and 4.7 .

4.2 Implementation Decisions

As per the discussion given in this chapter, the implementation decisions made are presented in the following table:

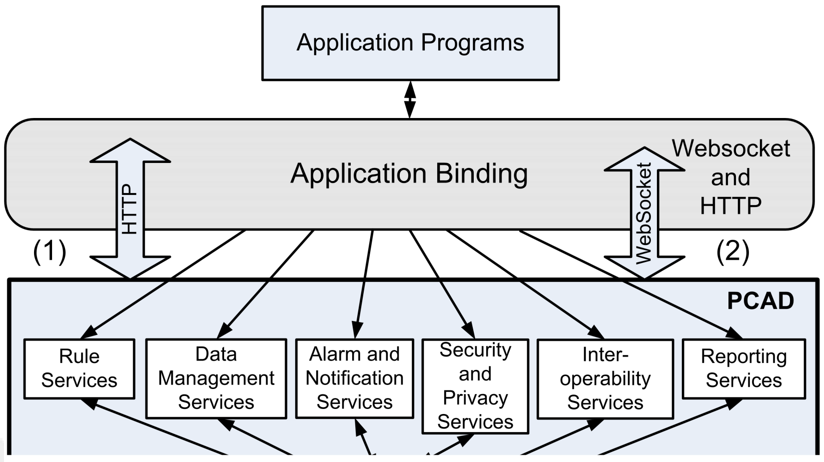


Figure 4.6: Application Bindings

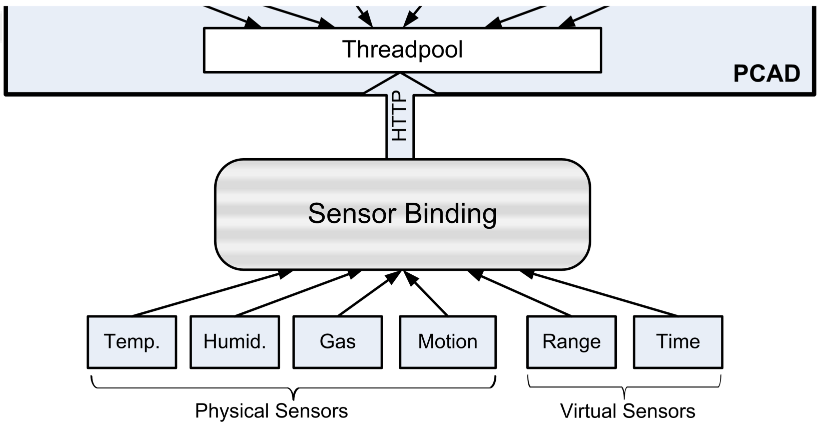


Figure 4.7: Sensor Bindings

Table 4.2: Implementation Decisions

Framework	:	AKKA, Play
Computation Model	:	Actor Model
Language	:	Scala, Python, Javascript, HTML
Operating System	:	Linux
Database	:	Relational-MySQL
Platform Interfaces	:	RESTful API, Python, WebSocket

Chapter 5

System Implementation

This chapter explains the implementation of the platform. The important implementation decisions are choosing a framework, a language and a computation model to address scalability, fault tolerance and performance needs of the platform. The platform must respond well to an increase in the number of applications using the platform and to an increase in the amount of data generated by the sensors. It has to peacefully recover from faults and continue to work, be responsive under high load and traffic without causing excessive latency. The Akka toolkit and the Actor model is chosen as the computational model since the Actor model [34] is tailored for developing scalable, responsive and resilient systems.

Several Actor model implementations exist in different languages. Akka [42] is the renown implementation of the Actor model which has been created first using the Scala language then is ported to Java. There exists a port for the .NET platform, but it does not contain all the features of Akka. Scala supports the functional programming paradigm besides the object oriented programming model and has more features than Java. It has immutability support, type inference, extensive type system, and is also statically typed. It runs on a Java Virtual Machine platform, therefore, one can use Java and Scala within the same program. JVM is a proven and longstanding technology created mainly for executing Java programs after they are translated into Java byte-code. It is designed with write-once-run-everywhere philosophy in mind to run applications on different host systems. The JVM provides a common platform for applications and permits the execution of other programming languages once they are compiled to byte-code.

For instance Scala and Kotlin languages work on JVM. AKKA is considered a toolkit as it has extra features besides a mere implementation of the Actor model. Akka toolkit, Scala and JVM are viable development tools in creating enterprise solutions. Therefore, they are chosen to implement the PCAD platform.

PCAD consists of two main components: a back-end, which is the server-side and a front-end which is the client-side. In the remaining sections, the server and client sides of the platform will be discussed in detail.

5.1 Server-Side of Platform

The server-side is the back-end component of PCAD. It is the indispensable and a core component of the platform that contains services and business logic. This section explains how the platform services work and interact with each other and how the platform communicates with the applications and sensors. The server-side of the platform will be presented in two sections. First, the services will be discussed and then the communication layers of PCAD which provide the application-platform and sensor-platform communications will be presented.

5.1.1 Services

Services are the backbone of PCAD and responsible for accomplishing the core tasks of the platform. Each service operates as an actor in PCAD and each service instance runs in a separate thread because an actor is executed within its own thread in Akka. The interaction among services are accomplished via message exchanges between actors, therefore, the messages must be defined carefully to distinguish one task from another in the system. Every message arriving at a service is first identified and then mapped to an action to be triggered. The identification of a message is done by using Scala's comprehensive pattern matching features based on regular expressions.

Each service has a predefined task assigned to it within PCAD. For example, the Security and Privacy Service manages the security related actions such as generating and decoding tokens and does not handle the database related actions such as query, insert, update which are performed by Data Management Service. When a request arrives at PCAD, multiple services may be involved in fulfilling the request, causing several message exchanges among services. For instance, the “list sensors” request is handled by executing Security and Privacy Service for authentication and Data Management Service for querying the sensor list. In this example, first, a message is sent to Security and Privacy Service, to initiate a security transactions, and then a second message is sent to Data Management Service to perform database transactions. In summary the system is composed of a series of service processes as a whole.

5.1.1.1 Security and Privacy Service (SPS)

SPS mainly performs two tasks, authentication and authorization, for the purpose of restricting and controlling data access in PCAD. Additionally, it performs the token generation and decoding tasks. Tokens are being used extensively to resolve who can access data while performing tasks. The clients need a token when accessing the data, and they get a token after they register with the platform. The clients request data access using this token and when their request is granted, they gain access. The application owners assign access rights to their applications, thus user created applications get their own set of rights to access the data. When a new user signs up or a new application is registered with the platform, a private access token is generated for them. Access tokens are implemented using Encrypted JSON Web Token [49]. JSON Web Token (JWT) is widely used in many integrated web applications in establishing a secure HTTP communication [50]. A User or an application makes requests using a previously issued access token(i.e Web token). The user registration, sign-in or sign-out requests do not need a token. However, all other requests will be rejected without an access token. The access token will be used for authenticating identities by SPS and the token is forwarded to Data Management Service along with a message to retrieve data after the authentication is successful.

Access mechanism to sensor data by applications and users is designed based on a very basic and simplified version of Role Based Access Control (RBAC) [46]. Role Based Access Control (RBAC) regulates access to sensor data based on the roles of individual users and applications by placing restrictions on the authority and types of access to the source. RBAC model is closely related to the Mandatory Access Control mechanism [47]. A privacy level is assigned to sensor data and an access level is assigned to applications which specifies what kind of information is available to the applications. When the access level is higher the than privacy level, access is granted. In contrast to conventional methods of access control which grant or revoke user access on a set of rigid access right assignments between objects and subjects, in RBAC, roles can be easily created, changed, or discontinued according to the platform policies without having to individually update the privileges for users and applications. Roles are defined according to platform-based policies.

Sensors which generate data for the platform also needs to register and authenticate similar to client registration and authentication. PCAD does not allow rogue senders to send data. When a sensor registers to the platform, the sensor is assigned a token which gets embedded into every subsequent data send request.

The main difference between a sensor and an application access is that access right of a sensor is not based on levels. Sensors only have one level of access. They are either allowed to send data or not. On the other hand, applications have three levels of access rights. Applications can access sensor meta data (i.e. location, manufacturer), sensor data values (i.e. temperature, humidity reading), or sensor status data such as sensor being on or off.

Access rights are assigned by the user. These rights can be combined into roles to simplify the assignment of rights. Therefore, instead of setting access rights individually, one can group rights into a role and assign the role to the application as a whole.

Access Permission Assignment: In order to gain access to the platform, several steps must be completed. The very first of these steps is registration, a process monitored by the administrators [51]. Users then can send request for permission to access to a sensor via web application or RESTful API which in turn creates a notification for the PCAD administrator to act on it. Permissions are granted to users if the administrator approves the request. Applications are assigned rights following a slightly different process. Application owners specify a set of sensor access permissions on behalf of their applications and ask the administrator to approve these rights. The administrator makes a decision based on a predefined platform policy, and application gains access to sensors if the administrative decision is positive. Permissions are categorized into three depending on the type of information being accessed: *information*, *data*, *status*.

Request Validation Process: There are a lot of clients which will request data access from the platform. In order to ensure the security of the platform, we have implemented validation levels for the requests, and every level must be passed to access the resources.

1. If the request requires an authentication token, the token must be sent along with the request. Each client (i.e. user, application, sensor) has a token generated for it by the platform, and this token is used while handling requests. Clients have to provide an authentication token for every request requiring a token. Only sign-up, sign-in and sign-out requests do not need a token.
2. When a token is present in a request, SPS attempts to decrypt it. If the decryption is successful, the token is valid and the request progresses to next the step. Otherwise, the request is denied.
3. The clients are classified as being user, application or sensor. Each class is capable of doing certain actions and has restricted access to the platform. The requests made by the clients must be compatible with their class. For example, the sensors cannot ask for data, but only provide data to the platform. In contrast, the applications can not send platform sensor data but are permitted to retrieve sensor data. The users cannot request sensor data, only the applications can. On the other hand, the users can access the meta data about sensors such as the type or the location of the sensor.
4. The requester must have the appropriate permission to access the data. In order to access context-data, the requester must possess a specific permission for the information sought after. For example if the requester wants to find out if a sensor is on or off, it must have the '*status*' permission. This is validated by looking up the access tables.

There are two methods that the users can use while accessing the platform as illustrated in Figure 5.1:

1. The user connects directly to the platform via a web browser, and sends the request to the current URL address using the REST-API, and accesses the sensor meta data (User).

2. The application users access the platform indirectly through an application (Application User).

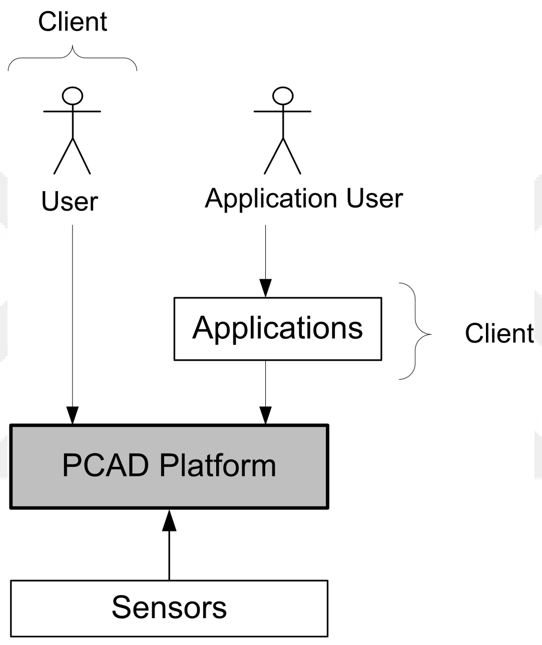


Figure 5.1: User and Application User Access to Platform

The application users are limited only to those rights authorized by the application for the users. This effectively reduces the permission rights of the user to the rights of the application. As the application users access the system through an application, the rights granted to the application will be used and these rights maybe further restricted by the application developer if needed. The applications restrict access for a particular user by creating their own databases and creating user profiles, thereby preventing the application users from having undesirable rights.

The applications and the sensors communicate with the platform differently than the user class. The users are able to identify and register themselves to the system by providing a username and a password. However, this authentication process is not practical for sensors and applications because sensors and applications are not capable of providing a username and password on their own. To

Example JSON Web Token

```
1  """
2  eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJQQ0FEIiwi
3  dWlkIjoxNSwiZ3JvdXAiOiJzZW5zb3IifQ.8ifEzEbLGwMWvs__IvGXK
4  X9Eh056HWZYXHkrcytcoPY
5  """
6
7  header = '{"alg":"HS256","typ":"JWT"}'
8  claim = '{"iss":"PCAD","group":"sensor","uid":"1"}'
```

Figure 5.2: Example JSON Web Token

initiate communication with the platform application and sensors have to first validate themselves with access tokens. If this process is successful, they get a session token, which is valid for a fixed period, and used during data exchange. This will complete the authentication process.

Connection between the platform and sensor also requires access and session tokens. Sensors connect to the platform and send data in a similar way. Sensors and sensor software first receive an access token during the platform registration and then subsequently acquire a session token. The sensor software also sends this session token along with data to the platform. Once the sensor has been authenticated, the sensor is allowed to send data. The most important difference between a user accessing the platform and the sensor accessing the platform is that the level of access for the sensor is single level. The sensor either has a right to access the platform or not. However, the access right for user is multi level.

Tokens are defined in JSON Web Token (JWT) format. An example JSON Web Token is given in Figure 5.2. Line 2-4 is the actual JWT as it is sent to the platform along with the request. JSON Web token is divided into three sections: header, claim and signature. In Figure 5.2, lines 7 and 8 show decrypted contents of the header and claim. This token was signed using HS-256 (HMAC-SHA256). Users can view their access tokens.

Each user or application is assigned a role which they use when accessing a sensor. A role is a collection of three authorizations, named as *information*, *data* and *status*. The union of these three authorizations determines the access capability of a role. The definition of these authorizations is given below.

- **Information**: The right to query sensor meta-data (i.e. location, type, manufacturer and other properties).
- **Data**: The right to receive data from the sensor (i.e. sensor measurement data).
- **Status**: The right to the query sensor status information (i.e. whether active or inactive).

For example, an application must be assigned a role containing information, data, and status rights in order to learn sensor location, query sensor state, and receive sensor data. Such a role is described as

```
RoleA = (I, D, S)
```

where

`I`, `D` and `S` stand for Information, Data and Status rights respectively. If a role does not possess a particular authorization, it is shown with a '-'. Other roles can be defined similarly.

```
RoleB = (-, D, -)

RoleC = (-, -, S)
```

5.1.1.2 Rule Service (RS)

The Rule Service is responsible for processing the raw data acquired from the sensors according to the rules specified in a rule file. For example, the end-users want notifications for certain events because notification information maybe more meaningful than the raw data itself for them. The rules are contained in special purpose files that are used while processing data. The applications sends those files along with a request, which is then taken out from request, parsed and executed properly by the Rule Service. In order to execute a rule file, a rule engine was designed and implemented as a part of PCAD. Rule engine mainly performs two jobs: parsing and interpreting a rule file. For this purpose, a small rule language was also designed.

Table 5.1: Rule BNF Notation

⟨*Rule*⟩	::=	⟨*RuleConf*⟩⟨*RuleAttr*⟩⟨*RuleBody*⟩
⟨*RuleConf*⟩	::=	‘`name`’ = ⟨*ident*⟩
⟨*RuleAttr*⟩	::=	⟨*VariableDef*⟩
	|	⟨*VariableDef*⟩⟨*whitespace*⟩⟨*RuleAttr*⟩
⟨*RuleBody*⟩	::=	‘`when` {’ ⟨*Condition*⟩ ‘} `then` {’ ⟨*Action*⟩ ‘}’
⟨*CompositePredicate*⟩	::=	⟨*Condition*⟩ | ⟨*Predicates*⟩
⟨*Predicates*⟩	::=	⟨*Predicate*⟩
	|	⟨*Predicate*⟩ ⟨*LogicalOperator*⟩ ⟨*Predicate*⟩
⟨*Predicate*⟩	::=	‘`(`’ ⟨*CompositePredicate*⟩ ‘`)`’ | ⟨*Condition*⟩
⟨*Condition*⟩	::=	⟨*Expression*⟩
	|	⟨*Expression*⟩ ⟨*Comparator*⟩ ⟨*Expression*⟩
⟨*Action*⟩	::=	⟨*Function*⟩
	|	⟨*Function*⟩ ⟨*whitespace*⟩ ⟨*Action*⟩
⟨*Expression*⟩	::=	⟨*Variable*⟩ | *number*
⟨*Comparator*⟩	::=	== | != | < | > | ≤ | ≥
⟨*Function*⟩	::=	⟨*ident*⟩‘`(`’⟨*Parameters*⟩‘`)`’
⟨*Parameters*⟩	::=	⟨*Parameter*⟩
	|	⟨*Parameter*⟩ ⟨*whitespace*⟩ ⟨*Parameters*⟩
⟨*Parameter*⟩	::=	⟨*VariableDef*⟩ | ⟨*Variable*⟩
⟨*Variable*⟩	::=	⟨*ident*⟩
⟨*VariableDef*⟩	::=	⟨*ident*⟩ ‘`=`’ ⟨*Definition*⟩
⟨*Definition*⟩	::=	⟨*Function*⟩ | ⟨*ident*⟩ | *number*
⟨*LogicalOperator*⟩	::=	‘`and`’ | ‘`or`’
⟨*ident*⟩	::=	*string*
⟨*whitespace*⟩	::=	‘ ’

The BNF notation [52] describing the rule language is given in Table 5.1, and an example rule written in this language is shown in Figure 5.3. As seen in the example, a rule consists of two main parts: the preamble and the body. The variable definitions, value and function assignments are done in the preamble part. In

Example Rule

```
## +------------------------------+
## |           Preamble           |
## +------------------------------+

## Part 1 - Rule Name
name = "notification"

## Part 2 - Variable Assignments
min_value = 10.5
current_value = get_value()
db_value = get_db_value(sensor=21)

## +------------------------------+
## |           Body               |
## +------------------------------+

## Part 3 - Logical Expressions
when {
  (current_value < min_value) and (current_value > db_value)
}

## Part 4  -- Actions
then {
  Notify(message="Unacceptable Temperature Value")
}
```

Figure 5.3: Example Rule

the preamble part, one retrieves data by invoking functions, and define constant values. The following part - body - contains a **when** block which houses logical expressions. Depending on the result of the logical expression, the rule engine triggers execution of the statements in the **then** block. Compound logical expressions are formed using ‘and’, ‘or’ connectives. System defined functions can be called in the **then** block. The example in Figure 5.3 defines a rule named “notification”, and a constant variable named `min_value`. The rule asks for a value from a sensor whose id is specified in the method call that sent the example rule file. Additionally, the rule asks for a value from the database for a second sensor with an id of 21. Then, it uses these two values in a logical expression to finalize the execution. If the expression yields a *true* result, it notifies the clients with a message specified in the *Notify* function call.

Table 5.2: Rule API

Name	Parameter	Description	Sample call
`get_value`	-	It retrieves value from default sensor. Default sensor is determined during the method call that establishes the connection.	`data = get_value()`
`get_time`	-	It retrieves hour value from default sensor.	`time = get_time()`
`get_rt_value`	*`sensor_id`*	It retrieves measured real-time value from defined sensor.	`data = get_rt_value(sensor=1)`
`get_rt_time`	*`sensor_id`*	It retrieves measured real-time hour value of time from defined sensor.	`time = get_rt_time(sensor=1)`
`get_db_value`	*`sensor_id`*	It retrieves measured database value from defined sensor.	`data = get_db_value(sensor=1)`
`get_db_time`	*`sensor_id`*	It retrieves measured database hour value of time from defined sensor.	`time = get_db_time(sensor=1)`

Applications retrieve data from the platform using the application-platform binding library provided by the platform. The library functions are shown in Table 5.2. The applications may ask for a one time instant data, or they can get data in periodical intervals. The applications can optionally provide a rule file to specify how to obtain data from the platform. When the applications use a rule file, they make use of a set of built-in functions to retrieve context data in more elaborate ways. For example, they can retrieve data from multiple sensors or from a database. Without a rule file only one sensor is permitted to send data and no control flow exists for given data. The following cases explain various ways of retrieving data from the platform:

- "Real-Time-No-Rule": Whenever a sensor pushes data to the platform, data is instantly delivered to the requester. The requester is blocked if there is no data available.

- "Periodical-No-Rule": The platform delivers data in fixed intervals to a requester. The data is obtained from the database, therefore requester is never blocked. However, the requester may get stale data in the case where sensor has not provided any fresh data to the database within the fixed interval.

- "Real-Time-With-Rule": This is similar to the first case with the addition of a rule file. Data delivery depends on constraints specified in the rule file. When a rule file is used, data from multiple sensors can be retrieved.
- "Periodical-With-Rule": This is similar to the second case with the addition of a rule file. Data is obtained from the database in fixed intervals as specified by the application. Data delivery frequency is specified in the rule file.

The cases mentioned above require a WebSocket connection between the application and platform. If an application needs a one-time data delivery from the database, a simple HTTP request/response mechanism would be sufficient.

The rule file uses the *StandardTokenParser* library written in Scala for syntax processing. This library utilizes Scala language pattern matching features to facilitate parsing. The rule file is parsed first to verify that it is in a valid form, then it is transformed into a rule object for execution. The rule object consists of a rule file and its associated metadata (i.e. metadata is related to the application that sent the rule). Finally, the interpreter executes step-by-step the rule file by interpreting its contents. Commands in a rule file are divided into four parts as shown in Figure 5.3 example : rule name, variable assignments, logical expressions and actions. Each component is run one after another. In the first step, the rule name is taken and the acknowledgment process is carried out (Line 6). In the second step, assignments of variables are completed (Lines 9-11) In the third step, the result of the logical expression is obtained (Lines 18-20), and according to the result, the operations in the last part are performed (Lines 23-25). The execution of the third and fourth steps is performed by the rule handler.

5.1.1.3 Data Management Service (DMS)

The Data Management Service is responsible for performing database related tasks. The database stores user, sensor, and application related information. Every action performed by PCAD is based on the stored data. Relational and non-relational type of databases were investigated for data storage. Relational databases have been widely used and are an accepted technology in information technology applications for years. They categorize data very well, are good when the data is modeled structurally. Various commercial and open source implementations do exist. The open source database MySQL[53] is one such implementation. The well-known language SQL [54] lets one access and manipulate relational databases. We have chosen the relational database as the database model and MySQL[53] as the open source relational database management system since in the long run a large amount of data can be stored in a more compact way in the form of tables. More importantly relational database establishes relationships among tables so that by using join operations one can get more insightful results about the data. Even though non-relational databases such as NoSQL are gaining popularity in web development in the recent years, extra effort has to be spent creating relationships in a NoSQL database like MongoDB [55].

The Data Management Service performs database actions such as *create*, *read*, *update*, *delete*. The platform uses Slick [56], a functional relational mapping library to accomplish database actions easily. Another advantage of Slick is that it supports multiple SQL database types such as PostgreSQL, Oracle and MySQL. It is written in Scala, and integrated seamlessly into the platform. It supports asynchronous query execution. Figure 5.4 presents MySQL schema and table definitions, and Tables 5.3 to 5.10 describe table attributes.

Table 5.3: Database Table Descriptions

Table Name	**Description**
`users`	The `users` table contains the user information and the access token used for authentication. There are two types of users of the platform: administrators and developers. The developers create sensor and application software. The administrators manage the platform (i.e. granting permission, blocking users.)
`applications`	This table contains information about applications using the platform. the access token needed for application authentication is also kept here. The access token is acquired when applications register to platform.
`sensors`	This table stores sensor meta data. Every data generator, whether it is physical or virtual, is saved in here.
`roles`	Applications and users need permissions to access sensor data. Roles specify those permissions rights, which are basically a combination of data, status, information authorizations. The role information is kept in this table.
`user_sensor_accesses`	Permissions of users on sensors are kept in this table. Each access determines users' access rights on sensors by using roles.

Continued on next page

Table 5.3 – *Continued from previous page*

`application_sensor_accesses`	Permissions of applications on sensors are stored in this table. Once again, roles are utilized to specify access rights on sensors.
`sensor_data`	Generated sensor data is stored in this table. All data is saved with a timestamp. Applications obtains the sensor data from this table.

Table 5.4: Users Table

Field Name	**Description**
`id`	Primary key that defines record.
`username`	A unique name that defines member of the platform.
`first_name`	First name of the user.
`last_name`	Last name of the user.
`email`	User e-mail address.
`company`	Institution that user has affiliated with.
`password`	Encrypted user password used for authentication.
`access_token`	A unique and encrypted value used for authorization.
`registration_time`	Time when user has registered to the platform.
`valid`	Approval status of the user.

Table 5.5: Applications Table

Field Name	**Description**
`id`	Primary key that defines record.
`name`	Unique name for application.
`user`	Defines owner of application which exists in `users` table.
`access_token`	A unique and encrypted value used for authorization.
`registration_time`	Time when application has registered to the platform.
`valid`	Approval status of application.

Table 5.6: Sensors Table

Field Name	**Description**
`id`	Primary key that defines record.
`type`	Type of sensor (i.e. temperature).
`unit`	Type of measure for sensor.
`company`	Institution that owns sensor.
`latitude`	Latitude value of sensor location.
`longitude`	Longitude value of sensor location.
`access_token`	A unique and encrypted value used for authorization.
`registration_time`	Time when sensor has registered to the platform.
`valid`	Approval status of sensor.

Table 5.7: Roles Table

Field Name	Description
`id`	Primary key that defines record.
`name`	Name of the role.
`info`	Specifies whether access permitted for sensor meta data.
`status`	Specifies whether access permitted for status information.
`data`	Specifies whether access permitted for data.

Table 5.8: User_Sensor_Accesses Table

Field Name	Description
`id`	Primary key that defines record.
`user`	Value which refers to a row in `users` table.
`sensor`	Value which refers to a row in `sensors` table.
`role`	Value which refers to a row in `roles` table.
`valid`	Approval status of access.

Table 5.9: Application_Sensor_Accesses Table

Field Name	Description
`id`	Primary key that defines record.
`application`	Value which refers to a row in `applications` table.
`sensor`	Value which refers to a row in `sensors` table.
`role`	Value which refers to a row in `roles` table.
`valid`	Approval status of access.

Table 5.10: Sensor_Data Table

Field Name	Description
`id`	Primary key that defines record.
`sensor`	Value which refers to a row in `sensors` table.
`time`	Time of sensor measurement.
`valid`	Measured value sent by sensor.

5.1.1.4 Alarm and Notification Service (ANS)

The Alarm and Notification Service is responsible for responding to data requests and notifying an application in accordance with the conditions set by the users and the applications. There are multiple ways that applications may request data, and PCAD offers two communication methods for the applications to interact with the platform: connection-oriented or connectionless methods.

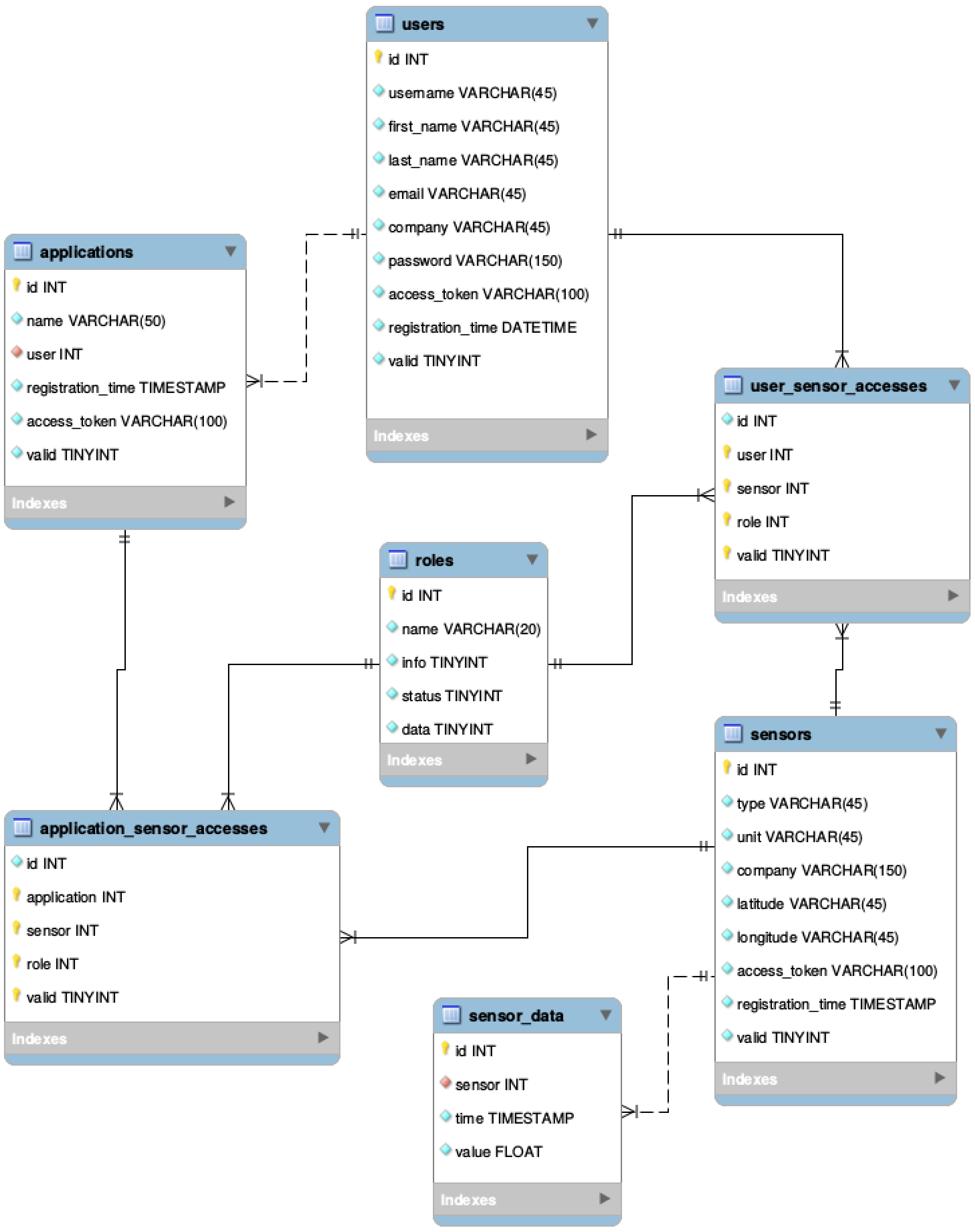


Figure 5.4: MySQL Schema

Connection-oriented Method In this method once a connection is established between an application and the platform, it remains active until it is terminated.

Connection-oriented communication is provided by opening a WebSocket connection with the platform. The main purpose of using connection-oriented communication is to provide continuous data transmission since there exists an *up* connection between the platform and the application.

This method can be operated in two different ways. In the first way as soon as sensor data is received by the platform, the raw data is delivered to client instantly, and in the meantime data is processed in parallel by the platform services, and eventually written to the database. The requesting client is blocked and waits until data is received from the sensor. This method of communication basically provides immediate data acquisition and is intended for applications which would like to receive data instantly as soon as a sensor sends data to the platform. In the second way, the data is read from the database instead of directly reading from a sensor, and the data is sent to the client at fixed intervals. In this way, the client is guaranteed to receive data at the end of a specified time interval. The client always receives the last data written to the database. If the queried sensor has not provided any new data to the platform within the given interval, the platform sends previously written, out-of-date data. It is the duty of the client to check whether data is up-to-date or not by looking at the timestamp of the data. Reading data from the database at periodic intervals is more suited for delay tolerant data delivery based systems.

Connectionless Method The connectionless communication method offers a simple data exchange. The data source is the database. It is a sessionless communication mechanism. The application sends a one-off request using the HTTP request/response mechanism and waits for a response. Once requested data is delivered to the client from the database, data exchange completes and the HTTP session is terminated. If one wants to receive one-time data directly from the sensor, the connection-oriented method should be used by establishing a connection and immediately closing the connection after data arrives.

The Alarm and Notification Service uses a rule file if one is specified. An application specifies the communication mechanism and a filtering criteria which is converted into a rule file by the application and sent to the platform. The ANS runs the rule file in cooperation with the Rule Service which is responsible for

Table 5.11: Interaction Mechanism Usage

	Connection-oriented	Connectionless	Filter	Real-time	Periodic
A1	+			+	
A2	+		+	+	
A3	+				+
A4	+		+		
B1		+			
B2		+	+		

running the rules. The applications may also request that data be sent without subjecting to any filtering, i.e. without specifying any rules.

An application exchanges data with PCAD using one of the six different interaction mechanisms:

A. Connection-oriented continuous data exchange between application and PCAD. This data exchange is carried out in four different ways.

  A1. Request data directly from the sensor without specifying any filter (real-time).

  A2. Request data directly from the sensor specifying a filter (real time).

  A3. Request data from the database without specifying any filter (periodic).

  A4. Request data from database specifying a filter (periodic).

B. Connectionless one-time data exchange between application and PCAD.

  B1. Request a one-time data from database without specifying any filter.

  B2. Request a one-time data from database by specifying a filter.

Table 5.11 lists a brief summary of which methods that each interaction mechanism utilizes when accessing the platform.

Table 5.12 shows entities and activities participate in connection-oriented interaction mechanisms A1 to A4. The shaded numbers indicate the entities that plays a role in the interaction and unshaded numbers indicate the activities.

Table 5.12: Interaction Mechanism Descriptions

#	Description
❶	Application: Applications that ask for sensor data.
❷	Rule file: This is the rule file created by the application.
❸	Parser: The rule file is parsed by the parser and transformed into functional elements according to syntactic and semantic rules.
❹	Rule object: The result of a parsed file is a rule object.
❺	Interpreter: The interpreter runs its own rule file and sends it to the rule handler.
❻	Rule handler: The rule handler executes the rules by importing the rule file and the sensor data.
❼	Scheduler: The scheduler forms the work piece according to a time schedule when there are periodic data requests.
❽	Database: holds data from sensors.
❾	Websocket: allows the application to send requests to the PCAD platform, and PCAD sends the requested data back to the application via the Alarm and Notification Service. A WebSocket is opened with a request to PCAD and it continues to feed data as long as connection is alive. Application decides when to stop, the transmission could end by a new API call.
①	The application is making a request from PCAD via a WebSocket using the interface functions in the application program.
②	The application creates the rule file.
③	The PCAD platform sends the data from the sensors to the rule handler.
④	The PCAD platform sends the data from the sensors through the rule handler.
⑤	The scheduler receives a rule sent by the rule handler.
⑥	The scheduler sends the data received from the database to the application via WebSocket, with or without considering the rule.
⑦	The Alarm and Notification Service sends the data to the application.

Figure 5.5 shows the activities of the first 4 interaction mechanisms that take place within the platform.

Interaction Mechanism A1 In interaction mechanism A1, an application requests data to be delivered directly from the sensor without specifying any filter. This mechanism is the simplest transfer method for sending unfiltered data. This method represents immediate data transfer. The data source is the sensor and the sensor data is sent directly to the application via WebSocket (Figure 5.5). The Rule Handler is idle in this mechanism. It follows the sequence ❶, ①, ❾, ❻, ③, ④, ❾, ⑦, ❶.

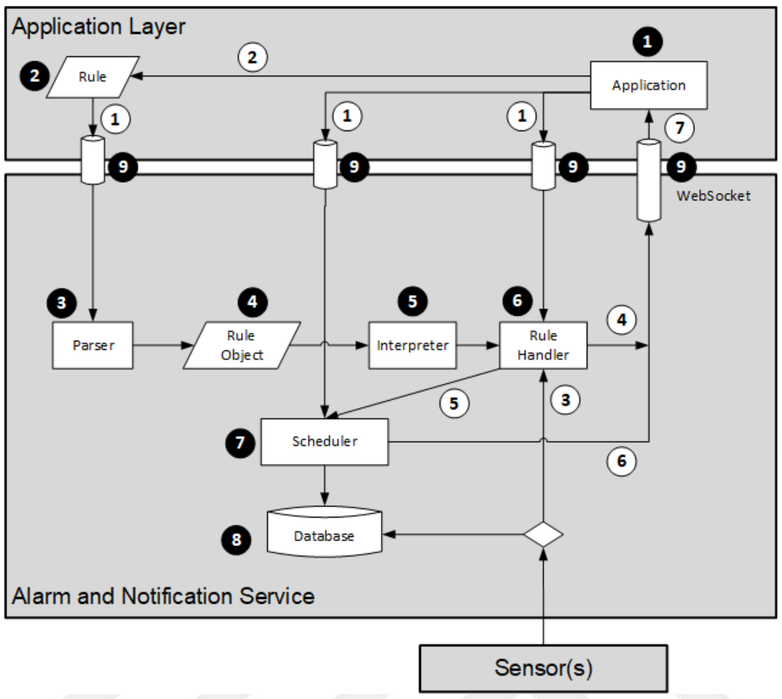


Figure 5.5: Connection-Oriented Interaction Mechanism Activities

Interaction Mechanism A2 When Interaction Mechanism A2 is used, an application requests data to be directly delivered from the sensor by specifying a rule file. The sensor data is filtered using the rule. To do this, the application sends a rule file along with the request. This rule file is first validated in the parser and transformed into a rule object. The rule object is run by the rule handler using the sensor data and then data is sent to the applications. It follows the sequence ❶ , ② , ❷ , ① , ❾ , ❸ , ❹ , ❺ , ③ , ❻ , ④ , ❾ , ⑦ , ❶ .

Interaction Mechanism A3 An application requests data to be delivered from the database at periodic intervals without specifying any rule file. The interval is determined by the application and sent to the Scheduler. Periodic sending process is done by the time scheduler. For instance, if an application requests to receive data every 5 minute, a task is created by the time plan implementer to request a database query from Data Management Service. The data is then sent to the requesters. This method represents delay tolerant data trasfer. It follows the sequence ❶ , ① , ❾ , ❼ , ❽ , ⑥ , ❾ , ⑦ , ❶ .

Query Payload

```
{
  "value": {
    "min": Float,
    "max": Float
  },
  "time": {
    "min": Timestamp,
    "max": Timestamp
  }
}
```

Figure 5.6: Query Payload Format

Interaction Mechanism A4 Interaction mechanism A4 is similar to A3, as they are both used to retrieve data from the database at certain intervals, however, mechanism A4 uses a rule file. After running this rule, the scheduler implements and executes the timely delivery of data. It follows the sequence ❶, ②, ❷, ①, ❾, ❸, ❹, ❺, ❻, ⑤, ❼, ❽, ⑥, ❾, ⑦, ❶.

Interaction Mechanisms B1 and B2 The previous 4 mechanisms, A1-A4, require a connection, and they maintain the connection until it is terminated by the client. The last two interaction mechanisms B1 and B2 do not use a continuous connection. A simple HTTP request/response mechanism is sufficient to implement the two mechanisms since they do not require uninterrupted communication. Therefore, there is no need to use WebSockets in B1 and B2. Database is the data source for both mechanisms. B1 does not specify any filter, whereas B2 uses a filtering mechanism albeit a very limited one. The filters are simple constraints enforced on the data. Filtering request is sent to the platform in the HTTP request payload. Query payload must be in JSON format given in Figure 5.6 so that filtering can be done adequately. The data that satisfies the filter conditions will be sent back in the response. When the filter is sent as part of the JSON query, data can be filtered based on two criteria: sensor data value and the time that data is obtained from the sensor. Each field has *min* and *max* boundary attributes which are optional. For example, if *min* of *value* is given as 10, the response data would contain sensor data which is greater than 10.

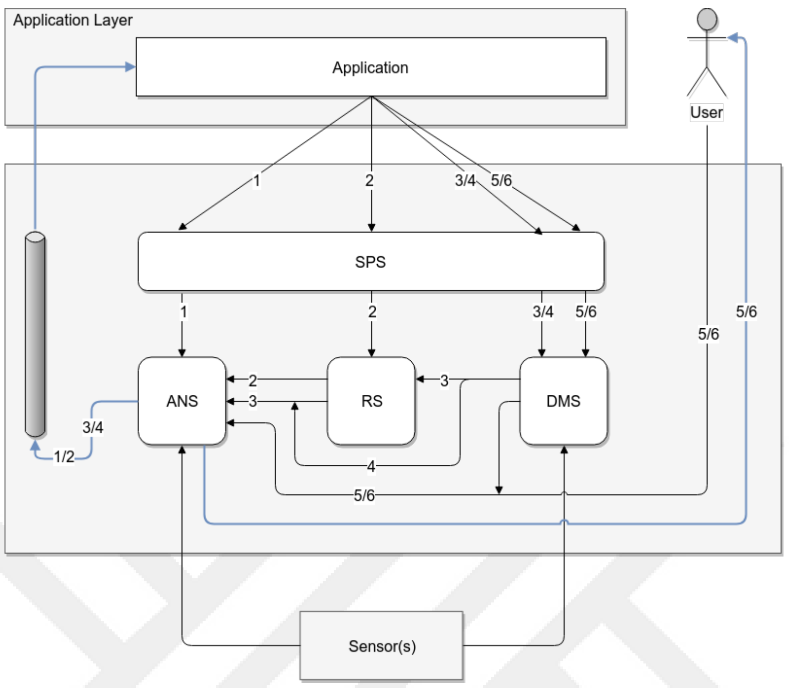


Figure 5.7: ANS Mechanism

Interaction Mechanism-Service Assignment When the clients interacts with the platform using one of the six mechanisms, the request is processed by the services of the platform after passing the security checks. Therefore, all requests coming to the platform use the Privacy and Security and Alarm and Notification services at a minimum. Mechanisms A1 and A2 take information directly from a sensor. Mechanism A1 does not filter data, while A2 uses a rule file to perform the filtering. Therefore, mechanism A2 uses functions provided by the Rule Service. Mechanisms A3 and A4 use database as the data source and send data to the user at fixed intervals using a Scheduler. They all use services of the Data Management Service which is responsible for retrieving the data from the database. Similar to mechanism A2, A4 uses rule service when data filtering is requested. Since mechanisms B1 and B2 use the HTTP request/response mechanism, there is no need for a connection establishment. In the last two mechanisms, data filtering is provided in a very simple and limited way, which does not require a rule file. For this reason, the Rule Service is not used. The basic filtering information is sent in the load section of the HTTP request. Figure 5.7 shows which services are used in the 6 interaction mechanisms described above. Table 5.13 gives mechanism to service usage mapping.

Table 5.13: Interaction Mechanism Service Usage Mapping

Interaction Mechanism	Rule Service	Data Management Service	Alarm and Notification Service	Security and Privacy Service
A1			X	X
A2	X		X	X
A3		X	X	X
A4	X	X	X	X
B1		X	X	X
B2		X	X	X

The Alarm and Notification Service of the platform uses *subscriber* and *publisher* paradigm [57] when transferring data between the data source and application. The Publisher-Subscriber paradigm creates a data stream between the data source and applications. In this paradigm the application has the role of subscriber and publisher acts like a front end for the real data source. Data source could be a sensor, database or some other third party repository. The data source sends data to its assigned publisher, and the publisher forwards it to its subscribers. The Publisher could pass sensor data to Subscribers as it is or after processing it based on rule execution. The application connects to PCAD and registers itself as a Subscriber. The application sends a message that includes the data source it wants to connect, the mode of operation and optionally a rule file. Using this information ANS creates and configures a Publisher associated with a Subscriber.

There are two modes for passing data: real and periodic. If the mode of operation specifies that data to be retrieved from the database at periodic intervals, a *Task* actor needs to be created. The *Task* actor executes in the background and remains alive as long as client connection is active. It fetches and sends data periodically. In such a case a Task-Publisher mapping is created with *Task* being the data source. The *Task* actor behaves like a facade acting as a data source on behalf of the database. From that point on, data sources, either *Task* or *Sensor* sends data to ANS via their assigned *Publisher*. *Publisher* uses the Rule Service and its rule engine if data filtering is needed. ANS conveys the final result to the *Subscriber* which is eventually delivered to the application. Since all messages are conveyed via ANS and the state of ANS keeps the subscriber-publisher mappings, the destination subscriber is correctly determined.

The Akka system scheduler is used for executing *Tasks* which are created for

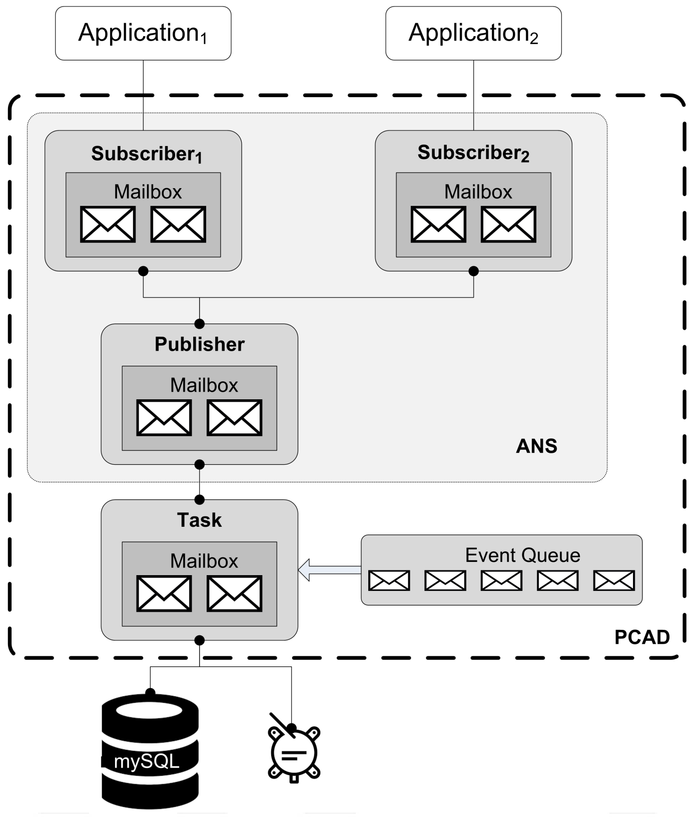


Figure 5.8: Task-Subscriber-Publisher Interaction.

the purpose of sending data at periodic intervals. The scheduler maintains an event queue and runs when necessary. Each event is created with the interval and actor reference information. Handling an event means sending a message at certain intervals to the *Task* Actor assigned to do work. For example, when an event's interval is set to 3 minutes, the result of handling this event is to send a message to the *Task* instance reference associated with the event every 3 minute. Upon receiving the message, the *Task* instance runs a query and pushes the result to ANS which delivers it to the *Publisher*. The publisher returns the result to the application via its Subscriber. The *Task-Publisher-Subscriber* relationship is shown in Figure 5.8 The sensors or the database feed *Task* with data which relays it to its assigned Publisher every so often triggered by an event generated by the Scheduler. The Publisher, via its assigned Subscriber forwards data to the correct application.

ANS maintains a state to hold the subscriber-publisher mapping. It is the only actor that has a state. Other actors do not have states because, services they implement do not require maintaining a state. ANS state is shared, mutable and updated dynamically. Therefore, ANS actor is implemented as a Singleton. Since its state is shared and multiple actors may attempt to change the state, the state changes must be protected against concurrent modifications. Actors can change their state in response to messages arriving at their mailboxes. Due to the nature of the mailbox (i.e. queue), messages are serialized upon entry to the queue. Messages are processed in the order they are placed in the queue and processing of a message can not be started before the previous one is completed. When a request asking for data from a data store is received by the platform, a pair of subscriber-publisher actor is created and the state of the ANS is updated to include this mapping.

Data through a Publisher does not have to come from just one source. When necessary, publisher processes data from multiple sources and send them to subscribers i.e. applications. Since multiple data sources can send data to the same application, another mapping is needed to map the real data source to a publisher as shown in Figure 5.9

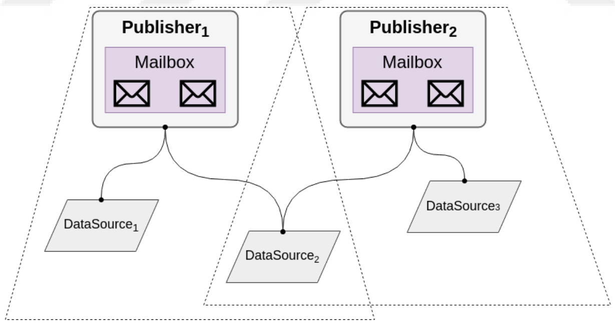


Figure 5.9: Multiple Source Using One Publisher.

Filtering operations Filtering has two forms: advanced filtering and simple filtering. Advanced filtering is done through the rule file described in the Rule Service section. The rule file can only be specified if the programming APIs provided in the Python library are used. Simple filtering is provided in the JSON load when using REST-API. In the real-time case, the data sent by the platform sensors are processed instantly, whereas in the periodic case, the data is processed using the interval specified by the applications and the database is used as the data source.

5.1.2 Sensor and Application Bindings

Several alternatives are researched to choose a communication protocol to use in the binding layer when transferring data to and from the platform. Communication protocols considered are listed below:

MQTT[58]: MQTT protocol is simply a publisher-subscriber model for messaging. It offers a lightweight solution for delivering messages. It is ideal for working with limited resources.

XMPP[59]: It is a real-time messaging protocol and mostly used in chat applications. It uses XML for exchanging messages and publisher-subscriber messaging model.

CoAP[60]: It is a lightweight RESTful interface for devices with limited resources. It uses the HTTP protocol but it works using UDP rather than TCP. Besides, it consumes low amount of resources comparing to the standard RESTful protocol.

AMQP[61]: It is similar to MQTT protocol, and has publisher-subscriber architecture. Besides, it has a queuing feature which is based on "topics". It offers security and reliability.

REST[62]: REST stands for REepresentational State Transfer and uses the HTTP protocol. It is a widely used and well-known model in building modern web applications. Both XML and JSON data format can be used for data transfer.

WebSocket[63]: It enables a full-duplex communication over sockets and makes server to send messages to clients, unlike HTTP protocol. For real-time communication, WebSocket is a good option. RESTful HTTP and WebSocket are chosen for this project. They are well-known, easy to understand and widely used mechanisms to exchange data. Other protocols are also acceptable candidates, however, they would not be as easy to implement when compared to the two protocols mentioned before. Websockets are used to implement real-time communication [48] in PCAD.

RESTful based communication is gaining momentum in standard web technologies for the primary reason that it is based on the HTTP protocol and provides a much simpler use than the SOAP based methods. HTTP is the de-facto internet standard and is universally supported by the operating systems and programming languages. RESTful APIs are indispensable components for single-page-applications. Single-page-application is a web application that the business logic take place on the client-side, and it interacts with the server-side only when data is needed. This is the perfect mechanism that fits for PCAD, and when we provide RESTful API, the platform will be suitable for up-to-date application development trends.

Akka's design goal is to provide a concurrent computation environment to develop scalable applications. It does not provide an extensive and robust functionality to support HTTP [50] and WebSocket [63] protocols. To remedy the absence of HTTP and WebSocket support, Play Framework [64] is used as it has support for both RESTful architecture and WebSockets and integrates these web protocols with Akka. Play Framework offers ready to use off-the-shelf features for Akka such as comprehensive testing libraries, encryption for security, and actor pooling for scaling platform. It is based on the Model-View-Controller [65] architectural pattern. The platform functions are represented as resources and accessed via service endpoints which are references to Uniform Resource Identifiers. Play uses URL which is a specific type of URI to send web requests using the underlying HTTP protocol. Each request contains a URL, a method (i.e. *PUT,GET,POST,DELETE*), list of headers and payload. Every URL is mapped to a controller action in Play and this mapping is called a *route*. The *router* is the component in charge of translating each incoming HTTP request to an action.

When a request is received, the controller first preprocesses it, then passes it to Akka actor system and finally returns the response received from the Actor system back to the client. Controllers in Play framework provide access to Actor system by converting web requests to messages so that actors could understand. Thus, controllers compose messages and pass them to actors. After a request arrives to an actor, it performs a task or a service as explained in Chapter 5. To summarize, the main component to bind HTTP requests to Akka is the route endpoints which defines a URL, an HTTP method, and a controller which handles requests. Each route endpoints forms a universal PCAD RESTful API. A sample RESTful API is shown below where secure access is achieved through the use of access token:

```
http://pcad.com/users?access_token=eykls12930KIM.9023JKDF...
```

Except for the *signup* and *signout* requests shown in Table 5.14, all other requests must use a token and send it as a query parameter via the URL. The four interaction mechanisms - A1 to A4 - mentioned in the Alert and Notification Service are handled using WebSocket protocol. There is only one URL for the WebSocket protocol. It is used to initialize WebSocket connection and keep it active until the connection is closed. This is in contrast to the HTTP protocol where no state information is kept from one request to the other. An example URL for a WebSocket connection is given below in which a session token is sent along with the URL when establishing a connection.

```
ws://pcad.com/sensors/stream?access_token=eykls12930KIM.9023JKDF...
```

Table 5.14: RESTful API

URL	**Method**	**Payload**	**Description**
`/auth/signup`	POST	`NewUser`	Registers new user.
`/auth/signin`	POST	`Credentials`	User, Application, and Sensor sign in.
`/auth/signout`	GET	-	User signs out.
`/users`	GET	-	Returns users.
`/users/:id`	GET	-	Returns a user.
	PUT	`User`	Updates a user.
	DELETE	-	Deletes a user.
`/sensors`	GET	-	Returns sensors.
	POST	`NewSensor`	Creates new sensor.
`/sensors/:id`	GET	-	Returns a sensor.
	PUT	`Sensor`	Updates a sensor.
	DELETE	-	Deletes a sensor.
`/applications`	GET	-	Returns applications.
	POST	`NewApplication`	Creates new application.
`/applications/:id`	GET	-	Returns a application.
	PUT	`Application`	Updates a application.
	DELETE	-	Deletes a application.
`/roles`	GET	-	Returns roles.
	POST	`NewRole`	Creates new role.
`/roles/:id`	GET	-	Returns a role.
	PUT	`Role`	Updates a role.
	DELETE	-	Deletes a role.
`/permissions/users`	GET	-	Returns user permissions.
	POST	`NewUserPermission`	Creates new user permission.
`/permissions/users/:id`	GET	-	Returns a user permission.

Continued on next page

Table 5.14 – *Continued from previous page*

URL	Method	Payload	Description
	PUT	`UserPermission`	Updates a user permission.
	DELETE	-	Deletes a user permission.
/permissions/applications	GET	-	Returns application permissions.
	POST	`NewAppPermission`	Creates new application permission.
/permissions/applications/:id	GET	-	Returns a application permission.
	PUT	`AppPermission`	Updates a application permission.
	DELETE	-	Deletes a application permission.
/sensor/:id/data	GET	-	Returns sensor datas.
	POST	`SensorData`	Creates new sensor data.
/sensors/:id/data/filter	POST	`Filter`	Returns sensor data.
/sensors/:id/info	GET	-	Returns sensor info.
/sensors/:id/status	GET	-	Returns sensor status.

A complete list of URLs is shown in Table 5.14. For instance, the first row in the table presents an API for signup where */auth/signup* is the URL, *POST* is the method to use, and *NewUser* is the payload information. The payload information is given in Table 5.15. The programming examples that demonstrate RESTful API use can be found in Appendix A.

Table 5.15: Payloads

NewUser

```
{
  "first_name": String,
  "last_name": String,
  "email": String,
  "company": String,
  "username": String,
  "password": String
}
```

User

```
{
  "id": Number,
  "first_name": String,
  "last_name": String,
  "email": String,
  "company": String,
  "username": String,
  "access_token": String,
  "registration_time": Date,
  "valid": Boolean
}
```

Continued on next page

Table 5.15 – *Continued from previous page*

NewSensor

```
{
  "type": String,
  "company": String,
  "unit": String,
  "latitude": String,
  "longitude": String
}
```

Sensor

```
{
  "id": Number,
  "type": String,
  "company": String,
  "unit": String,
  "latitude": String,
  "longitude": String,
  "access_token": String,
  "valid": Boolean
}
```

NewApplication

```
{
  "user": Number,
  "name": String
}
```

Application

```
{
  "id": Number,
  "user": Number,
  "name": String,
  "registration_time": Date,
  "access_token": String,
  "valid": Boolean
}
```

NewRole

```
{
  "name": String,
  "status": Boolean,
  "info": Boolean,
  "data": Boolean
}
```

Role

```
{
  "id": Number
  "name": String,
  "status": Boolean,
  "info": Boolean,
  "data": Boolean
}
```

NewUserPermission

```
{
  "user": Number,
  "sensor": Number,
  "role": Boolean
}
```

UserPermission

```
{
  "id": Number,
  "user": Number,
  "sensor": Number,
  "role": Boolean,
  "valid": Boolean
}
```

Continued on next page

Table 5.15 – *Continued from previous page*

NewApplicationPermission

```
{
  "application": Number,
  "sensor": Number,
  "role": Boolean
}
```

ApplicationPermission

```
{
  "id": Number,
  "application": Number,
  "sensor": Number,
  "role": Boolean,
  "valid": Boolean
}
```

SensorData

```
{
  "sensor": Number,
  "time": Date,
  "value": Number
}
```

Filter

```
{
  "sensor": Number,
  "value": {
    "min": Number,
    "max": Number
  },
  "time": {
    "min": Date,
    "max": Date
  }
}
```

5.2 Client-Side of Platform

Client-side is the front-end of PCAD and made up of two main components: programming language bindings (i.e. Application Programming Interface) and user interface bindings (Graphical User Interface). The programming interfaces are used for developing sensor and application software. User interface is the web panels for user and application management, and it has certain features to ease platform management. Application Programming and Graphical User Interfaces are explained in the following sections.

5.2.1 Application Programming Interface Bindings

PCAD provides a library for applications and sensors to use the services of PCAD. Applications can retrieve data or sensors provide data to platform using one of the following two methods:

- Using a set of library functions written in Scala/Python
- RESTful API (HTTP Request/Response)

RESTful API is used to perform various actions on the server using HTTP requests. All capabilities of PCAD are available through the API. Each HTTP REST request has a method type, URL and a payload. Required elements of a request has to be defined properly, otherwise, requested action will not be completed successfully. Alternatively, the application programming interface can be used to access the platform services. Programming interface is used for authentication, querying data and getting real time data. Programming language bindings provide a better development interface than RESTful API. By using programming bindings, developers are less prone to errors when communicating with PCAD. Similar to applications, sensors can send data to the platform using both the programming language bindings or RESTful API. Sensor software only needs authentication and data posting features, therefore the number of available library functions are less than the ones available for applications. In Figure 5.10, we can see how clients, sensors and applications, access the platform.

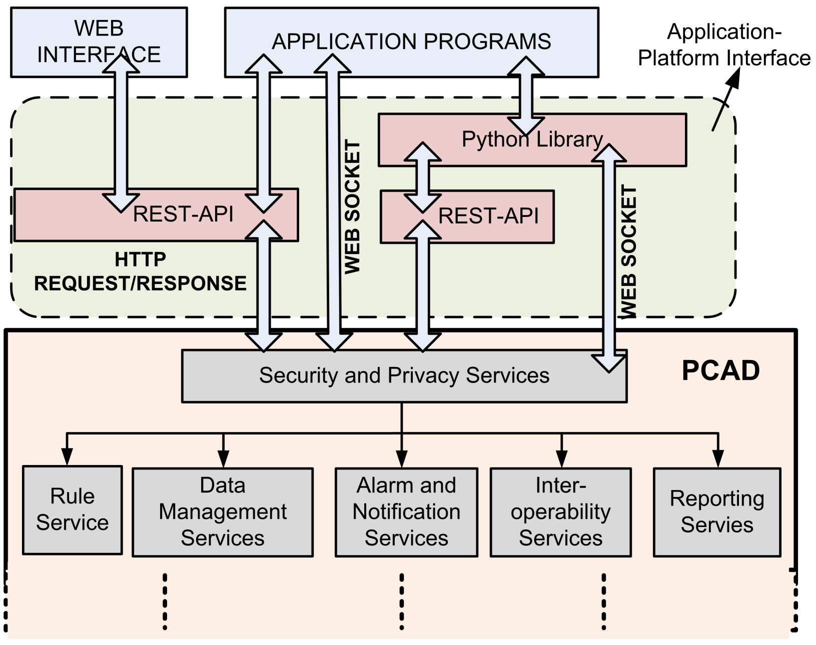


Figure 5.10: Application Platform Binding Protocols

The biggest rationale in using a library when communicating with the PCAD platform is provisioning of a data channel to the sensors and applications to enable instant data delivery to and from the platform. When applications and sensors need to communicate with the platform for instant data delivery, application programming interface must be used. This type of communication is supported via WebSocket protocol [63]. It establishes a continuous connection between clients and server, where PCAD is the server and sensors and applications are the clients. PCAD plays the conveyor role among sensors and applications. When using WebSocket for communication, clients connect to the platform first. Every connection instance runs in its own thread. The applications can get data from multiple sources simultaneously, if they are configured so. To receive data, the applications initialize a context to store session parameters between the application and the platform after a successful authorization. This connection context is used in subsequent communications. When the connection is established, the application specifies the payload to specify communication parameters by using a connection instance. The same connection instance could be used in receiving data from different resources. All an application has to do is, to specify different payloads for different resources. Callback function mechanism is used to send data to the application. Platform calls the callback function specified by the application whenever data is ready. Callback function handles data and performs the operations that it is assigned to, such as for example drawing real-time graph. A summary of the API is given in Table 5.16. The full programmer's guide is given in Appendix A.

Table 5.16: Application Programming Interface

Name	**Parameter**	**Description**
`pcad_init_context`	*`config_file_path`*	Library function to initialize communication between the platform and the clients (i.e sensors and applications). `config_file_path` parameter is used to deliver the configuration data needed to start a communication. The file contains access token information which is unique for each client.
`pcad_get_credentials`	*`context`*	Function to obtain session token after authentication using the access token. Session token is special one that has expiration time. When the session token expires, it must be re-acquired again by using this library method.
`pcad_free_context`	*`path`*	Function for cleaning session token footprint on the disk.

Continued on next page

Table 5.16 – *Continued from previous page*

`pcad_post_data`	*`sensor_data`*	Function that sends sensor data to the platform. It accepts data in JSON format.
`pcad_init_connection`	*`secure_context`*	It starts the communication between an application and the platform for notification purposes. Communication is based on WebSocket protocol and remains alive until terminated.
`pcad_attach_sensor`	*`payload, rule_path`*	Function to subscribe to a data source (i.e. sensor). Optionally, a rule can be passed to filter sensor data. Payload parameter specifies the `mode` of notification. If *mode* is *periodical*, an interval must be given as well. The other mode is `real`.
`pcad_on_message`	*`connection, callback`*	Function to receive notification messages. By using this method, an application enables listening of incoming messages. Received messages are passed to a callback function which is specified as a parameter.
`pcad_terminate _connection`	*`connection`*	Function to terminate a communication.
`pcad_send_query`	*`secure_context, sensor, filter`*	Function to query the sensor data and time. Both data and time may have maximum and minimum values to limit query. Response returned is in JSON format. This function performs a one time query which does not require a continuous connection.

An example program in Python language that illustrates the use of the API to establish communication between application and the platform is shown below. Other example programs can be found in Appendix A.

The following sequence of events takes place in the program shown in Figure 5.11

1. Authentication to the platform is done using `pcad_init_context`, and `pcad_get_credentials` function calls.

```
INSTANCE = Connection()
CONTEXT = INSTANCE.pcad_init_context("application_config.json")
SECURE_CONTEXT = INSTANCE.pcad_get_credentials(CONTEXT)
```

As a result a secure context is established between the application and the platform.

2. `pcad_init_connection` call establishes a new communication channel. Once the channel is established, data flow from the platform to the application takes place.

   ```
   CONNECTION = INSTANCE.pcad_init_connection(SECURE_CONTEXT)
   ```

3. Data request is made by calling `pcad_attach_sensor` method. The sensor to be attached to and other information are provided in the payload as arguments to the method. The method takes the connection instance as the first parameter which is initialized as a result of a prior `pcad_init_connection` call used to establish data exchange channel. The second parameter is of dictionary data type which stores mode, sensor key-value pairs, and optionally an interval. Interval is only required when the mode is 'periodic'. The third parameter is the path of a rule file, which is by default `null` when not provided.

   ```
   INSTANCE.pcad_attach_sensor(CONNECTION, {"mode":"real", "sensor": 19})
   ```

4. The `pcad_on_message` method allows the processing of incoming messages via the active connection. For example, a function that outputs incoming messages to the screen is given as a parameter to this method and the messages is printed on the screen.

   ```
   INSTANCE.pcad_on_message(CONNECTION, print_message)
   ```

5. `pcad_terminate_connection` is used to end an active connection.

   ```
   INSTANCE.pcad_terminate_connection(CONNECTION)
   ```

5.2.2 User Interface

This section describes the web interface for the platform users. The web interface is implemented as a separate package. It also communicates with the platform using RESTful API. It is primarily intended for performing administration tasks such as registration of users, applications, and sensors, password management, authorization, listing sensor information and update. There are two kinds of users defined on the system: *user* and *admin*. The web panel has an adaptive interface which alters its presentation based on user type. The web panel is implemented

Real-Time Application Program

```
# -*- coding: utf-8 -*-
from connection import Connection

def print_message(message):
  print(message)

if __name__ == "__main__":
  try:
    INSTANCE = Connection()
    CONTEXT = INSTANCE.pcad_init_context("application_config.json")
    SECURE_CONTEXT = INSTANCE.pcad_get_credentials(CONTEXT)
    CONNECTION = INSTANCE.pcad_init_connection(SECURE_CONTEXT)
    INSTANCE.pcad_attach_sensor(CONNECTION, {"mode": "real", "sensor":
    ↪  19})
    INSTANCE.pcad_on_message(CONNECTION, print_message)
  except (Exception, KeyboardInterrupt, SystemExit) as e:
    INSTANCE.pcad_terminate_connection(CONNECTION)
```

Figure 5.11: Example Real-Time Application Program

using AngularJS [66]. It is a JavaScript framework to develop web applications, in particular front-end development. Single-page-application is used for comprising application which share same principles and features. It has a simple working principle. Basically, one page is presented by server side and all of other functionalities are provided by client-side business logic. Only data requests is made to the server and the page dynamically alters its behavior based on response data. Therefore, the single-page-application concept perfectly fits for the RESTful interface because, there is no need for rewriting or refactoring server-side logic to create such a user interface. The web panel user interface screen shots are give in Appendix B. Each interface was created by using Javascript and HTML [67], it has no extra burden on server-side.

Chapter 6

Tests

This section describes the test methodology utilized in testing PCAD platform based on the IEEE Std 829-2008 standard [68]. The IEEE standard specifies the testing process in four phases: Test Planning, Test Design, Test Execution and Analysis of Results. The platform is tested to determine whether it is performing its expected functions and to reveal bugs present in it. The tests are grouped into four categories:

- API tests
- GUI tests
- Integration tests
- Performance tests

The API tests are intended to test the RESTful interface which is the mechanism to connect to the platform. The GUI tests are executed to test the web interfaces of the platform. The Integration tests ensure that the system performs correctly as a whole and the platform services provide the required functionality working individually or as an aggregate. The overall performance of the platform is verified via performance tests. In order to carry out the tests presented above, first inputs, operations and outputs related to the test cases were specified and developed during the design stage. Following the design phase, the test cases are executed, necessary corrections and changes were made based on the bugs discovered and results are documented. The testing process concluded when 95 percent success rate has been reached without any severe defect left unresolved. Thus, the testing process is completed.

6.1 Test Design

Test were written for the four categories given above. API and Integration tests are common. The user interaction in the GUI tests are done automatically by the help of a test library to test functional correctness of the user interface. However, style related verification is done visually. Performance tests are done by simulating the data source.

6.1.1 API and Integration Tests

This section describes the conversion of the application programming interfaces presented within the RESTful interfaces of the PCAD platform into concrete test conditions and test cases. These test cases also serve as the API and integration tests. Since the programming interface of the platform is based on REST, HTTP protocol is used when making a request. Each test case consists of 8 fields: test case ID, purpose, method used (e.g. POST, GET), HTTP request URL, query parameters, payload, response code and content. A total of 87 API and Integration test cases are developed.

Table 6.1, presents a sample test case with actual contents used in testing. While, purpose, method, and URL fields may contain identical information spanning multiple test cases, input/output parts will vary from one test to another. The query parameter refers to the entries given in the URL and are indicated by '-' if they are not needed. Payload shows the input sent in the payload part of the request. Code and content are the results returned in the HTPP response. The code and content values returned must be exactly same as the ones specified in the test cases. Otherwise, the test case causes a failure.

Table 6.1: Example Test Case

ID	Test Case 35.1
Purpose	To post sensor data
Method	POST
URL	/api/v1/sensors/:id/data
Input : Query	id:1, access_token=VALID_SESSION_TOKEN
Input : Payload	`{sensor:1, time: "2016-02-15 13:10:12", value: 23.4}`
Output : Code	*200*
Output : Content	`1 (Randomly generated sensor data id.)`

The goal of integration testing is to ensure that all four platform services interact with each other, with applications and sensors as designed, and the platform functions correctly as a whole. Even though integration tests are given as a separate category, no separate test cases are written for the integration testing. API tests are used for integration testing as well. As stated in previous chapters, four services have been implemented in the platform: DMS, ANS, SPS, RS. The aim of integration testing is to exercise these services. However, not every service within the platform communicates with every other service. HTTP requests sent to the platform exercises a subset of these services. Table 6.2 presents the REST-API endpoints used by these services and the services that are exercised during processing a request. Each service indicated with a check box in the table must work correctly, and participate in handling the request otherwise, it would cause the request fail. For example, writing sensor data on the database depends on SPS and DMS, failure of this of request means either SPS or DMS has failed individually or these two services are not working correctly together.

The database should be populated with sample data prior to the tests by running `init_test_db.sh` script. This script populates `user`, `user_permissions`, `application_permissions`, `sensors`, `applications`, and `roles` tables. Tests that require authentication, there must exist a `VALID_SESSION_TOKEN` which is sent along with the URL to complete the authentication process. `VALID_SESSION_TOKEN` is obtained by making a `POST` request on `/auth/signin` endpoint. Test cases also require an `INVALID_SESSION_TOKEN` to test error cases. A random string value is sufficient to make a token invalid.

6.1.2 GUI Tests

This section provides the test cases to test whether web interfaces of the platform function correctly or not and, whether they perform the functions expected from them.

GUI tests are automated with the help of a sophisticated user interface testing library. It is a full package for testing browser based applications, This library provides a utility that simulates browser actions programatically. Due to this library, GUI test are automated without any user intervention. For instance, filling form inputs, clicking buttons, submitting forms and executing background tasks such as listening HTTP responses, AJAX requests are done by the browser automatically. However, it requires to precisely specify each and every event sequence generated by the user. For each GUI test we have listed every action that must be done such as opening a page, clicking a button, filling a form then submitting it. Each test required at most 4 steps. In total 69 test cases are developed.

GUI test cases do not include testing of style-related properties, such as positioning of widgets, web page color, sizes etc. Such style related properties are verified visually during the tests by the tester performing the tests. For this purpose, the screenshots of the web interfaces of the platform were taken and the test cases were exercised visually. Appendix B lists the screen shots of the web interface. If the interface looked peculiar or exhibited incorrect visual behavior such as inconsistent font size, the issue is fixed immediately. The test case is not marked as a failure however, as the function worked correctly.

Table 6.2: API Tests and Services

Inputs				Services			
URL	Method	Query	Payload	SPS	DMS	RS	ANS
/auth/signup	POST	-	+	+	+	-	-
/auth/signin	POST	-	-	+	+	-	-
/auth/signin	POST	+	-	+	+	-	-
/auth/signout	GET	+	-	+	-	-	-
/users	GET	+	-	+	+	-	-
/users	POST	+	+	+	+	-	-
/users/:id	GET	+	-	+	+	-	-
/users/:id	PUT	+	+	+	+	-	-
/users/:id	DELETE	+	-	+	+	-	-
/sensors	GET	+	-	+	+	-	-
/sensors	POST	+	+	+	+	-	-
/sensors/:id	GET	+	-	+	+	-	-
/sensors/:id	PUT	+	+	+	+	-	-
/sensors/:id	DELETE	+	-	+	+	-	-
/sensors/stream	GET	+	-	+	+	+	+
/applications	GET	+	-	+	+	-	-
/applications	POST	+	+	+	+	-	-
/applications/:id	GET	+	-	+	+	-	-
/applications/:id	PUT	+	+	+	+	-	-
/applications/:id	DELETE	+	-	+	+	-	-
/permissions/users	GET	+	-	+	+	-	-
/permissions/users	POST	+	+	+	+	-	-
/permissions/users/:id	GET	+	-	+	+	-	-
/permissions/users/:id	PUT	+	+	+	+	-	-
/permissions/users/:id	DELETE	+	-	+	+	-	-
/permissions/applications	GET	+	-	+	+	-	-
/permissions/applications	POST	+	+	+	+	-	-
/permissions/applications/:id	GET	+	-	+	+	-	-
/permissions/applications/:id	PUT	+	+	+	+	-	-
/permissions/applications/:id	DELETE	+	-	+	+	-	-
/sensors/:id/status	GET	+	-	+	+	-	-
/sensors/:id/info	GET	+	-	+	+	-	-
/sensors/:id/data	GET	+	-	+	+	-	-
/sensors/:id/data	POST	+	+	+	+	-	-

6.1.3 Performance Tests

This section presents the results of tests related to performance. The goal is to determine data transfer rate and any computing bottleneck present in the services. We have measured the response time as the amount of time that have elapsed from the time a data provider sends data to the system till the time data is written to the database. In order to load the system, a number of sensors are simulated, sending concurrent requests over a period of time to verify the expected write times to the database. Table 6.3, shows performance results. Latency between submission of data by the provider and storing it to database is given in Elapsed Time column. This value is calculated by taking the average of elapsed times for each individual data write. Sensor column specifies the number of simulated sensors to generate the required data. Data Quantity column specifies total number of data write requests sent to the platform. For performance tests, a program was created to simulate sensor behavior by randomly generating values as sensor data. The data is sent using the Application Programming Interface given in Appendix A. Using a program to simulate sensors allows us to better control loading of the system, despite the fact that values used are not real, but random.

Table 6.3: Performance Test Results

# Sensor	Data Quantity	Elapsed Time(ms)
1	30	3
10	300	56
50	1500	52
100	3000	51
250	7500	60
375	11250	66
500	15000	87

6.2 Test Analysis and Results

87 test cases for API Testing and 69 test cases for GUI, with a total of 156 test cases were developed. The testing libraries provided in the Play Framework (Web application framework) for Java and Scala are used to run tests. API and GUI test were done programatically. API tests simulates HTTP actions, GUI tests simulates browser actions which is explained in GUI tests section.

Upon completion of the platform development, test cases have been executed which the results are shown in Table 6.4. Some errors from the test runs were not due to platform specific bugs but rather caused by the incorrect interpretation of the results by test cases. For instance an HTTP request that has an incorrectly formatted JSON payload causes HTTP error with a code of 500. On the other hand, test cases were programmed expecting a response code of HTTP 400. This mismatch caused many failures despite the fact that platform functioned correctly (i.e. correctly detected the invalid JSON payload). Faults in GUI tests are due to form inputs. Form input values were not using the related JSON variable correctly, for example, JSON must have property called 'sensor', but it was written as 'sensorId'. Similar kind of errors caused failures, and they were fixed immediately. After the correction of errors, the test cases were executed again and the obtained results are presented in Table 6.5. The PCAD platform has successfully passed the acceptance tests.

Table 6.4: Beta Test Results

Tests	Planned	Executed	Successful	Failure
API	87	87	73	14
GUI	69	69	59	10

Table 6.5: Acceptance Test Results

Tests	Planned	Executed	Successful	Failure
API	87	87	87	0
GUI	69	69	69	0

As a result, many implementation errors were revealed by the tests and necessary corrections were made. After the revisions, test cases were run again and the system was found to be working without error with respect to the designed test cases. The performance tests were recorded as benchmark for further development of the platform, therefore when a new version of the platform is completed, the current performance could be compared against the benchmark to determine how new features would affect the system. In fact, this approach has been used during development. New features of the platform were added step by step, and performance tests were repeated after every improvement in order to monitor the performance. Only trivial performance differences were observed in each new feature addition.

Chapter 7

Conclusion and Future Work

This thesis presents the design and implementation of a context-aware application development platform - PCAD. Its primary goal is to relieve context aware applications from acquisition and storage of data, thus allowing rapid application development with minimal effort. The platform also provides data providers with a simple-to-use interface to store their data on the platform. The platform acts as a middleware between data providers and data users and was inspired by operating system design. It adapts a service based approach. The platform functions are designed as services, with each service assigned to perform a specific task. The service based approach enables loose coupling between transacting entities within the platform thus making the system flexible and extensible. New features can be easily added as a service.

Services in PCAD are modeled using the Actor model. In addition to many other advantages, the Actor model provides load distribution to other machines thus making the system scalable. Actors use a small footprint and due its asynchronous message passing capability, an Actor does not exhibit some potential problems one may see in parallel computation. PCAD is implemented using the Akka toolkit, Scala language and Java Virtual Machine Platform. The Akka toolkit is the reference implementation for the Actor Model. Several other supporting technologies played a part in PCAD implementation; Play framework supports HTTP, WebSocket and RESTful architecture. Slick library and MySQL supports relational database storage.

The platform has implemented four services: Security and Privacy, Data Management, Rule, Alarm and Notification. The Security and Privacy service uses a simple Role Based Access Control scheme together with JSON web token to enforce authentication and authorization. The Data Management Service provides relational data storage functionality. A simple rule language and a rule engine is designed to filter data based on application specified criteria. The Rule, and Alarm and Notification services work together to send data to applications based on the criteria provided by the applications. The Alarm and Notification Service uses subscriber-publisher paradigm to connect data providers to data consumers. The remaining two services -Interoperability and Reporting- were architected and designed but not implemented for the reason that these services are considered lower priority in comparison to core platform features. They are planned to be integrated and implemented in the next release of the platform.

The platform has the following main benefits: it supports a diverse range of context sources, allows applications to access context sources synchronously or asynchronously using filters, decouples application code form context acquisition, and provides a simple rule engine for context processing. The platform supports instant data delivery requirements by opening a direct WebSocket channel between the context source and applications. The applications that are delay tolerant can use the RESTful API over HTTP Request/Response protocol to fulfill their data needs. Due to its architecture, the platform is flexible, scalable and extensible. The graphical and application programming interfaces provided by the platform facilitates rapid application development and delivers a easy-to-use user experience.

As a result, this thesis has achieved the objective of implementing a novel service based application development platform based on the notion of context-awareness.

7.1 Future Work

1. Akka toolkit is the reference implementation of the Actor model and is still being actively developed. Rather than using third party library support for HTTP as an add-on to Akka toolkit, using the version of Akka that has

built-in support for HTTP will reduce the overall footprint of the platform and make the implementation more compact and lightweight.

2. The rule language and rule engine will be enhanced to support more complicated filtering.

3. Context modeling and context reasoning functions are intentionally left out in current version since the focus of this work was to lay out the critical components of the platform. Currently, these functions are delegated to applications. In the future version of the platform it is planned to include some context modeling and reasoning activity in the platform.

4. Pluggable database support as described in Chapter 4 will be added. As a result NoSQL, cloud and other databases will be supported without the need for any changes to the core platform.

5. The current system is designed to work on a single machine. We consider adding support so the services can work on remote machines. This work will be based on remote actor feature of Akka that enables sharing tasks across multiple machines, thus enhancing scalability.

6. To improve scalability, demand based system tuning will be added by adding a configurable actor pool mechanism so that system performance will be enhanced and resource usage will be balanced.

Appendix A

Application Programming Interface Manual

Name:

pcad_init_context: Creates a context object which is used when calling library APIs.

Synopsis:

context = pcad_init_context(context_path)

Description:

It accepts a configuration file as a parameter and initializes the sensor connection interface. The configuration file which is in JSON format contains information about sensor which includes access_token for authentication.

Parameters:

context_path: The location of the configuration file.

Return Value:

It returns a context for to be used in subsequent API calls.

Example:

```
context_path = "/opt/pcad/config/app_config.json"
context = pcad_init_context(context_path)
```

Name:

`pcad_free_context`: It frees the specified context.

Synopsis:

`void pcad_free_context(context)`

Description:

It deletes `context` object.

Parameters:

`context`: The context acquired as a result of `pcad_init_context` call.

Return Value:

It returns void, it only logs.

Example:

```
context_path = "/opt/pcad/config/example.cfg"
context = pcad_init_context(context_path)
pcad_free_context(context);
```

Name:

`pcad_get_credentials`: Retrieves session credentials.

Synopsis:

`pcad_get_credentials(context)`

Description:

It takes context object as an argument to retrieve session credentials from PCAD. It returns the modified context object augmented with the session credentials. Unlike `access_token`, session credential has duration. When it expires new credentials must be acquired.

Parameters:

`context`: The context acquired as a result of `pcad_init_context` call. At the end of the call context will be modified with the session credentials.

Return Value:

It returns a secure `context` that contains session credentials.

Example:

```
context_path = "/opt/pcad/config/app_config.json"
context = pcad_init_context(context_path)
secure_context = pcad_get_credentials(context)
```

Name:

`pcad_init_connection`: It opens a connection between PCAD and application which will be used for continuous data transfer.

Synopsis:

`connection = pcad_init_connection(secure_context)`

Description:

It opens a connection with PCAD.

Parameters:

The secure `context` acquired as a result of `pcad_get_credentials` call.

Return Value:

It returns connection instance which ia used for data exchange.

Example:

```
context_path = "/opt/pcad/config/app_config.json"
context = pcad_init_context(context_path)
secure_context = pcad_get_credentials(context)
connection = pcad_init_connection(secure_context)
```

Name:

`pcad_on_message`: It receives incoming messages from PCAD.

Synopsis:

`pcad_on_message(connection, callback)`

Description:

It listens incoming messages coming through connection instance. It cannot be used without a connection instance. Therefore, firstly, a connection must be established.

Parameters:

`connection`: It is the connection handle that the applications receive messages and data. The connection handle is initialized by using `pcad_init_connection` method.

`callback(connection, message)`: callback function is taken as parameter. This function is written by the application developer. It will be registered to connection instance for listening messages. Method will take two parameters and the first one is connection handle, and other one is message received. The function uses message parameter to access to the data.

Return Value:

-1: for error

non-negative: Success

Example:

```
    .
    .
source = {"mode": "periodic", "sensor": 51 ,"interval": 5}
connection = pcad_init_connection(secure_context)
pcad_attach_sensor(connection,source,null)
def foo(conn, message):
        print(message)
pcad_on_message(connection, foo)
```

Name:

`pcad_attach_sensor`: Binds the application to particular sensor for data retrieval.

Synopsis:

`pcad_attach_sensor(connection, source, rule_file=None)`

Description:

It binds the application to a particular sensor for data retrieval. The applications use this call to get data and notifications for a particular sensor.

Parameters:

`connection`: It is the connection handle that the applications receive messages and data. The connection handle is initialized by using `pcad_init_connection` method.

`source`: The source of data. It is a JSON formatted object which specifies the data source. It has three fields: mode of transfer, sensor id and interval. A valid JSON source object is given in the following code snippet.

```
source = {
  "mode": 'real' | 'periodic',
  "sensor_id": Number,
  "interval": Number # minute value and only required for periodic
}
```

The `mode` field indicates if the data is to be sent directly from the sensor or not. If this field is set to ‘real’, platform sends data as soon as it becomes available (i.e. delivered by the sensor) . When this field is set to ‘periodic’, data will be delivered at predefined intervals from the database.

`rule_file`: It is a rule file to be interpreted by PCAD. If no rule file is given then this field must be specified as null.

Return Value:

`void`

Example:

```
       .
       .
source = {"mode": "real" , "sensor_id": 23}
connection = pcad_init_connection (secure_context)
pcad_attach_sensor(connection,source,null)
```

Name:

`pcad_terminate_connection`: Terminate connection.

Synopsis:

`pcad_terminate_connection(connection)`

Description:

Connection instance will be destroyed and communication will be ended.

Parameters:

`connection`: It is handle of the connection to be destroyed.

Return Value:

-1: for error

non-negative: Success

Example:

```
      .
      .
secure_context = pcad_get_credentials(context)
connection = pcad_init_connection(secure_context)
result = pcad_terminate_connection(connection)
```

Name:

`pcad_send_query` - It runs query on PCAD's sensor data.

Synopsis:

`pcad_send_query(secure_context, sensor_id, query)`

Description:

It is used for one time data requests. Querying pastime data is one of the use case for this function.

Parameters:

`secure_context`: The secure context acquired as a result of `pcad_get_credentials` call.

`sensor_id`: Sensor id.

`query`: It is a JSON object that filter data in database. It must be composed as following.

Return Value:

It returns JSON response which consists of related sensor data. It may return empty JSON object, if no data to be found.

Example:

```
    .
    .
filter = {
  "value": null,
  "time":          {
    "min": "2016-12-02T23:39:30.0+03:00",
    "max": "2016-12-07T23:39:30.0+03:00"
  }
}
response = pcad_send_query(secure_context, 1, filter)
```

Name:

`pcad_post_data`: Posts data to the platform.

Synopsis:

`pcad_post_data(context, data)`

Description:

Data sources use this API to post data to the platform. It accepts a context and data in JSON format and sends the data to the platform.

Parameters:

`context`: Configuration parameters which will be utilized for authentication.

`data`: It consists of sensor measurement and complementary information. It is a JSON formatted object as shown below.

Return Value:

-1: for error

non-negative: Success

Example:

```
context_path = "/opt/pcad/config/sensor_config.json"
context = pcad_init_context(context_path)
secure_context = pcad_get_credentials(context)
data = {"sensor_id": 1 , "time":"2016-09-20", "value": 23.2}
response = pcad_post_data(secure_context, data)
pcad_free_context(context)
```

Full Program Examples

RESTful Real-Time Application Program

```python
# -*- coding: utf-8 -*-
import requests
import websocket

AUTH_URL = "http://localhost:9000/api/v1/auth/signin"
QUERY_STRING = {"access_token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJQQ0\
FEIiwidWlkIjoxLCJncm91cCI6ImFwcGxpY2F0aW9uIn0.\
_DjfVW2emwWuXeZvCSipZKReWb06b9zq3_JM5CRkbEE"}
HEADERS = {'content-type': 'application/json'}
RESPONSE = requests.request("POST", AUTH_URL, data="{}", headers=HEADERS, params=QUERY_STRING)

DATA_URL = "ws://localhost:9000/api/v1/sensors/stream"
WEBSOCKET = websocket.WebSocket()
PAYLOAD = "{\"mode\":\"real\",\"sensor\":19}"
WEBSOCKET.connect(DATA_URL+"?access_token="+RESPONSE.content.replace("\"", ""))
WEBSOCKET.send(PAYLOAD)
RESPONSE = WEBSOCKET.recv()
print(RESPONSE)
WEBSOCKET.close()
```

RESTful Real-Time Application Program using Rule

```python
# -*- coding: utf-8 -*-
import requests
import websocket

AUTH_URL = "http://localhost:9000/api/v1/auth/signin"
QUERY_STRING = {"access_token":"eyJhbGciOiJIUzI1NiIs-
 ↪  InR5cCI6IkpXVCJ9.eyJpc3MiOiJQQ0\
FEIiwidWlkIjoxLCJncm91cCI6ImFwcGxpY2F0aW9uIn0.\
_DjfVW2emwWuXeZvCSipZKReWb06b9zq3_JM5CRkbEE"}
HEADERS = {'content-type': 'application/json'}
RESPONSE = requests.request("POST", AUTH_URL, data="{}",
 ↪  headers=HEADERS, params=QUERY_STRING)

DATA_URL = "ws://localhost:9000/api/v1/sensors/stream"
RULE = "name = notification\nmin_value = 10.5\ncurrent_value =
 ↪  pcad_get_time()\ndb_value = pcad_get_db_time(sensor=17) \nwhen
 ↪  {\n(current_value < min_value) and (db_value < min_value)}\nthen\n{
 ↪  Notify()}"
WEBSOCKET = websocket.WebSocket()
PAYLOAD = "{\"mode\":\"real\",\"sensor\":19, \"rule\":"+RULE+"}"
WEBSOCKET.connect(DATA_URL+"?access_token="+RESPONSE.content.re-
 ↪  place("\"",
 ↪  ""))
WEBSOCKET.send(PAYLOAD)
RESPONSE = WEBSOCKET.recv()
print(RESPONSE)
WEBSOCKET.close()
```

RESTful Non-Real-Time Application Program

```
# -*- coding: utf-8 -*-
import requests
import websocket

AUTH_URL = "http://localhost:9000/api/v1/auth/signin"
QUERY_STRING = {"access_token":"eyJhbGciOiJIUzI1NiIs-
 ↪  InR5cCI6IkpXVCJ9.eyJpc3MiOiJQQ0\
FEIiwidWlkIjoxLCJncm91cCI6ImFwcGxpY2F0aW9uIn0.\
_DjfVW2emwWuXeZvCSipZKReWb06b9zq3_JM5CRkbEE"}
HEADERS = {'content-type': 'application/json'}
RESPONSE = requests.request("POST", AUTH_URL, data="{}",
 ↪  headers=HEADERS, params=QUERY_STRING)

DATA_URL = "ws://localhost:9000/api/v1/sensors/stream"
WEBSOCKET = websocket.WebSocket()
PAYLOAD = "{\"mode\":\"periodic\",\"sensor\":1, \"interval\":5}"
WEBSOCKET.connect(DATA_URL+"?access_token="+RESPONSE.content.re-
 ↪  place("\"",
 ↪  ""))
WEBSOCKET.send(PAYLOAD)
RESPONSE = WEBSOCKET.recv()
print(RESPONSE)
WEBSOCKET.close()
```

RESTful Non-Real-Time Application Program using Rule

```
# -*- coding: utf-8 -*-
import requests
import websocket

AUTH_URL = "http://localhost:9000/api/v1/auth/signin"
QUERY_STRING = {"access_token":"eyJhbGciOiJIUzI1NiIs-
 ↪  InR5cCI6IkpXVCJ9.eyJpc3MiOiJQQ0\
FEIiwidWlkIjoxLCJncm91cCI6ImFwcGxpY2F0aW9uIn0.\
_DjfVW2emwWuXeZvCSipZKReWb06b9zq3_JM5CRkbEE"}
HEADERS = {'content-type': 'application/json'}
RESPONSE = requests.request("POST", AUTH_URL, data="{}",
 ↪  headers=HEADERS, params=QUERY_STRING)

DATA_URL = "ws://localhost:9000/api/v1/sensors/stream"
RULE = "name = notification\nmin_value = 10.5\ncurrent_value =
 ↪  get_value()\ndb_value = get_db_value(sensor=17) \nwhen
 ↪  {\n(current_value < min_value) or (db_value < min_value)}\nthen\n{
 ↪  Notify()}"
WEBSOCKET = websocket.WebSocket()
PAYLOAD = "{\"mode\":\"periodic\",\"sensor\":19,
 ↪  \"interval\":5,\"rule\":"+RULE+"}"
WEBSOCKET.connect(DATA_URL+"?access_token="+RESPONSE.content.re-
 ↪  place("\"",
 ↪  ""))
WEBSOCKET.send(PAYLOAD)
RESPONSE = WEBSOCKET.recv()
print(RESPONSE)
WEBSOCKET.close()
```

RESTful Single-Time Data Retrieval Application Program

```
# -*- coding: utf-8 -*-
import requests

AUTH_URL = "http://localhost:9000/api/v1/auth/signin"
QUERY_STRING = {"access_token":"eyJhbGciOiJIUzI1NiIs-
 ↪  InR5cCI6IkpXVCJ9.eyJpc3MiOiJQQ0\
FEIiwidWlkIjoxLCJncm91cCI6ImFwcGxpY2F0aW9uIn0.\
_DjfVW2emwWuXeZvCSipZKReWb06b9zq3_JM5CRkbEE"}
HEADERS = {'content-type': 'application/json'}
RESPONSE = requests.request("POST", AUTH_URL, data="{}",
 ↪  headers=HEADERS, params=QUERY_STRING)

DATA_URL = "http://localhost:9000/api/v1/sensors/1/data"
QUERY_STRING = {"access_token":RESPONSE.content.replace("\"", "")}
RESPONSE = requests.request("GET", DATA_URL, headers=HEADERS,
 ↪  params=QUERY_STRING)
print(RESPONSE.text)
```

RESTful Single-Time Data Retrieval Application Program using Filter

```python
# -*- coding: utf-8 -*-
import requests

AUTH_URL = "http://localhost:9000/api/v1/auth/signin"
QUERY_STRING = {"access_token":"eyJhbGciOiJIUzI1NiIs-
  ↪  InR5cCI6IkpXVCJ9.eyJpc3MiOiJQQ0\
FEIiwidWlkIjoxLCJncm91cCI6ImFwcGxpY2F0aW9uIn0.\
_DjfVW2emwWuXeZvCSipZKReWb06b9zq3_JM5CRkbEE"}
HEADERS = {'content-type': 'application/json'}
RESPONSE = requests.request("POST", AUTH_URL, data="{}",
  ↪  headers=HEADERS, params=QUERY_STRING)

DATA_URL = "http://localhost:9000/api/v1/sensors/1/data/filter"
PAYLOAD = "{\"value\": null, \"time\": {\"min\":
  ↪  \"2016-12-02T23:39:30.0+03:00\", \"max\":
  ↪  \"2016-12-07T23:39:30.0+03:00\"}}"
QUERY_STRING = {"access_token":RESPONSE.content.replace("\"", "")}
RESPONSE = requests.request("POST", DATA_URL, data=PAYLOAD,
  ↪  headers=HEADERS, params=QUERY_STRING)
print(RESPONSE.text)
```

Real-Time Application Program

```python
# -*- coding: utf-8 -*-
from connection import Connection

def print_message(message):
  print(message)

if __name__ == "__main__":
  try:
    INSTANCE = Connection()
    CONTEXT = INSTANCE.pcad_init_context("application_config.json")
    SECURE_CONTEXT = INSTANCE.pcad_get_credentials(CONTEXT)
    CONNECTION = INSTANCE.pcad_init_connection(SECURE_CONTEXT)
    INSTANCE.pcad_attach_sensor(CONNECTION, {"mode": "real", "sensor":
  ↪  19})
    INSTANCE.pcad_on_message(CONNECTION, print_message)
  except (Exception, KeyboardInterrupt, SystemExit) as e:
    INSTANCE.pcad_terminate_connection(CONNECTION)
```

Real-Time Application Program using Rule

```python
# -*- coding: utf-8 -*-
from connection import Connection

def print_message(message):
  print(message)

if __name__ == "__main__":
  try:
    INSTANCE = Connection()
    CONTEXT = INSTANCE.pcad_init_context("application_config.json")
    SECURE_CONTEXT = INSTANCE.pcad_get_credentials(CONTEXT)
    CONNECTION = INSTANCE.pcad_init_connection(SECURE_CONTEXT)
    INSTANCE.pcad_attach_sensor({"mode":"real", "sensor": 19},
 ↪   "example.rl")
    INSTANCE.pcad_on_message(CONNECTION, print_message)
  except (Exception, KeyboardInterrupt, SystemExit) as e:
    INSTANCE.pcad_terminate_connection(CONNECTION)
```

Non-Real-Time Application Program

```python
# -*- coding: utf-8 -*-
from connection import Connection

def print_message(message):
  print(message)

if __name__ == "__main__":
  try:
    INSTANCE = Connection()
    CONTEXT = INSTANCE.pcad_init_context("application_config.json")
    SECURE_CONTEXT = INSTANCE.pcad_get_credentials(CONTEXT)
    CONNECTION = INSTANCE.pcad_init_connection(SECURE_CONTEXT)
    INSTANCE.pcad_attach_sensor({"mode":"periodic", "sensor": 1,
 ↪   "interval": 10})
    INSTANCE.pcad_on_message(CONNECTION, print_message)
  except (Exception, KeyboardInterrupt, SystemExit) as e:
    INSTANCE.pcad_terminate_connection(CONNECTION)
```

Non-Real-Time Application Program using Rule

```
# -*- coding: utf-8 -*-
from connection import Connection

def print_message(message):
  print message

if __name__ == "__main__":
  CONNECTION = None
  try:
    INSTANCE = Connection()
    CONTEXT = INSTANCE.pcad_init_context("application_config.json")
    SECURE_CONTEXT = INSTANCE.pcad_get_credentials(CONTEXT)
    CONNECTION = INSTANCE.pcad_init_connection(SECURE_CONTEXT)
    INSTANCE.pcad_attach_sensor({"mode":"periodic", "sensor": 1,
 ↪   "interval": 10}, "example.rl")
    INSTANCE.pcad_on_message(CONNECTION, print_message)
  except (Exception, KeyboardInterrupt, SystemExit) as e:
    INSTANCE.pcad_terminate_connection(CONNECTION)
```

Single-Time Data Retrieval Application Program

```
# -*- coding: utf-8 -*-
from connection import Connection

if __name__ == "__main__":
  try:
    INSTANCE = Connection()
    CONTEXT = INSTANCE.pcad_init_context("application_config.json")
    SECURE_CONTEXT = INSTANCE.pcad_get_credentials(CONTEXT)
    result = INSTANCE.pcad_send_query(SECURE_CONTEXT, 1)
    print(result)
  except (Exception, KeyboardInterrupt, SystemExit) as e:
    print "connection killed " + str(e)

```

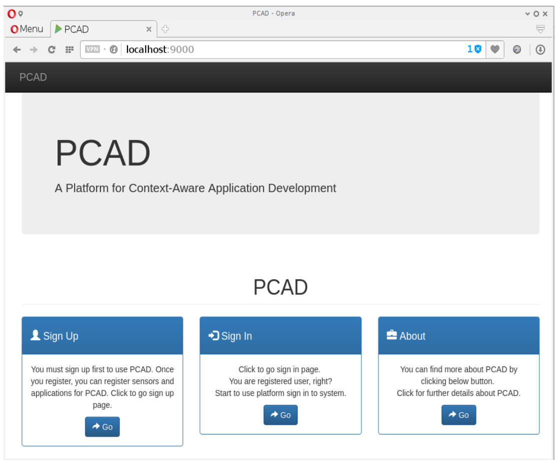
Single-Time Data Retrieval Application Program using Filter

```python
# -*- coding: utf-8 -*-
from connection import Connection

if __name__ == "__main__":
  try:
    INSTANCE = Connection()
    CONTEXT = INSTANCE.pcad_init_context("application_config.json")
    SECURE_CONTEXT = INSTANCE.pcad_get_credentials(CONTEXT)
    FILTER = {"value": None, "time": {"min":
 ↪  "2016-12-02T23:39:30.0+03:00", "max":
 ↪  "2016-12-07T23:39:30.0+03:00"}}
    RESULT = INSTANCE.pcad_send_query(SECURE_CONTEXT, 1, FILTER)
    print(RESULT)
  except (Exception, KeyboardInterrupt, SystemExit) as e:
    print "connection killed " + str(e)
```

Appendix B

User Interface



PCAD

A Platform for Context-Aware Application Development

PCAD

Sign Up

You must sign up first to use PCAD. Once you register, you can register sensors and applications for PCAD. Click to go sign up page.

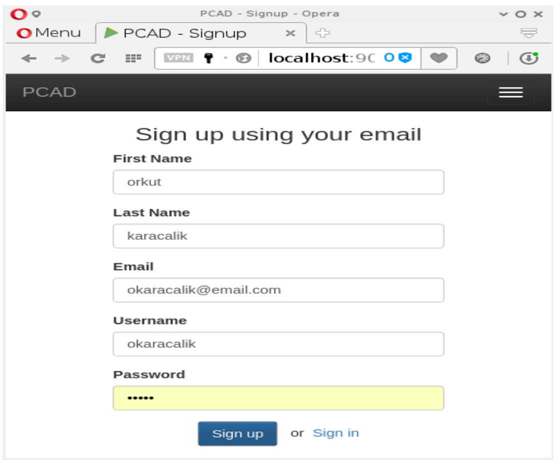
Sign In

Click to go sign in page.
You are registered user, right?
Start to use platform sign in to system.

About

You can find more about PCAD by clicking below button.
Click for further details about PCAD.

Figure B.1: Main Page.



Sign up using your email

First Name: orkut

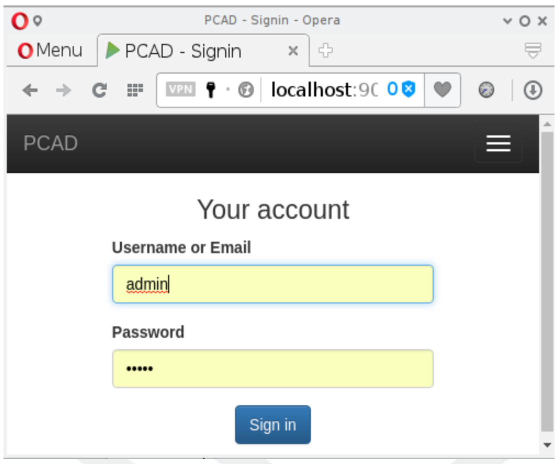
Last Name: karacalik

Email: okaracalik@email.com

Username: okaracalik

Password: •••••

Figure B.2: User Registration Page.



Your account

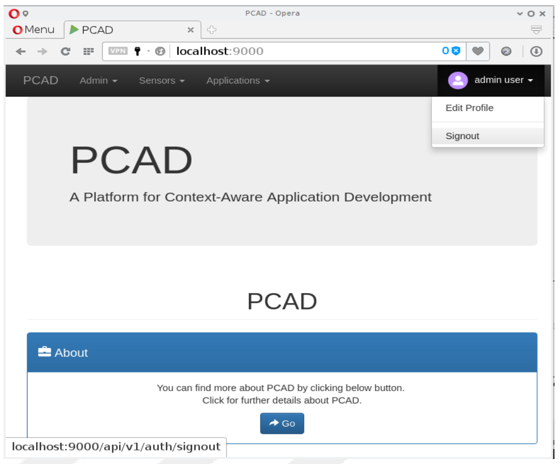
Username or Email

admin

Password

•••••

Figure B.3: Sign-in Page.



PCAD

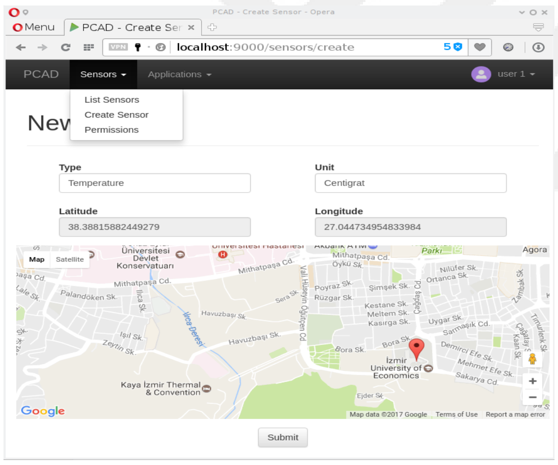
A Platform for Context-Aware Application Development

PCAD

You can find more about PCAD by clicking below button.
Click for further details about PCAD.

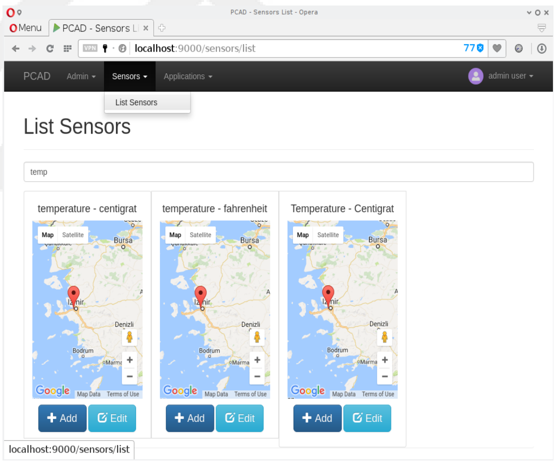
localhost:9000/api/v1/auth/signout

Figure B.4: Sign-out Action.



Type	Unit
Temperature	Centigrat
Latitude	Longitude
38.388158822449279	27.044734954833984

Figure B.5: Sensor Registration.


PCAD - Sensors List - Opera
Menu
PCAD - Sensors L
localhost:9000/sensors/list
PCAD
Admin
Sensors
Applications
admin user
List Sensors
Add
Edit
localhost:9000/sensors/list

List Sensors

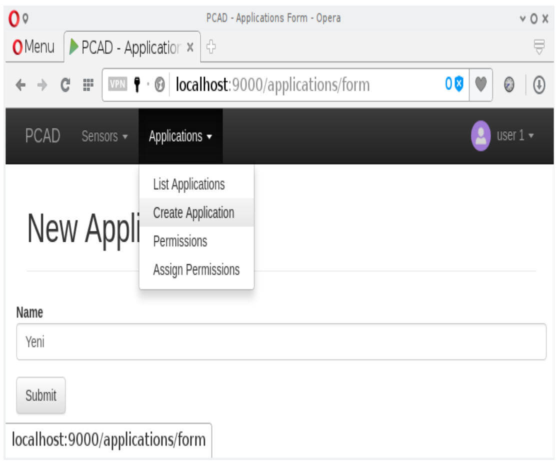
temp

temperature - centigrat

temperature - fahrenheit

Temperature - Centigrat

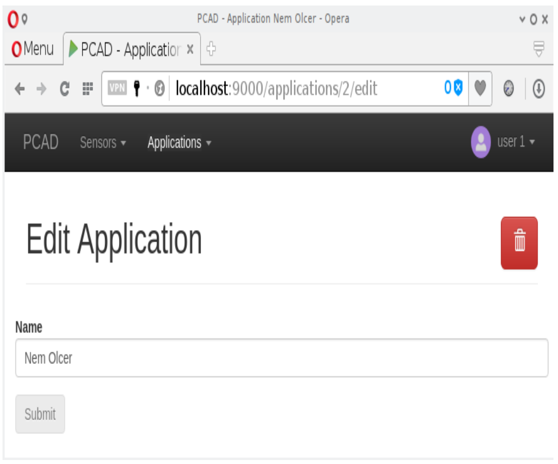
Figure B.6: Sensor Update for *admin*.



Name

Yeni

Figure B.7: Application Registration.

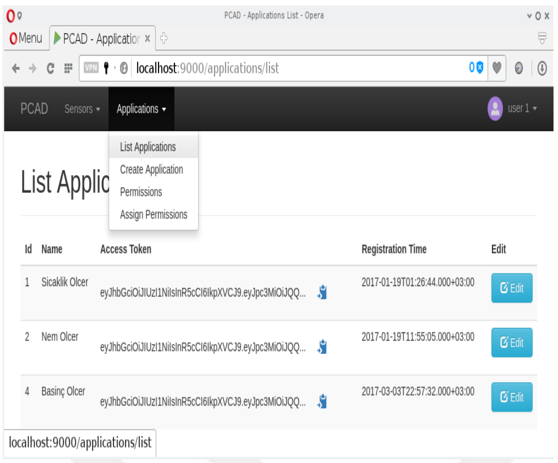

PCAD - Application Nem Olcer - Opera
Menu
PCAD - Applicatior
localhost:9000/applications/2/edit
PCAD
Sensors
Applications
user 1
Submit

Edit Application

Name

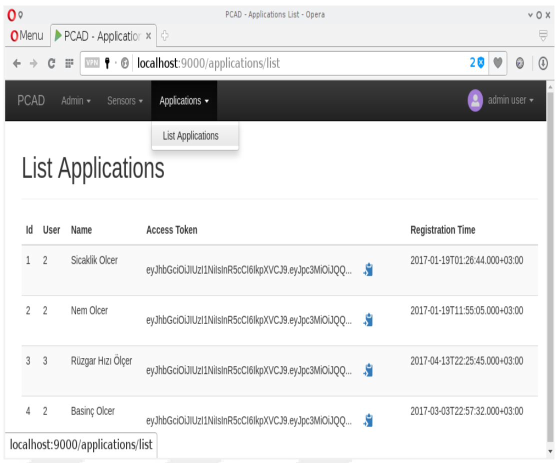
Nem Olcer

Figure B.8: Application Update.



Id	Name	Access Token	Registration Time	Edit
1	Sicaklik Olcer	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJQQ...	2017-01-19T01:26:44.000+03:00	Edit
2	Nem Olcer	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJQQ...	2017-01-19T11:55:05.000+03:00	Edit
4	Basinç Olcer	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJQQ...	2017-03-03T22:57:32.000+03:00	Edit

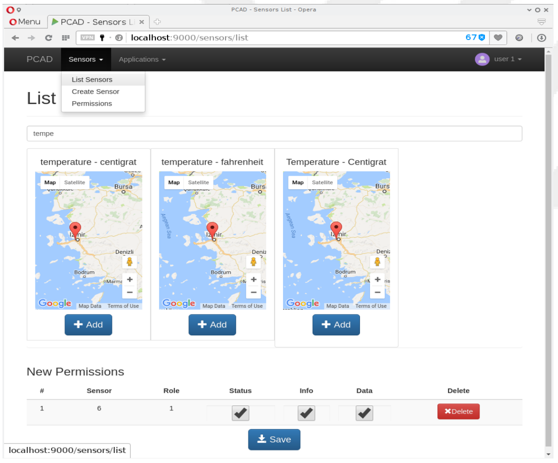
Figure B.9: Application List for *users*.


PCAD - Applications List - Opera
Menu
PCAD - Applicatior
localhost:9000/applications/list
PCAD
Admin
Sensors
Applications
admin user
List Applications
localhost:9000/applications/list

List Applications

Id	User	Name	Access Token	Registration Time
1	2	Sicaklik Olcer	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJQQ...	2017-01-19T01:26:44.000+03:00
2	2	Nem Olcer	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJQQ...	2017-01-19T11:55:05.000+03:00
3	3	Rüzgar Hızı Ölçer	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJQQ...	2017-04-13T22:25:45.000+03:00
4	2	Basinç Olcer	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJQQ...	2017-03-03T22:57:32.000+03:00

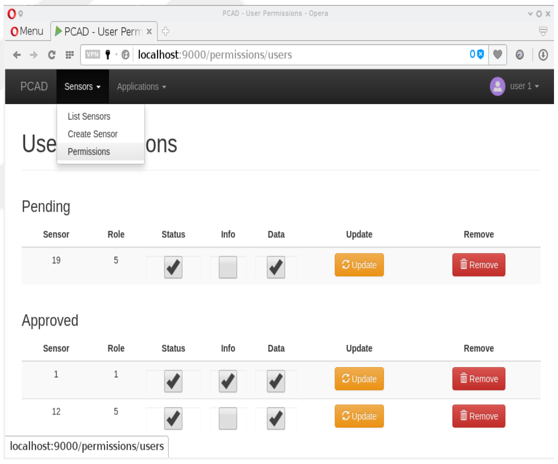
Figure B.10: Application List for *admin*.



New Permissions

#	Sensor	Role	Status	Info	Data	Delete
1	6	1	✓	✓	✓	Delete

Figure B.11: Requesting Permission for Application.


PCAD - User Permissions - Opera
Menu
PCAD - User Perm
localhost:9000/permissions/users
PCAD
Sensors
Applications
user 1
List Sensors
Create Sensor
Permissions
Use ons
localhost:9000/permissions/users

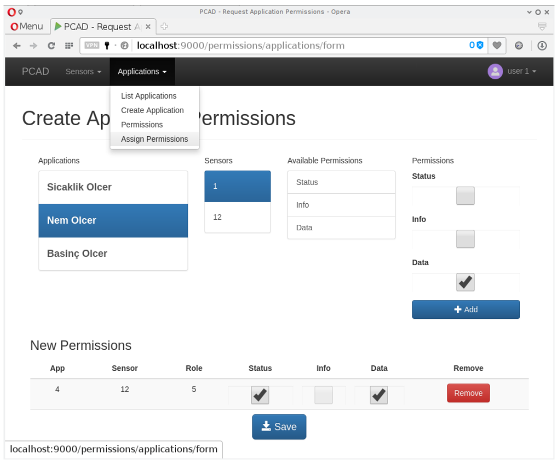
Pending

Sensor	Role	Status	Info	Data	Update	Remove
19	5	✓		✓	Update	Remove

Approved

Sensor	Role	Status	Info	Data	Update	Remove
1	1	✓	✓	✓	Update	Remove
12	5	✓		✓	Update	Remove

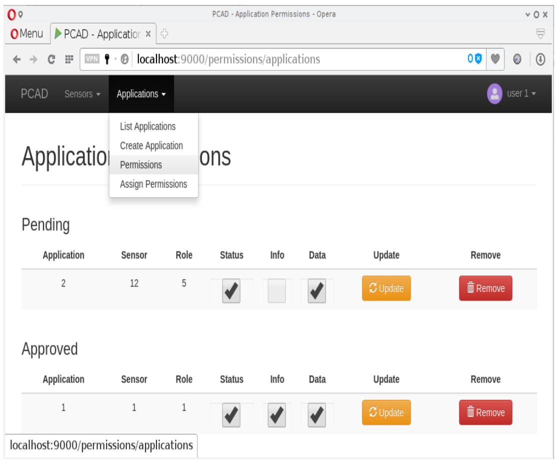
Figure B.12: Updating Permissions for Application.



New Permissions

App	Sensor	Role	Status	Info	Data	Remove
4	12	5	✓		✓	Remove

Figure B.13: Requesting Permission.



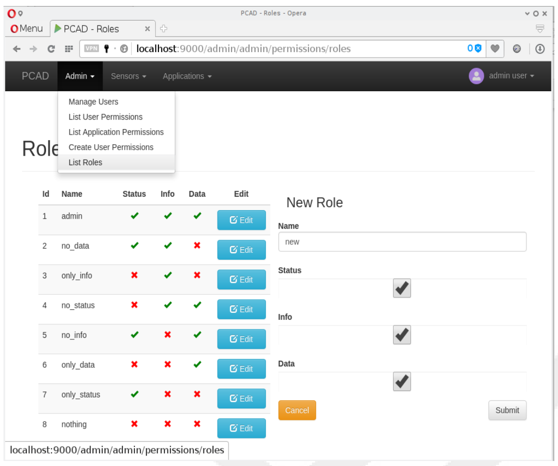
Pending

Application	Sensor	Role	Status	Info	Data	Update	Remove
2	12	5	✓		✓	Update	Remove

Approved

Application	Sensor	Role	Status	Info	Data	Update	Remove
1	1	1	✓	✓	✓	Update	Remove

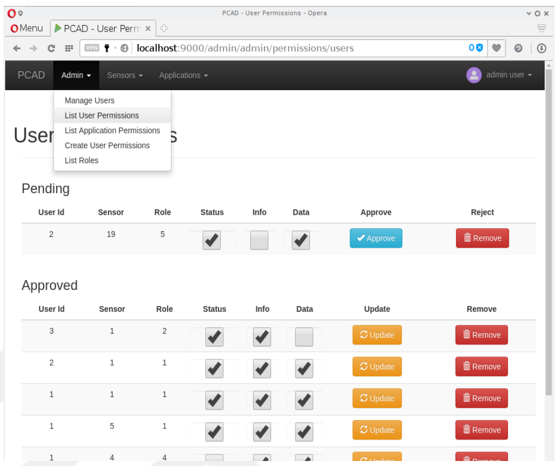
Figure B.14: Updating Permissions.



Id	Name	Status	Info	Data	Edit
1	admin	✓	✓	✓	Edit
2	no_data	✓	✓	✗	Edit
3	only_info	✗	✓	✗	Edit
4	no_status	✗	✓	✓	Edit
5	no_info	✓	✗	✓	Edit
6	only_data	✗	✗	✓	Edit
7	only_status	✓	✗	✗	Edit
8	nothing	✗	✗	✗	Edit

Name: new
Status: ✓
Info: ✓
Data: ✓

Figure B.15: Role Related Actions for *admin*.



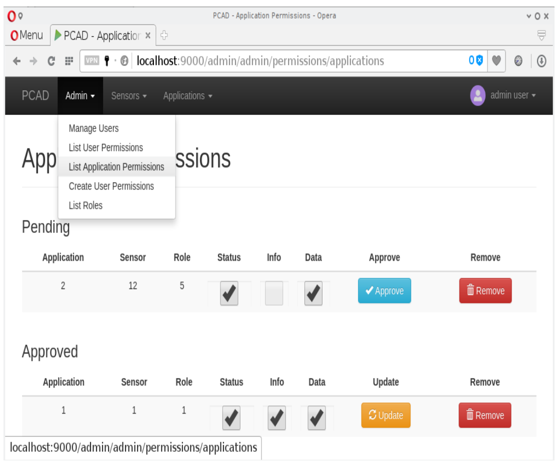
Pending

User Id	Sensor	Role	Status	Info	Data	Approve	Reject
2	19	5	✓	☐	✓	✓ Approve	Remove

Approved

User Id	Sensor	Role	Status	Info	Data	Update	Remove
3	1	2	✓	✓	☐	Update	Remove
2	1	1	✓	✓	✓	Update	Remove
1	1	1	✓	✓	✓	Update	Remove
1	5	1	✓	✓	✓	Update	Remove
1	4	4					

Figure B.16: User Permission Actions for *admin*.


PCAD - Application Permissions - Opera
Menu
PCAD - Applicatior
localhost:9000/admin/admin/permissions/applications
PCAD
Admin
Sensors
Applications
admin user
Manage Users
List User Permissions
List Application Permissions
Create User Permissions
List Roles
App
ssions
localhost:9000/admin/admin/permissions/applications

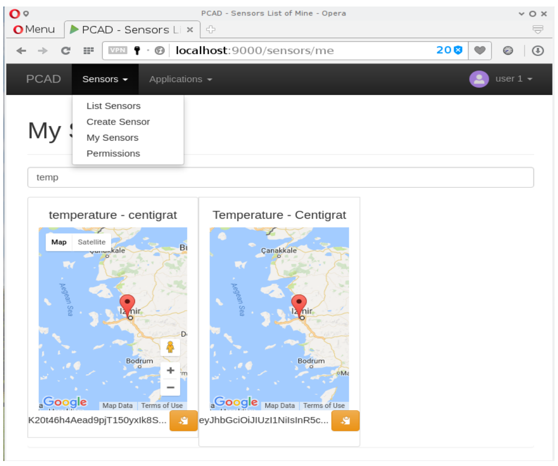
Pending

Application	Sensor	Role	Status	Info	Data	Approve	Remove
2	12	5	✓	☐	✓	✓ Approve	Remove

Approved

Application	Sensor	Role	Status	Info	Data	Update	Remove
1	1	1	✓	✓	✓	Update	Remove

Figure B.17: Application Permission Actions for *admin*.



K20t46h4Aead9pjT150yxIk8S...

eyJhbGciOiJIUzI1NiIsInR5c...

Figure B.18: Listing Permitted Sensors for *users*.

Appendix C

Installation

The platform has two methods of installation tailored towards two different audience. The first method of installation is intended for platform end users including application developers. The other is intended for those who would like to work on the platform itself by extending and contributing to the platform functionality. This section describes installation instructions to deploy the system for end users only. This type of installation requires minimal effort. The platform customization can be made by modifying `application.conf` file such as changing the port number on which PCAD runs.

Pre-requisite software:

- Linux Ubuntu 14.04
- Java 8: `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
- MySQL: `https://dev.mysql.com/downloads/`

The installation of the required software, should be verified with the following commands. 'x' indicates minor version, and may vary.

```
$ java -version
java version "1.8.x"
$ mysql version
mysql 5.7.x
```

The platform currently runs on the Linux operating system. The software is packaged in *pcad-play-1.0.zip* file. This file, when extracted, lays out the following directory structure on the file system:

```
pcad-play-1.0/
├── bin
│   ├── logs (It keeps logs about application.)
│   └── pcad-play.sh (It starts application as an executable file.)
├── conf
│   ├── application.conf (It keeps configuration variables.)
│   ├── logback.xml (Configuration for logging.)
│   └── routes (It keeps existing HTTP URLs for applications.)
├── lib (It involves library packages that application requires.)
├── samples (Sample programs that works with PCAD.)
│   ├── api_01_app_real_time.py
│   ├── api_02_app_real_time_rule.py
│   ├── api_03_app_non_real_time.py
│   ├── api_04_app_non_real_time_rule.py
│   ├── api_05_app_one_time.py
│   ├── api_06_app_one_time_filter.py
│   ├── api_sensor.py
│   ├── application_config.json
│   ├── application_rest.py
│   ├── connection.py
│   ├── example.rl
│   ├── rest_01_real_time.py
│   ├── rest_02_real_time_rule.py
│   ├── rest_03_non_real_time.py
│   ├── rest_04_non_real_time_rule.py
│   ├── rest_05_one_time.py
│   ├── rest_06_one_time_filter.py
│   ├── rest_sensor.py
│   └── sensor_config.json
├── init.sh (Script that must be executed before starting application.)
└── README.md (Simple notes about application.)
```

The `init.sh` file in the zip file should be run first. This creates the database schema. However, before doing this, `application.conf` file must be edited to add a database name and password.

```
$ chmod +x init.sh
$ ./init.sh
```

This is followed by the execution of the *pcad-play* command found in the `bin` directory, as shown below. The argument given to the command corresponds to the string variable used for encryption.

```
$ pcad-1.0/bin/pcad-play -Dplay.crypto.secret=ABCD
```

After completion of the command, PCAD starts working fully. *nginx* must be installed in order for the system to communicate with the external world. *nginx* provides the web server functionality that PCAD needs. Using the domain name it also redirects the HTTP requests sent to the standard port (i.e. 80) to the port where the PCAD configured to run. *nginx* installation can be done using the following link: `http://nginx.org/en/download.html`. What should be done for Linux is to open the file */etc/nginx/sites-available/default* and change it as follows:

1. replace *domain.com* with the actual domain name registered for PCAD.
2. `localhost:9000` indicates the machine and port number of the PCAD system. If the default URI and the port number that PCAD uses is altered one needs to replace them in this file as well.

]

```
server {
        listen 80;
        server_name domain.com;
        location / {
                proxy_pass http://localhost:9000;
                proxy_http_version 1.1;
                proxy_set_header Upgrade $http_upgrade;
                proxy_set_header Connection "upgrade";
                proxy_set_header Host $host;
                proxy_cache_bypass $http_upgrade;
        }
}
```

Bibliography

[1] Gubbi, J. et al. (2013). "Internet of Things (IoT): A vision, architectural elements, and future directions". In: *Future Generation Computer Systems* 29.7, pp. 1645–1660. URL: http://scholar.google.de/scholar.bib?q=info:h5t3ZYhcvvcJ:scholar.google.com/&output=citation&hl=de&ct=citation&cd=0.

[2] Al-Fuqaha, A. I. et al. (2015). "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications." In: *IEEE Communications Surveys and Tutorials* 17.4, pp. 2347–2376. URL: http://dblp.uni-trier.de/db/journals/comsur/comsur17.html#Al-FuqahaGMAA15;%20http://dx.doi.org/10.1109/COMST.2015.2444095.

[3] Dey, A. K., Abowd, G. D., et al. (2000). "The context toolkit: Aiding the development of context-aware applications". In: *Workshop on Software Engineering for wearable and pervasive computing*, pp. 431–441.

[4] Baldauf, M., Dustdar, S., and Rosenberg, F. (2008). "A survey on context-aware systems." In: *IJAHUC* 2.4, pp. 263–277. URL: http://dblp.uni-trier.de/db/journals/ijahuc/ijahuc2.html#BaldaufDR07;%20http://dx.doi.org/10.1504/IJAHUC.2007.014070.

[5] Ailisto, H. et al. (2002). "Structuring context aware applications: Five-layer model and example case". In: *Proceedings of the Workshop on Concepts and Models for Ubiquitous Computing*, pp. 1–5.

[6] Rehman, K., Stajano, F., and Coulouris, G. (2007). "An architecture for interactive context-aware applications". In: *IEEE Pervasive Computing* 6.1.

[7] Winograd, T. (2001). "Architectures for Context." In: *Human-Computer Interaction* 16.2-4, pp. 401–419. URL: http://dblp.uni-trier.de/db/journals/hhci/hhci16.html#Winograd01;%20http://dx.doi.org/10.1207/S15327051HCI16234_18.

[8] Kwon, J. H. and Kim, S. R. (2004). “Context-Aware Recommendation Using Pattern Discovery in Ubiquitous Computing”. In: *On the Convergence of Bio-, Information-, Enrivonmental-, Energy-, Space- and Nano-Technolgies.* Vol. 277. Key Engineering Materials. Trans Tech Publications, pp. 278–286.

[9] Munoz, M. A. et al. (2003). “Supporting context-aware collaboration in a hospital: An ethnographic informed design”. In: *International Conference on Collaboration and Technology.* Springer, pp. 330–344.

[10] Abowd, G. D. et al. (1997). “Cyberguide: A mobile context-aware tour guide”. In: *Wireless networks* 3.5, pp. 421–433.

[11] Hong, J.-y., Suh, E.-h., and Kim, S.-J. (2009). “Context-aware systems: A literature review and classification”. In: *Expert Systems with applications* 36.4, pp. 8509–8522.

[12] Lu, Y. and Yu, H. (2010). “A Flexible Architecture for RFID Based Laundry Management Systems”. In: *Wireless Communications Networking and Mobile Computing (WiCOM), 2010 6th International Conference on.* IEEE, pp. 1–4.

[13] Tajima, N., Tsukada, K., and Siio, I. (2011). “AwareHanger: Context-aware hanger for detecting the status of laundry”. In: *Pervasive 2011.*

[14] Van, N. T. et al. (2012). “An implementation of Laundry Management System based on RFID hanger and wireless sensor network”. In: *Ubiquitous and Future Networks (ICUFN), 2012 Fourth International Conference on.* IEEE, pp. 490–493.

[15] Noor, M. et al. (2012). “Design and development of ‘smart basket’system for resource optimization”. In: *Control and System Graduate Research Colloquium (ICSGRC), 2012 IEEE.* IEEE, pp. 338–342.

[16] Capra, L., Emmerich, W., and Mascolo, C. (2003). “Carisma: Context-aware reflective middleware system for mobile applications”. In: *IEEE Transactions on software engineering* 29.10, pp. 929–945.

[17] Roman, M. et al. (2002). “Gaia: A Middleware Infrastructure to Enable Active Spaces”. In: *IEEE Pervasive Computing*, pp. 74–83.

[18] Fahy, P. and Clarke, S. (2004). “CASS-middleware for mobile context-aware applications”. In: *Proceedings of the Workshop on Context Awareness,(WCA'04), CiteSeerX*, pp. 1–6.

[19] Gu, T., Pung, H. K., and Zhang, D. Q. (2005). "A service-oriented middleware for building context-aware services". In: *Journal of Network and computer applications* 28.1, pp. 1–18.

[20] Conan, D., Rouvoy, R., and Seinturier, L. (2007). "Scalable processing of context information with COSMOS". In: *IFIP International Conference on Distributed Applications and Interoperable Systems.* Springer, pp. 210–224.

[21] Chen, H. et al. (2004). "Intelligent agents meet the semantic web in smart spaces". In: *IEEE Internet computing* 8.6, pp. 69–79.

[22] Devaraju, A., Hoh, S., and Hartley, M. (2007). "A context gathering framework for context-aware mobile solutions". In: *Proceedings of the 4th international conference on mobile technology, applications, and systems and the 1st international symposium on Computer human interaction in mobile technology.* ACM, pp. 39–46.

[23] Firner, B. et al. (2011). "Poster: Smart buildings, sensor networks, and the internet of things". In: *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems.* ACM, pp. 337–338.

[24] Hofer, T. et al. (2003). "Context-awareness on mobile devices-the hydrogen approach". In: *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on.* IEEE, 10–pp.

[25] Badii, A., Crouch, M., and Lallah, C. (2010). "A context-awareness framework for intelligent networked embedded systems". In: *2010 Third International Conference on Advances in Human-Oriented and Personalized Mechanisms, Technologies and Services.* IEEE, pp. 105–110.

[26] Katasonov, A. et al. (2008). "Smart Semantic Middleware for the Internet of Things." In: *Icinco-Icso* 8, pp. 169–178.

[27] Luckenbach, T. et al. (2005). "TinyREST-a protocol for integrating sensor networks into the internet". In: *Proc. of REALWSN*, pp. 101–105.

[28] Merezeanu, D., Vasilescu, G., and Dobrescu, R. (2016). "Context-aware control platform for sensor network integration in IoT and Cloud". In: *Studies in Informatics and Control* 25.4, pp. 489–498.

[29] Antonic, A. et al. (2014). "A Mobile Crowdsensing Ecosystem Enabled by a Cloud-Based Publish/Subscribe Middleware." In: *FiCloud.* IEEE, pp. 107–114. URL: http://dblp.uni-trier.de/db/conf/ficloud/ficloud2014.html#AntonicRMPZ14;%20http://dx.doi.org/10.1109/FiCloud.2014.27.

[30] Martin, D. et al. (2014). "Empowering End-Users to Develop Context-Aware Mobile Applications Using a Web Platform." In: *FiCloud*. Ed. by M. Younas, I. Awan, and A. Pescape. IEEE Computer Society, pp. 139–145. URL: http://dblp.uni-trier.de/db/conf/ficloud/ficloud2014.html#MartinLAT14.

[31] Pokraev, S. et al. (2005). "Service Platform for Rapid Development and Deployment of Context-Aware, Mobile Applications." In: *ICWS*. IEEE Computer Society, pp. 639–646. URL: http://dblp.uni-trier.de/db/conf/icws/icws2005.html#PokraevKSBCWES05;%20http://doi.ieeecomputersociety.org/10.1109/ICWS.2005.106.

[32] Paganelli, F. et al. (2007). "ERMHAN: A multi-channel context-aware platform to support mobile caregivers in continuous care networks." In: *ICPS*. IEEE Computer Society, pp. 355–360. URL: http://dblp.uni-trier.de/db/conf/icps/icps2007.html#PaganelliSMBB07;%20http://eudl.eu/doi/10.1109/PERSER.2007.4283939.

[33] *Scala Documentation*. URL: http://docs.scala-lang.org. [21 September 2018]

[34] Hewitt, C., Bishop, P., and Steiger, R. (1973). "A Universal Modular ACTOR Formalism for Artificial Intelligence". In: URL: http://citeseerx.ist.psu.edu/viewdoc/summary.

[35] Hewitt, C. (2012). *Actor Model of Computation: Scalable Robust Information Systems*. Tech. rep. v24. URL: http://arxiv.org/abs/1008.1459v24.

[36] Karmani, R. K. and Agha, G. (2011). "Actors." In: *Encyclopedia of Parallel Computing*. Ed. by D. A. Padua. Springer, pp. 1–11. URL: http://dblp.uni-trier.de/db/reference/parallel/parallel2011.html#KarmaniA11;%20http://dx.doi.org/10.1007/978-0-387-09766-4_125.

[37] Silberschatz, A., Galvin, P. B., and Gagne, G. (2008). *Operating System Concepts*. 8th. Wiley Publishing.

[38] Agha, G. (1986). *ACTORS: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press. URL: http://portal.acm.org/citation.cfm?id=7929&dl.

[39] *Erlang Reference Manual User's Guide*. URL: http://erlang.org/doc/reference_manual/users_guide.html. [21 September 2018]

[40] *What is Akka.NET?* URL: https://getakka.net/articles/intro/what-is-akka.html. [21 September 2018]

[41] *What is Kilim?* URL: http://www.malhar.net/sriram/kilim/. [21 September 2018]

[42] *Akka*. URL: http://doc.akka.io/docs/akka/2.5.3/scala. [21 September 2018]

[43] *What you should not concern yourself with.* URL: https://doc.akka.io/docs/akka/current/general/actor-systems.html%5C#what-you-should-not-concern-yourself-with. [21 September 2018]

[44] Celikkan, U. and Kurtel, K. (2015). "A Platform for Context-Aware Application Development: PCAD." In: *FedCSIS*. Ed. by M. Ganzha, L. A. Maciaszek, and M. Paprzycki. IEEE, pp. 1481–1488. URL: http://dblp.uni-trier.de/db/conf/fedcsis/fedcsis2015.html#CelikkanK15;%20http://dx.doi.org/10.15439/2015F49.

[45] Gamma, E. et al. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison-Wesley. URL: http://www.worldcat.org/search?qt=worldcat_org_all&q=0201633612.

[46] Ferraiolo, D. F., Kuhn, D. R., and Chandramouli, R. (2003). *Role-Based Access Control*. Norwood, Massachusetts: Artech House.

[47] Bishop, M. (2003). *Computer security: art and science*. Addison-Wesley Professional.

[48] Pimentel, V. and Nickerson, B. G. (2012). "Communicating and Displaying Real-Time Data with WebSocket." In: *IEEE Internet Computing* 16.4, pp. 45–53. URL: http://dblp.uni-trier.de/db/journals/internet/internet16.html#PimentelN12;%20http://doi.ieeecomputersociety.org/10.1109/MIC.2012.64.

[49] Jones, M., Bradley, J., and Sakimura, N. (2015). *Json web token (jwt)*. Tech. rep.

[50] Fielding, R. et al. (1999). *Hypertext Transfer Protocol – HTTP/1.1*. Tech. rep. URL: http://www.ietf.org/rfc/rfc2616.txt.

[51] Park, J. S., Sandhu, R. S., and Ahn, G.-J. (2003). "Role-based access control on the web." In: *ACM Trans. Inf. Syst. Secur.* 4.1, pp. 37–71. URL: http://dblp.uni-trier.de/db/journals/tissec/tissec4.html#ParkSA01;%20http://doi.acm.org/10.1145/383775.383777.

[52] Backus, J. W. et al. (1960). "Report on the algorithmic language ALGOL 60". In: *Numerische Mathematik* 2.1, pp. 106–136.

[53] *MySQL*. URL: https://dev.mysql.com/doc/. [21 September 2018]

[54] Date, C. J. and Darwen, H. (1987). *A Guide to the SQL Standard*. Vol. 3. Addison-Wesley.

[55] *MongoDB*. URL: https://docs.mongodb.com/. [21 September 2018]

[56] *Slick*. URL: http://slick.lightbend.com/docs. [21 September 2018]

[57] Rajkumar, R., Gagliardi, M., and Sha, L. (1995). "The real-time publisher/subscriber inter-process communication model for distributed real-time systems: design and implementation." In: *IEEE Real Time Technology and Applications Symposium*. IEEE Computer Society, pp. 66–75. URL: http://dblp.uni-trier.de/db/conf/rtas/rtas1995.html#RajkumarGS95;%20http://doi.ieeecomputersociety.org/10.1109/RTTAS.1995.516203.

[58] Locke, D. (2010). "Mq telemetry transport (mqtt) v3. 1 protocol specification". In: *IBM developerWorks Technical Library*.

[59] Saint-Andre, P. (2004). *Extensible Messaging and Presence Protocol (XMPP): Core*. RFC 3920 (Proposed Standard). Internet Engineering Task Force. URL: http://www.ietf.org/rfc/rfc3920.txt.

[60] Shelby, Z., Hartke, K., and Bormann, C. (2014). "The constrained application protocol (CoAP)". In:

[61] Godfrey, R., Ingham, D., and Schloming, R. (2012). *OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0; OASIS Standard*.

[62] Masse, M. (2011). *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media.

[63] Fette, I. and Melnikov, A. (2011). *The WebSocket Protocol*. RFC 6455. RFC Editor. URL: http://www.rfc-editor.org/rfc/rfc6455.txt.

[64] *Play 2.5.x Documentation*. URL: https://www.playframework.com/documentation/2.5.x/Home. [21 September 2018]

[65] Hilton, P., Bakker, E., and Canedo, F. (2014). *Play for Scala*. Manning.

[66] *AngularJS*. URL: https://angularjs.org. [21 September 2018]

[67] Berners-Lee, T. and Connolly, D. (1995). *Hypertext markup language-2.0*. Tech. rep.

[68] Software & Systems Engineering Committee (2008). "IEEE standard for software and system test documentation". In: *Fredericksburg, VA, USA: IEEE Computer Society*.