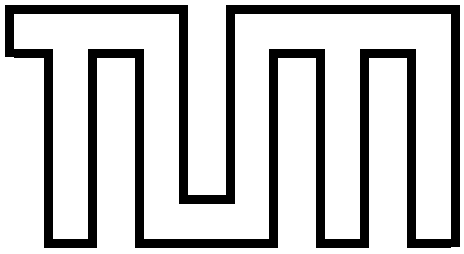


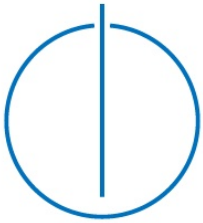
Technische Universität München

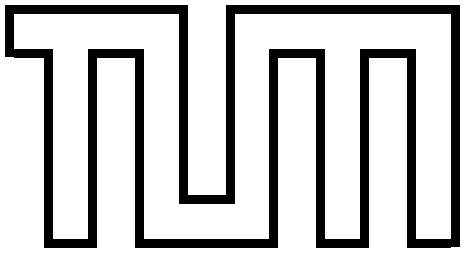
Fakultät für Informatik

Master's Thesis in Informatik

An Architecture for Composable Distributed Applications and Services in Disconnected Information-centric Networks

Chrysa Papadaki





Technische Universität München

Fakultät für Informatik

Master's Thesis in Informatik

An Architecture for Composable Distributed Applications and Services in Disconnected Information-centric Networks.

Eine Architektur für modulare verteilte mobile Anwendungen und Dienste in unterbrechungstoleranten inhaltsorientierten Netzen

Author: Chrysa Papadaki

Supervisor: Prof. Dr.-Ing. Jörg Ott

Advisor: M.Sc. Teemu Kärkkäinen

Submission: 05.08.2016

I assure the single handed composition of this master's thesis only supported by declared resources.

München, 05.08.2016

(Chrysa Papadaki)

Abstract

Even though, nowadays, there is a tremendous increase in the usage of powerful mobile consumer devices, most of the application frameworks support the development of standard client/server mobile applications in infrastructure networks (Internet), without making use of the capabilities of the consumer devices. The majority of the modern mobile devices is equipped with a large variety of sensors and networking capabilities. Considering those resources that such devices can provide, their increased number and the large amount of user-generated content, we envision a wide variety of localized networking scenarios in various contexts, e.g., social networking, location-based services, navigation services, urban orientation systems, real-time streaming and others. This can be realized without employing a central system entity similar to content-servers used in standard web-based applications. This is feasible by leveraging peer-wise contacts using the networking capabilities of the devices. The actual question that comes at this point is how these scenarios can be realized considering that the probability of the occurrence of a peer-wise contact is unknown.

To answer this question, in this work we propose an architecture that enables applications and services to run on top of opportunistic networks. This is achieved by leveraging the following: (i) the composability principle to enable modular systems and reduce complexity, (ii) the publish-subscribe interaction model to decouple senders and receivers and enable efficient coordination among the subsystems without the need of central entities and, at last (iii) the story-carry-forward paradigm to enable the storing and dissemination of the system state. The fundamental concept of this work is the *Composable Component*, i.e., a simple, self-contained unit that can communicate opportunistically with other local and remote components and provide a UI to the users.

Initially, given concrete scenarios, we provide the fundamental requirements analysis for enabling opportunistic communication and structuring the system in a manner that allows efficient interactions among application-level processes without using centralized infrastructure. Subsequently, we discuss the architecture based on those requirements and present a framework that enables these features. In order to prove the viability of this solution, we evaluate the architecture by providing a set of application designs of various context built up using the discussed framework, the implementation of the framework for both Java and Android and the implementation of the one of the application designs, i.e., polling application. At last, we carry out a set of experiments by deploying the implementations on real devices and examining the system interactions in detail.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Despite the intensive research work on opportunistic and delay tolerant networks that has taken place in the recent years and the widespread use of powerful consumer mobile devices, there is still lack of frameworks that support the development of opportunistic mobile applications. Most of them focus on helping accessing web services hosted in remote centralized infrastructure, and thus the peer-to-peer communication is non-existent even though the hardware is capable and might be the case that the requested content is locally relevant. Exploiting peer-wise contacts utilizing consumer mobile devices to dynamically create applications and services, enabling interactions when fixed network infrastructures may not be available is a challenging task, rather reasonable.

More specifically, it makes more sense to distribute locally relevant user generated content through localized networks and services, rather than storing the content to remote servers for supporting interactions needed to exchange content among co-located users. This implies unnecessary traffic and energy consumption by resources used to provide large scalable systems in order to enable this kind of services, when the necessary hardware can be provided by the end users. This can be backed up by the fact that the estimated number of mobile phone users is 3.3 billion worldwide, which is more than half of the world population. Most of the mobile phones in the current era are equipped with Wi-Fi, Bluetooth, cameras, sensors, and numerous other components. Such an availability of mobile communication devices creates a huge number of contact opportunities among humans, and become the key to establishing opportunistic mobile social networks [Cont 10]. Furthermore, the use of Internet is still constrained by cost (e.g., cellular data charges) and provisioning limitations (e.g., availability, capacity) in many areas [Kark 14]. However, the fact that even standard web-based applications might be characterized by increased complexity that decreases their scalability, maintainability and makes it difficult to get rid of obsolete technologies, implies further effort on building viable opportunistic applications capable of supporting functionalities resembling existing modern web-based applications. The study of architectural patterns and design principles that would not only enable the efficient design and implementation of applications of

similar complexity but also allow us to tackle with the absence of robust centralized content-servers, is considered as necessary.

In this work, we introduce the Composable Component (CC), a self-contained unit that encapsulates a single functionality and can communicate opportunistically with other components with the purpose of providing a composable service. We present an architecture for modeling content-centric services and applications suitable to run in disconnected networks, out of autonomous, configurable CCes that are capable of managing any type of content and interacting with the user and/or other components dynamically. This is a novel approach for modeling systems that tends to reduce the overall design and development effort and enable user to decide on the features of the end-system. The following section presents the concrete goals of the thesis in detail.

1.1 Scope and Goals

In this work, we present a system architecture and a framework for building distributed composable applications and services running on top of opportunistic networks utilizing the in-network resources and enabling dynamic interactions by exploiting the random presence of smartphone users in the network.

More specifically, the system is suitable for developing applications to exploit random encounters between nearby nodes, utilizing the publish-subscribe interaction paradigm. Those apps are composed of a set of autonomous components each one of them manages different subset of the network content space, can cooperate with each other to provide the network services and can be managed at runtime by the user. The dissemination mechanism of the system relies on opportunistic content forwarding while the architecture does not assume a traditional network layer.

The system by using publish-subscribe paradigm decouples the communicating entities from the contents and thus it inherently allows for asynchronous communication and leverages looser delay constraints. This approach is derived by the ICN where nodes receive only the information which they have requested or subscribed to. The request model and content transfer from sources to receivers is connectionless, hence, in cases of devices reposition, re-issue of requests for information can be done and thus delay/disruption tolerant operation in addition to mobility is supported without requiring cumbersome solutions. In addition, in this work, we discuss the possibility of realizing synchronous interactions on the top of opportunistic networks by introducing in the architecture design an application-level bus abstraction that is responsible for taking care of the queries to the CCs' content.

We demonstrate the framework feasibility by implementing and evaluating an application framework, designing three applications using the proposed architecture and implementing and evaluating one of them, i.e., the polling application. To realize this architecture, the

framework implementation leverages the publish-subscribe interaction paradigm and the SCAMPI opportunistic router [Kark 12]. The discussed applications comprise a set of self-contained units that can communicate opportunistically and present a UI to the user, out of which the user can compose an application or view that lets them access meaningful information. In this sense, the system attempts to improve usability and user experience. Due to time constraints, the evaluation of the user experience is out of scope. The resulting system will (i) accelerate the design and implementation of content-centric composable applications which enable opportunistic interactions and services when fixed infrastructure is not available, (ii) expand the applications and services range over different networks without using the Internet, (iii) provide a technology-agnostic framework that enforces high level of modularity and allows the efficient design of complex applications, and (iv) allow mobile users to interact with their surroundings by using location-relevant applications that can be installed dynamically over an in-network distribution mechanism.

1.2 Structure of the Thesis

The structure of the thesis consists of three sections, i.e., (i) the introduction to the discussed problem, the proposed solution and the goals of this work, (ii) elaboration on the proposed solution and (iii) the evaluation of the solution. More specifically, Chapter 1 defines the scope and the goals of the thesis, which mainly focus on discussing a system design that enables development of mobile opportunistic applications. Chapter 2 references significant concepts in the area of ICN, DTNs and system decomposition. In Chapter 3, we present the main contribution of this work, which is an architecture for building distributed, composable applications and services in disconnected ICNs. Chapter 4, 5 and 6 evaluate the proposed solution by designing applications and services of different context with the same discussed architecture, presenting an application development framework implementation and the implementation of a polling application developed using the framework and finally demonstrating the viability of this system by presenting the results of experiments on real devices. At last, Chapter 7 concludes this thesis by summarizing its results.

Chapter 2

Background and Related Work

This chapter describes the ICN architecture and a strawman proposal by Dirk Trossen, presents the Delay Tolerant Networking (DTN) and related work in mobile and opportunistic networks as well as the SCAMPI opportunistic networking middleware that is used in this work. In addition, it elaborates on the notion of composability from the system design point of view and discusses existing composable architectures and industrial services composition and management technologies.

Information Centric Networking (ICN) [Tros 12] is an alternative approach to the current Internet architecture, where the focus is on the content, rather than the communication between hosts. The main goal of ICN architecture is to decentralize content sharing and offload traffic from the content-servers. To achieve this goal, the architecture takes advantage of in-network caching, multiparty communication through replication, and the publish-subscribe interaction model, that decouples senders and receivers. Any content type is represented as an object which can be stored in different locations in the network. In addition, not only servers but also consumer devices that are capable of maintaining a copy of an object, can implement caching. In practice, once a request is issued by a user, the network will locate the requested object in one of the caches and return a copy to the requester. This architecture intends to be more suitable for content distribution and mitigate the impact of communication disruptions. There are four ICN approaches [Ahlg 11]; Data-Oriented Network Architecture (DONA), Content-Centric Networking (CCN), Publish-Subscribe Internet Routing Paradigm (PSIRP), and Network of Information (NetInf), that share common design concepts which make them differentiate from the existing Internet host-centric architecture, focusing on *WHAT* is exchanged rather than *WHO* exchanges data.

Those approaches, in contrast with this work, focus on well-connected networks. In this thesis, we propose an architecture that addresses disconnected information-centric networks, where there are limitations in connectivity due to nodes mobility and poor infrastructure. Designing information-centric applications and services running on top of such challenged networks, where there is no centralized infrastructure and the content

dissemination can be realized over opportunistic contacts, requires the employment of different approaches and mechanisms than those considered in standard systems. One of the concepts adopted in this work is the opportunistic content sharing, that is built upon the ICN architectures and allows users in proximity, who request the same content to retrieve it from each other, rather than from the original content-servers. The idea of distributed dissemination of user-content is to make relevant content easily available where it is most likely to be requested. In our work, we envision systems that run on top of opportunistic networks (OppNets) and encourage frequent interactions in neighborhood networks where location relevant data can be exchanged and eventually disseminated not only locally but even further in distant networks by leveraging users' mobility. A neighborhood network is seen here as a set of two or more static and/or mobile nodes in proximity, the mobile nodes move to nearby networks and offload the collected information, doing so the neighborhood is being expanded.

Disruption Tolerant Networks (DTNs) [Neum 95] are designed to overcome limitations in disconnected networks. DTNs rely on the inherent mobility in the network to deliver packets using a store-carry-and-forward paradigm [Burg 06], [Burn 05]. In this work, we utilize autonomous mobile and stationary store-carry-forward routers that use various discovery mechanisms, including IP multicast/broadcast beaconing, to pass messages. This approach is similar to the use of throwboxes [Zhao 06] and can enable multiple scenarios such as contextual quiz games, polling applications, messaging applications, newsfeed service or even an urban orientation system, that is based on the idea that an user's journey can be enriched by providing contextual information about points of interest along their route. Users can drop and query information about the surroundings and these queries can be satisfied with minimal energy consumption and memory occupation by leveraging other consumer devices in proximity. More services can be built upon such networks that are tolerant to modest delay is done such us social networking, local surveys, services for students in campuses or local services for citizens and more.

Nevertheless, building such applications and services requires design decisions to support features such as: (i) subsystems orchestration without the utilization of a central coordinator, (ii) systems decoupling, (iii) client/server communication style realization since this kind of applications and services might resemble the functionalities of standard web-based applications, yet running on top of opportunistic networks, (iv) decentralized application distribution mechanism and (v) ease of development and deployment. To achieve the aforementioned, in this chapter we discuss the composability principle in detail and elaborate on the system decomposition technique. Moreover, we provide an overview of the Microservices Architecture and its commonly-used patterns and discuss the containerization with Kubernetes, which provides management and administration of large sets of systems which are partitioned into smaller reusable subsystems each of those is isolated within a Docker container.

2.1 Information Centric Networking

Information-Centric Network architectures differentiate from the current Internet's host-centric end-to-end communication paradigm and instead adopt a content-centric communication paradigm, where information, rather than hosts, are named. The main research effort that has been made in this direction is by Dirk Trossen et al., who proposed an information-centric internetworking architecture [Tros 10], an information-centric internetworking architecture, with which they attempt to replace the current Internet architecture. In the following section, we outline the architecture and introduce the fundamental design elements of ICN architecture.

The architecture proposal focuses on the information that is being exchanged among entities rather than on the entities themselves. To achieve that, the authors propose as the underlying service model, the publish-subscribe model that decouples the senders and the receivers. In their study, they identified four key challenges, that they believe that an ICN architecture could tackle, i.e., (i) information-centrism of applications, (ii) supporting and exposing tussles,(iii) increasing accountability, and (iv) addressing attention scarcity. Their main attempt is to replace the interworking layer as a whole and contribute to solving human-centric information problems rather than shifting the solution to the application layer. The key design concepts of this proposal are summarized below:

- Everything is information: Trossen et al. define an information item as the simplest unit of data transmitted through the network, for which a rendezvous identifier (RId) is used for its localization. The RIds are unique global identifiers and assist in composing more complex and larger information items out of other RIds. An information item can also represent service information and other Rids. With this, the authors introduced the concept of service metadata, which, as in the standard Internet architecture, can be used for declaring access control policies or quality of service parameters.
- Information is scoped: the authors introduced the concept of scope. From the application's perspective, a scope denotes a group of related data and hence, it is considered as information itself and their RIds are the scope identifiers (SId). From the network's perspective, it represents the party being responsible for locating a copy of the data in the network. This concept refers to information grouping considering a particular application domain and that in turn reduces the space to be searched for a RId. Therefore, it supports composition of information that allows the mapping of application-level semantics to information items and scopes. The underlying network is application-agnostic and in fact it provides an information naming structure that allows the applications to create their own naming schemes and ontologies.
- Equal control: the architecture provides a balance of power between publisher and subscriber(s), offering a new set of network services that depart from the standard

send-receive communication between endpoints to a publish-subscribe model of information. Endpoints do not require unique identifiers to be addressed. Instead, it is the information itself that is being located in host endpoints.

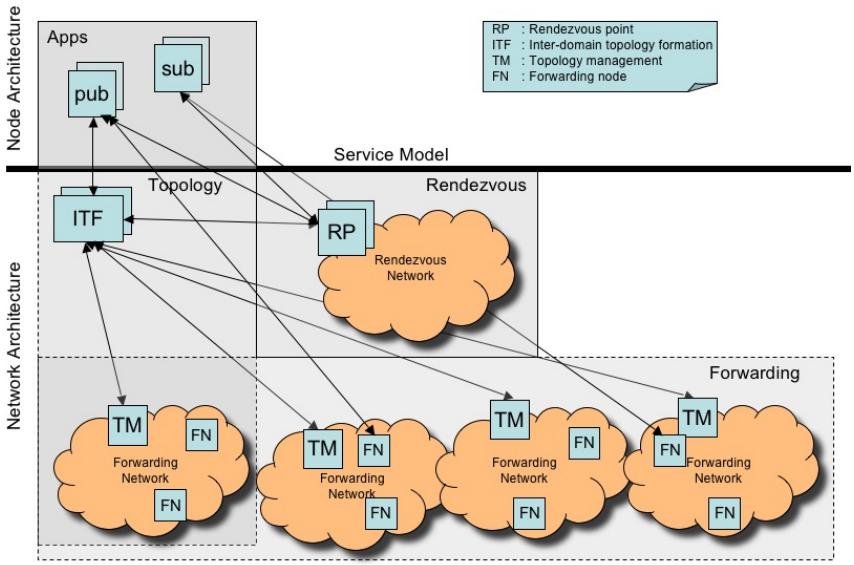


Figure 2.1: An information-centric internetworking architecture. *Adopted from [Tros 10]*

The proposed conceptual architecture relies on the three above-mentioned design concepts and aims at keeping the network architecture simple and yet enabling more complex application-level naming structures. In Figure 2.1, the conceptual architecture is presented. The pubs and subs implement applications based on pub/sub network services, by enabling publications and subscriptions to specific information items within a particular scope. The three main fucntions of the underlying network architecture are:

1. The Rendezvous function is responsible for logically linking the publishers and subscribers of an information item using a RId. It provides a global rendezvous system that defines the Rendezvous Points (RP) that are used to match the subscriptions with the publications. More specifically, an information item should be logically located in at least one scope and there should be one RP per scope, i.e., each RP subscribes to a SId. Thus, when a subscription to an information item is being identified, the request is forward to the corresponding RPs.

2. The Inter-domain Topology Formation (ITF) function gets involved in the forwarding topology formation for the RP. The formation is done based on the location of the publishers and subscribers on the level of autonomous systems (ASes) and the ITF information that includes the interconnections between the ASes. In practice, it builds the inter-domain paths between the forwarding networks.

3. The Topology Management (TM) function exists in every AS and is responsible for managing its forwarding nodes (FN) to serve as links between the ASes. In practice,

the locations of the subscribers and publishers are identified as local link identifiers that require only local uniquess in the AS-level and thus, the inter-domain paths assures the forwarding of information across the ASes.

Information-centrism is a main concept of this architecture. Information items and scopes enable the mapping of the application-level structures to the information structures on network level. In addition, the linking of information through metadata enables the governance and provenance of information on low level. Location transparency is achieved, since on the network level, the topology and forwarding functions determine the locations of the endpoints without exposing this functionality to the applications.

With respect to the challenge of supporting and exposing tussles, the tussle of economics is addressed by the provision of an efficient information delivery service, while the inter-domain topology function constructs the path between the publisher and subscriber(s) across domains give the policies that are specific to each scope. The concept of metadata is utilized to facilitate a low level system of policies and the construction of forwarding paths considering for instance security policies. The tussle of trust is addressed in the rendezvous point by implementing authorization methods for the exchange of information and thus, limiting the access rights of the endpoints. As far as the problem of accountability is concerned, the proposed solution is to provide a network where information itself as well as its structures are visible throughout the internetworking infrastructure, without relying on certain application semantics. The scopes and the ability of composition on low level enables the identification of single entities as well as organizations.

2.2 Delay Tolerant Networking

In this work, we employ autonomous store-carry-forward routers for content dissemination, by using the SCAMPI opportunistic networking middleware, as described in the following section. SCAMPI is built on the Delay-Tolerant Networking (DTN) protocols with the aim to enabling communication in challenged networking environments. This section provides an overview of the DTN architecture developed by the Delay-Tolerant Networking Research Group (DTNRG), as defined in RFC 4838 [Cerf 07].

The DTN architecture is an end-to-end message-oriented overlay network architecture that aims at interconnecting highly heterogeneous networks together even if end-to-end linking may never take place. It comprises DTN nodes deployed in a challenged network, which have capabilities of storing and forwarding messages for an extended period of time. It is an information-centric networking overlay that diverges from the existing host-centric Internet infrastructure, i.e., DTN is message-switched instead of packet-switched and it creates transport links on opportunistic contacts instead of forwarding packets over persistent links.

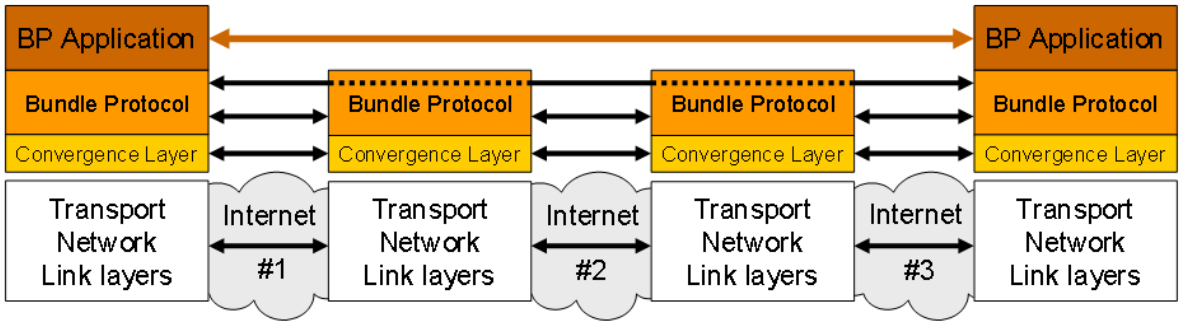


Figure 2.2: Delay Tolerant Networking Architecture. *Adopted from [Dela]*

This architecture is enabled by the Bundle Protocol [Scot 07], a delay-tolerant protocol stack to support intermittent connectivity. The destinations of messages, i.e., bundles, are identified by endpoint identifiers. Nodes register in endpoint identifiers and these registrations are exchanged when two devices meet. Thus, bundles are transmitted in bursts and stored locally until the next forwarding opportunity arises.

More specifically, the DTN architecture is shown in Figure 2.2 and presents the overlay layer implemented by the Bundle Protocol, the application layer (the BP Application layer at the top of the figure) as well as the lower layers of convergence and the link transport layers. In practice, the message passing is done as follows: the applications that are built on top of DTN, communicate by exchanging large application data units (ADU). The ADU is passed to the Bundle Protocol layer, which attaches control information to the ADU by encapsulating it in a bundle. The bundle consists of blocks that apart from the ADU, contain routing, security, caching information and other. Consequently, the bundle is passed to the Convergence layer, the function of which is to break down the bundle into smaller transmission units and pass them to the underlying link layers, i.e., transport layer, network layer, data link layer and physical layer. The Covergence layer abstracts the end-to-end delivery (bundles passed hop-by-hop between the DTN nodes) realized through the lower layers from the higher layers.

A DTN endpoint is identified by a single EID and can be composed by one or more nodes instead of being mapped one-to-one. In order to be a message delivery successful, a message must reach a subset of the nodes, called minimum reception group (MRG) that correspond to the endpoint. This allows the DTN architecture to support unicast, broadcast, multicast and anycast.

2.3 SCAMPI Opportunistic Router

This section presents the SCAMPI opportunistic Router [Kark 12], a store-carry-forward router that enables the opportunistic communication of the resulting system in this work.

SCAMPI platform [Pitk 12] is a service-oriented platform for mobile and pervasive networks, which offers a middleware that hides routing and opportunistic networking from the applications. It consists a communication subsystem, which is responsible for detecting peers and exchanging messages. Direct peer sensing mechanisms are applied to discover peers in proximity and services within communication range based on IP multicast or static IP discovery. To discover remotely located nodes, the nodes exchange information about other nodes they have discovered. The SCAMPI routing subsystem is responsible for the routing of messages in the network based on the discovered peers.

The SCAMPI opportunistic router is developed based on the DTNRG architecture and protocols. The router provides message caching, peer discovery and message routing over multiple hops. The supported discovery mechanisms include IP multicast/broadcast beaconing, TCP unicast discovery and subnet scanning for known ports. The combination of different discovery mechanisms is supported and results in higher probability of successful discovery. The SCAMPI Router opens links between the peers as soon as they have been discovered and uses TCP Convergence Layer (TCPCL) for all message transmissions in IP-based networks.

The router provides a native Application Programming Interface (API) over TCP that can be used by native applications written in any language by implementing the SCAMPI client. The exposed API operations are: publish/subscribe for messages, automatic framing for structured messages, searching for content-based metadata, and peer discovery of nearby nodes. The key entity in SCAMPI that allows the message passing over the Bundle Protocol layer is the SCAMPIMessage, which is an application layer object that provide a map structure where arbitrary string keys map to binary buffers, strings, numbers or file pointers. In addition, there is no need of developers handling the serialization and framing of the map, since it is taken over by the router. SCAMPI API offers message versioning, automatic merging and cache management. SCAMPIMessage can be tagged with namespaced metadata that describes the content, this allows search queries against this metadata by any SCAMPI router. These messages can then be published to services identified by opaque strings and any node subscribed to the service will receive copies of the published messages. Floating content [Hyyt 11] style distribution is supported for limiting the geographical spreading of the messages.

With respect to the router implementation, it has been developed in plain JavaSE, allowing it to run on any platform with a Java Virtual Machine (JVM). Furthermore, the authors developed an Android application that runs the router as a persistent background process on Android devices. In this work, both the Android and Java version of the router are utilized.

2.4 Composability

In this chapter, we discuss the composability design principle, which is the fundamental concept of this thesis, and elaborate on its benefits and applicability in systems architecture. Furthermore, in order to show how composability is essential in industrial software engineering, with the aim to solving complexity in software systems, we present the Microservices Architecture, Docker containers and Google's Kubernetes container management tool. In our work, we have taken the concepts and principles behind those technologies into account, in order to achieve the proposed architecture.

Composability design principle refers to the inter-relationships of components given any system granularity. By ensuring composability in a system design, the resulting system comprises independent, self-contained components that can be assembled in various combinations to satisfy specific requirements. Composability is beneficial at many layers of abstraction, for components, subsystems, networked systems, and networks of networks. A CC can be either stateless which means that it handles each request independently from any previous requests, taking only the given parameters into account or stateful being responsible for a subset of the state of the system and providing a service to other components. Additionally, a CC is characterized by modularity, which means it can be deployed independently, yet it can cooperate with other components.

Composable systems tend to be more inherently available, and reliable than non-composable systems, since the evaluation of the individual parts is easier. However, they tend to be more complex in order to achieve the proper component orchestration. In this work, the developed framework structures the interactions among the CCes in a manner that does not induce extra complexity and efficient coordination can be achieved.

Concerning statelessness, it is less likely that stateless system components have adverse interactions when they are composed with other components, although it is always possible that interoperability and compatibility among the systems and subsystems might not be totally feasible. For instance, in the case of services composition, the Service Stateless Principle [Powe] is applied with the purpose of designing scalable services that consume a reduced amount of resources and thus, they increase their performance and act more reliably. Furthermore, one of the key benefits of composability is the reusability of modules and subsystems, which eventually increases maintainability, replaceability and reduces the overall development effort. The resulted modularization in composable systems allows work distribution, aims to increase understandability and reduces complexity. The following section provides a detailed overview of the system decomposition technique.

2.4.1 Modularity

This section introduces the system decomposition and discusses the fundamental concepts for designing a composable and modular system architecture.

The definition of decomposition stated by Jim Horning [Neum 95] is as follows:

> Decomposition into smaller pieces is a fundamental approach to mastering complexity. The trick is to decompose a system in such a way that the globally important decisions can be made at the abstract level, and the pieces can be implemented separately with confidence that they will collectively achieve the intended result. (Much of the art of system design is captured by the bumper sticker: Think globally, act locally.)

In practice, the decomposition of a system into modules (components) shows the extent to which the systems modules can be separated and re-assembled again. The purposes of that are (i) to reduce complexity ("divide and conquer") by managing small pieces of work independently and (ii) the ability to deal with changes on the systems behavior in the long term by being able to modify or even replace a system component when it is required. In a software system project, the need of modularity is beneficial across all stakeholders. For instance, developers and testers require less coordination effort by working on separate modules and intuitively gain more guidance from the systems structure. Architects can easily adapt the system designs to meet new requirements, manage local changes more efficiently and have the freedom to employ new technologies.

Decomposing a system includes analyzing of the dependencies between elements and forming components out of a set of elements with strong dependencies. The dependencies between the components are captured by well-defined interfaces, which enable information hiding. In functional decomposition, the system is decomposed into modules and each module is a major process step (function) in the application level. Those modules can be decomposed into smaller modules. This kind of decomposition causes several other issues that should be tackled i.e. the functionality is spread all over the system and thus, there is need to understand the entire system in order to make a change. That leads to increase of code complexity and hardens maintainability. An alternative approach is the modular decomposition, where the system is decomposed into modules and each module represents a major abstraction in the application domain. Each module can then be decomposed into smaller submodules.

As a rule of thumb for landing system modularity, the following set of design principles is recommended.

Low coupling and high cohesion

Before we continue with the definition of coupling and cohesion, lets formally define the structure of a system according to [Vlie 08]. The system S is defined by the tuple:

$$S = (C, I, CON)$$

where C denotes the components, $env \ \epsilon \ C$ denotes the system environment, I denotes the interfaces of the components and $CON \subseteq I \times I$ represents the connection between interfaces.

Coupling of a system is the normalized number of connections between components at the same hierarchical level. The formal definition is as follows [Vlie 08]:

$$coupling(S) = \frac{|con \epsilon S.CON | \exists i,j \epsilon S.I : con = connected(i,j) \wedge parent(assigned(i)) = parent(assigned(j))|}{|S.C|}$$

$coupling : S \rightarrow \mathbb{R}$, if a is a component and i its interface then $assigned(i) = a$

There are six types of coupling in a scale of high, loose and low values [Bria 97]. (i) The content coupling (highest) occurs when a component directly affects the working of another component. (ii) The common coupling occurs when two components have shared data, there is lack of clear responsibility for the data, reduces readability, it is difficult to reuse components. (iii) The external coupling refers to components that communicate via an external medium such as file, device interface, protocol, data format. (iv) The control coupling occurs when one component directs the execution of another component by passing the necessary control information. (v) The stamp coupling refers to the common property of the components that declares the precise format of the data structures passed as a complete dataset from one component to the other and at last (vi) the low coupling, called data coupling, occurs when a component passes only the needed data to other components not entire data structures.

Cohesion describes how closely related are the different responsibilities of a component and the level of interactions within that component. The aim in system design is to achieve as high cohesion as possible. The types of cohesion are the following [Bria 97]: (i) The coincidental cohesion (lowest) occurs when the elements of a component are grouped accidentally and there is no significant relation between the elements. (ii) The logical cohesion occurs when the elements of a component are grouped logically and not functionally. (iii) The temporal cohesion takes place when the elements are independent but initiated at about the same point in time. (iv) The procedural cohesion occurs when elements of a component are related only to ensure a particular order of execution. (v) The communicational cohesion refers to operations of component's elements on the same external data. (vi) The sequential cohesion occurs when the output of one part is the input to another, this usually happens in functional programming languages. (vii) The functional cohesion takes place in components that transform a single input into a sing output, only the essential elements to a single computation are contained in the component. High cohesion increases reusability, testability and understandability.

Single responsibility principle (SRP)

The SRP forces the assignment of a single responsibility, i.e., a set of functions that work collectively for a single result, to a single class. A class should only have one reason to change, if there are more then the functionality should be split into more classes.

Separation of concerns (SOC)

A key principle of software engineering which is closely related to SRP. The target of this principle is to minimize responsibilities-per-component ratio. Each component must fulfill its own concrete purpose and its functionalities are encapsulated within the component.

At an architectural level, SOC is the key to building layered applications, such layers might be presentation, business logic, data layer. In addition, it is possible to separate concerns along application feature sets, thus, it is easier to add or remove features in a modular fashion.

Liskove substitution principle (LSP)

> Let $q(x)$ be a property provable about objects x of type T. Then $q(y)$ should be provable for objects y of type S where S is a subtype of T.

The LSP as stated by Barbara Liskov, forbids the child classes from breaking the parent class' type definitions. That means that child classes must not remove base class behavior and violate base class invariants. This principle leads to creation of abstractions.

Don't Repeat Yourself (DRY)

DRY refers to duplication (inadvertent or purposeful duplication) in architecture, requirements, code or documentation. It leads to difficulties in maintenance, poor factoring, and logical contradictions. In practice, that might cause mis-implemented code, developer's confusion or even complete system failure.

Applying the above-mentioned principles and practices leads to a modular design with high level of composability, which improves maintainability, testability, scalability and performance of the resulting system.

2.4.2 Microservices Architecture

In recent years, a new architectural pattern for designing applications out of autonomous services has emerged, called Microservices Architecture Pattern [Rich 15]. This new approach is not precisely defined and, in practice, derives from current trends in industry such as domain-driven design, continuous delivery, small autonomous teams, intelligence in the endpoints, and decentralized data management. In fact, before Microservices have come to play, Alistair Cockburn's concept of hexagonal architecture [Cock 05] drove the organizations away from the traditional layered architectures by isolating the core domain of the application from the technical infrastructure (databases, web services, message queues e.t.c.) that enables it to communicate with the outside world. Large companies such as Amazon and Google turn to small team project organization to encourage autonomy, allow more efficient communication and thus, speed up development. A highlight of this approach is the rule "Two pizza team", as stated by the founder and CEO of Amazon, Jeff Bezos. Those trends triggered the solution of Microservices i.e. applications split into a set of smaller, loosely-coupled services that can be developed, deployed and scaled independently. The intent of this architecture is to enable organizations to deliver software faster, integrate with new technologies and react faster to changes.

Microservices are small, autonomous services that work together to provide a scalable

system. The above-mentioned principles SRP and low coupling high cohesion (section 4.1.) are closely related to the nature of the microservice, in the sense that the boundaries of the service must be explicitly defined and it must serve one single purpose. Following this, we avoid to build large services and we keep the systems simple. understandable and easily adaptable to changes. At this point, the question "how small a microservice should be?" comes to the front. The answer to this question depends on many factors, such as the development team size suitable for managing the service's codebase and the level of intended complexity within the service.

2.4.2.1 Pattern Description

Microservices represent a distributed architecture that simplifies communication among the remote subsystems and minimizes the orchestration needs. It enables a fully decoupled system, where entities communicate via remote access protocols (e.g. REST, SOAP, JMS, AMQP e.t.c.). This is the key characteristic behind the increase of system scalability and faster deployment.

A microservice can be interpreted as a self-contained unit of functionality, part of the entire application, that might be consisted of multiple subunits. In this sense, this kind of service can be defined as a *service component* of various granularity that is part of the system design. The subunits of the *service component* are one or more modules that correspond to either a single-purpose functionality or even a complex subsystem. The *service component* is independently deployable, which means easier deployment, high scalability and component decoupling within the application. They offer well-defined APIs (application programming interfaces) that allows simple interactions within the system. This concept is illustrated in figure 2.3 [Rich 15], where the service components can be plugged to the overall design without affecting the system and being accessed using a REST-based interface implemented by a separately deployed API layer.

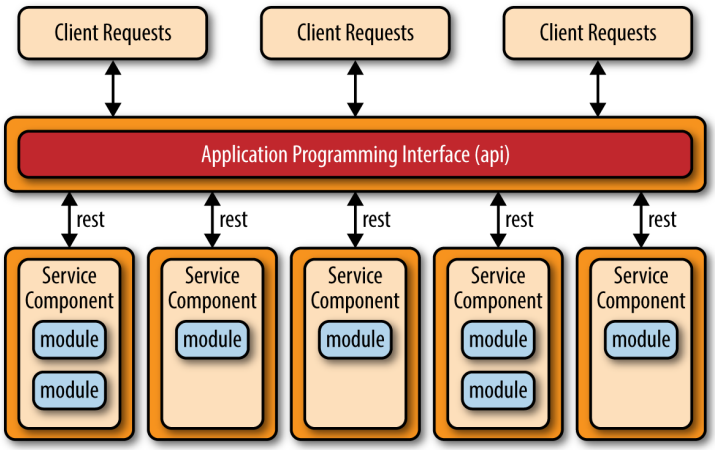


Figure 2.3: API REST-based Microservices architecture pattern. *Adopted from [Rich 15]*

The need of dealing with the increased complexity and inefficiency of monolithic applications invoked the solution of Microservices. One of the key problems in monolithic applications is reliability. When all modules are running within the same process, a failure of any module can potentially shut down the entire process, and that might cause the failure of entire system. Microservices provide module isolation that brings a layer of security. Furthermore, in monolithic designs, embracing new technologies is difficult since all modules are tightly coupled and reside on the same machine. By building microservices, we increase the level of composability which makes our design modular and allow developers to freely choose the technologies fit the most their own service and manage to rewrite or replace it without requiring cumbersome solutions.

As mentioned in [Rich 15], besides the topology depicted in figure 2.3, where the system consists of fine-grained, single-purposed services each one dedicated to a specific business function accessed via an API layer, there are two more common-used approaches to implement the microservices pattern i.e. Application REST-based topology and centralized messaging topology. In the first approach, compared to API REST-based, the system consists of coarse-grained service components responsible for a considerable amount of the overall business functionality. Furthermore, instead of implementing a simple API layer, a user-interface layer takes its place, so that the clients requests are received through traditional fat-client screens. In this case, we distinguish the distributed nature of the architecture, where the user-interface layer and the service components are separately deployed and interact over REST-based interfaces. Concerning the latter approach, as figure 2.4 illustrates, a messaging broker layer is introduced to impose advanced control between the user interface and the service components. This is mostly applied in larger business applications, where there is need of asynchronous messaging, queuing mechanisms, load balancing and scalability. Broker clustering is implemented to deal with the single point of failure on the broker layer.

2.4.2.2 Key Benefits

Microservices offer a plethora of advantages originating from the concepts of distributed systems and service-oriented architecture. This section outlines the key benefits of this architecture as follows:

Composability

The Microservices Architecture pattern enforces a level of modularity that in practice is very difficult to achieve with monolithic applications. Consequently, individual services are much faster to develop, and easier to understand, test and maintain. Microservices lead to high level of reuseability, which means the ability of a piece of functionality being consumed in different ways for different purposes.

Heterogeneity

A system architecture that comprises a set of multiple, collaborating services, offers the

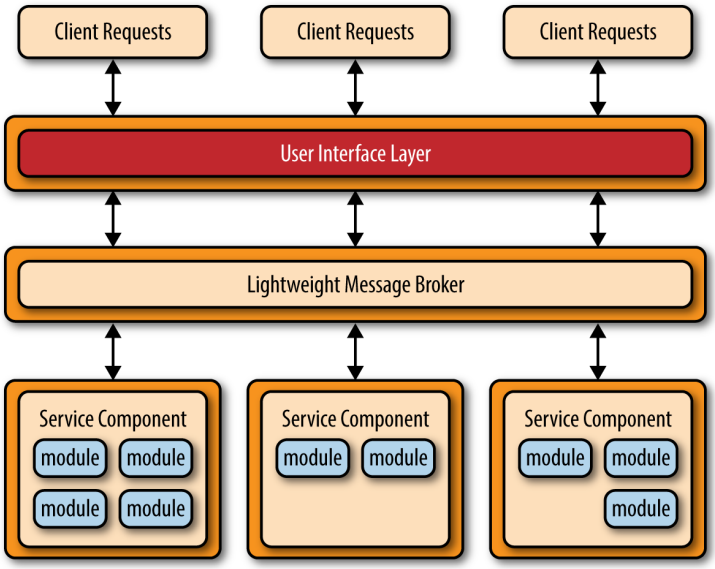


Figure 2.4: Messaging Broker Microservices topology. *Adopted from [Rich 15]*

freedom of employing the best technology for each service, rather than opting for the one-size-fits-all approach. If one part of the system needs to improve its performance, it is much easier to use a different technology to speed up performance. Furthermore, when it comes to the data storage, it is easier to support or interchange between traditional relational and document-based database on the services that it is more reasonable for the application. For instance, many e-commerce applications use a combination of MongoDB and MySQL. The product catalog, which includes multiple products with different attributes, is a good fit for MongoDB's flexible data model. On the other hand, the checkout system, which requires complex transactions, would likely be built on MySQL or another relational technology.

Scalability

Monolithic applications require scaling of the entire system together, even if only one part of the system needs performance improvement. With a large, monolithic service, we have to scale everything together. With smaller services, we can just scale those services that need scaling, allowing us to run other parts of the system on smaller, less powerful hardware.

Faster Deployment

Microservices allows changes to a single service and independent deployment from the rest of the system. Thus, the deployment accelerates. In case of a problem on the particular deployment, it can be easily isolated and a roll back can be done at once. In addition, that means the delivery to end-customers is much faster.

Replaceability

Building a system out of smaller, manageable services leads to easily upgrade a legacy system. When the time comes for embracing new technologies, it is clearly less risky and expensive to completely remove an small component and plug in a newly developed one, rather than replacing a large codebase with one written in a new technology.

2.4.2.3 Considerations

One of the most significant decisions on building a system using Microservices architecture is the correct level of granularity for the service components. There are cases where coarse-grained services might not lead to fully taking advantage of the pattern in terms of low coupling, scalability or testability. On the other hand, if the service components are fine-grained, the need of orchestration arises and that means additional complexity and overhead. In case the need of orchestrating within a user-interface or API layer of the application, it is possible that the service components are too fine-grained. The same applies, in the scenario, where inter-service communication between services is required in order to handle one single client request. In such case, this could be tackled by using a shared database. For example, if a service component handing Internet orders needs customer information, it can go to the database to retrieve the necessary data as opposed to invoking functionality within the customer-service component. Concerning the shared functionality, If a service component needs functionality that resides within another service component or common to all service components, developers violate the DRY principle (see 4.1.) and copy the shared functionality in order to keep the services independent and separate their deployment.

2.4.3 Containerization using Docker and Kubernetes

This section provides an overview of Docker system [Dock], discusses the differences between virtual machines (VM) and containers to better understand the benefits of containerization and describes Google's Kubernetes [Kube] key concepts.

Docker system is an open-source application container engine that provides a platform that allows to pack, ship and run an application as a container. The intent is to allow developers to containerize any application and run it on any infrastructure. To gain an understanding of this technology, we can think of a Docker container as a shipping container that provides a standard, consistent way of shipping any kind of products. In the case of Docker the products correspond to applications.

In order to better understand the benefits of containerization, we discuss containers versus virtual machines, as illustrated in figure 2.5. A hypervisor or virtual machine monitor (VMM) that works by having the host operating system emulate machine hardware and then managing to host VMs as guest operating systems (OS) on top of the hardware. That means that the communication between the guest and host OSes is done through

hardware. In the case of containers, the virtualization is realized at the OS level instead of hardware. The benefits of that are: (i) each of guest OSes share the same kernel and parts of the OS with the host, and, (ii) it is smaller than hypervisor guests. On the other hand, with container virtualization, challenges in terms of isolation and security arise, i.e., the isolation between the host and the container is not as strong as hypervisor-based virtualization since all containers share the same kernel.

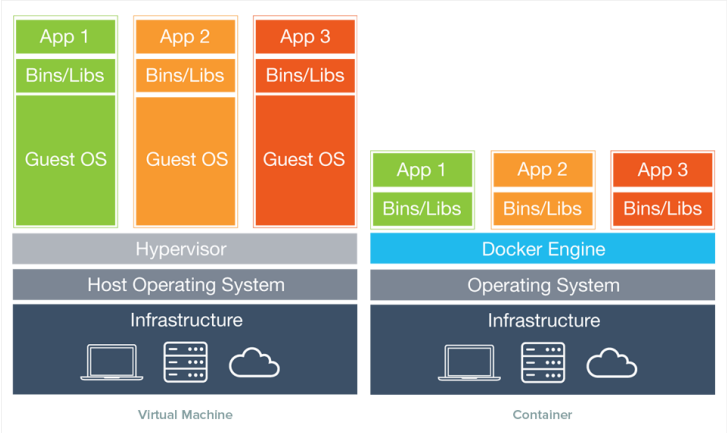


Figure 2.5: Containerization versus hypervisor. *Adopted from [Dock]*

Kubernetes [20] is an open-source platform for automating deployment, scaling, and operations of application containers across clusters of hosts, providing container-centric infrastructure. It essentially provides container management at scale and it is made to manage applications and not machines. The main focus is the management of a cluster of Linux containers as a single system to accelerate development and simplify operations. It supports cloud and bare-metal deployments. Some of its characteristics are the automatic restart feature and placement of containers similar to VMware DRS. In essense, with Kubernetes, we can deploy applications quickly and predictably, scale the applications on the fly, seamlessly roll out new features and optimize use of the hardware by using only the resources needed.

More specifically, the key concepts of Kubernetes technology, as presented in figure 2.6 are the following:

- *Minion*: it is a physical or virtual machine that acts as a Kubernetes worker. Each node runs the following key Kubernetes components: (i) Kubelet: is the primary node agent, (ii) kube-proxy: used by Services to proxy connections to Pods, as explained below, and, (iii) the container technology that Kubernetes use, i.e., Docker.
- *Cluster*: it represents a group of nodes that can be physical servers or virtual machines that has the Kubernetes platform installed.
- *Master*: it is the controlling unit of the cluster, that provides a unified view into the cluster and has a number of components such as the Kubernetes API Server,

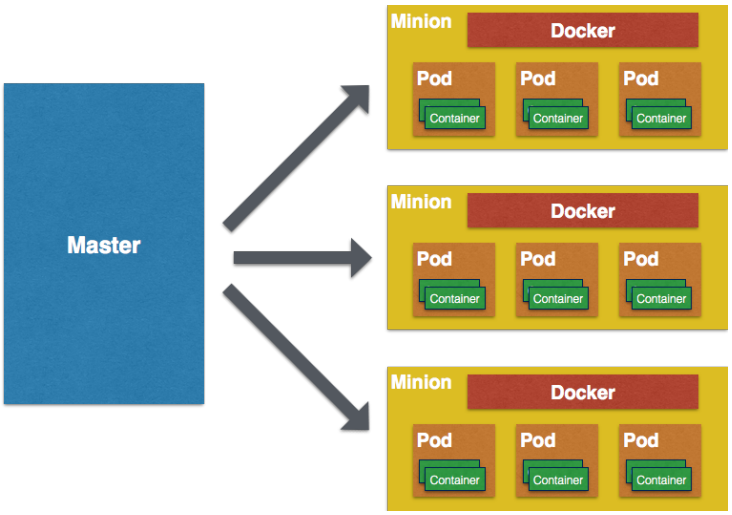


Figure 2.6: Kubernetes Architecture. *Adopted from [Kube]*

that provides a REST endpoint that can be used to interact with the cluster. The master also includes the Replication Controllers used to create and replicate Pods.

- *Pods*: small group of Docker containers that work together, which is the smallest deployable unit that can be created, scheduled, and managed with Kubernetes. So, Kubernetes will place that pod somewhere i.e. lab, cloud, dev environment, and run its containers there.
- *Replication Controller (RC)*: for running multiple instances of a container e.g. 4 copies of a web server. The RC ensures that a specified number of pods are running at any given time. It creates or kills pods as required.
- *Services*: is an abstraction that defines a set of Pods that work together and a policy to access them. Services find their group of Pods using Labels. The pod contains a number of containers that cooperate and the services essential tie pods together. For instance, a Kubernetes service can be a cluster of two nodes where, one of them contains a backend system and a frontend system and the other a replica of the backend system with the purpose of achieving load balancing. Thus, the service defines the routing of the frontend requests to the backend systems.
- *Labels*: are used to organize a group of objects using key-value pairs. So, using those labels it is easy to search for a specified collection of services, backends wherever they are deployed to.

One significant key point of Kubernetes that fits our architecture is the focus on composing applications that are consisted of loosely coupled and distributed microservices, which in our work correspond to small, independent components that can be deployed and managed independently, rather than using a fat monolithic stack running on one big single-purpose machine. On the other hand, our work differs from Kubernetes on the

coordination and discovery of the composable parts of a system, in the sense that in opportunistic networks it is not feasible to provide a centralized control component such as the Master in Kubernetes. We discuss this further in Chapter 3.

2.5 Summary

This chapter provided an overview of the areas of ICN and DTN, and described the SCAMPI opportunistic networking middleware, that has been used in this work. In addition, it elaborated on the composability design principle, its benefits and applicability in system designs by going over modularity and discussing all the design principles that enable the design of modular architectures. Subsequently, we described in detail the microservices architecture and common-used topologies in modern systems and we provided an outline of containerization using Docker technology and the key concepts of Kubernetes platform. The following chapter presents the contribution of this work, which is an architecture for composable, opportunistic applications and services.

Chapter 3

An Architecture for Composable Opportunistic Applications

This chapter presents an architecture for building composable and distributed applications on top of OppNets. Initially, we discuss the motivation and intent of this work by providing a set of real-world use cases for which those kind of applications could be developed, and their requirement analysis. Subsequently, we elaborate on the key concepts that enable this architecture and its structure. At last, we discuss important considerations and security issues.

3.1 Intent and Motivation

The fact that modern mobile devices such as tablets, smartphones, smart watches and others are equipped with powerful computing, storage and sensor capabilities (GPS, camera, accelerometer and others), as well as multiple network interfaces, i.e., Wi-Fi, 4G, Bluetooth, NFC and others, empowers the idea of building application making use of those resources by enabling peer-to-peer communications. Furthermore, in recent years, social networking, proximity-based services and applications such as Nextdoor [Next], Groupon [Grou] and Foursquare [Four], concentrating on meeting people, finding nearby restaurants or special offers, have gained great popularity. However, in most cases, the utilized infrastructure is Internet-based using centralized content-servers, even though the content is ephemeral and locally relevant. There have been numerous cases where this kind of centralized services, e.g. cloud service providers, that maintain important and sensitive user data have been targets for attack, where data disclosures and malicious break-ins have taken place. Let alone, cases of user traffic monitoring with the aim of collection and analysis of user content in the shake of political and industrial interests. Considering the above-mentioned as well as the indisputable fact that there are many areas outside the developed world, where Internet provision is highly costly or even impossible, led us to

examine the feasibility of designing and implementing localized mobile applications and services running on top of challenged networks by leveraging opportunistic contacts.

3.1.1 Scenarios

Below we present scenarios where this sort of opportunistic services could be developed to overcome the above-mentioned issues.

3.1.1.1 Polling Application

In this scenario, we consider a dynamic environment of mobile device (i.e. laptop, smartphone and tablet) users who are called at a random point of time to respond to a poll. A talk is being held in a lecture hall at a university and the speaker wants to request the opinion of the audience on a talk-relevant topic. However, there is no Wi-Fi access point and thus, the speaker is not able to create a poll online and request the audience to access it and respond to it. The speaker then takes advantage of an opportunistic poll creator application that has installed on her smartphone and her laptop's storage and computing capabilities. She turns her laptop to an access point and asks from the audience to connect to it, she also connects her smartphone. Then she opens the application, which starts publishing all the essential poll service binaries to the network. Those files correspond to different versions of a mobile poll participant application suitable to be run on different devices and a poll service able to collect and process the user generated content in order to provide the poll results. At the end, the speaker is able to create a poll with her smartphone and the mobile device users can receive it on their device and respond to it. The results are calculated and shown on the projector connected to the speaker's laptop. In addition, latecomers can also join the polling service and respond to the published poll.

3.1.1.2 Course Rating System

We envision a course rating system in a university setting that allows students to review their courses, professors get feedback on their lectures and view the ranking of their course at the end of each semester. A professor decides to request from the students to provide their review on the course anonymously. In order to do that, the professor distributes the questionnaires using a mobile application, like it was described in the polling application scenario. The students connected to the class network receive the questionnaire on their mobile devices and start filling it in. As soon as they finish, they submit the questionnaire. The submitted questionnaires are now stored and processed in an in-network throwbox-type node located in the university campus that collects the reviews of courses during the academic year and calculates a ranking list based on the students reviews and publishes it along with the reviews of each course at the end of each semester. At that time, when mobile device users pass by the throwbox node, they

receive the ranking list and the reviews which are also shown on the monitor of the central university hall.

3.1.1.3 Event Registration and Check-in Application

This scenario refers to a university event application that assists in notifying students and the university staff about events, creating guests lists and checking in the event attendants. One of the members of the event organization team is responsible for distributing the event poster using the event-poster mobile application. The university is separated in neighborhood networks where the access point is a throwbox-type node with storage and computing capabilities. The event poster distributor, moves from one neighborhood to the other and offloads the event poster on the throwbox node which publishes the event poster in the network. In addition, on the way to a neighborhood network, the distributor can turn the smartphone to an open access point and publish the event poster to the students that get connected to it. The students who are connected to a neighborhood network or the distributor's device network receive the event poster and with it they can submit their attendance to the event. Also, the students can also share the event poster to help in the distribution. In case they join the event, their response is sent to the network node that creates the guest list. The member who is responsible for holding the event guest list goes to the neighborhood networks in the campus and collects the guest lists and merges them with the event guest list application. At the entrance of the event building, the guest list holder's device is waiting to get a notification from the guests device that they have arrived and checks them in.

3.1.2 Requirements

The goal of this work is to enable the design of complex mobile opportunistic applications given that the employment of centralized infrastructure used in standard web applications is not possible. This section enlists the main requirements that we derived from the above-described scenarios:

1. *Dynamic service instantiation*: taking into consideration the absence of centralized infrastructure in the given scenarios, the service instantiation must be done by utilizing the temporary in-network resources of the mobile devices that happen to be connected to a particular network. In all three scenarios, the systems employ the end-users' devices to instantiate the service by publishing executable files to the network. There is no access to a centralized application distribution store. The users happen to need a service at a certain point and they create it by using the available resources.

2. *Single-purpose, autonomous subsystems*: in every scenario, the overall system is partitioned into different subsystems with separate responsibilities. For instance, in

the case of the polling application, the subsystems are the following: an application running on the speakers smartphone that creates the poll, a participant application is running on the audience devices allows them to respond to the poll, an application is running on the laptop and shows the poll results. In addition, there are subsystems that do not offer any interaction with the user e.g. calculation of poll results. All those subsystems have a particular functionality and operate independently on different devices. In addition, in such challenged networks, the possibility of a node to leave the network is high, therefore, the interaction model that should be used must support loose coupling among those subsystems. Partitioning the system into smaller subsystems, is a common technique to deal with complexity which is crucial in this kind of scenarios where additional complexity might be induced, since centralized content-servers and orchestration cannot be used.

3. *No central controlling unit*: utilizing a centralized coordinator to orchestrate the various applications and services running on top of OppNets is difficult because at any moment it can get out of direct contact with any other nodes, or it might leave the network and never return. We need to employ a different approach that achieves the applications synergy tolerating those kind of challenged communications.

4. *Publish-Subscribe (PubSub) message delivery*: in all scenarios, the main function is publishing user-generated content to multiple recipients. This can be supported by the publish-subscribe interaction paradigm.

5. *Remote procedure call (RPC) messaging*: there are cases where a subsystem requires to perform request-response style communication with an other subsystem residing on a different process on the same or a remote machine.

6. *Supporting heterogeneous devices*: we need to take advantage of the in-network resources to the greatest extent. To achieve that the system should support the usage and collaboration of heterogeneous devices. In all scenarios, mobile users might carry smartphones, laptops, tablets or smart watches, thus, supporting multiple screens is essential.

7. *Security*: in all scenarios, applications and content is shared with end-user devices from unknown sources. We consider that the applications are set in a university environment which means that those systems support proper authorization, however such networks are often subject to different kinds of attacks by malicious nodes, thus, an additional security layer is required to address threats such as injection attacks, resource depletion, confidentiality disclosure and privacy invasion. The security layer is out of the scope of the thesis and it is only discussed briefly later in this chapter.

3.2 Key Concepts

This section presents the key concepts adopted in this work to design systems that fulfill the above requirements and enable opportunistic applications scenarios.

3.2.1 Neighborhood Networks

In the scenarios presented in the previous section, various network topologies and interactions have been described. Figure 3.1 illustrates the opportunistic interactions that occur in neighborhood networks, which are composed of small, wirelessly connected static and/or mobile store-carry-forward nodes and are built and controlled by the users themselves, and points out the occurrence of this kind of interactions in our university scenarios.

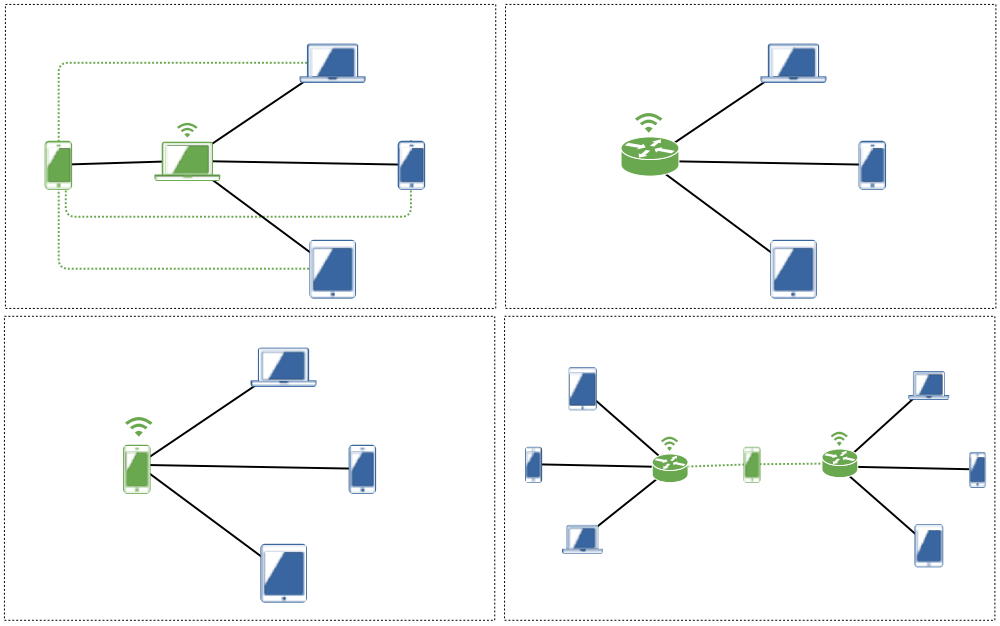


Figure 3.1: Neighborhood networks topologies

More specifically, (i) in the top left figure, there is a mobile access point (green laptop) to which the remaining devices are connected to, communication links are created directly among the mobile nodes (dashed green lines) or through the access point node (black lines). Such a topology is used in the scenario described in section 3.1.1.1, i.e., the speaker turns the laptop connected to the projector to an access point and the audience mobile devices connect to it to publish and receive content. In addition the speaker uses a smartphone to directly interact with the audience devices. (ii) In the top right figure,

the mobile devices are connected to a stationary store-carry-forward node (green router) that publishes content in the network. In scenario 3.1.1.2, the network is consisted of a stationary node, called throwbox, able to persistently maintain content, and mobile devices that connect to throwbox to publish and receive content i.e. course reviews and final course rankings respectively. (iii) In the bottom left figure, a smartphone is turned into an access point and publishes content to the devices connected to it. In 3.1.1.3, a mobile device user turns a smartphone on-the-go to an access point and other passengers connect their devices to it, (iv) In the bottom right figure, two separate networks created by two stationary nodes (green routers) share a subset of their network content provided by a mobile device (green smartphone) that moves from one network to the other. In 3.1.1.3, a mobile device user receives the poster event published in the network and by moving to an other network spreads it further.

3.2.2 Composable Component

The fundamental key concept of the proposed architecture is the Composable Component (CC), a simple, self-contained unit of functionality. The major goal of introducting this concept is to enable system componentization in order to facilitate the design and development of composable distributed applications, i.e., applications designed as a suite of components that collaborate to provide the resulting system. The main benefits are (i) the ability to build a system that fulfills the requirements described in section 3.1.2, (ii) the reduced complexity and (iii) easy deployment in opportunistic topologies. A CC must have the following characteristics:

1. *small*: it is a small manageable piece of work that provides a distinct, single-purpose functionality, which is part of the overall system.
2. *modular*: it is self-contained and has well-defined APIs to communicate with other CCes to provide the resulting system.
3. *opportunistic*: it communicates opportunistically with other CCes, which reside on different processes on the same or a remote machine, using RPC or PubSub communication style. The duration of the presence of the component in the network is unknown.
4. *independently deployeable*: it comes in binary executable code, ideally with their own runtime environment, if it is not supported by the platform.
5. *indepentently pluggable* and *replaceable*: a CC can be (un)plugged to the system and operate without any manual administration. It can be replaced and upgraded without affecting the other CCes.
6. *stateful* or *stateless*: it might be stateful by encapsulating a database layer or stateless, in which case it interacts with a remote database to either transform further the stored content and act as a service or display content to the end-user.

7. *service* or *widget*: the CC can take two forms; (i) the service component (stateful or stateless), which contains business logic and its client might be an other service or a view and (ii) the widget component (stateless), which is the application the provides a UI to the end-users. We can match those two terms to the commonly-used terms backend and frontend system respectively. A set of service components compose the system's backend and a set of widget components compose the system's frontend. We elaborate further on the CC's forms in the following section.

3.2.3 Decentralized Event-based Orchestration

One of the most significant design decisions, we had to take, is how to coordinate the CCes to efficiently work together and provide the desirable result. Considering the distributed nature of the neighborhood network scenarios, we decided to employ a choreography technique rather than centralized orchestration.

There are three different approaches to enable the system processes work together: (i) when calling multiple services to support a given requirement, we need to introduce a gateway layer which orchestrates the service calls to the required services and aggregates the final response and sends it back to the original consumer. The gateway layer should only have orchestration logic. Since we target OppNets, it is impossible to have a central node because at any moment it can loose direct contact with any other node, or it might leave the network and never return. (ii) Point-to-Point interaction is a commonly-used integration model, where services call directly other services asynchronously or synchronously. This approach introduces a great level of complexity and makes the system hard to change and maintain. (iii) A more appropriate mechanism is to use an asynchronous messaging style and an event stream to choreograph the services. By employing the PubSub interaction paradigm, this approach ensures that there are no bottlenecks in the final application-system since the communication protocol is based on non-blocking asynchronous calls and the services are completely decoupled since they publish events without the need of directly connecting to the recipients. The recipients are smart enough to process the request when they can and return a response without blocking the sender. In this case there is no central entity that takes care of the services interactions and the services coordinate by themselves.

In our design, we adopted an event-driven approach. A CC publishes an event for content dissemination, database operations, or other requests and the interested CCes subscribe to those events. When the CCes receive the events, they process them and perform the required tasks, if the event represented a request, and publish a new event that is the response or use the encapsulated event data to display information to the user. The CCes can store the events and thus keep track of the state of other CCes and still be completely decoupled.

More specifically, in this work we applied the PubSub and RPC messaging models using

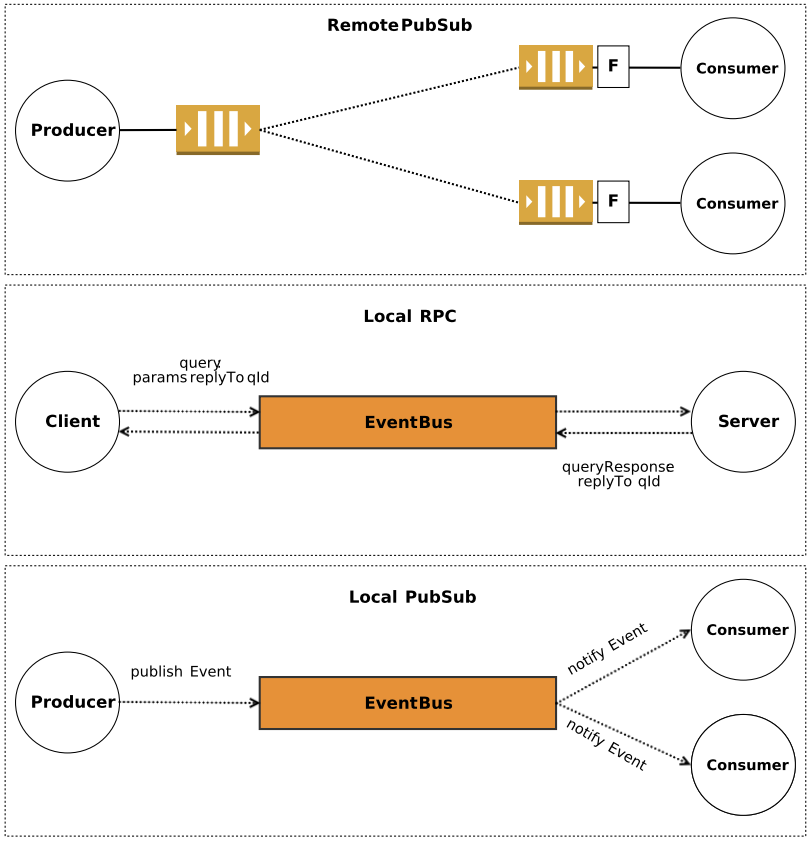


Figure 3.2: Messaging models

intermediary bus abstractions to decouple the components. In this section, we discuss the models and in section 3.3, we present the bus abstractions in detail. Concerning the PubSub model applied in this work, the main high-level elements are the producer and consumer of the event, the event tag and a bus with routing logic that is consisted of a message queue for storing the events with additional information e.g. the event tag, and a filtering module that assists in filtering the events at the consumers end. Given the challenge of the opportunistic communication links, in practice, the bus consists of a queue on each producer and consumer that buffers all the events to be sent and the received ones respectively. Once producers and consumers meet the message exchange occurs. A CC might act as both consumer and producer. In the top of the Figure 3.2, an abstract representation of this model, called in this work Remote PubSub, is illustrated in the context of CCes communicating opportunistically. In the bottom figure of 3.2, the Local PubSub is shown. This kind of messaging model is used among CCes that communicate using an event bus (see 3.3.5) and exchange messages based on the stored state of the system. It enables pull-style communication among CCes exchanging content previously received with the Remote PubSub. These interactions are realized at a higher layer than

the interactions with Remote PubSub. We discuss about it more in the section 3.3. At the middle of Figure 3.2, the Local RPC is presented. This messaging model, as with the Local RPC, occurs at a higher level utilizing an event bus. The Client publishes a query using a query id and the query parameters to the event bus and the bus notifies the server who has subscribed to those queries. Server processes the request and publishes a response event to the bus. The client has subscribed to those events and receives the response.

3.2.4 Decentralized Data Management

In the discussed design, we assume that a CC can be stateful and encapsulate its own database with which shares data only through push and pull-style subscriptions to content tags. To realize that, we need to properly decompose the system into smaller CCes that can be developed and deployed independently, select the CCes that act as services and decide which should be responsible for what data models. This contrasts with the classical three-tier architecture which leverages one central database, that makes deployment and development harder. Realizing this kind of design has the following benefits: (i) no implementation dependencies among the CCes, for each one a different technology for particular purposes (e.g. MongoDB, SQL) can be used without affecting the development and deployment of other application CCes. (ii) Each CC is designed and built around an application domain subset. That creates barriers among the CCes which leads to design modularity. (iii) By following this approach in combination with the event-driven communication (3.2.3), we achieve efficient data sharing without the need of a shared database, by replicating the data to the CCes that need it.

Nevertheless, it is important to correctly decompose the system and decide on the services, otherwise unnecessary complexity might be introduced. For instance, communication among many remote CCes to serve a single request induces communication delays and complexity due to additional coordination, thus, it is better to rethink of the system granularity and merge service components that serve related purposes into one, i.e. if a change requires an update in another part of the system, those parts should be close to each other.

An alternative technique that could be applied in particular use cases is to use a shared database, but with a private schema per service in order to keep clear boundaries among the services. In this case, there are service CCes that are stateless and access a remote database. In our scenarios, this is feasible in the cases of stationary store-carry-forward nodes utilization e.g. in the scenario 3.1.1.2, the throwbox node publishes the final ratings and reviews to other devices on which CCes might be running and using this content for further purposes.

3.3 Framework

This section presents a composable system architecture that provides a framework suitable for enabling mobile opportunistic applications and services, by leveraging the above-mentioned key concepts. The discussed network topologies presented earlier are composed of arbitrary number of devices, i.e. mobile nodes (laptops, smartphones e.t.c.) and/or static nodes (store-carry-forward routers), the collaboration of which is enabled by the framework. The architecture enforces separation of concerns and a decoupled system that allows the developers to work on different parts of the system and build applications and services independently that can be deployed on different devices. In this work, we attempt to adapt the Microservices architecture to opportunistic environments and provide a structure that reduces the additional complexity that occurs due to the extra coordination interactions among the microservices required to enable the resulting composed system. This section provides details on the elements of the framework and design decisions taken to this goal.The implementation of the application framework is presented in Chapter 5.

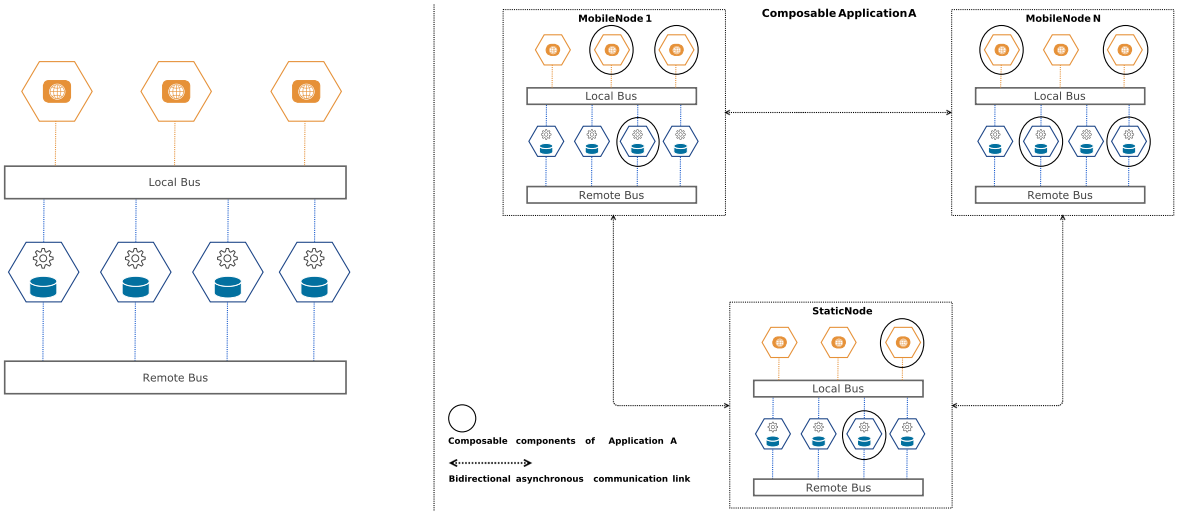


Figure 3.3: Composable System Architecture

The left part of the Figure 3.3 presents the system structure within a node instance, i.e., the system is composed of a set of CCes that come in two types, i.e. *service* and *widget* which represent the services and applications that wish to communicate opportunistically and two bus abstraction layers i.e. the *Local Bus* and the *Remote Bus* that are responsible for the communication between those CCes. More specifically, with the Local Bus we enable an event-driven communication model for interaction among the CCes running in the node instance, whereas with the Remote Bus, we support communication among service CCes running within the same node or among remote nodes. To enable the

dynamic instantiation of services composed of multiple applications and services running on different nodes, an instance of the framework must run on each device in order to enable the event-driven communication models both within the node and among remote nodes. On the right of the Figure 3.3, a composable application built out of multiple CCes (marked with a black circle) on different node instances is presented. The nodes in the figure host a set of services and UI widgets that serve different domain-specific purposes and communicate locally using the Local Bus and remotelly using the Remote Bus that enables peer-wise asynchronous communication links.

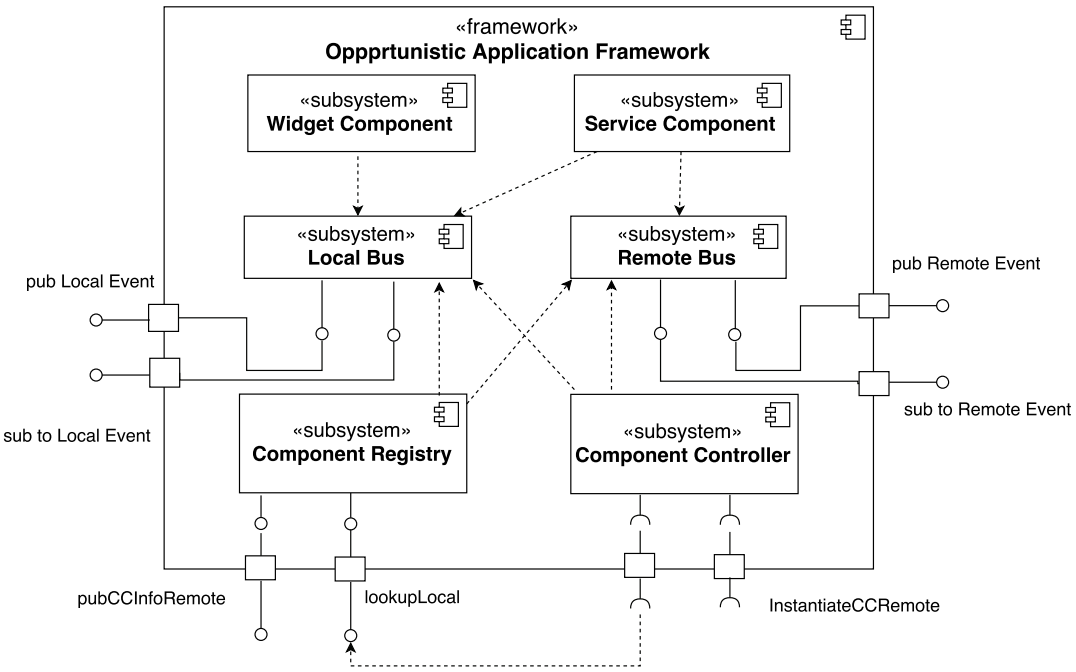


Figure 3.4: UML component diagram of the Opportunistic Application Framework

Figure 3.4 presents the internal subsystems of the opportunistic application framework and their dependencies. The subsytems of the framework are the bus abstractions as introduced previously, the component registry and the component controller. The Local Bus provides pub/sub interfaces for local events and the Remote Bus for remote events. The component registry maintains information of the CCes installed within a node instance and the component controller uses the component registry to check the state of the running CCes, manage their lifecycle and install new CCes on the node. To enable communication among remote nodes, the framework must run on every node.

The following sections describe the framework subsystems and their interactions in detail and Chapter 4 presents a set of distributed application designs that show how concrete applications and services are being created by different CCes in multiple node instances.

3.3.1 Service Tag and Event Message

The *service tag* corresponds to a service identifier that can be used to publish content and subscribe to receive the particular content from a network service. For instance, a CC may wish to publish content related to nature thus it publishes messages identified by the tag *nature* and other CCes that wish to receive this content subscribe to the tag *nature*. The *event message* is a message exchanged between CCes with the following properties: (i) service tag, (ii) event type, i.e., local and remote, the remote event is published to the Remote Bus and the local to the Local Bus, (iii) event content, the encapsulated generated content and (iv) event tag, a unique description of the event within the *service tag* scope. The event messages enable CCes to issue commands, requests and publish content. All CCes must agree on a common set of event messages in order to achieve proper coordination.

3.3.2 Service Component

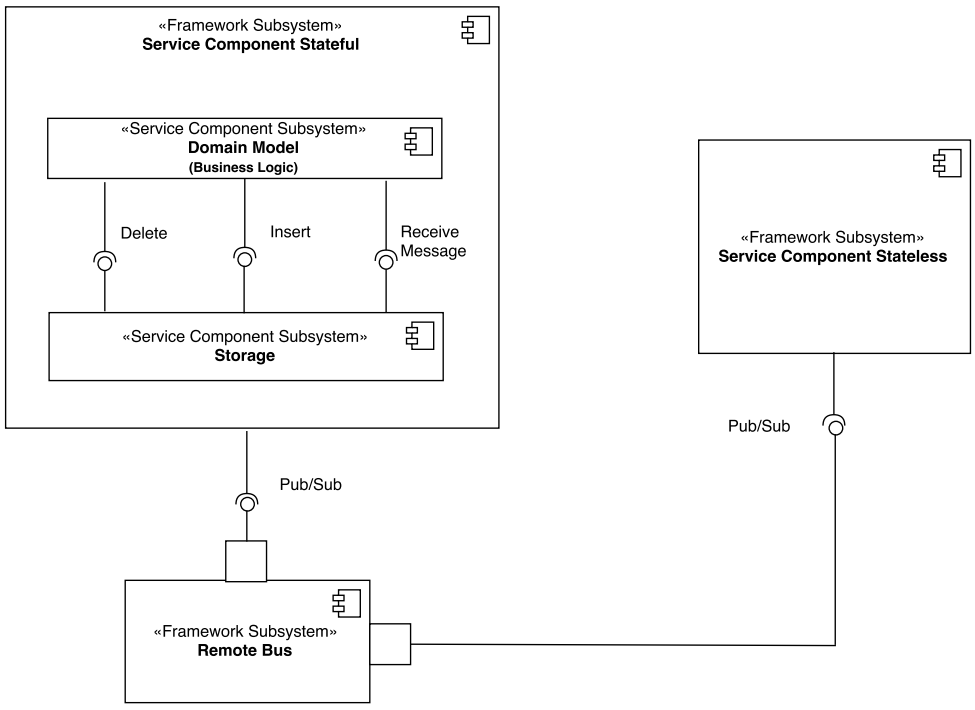


Figure 3.5: UML component diagram of stateful and stateless service components

The Service Component is a self-contained, composable component that represents a service built for a single domain purpose and might be composed of the storage and domain model (business logic) layers. It can be attached into an event message as a binary and published to the network. The framework allows nodes to receive and execute the binaries. This functionality is realized by the Component Controller as described in

section 3.3.7. Each service component uses the pub/sub APIs exposed by the Remote Bus to communicate opportunistically with other service components. More specifically, it publishes and receives event messages to/from the Remote Bus mapped to the subscribed service tags. As soon as, it receives an event message from the Remote Bus, it stores it and processes it properly. Subsequently, it might publish a local event to the Local Bus which, in turn, will publish it to the local *widget components* which have subscribed to it, and/or create a new remote event and publish it to the Remote Bus, which, in turn, will notify in-network service components.

The storage layer is an abstraction that provides an API to be used by the business logic layer in order to manage the received and published event messages and their content. The storage layer notifies the business logic layer about a new event message as soon as it has been stored. Then the business layer can handle the message as required. It can even discard it. The Service Component can also be designed as a stateless component. In this case, it does not contain a storage layer, instead the business logic interacts with a remote database hosted by an other service component. Figure 3.5 illustrates a stateful service component and its inner subsystems as well as the pub/sub interface of Remote Bus that the service consumes. In addition, the dependency among the stateful service component and a stateless one is implied in the figure, since those two components can communicate with each other using a publish-subscribe messaging model via the Remote Bus. For instance, this can be realized in scenarios where a static node exists in the network and stateful service components run on it publishing aggregated content to the network and stateless service components run on mobile nodes who come and go and are interested in just reading that content.

In addition, a service component might even encapsulate logic that includes connecting to a remote web service and post the localized content to it. The behavior of the service component is hidden from the rest of the system and it only exposes its local and remote pub/sub interfaces to communicate with the in-network CCes.

3.3.3 Widget Component

The Widget Component represents a platform-dependent user application that contains all the views and view controllers to display the content to the user and handle the user events. The only dependency it has to the framework is the Pub/Sub interface of Local Bus abstraction through which it can read and write user-generated content. It is not aware of the underlying networking infrastructure and it only subscribes to local events and listens to messages coming from the layers beneath. The widget can be either a standard user application with input views or even an advanced application that leverages hardware capabilities through platform APIs such as sensors, camera, GPS and others. The architecture of the widget is decided by the widget developer.

Figure 3.6 shows the framework local bus, a sensor and a view widget component. Both

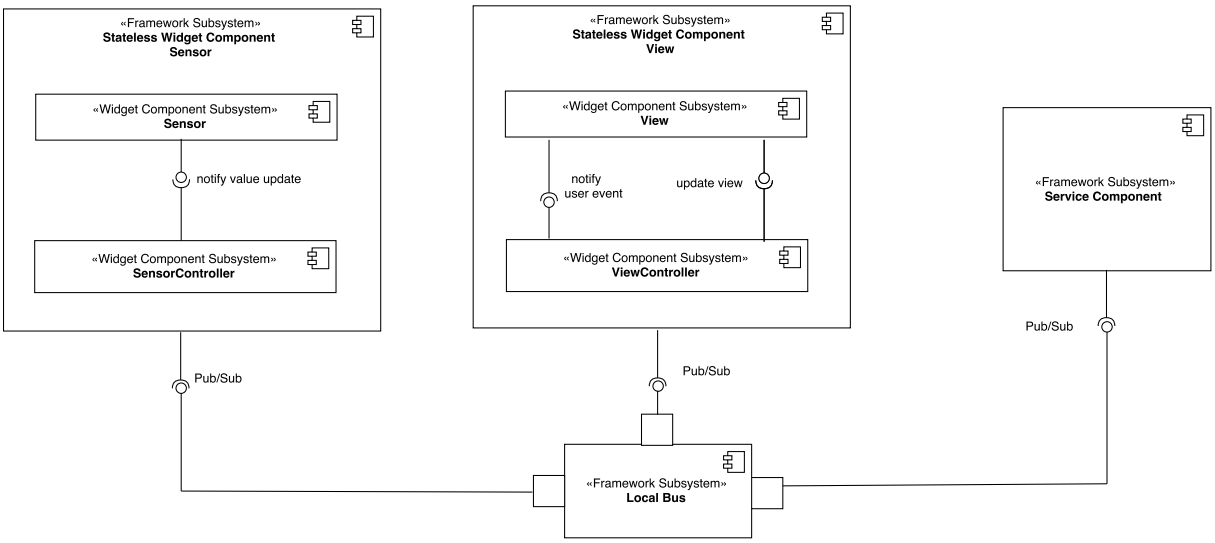


Figure 3.6: UML component diagram of widget components

view and sensor widgets consume the Pub/Sub interface provided by the local bus in order to publish and subscribe to local events. The view widget is composed of the View subsystem which contains all the application view elements and the ViewController subsystem which is responsible for receiving the local events published by the local bus and handle them properly in order to pass the required information to the views. In addition, it receives the user input and publishes it to the local bus. In the case of the sensor widget, it is composed of the Sensor subsystem which, in this figure, is a platform sensor module that provides an interface for getting notified when a new sensor value has been recorded and the SensorController subsystem which gets the sensor value and prepares a local event encapsulating the value and publishes it to the local bus so that the underlying service components can get the value.

3.3.4 Remote Bus

The Remote Bus abstraction is the fundamental architectural element that enables opportunistic communication links between the network processes using an event-based approach. It provides publish-subscribe messaging (Remote PubSub) as described in section 3.2.3 and runs on all network nodes. It encapsulates an opportunistic networking middleware that enables it to act as the intermediary between co-located and remote service components (see 3.3.2), which have agreed on a common set of event messages to communicate and can subscribe and publish to service tags. The middleware was used in this work is SCAMPI and its main features are the following:

1. *peer discovery*: is responsible for detecting peers through the communication interfaces. The middleware provides IP multicast discovery used for both IPv4 and IPv6, and unicast discovery using known targets (IP/port). In addition, the

middleware offers the possibility to indirectly connect nodes that might have never directly met. This is done with a discovery mechanism on top of direct discovery, called multihop. A node keeps track of all the peers that has discovered and share the information with all other encountered nodes. This is mostly used in networks with a large amount of nodes where it is not efficient to open a direct link to each one. The middleware takes over the decision of which links should be opened and remain active and which are unnecessary and must close, thus, scaling can be achieved in dense networks.

2. *routing*: is responsible for deciding which messages should be passed to which peers. It supports a framework of routing algorithms that can be used in a variety of network scenarios. For instance, in case we need to deliver presence information, flooding-based protocols should be used, whereas to efficiently deliver instant messages, unicast routing protocols are used.

3. *storage*: the Remote Bus supports a storage abstraction for caching the network service event messages and peer information. It supports add/remove/update/fetch operations with messages. The storage size can be configured and the messages can be marked as persistent or temporary. Versioning and duplicate detection is supported.

4. *publish/subscribe messaging*: is the main communication mechanism offered by the middleware. It resembles the store-carry-forward abstraction of DTN architecture, in that the messages a self-contained and are forwarded in the network hop-by-hop as complete transmission units. Nodes express interest in receiving messages belonging to a particular service by subscribing to its identifier. Any node can publish messages to any identifier and then the middleware tries to deliver to all interested subscribers.

Using the Remote Bus, a service component that needs to access the data of an other component can subscribe to a service tag and a set of event messages of the particular service that it needs to consume and receive all those event messages. Similarly, an other service component can publish event messages to a particular service tag in the network.

3.3.5 Local Bus

The Local Bus abstraction is used to decouple the widgets layer from the services layer and realize the messaging models Local PubSub and Local RPC as described in section 3.2.3. The purpose of widgets having no dependency to the remote bus is that widget is assumed to be a simple component, which encapsulates views and/or platform-dependent functionality such as sensors only. No network code and no business logic belongs to this kind of composable component. It can provide application logic and subscribe to local events to receive and display content to the user or receive user input and publish it to the local bus, which in turn is responsible to forward it to all interested subscribers for further processing. In practice, if we want to deploy a widget that listens to a remote

service component, we need then to deploy on the same node a service component that it actually listens to the remote service component feed via the remote bus and publishes events to local bus, so that the widget can receive the feed.

The framework enforces separation of concerns and applies the SRP, which leads to a more modular design that implies increase of composability. The widget and service developers can work separately and concurrently, having initially agreed on the local and remote events that must be used for the components communication. The widget developer builds the UIs and is the expert on the supported application platform, and the service developer is responsible for handling database management, (de)serialization and logic without depending on the technologies used for the widget development. The developers work is a black box to each other and the only common concern is the communication protocol. This approach promotes a clean and modular architecture that improves testability and maintainability. In addition, developers can replace/upgrade anytime their composable component or deploy more components that can listen to local events, without getting concerned on side effects on the system.

Furthermore, by using the local bus, the system is more robust, because if the widget fails, till it gets up and running again, the service component will keep listening to the network content and executing its tasks and as soon as the widget is back, it will notify it with the collected content.

3.3.6 Component Registry

The Component Registry is responsible for keeping track of the CCes. It persists the CCes binaries and metadata helpful for their installation and configuration. It runs in its own process on every node and exposes a pub/sub interface, enabled by the Remote Bus, to access information about the CCes such as name, state (active, running, disabled), type, version and others. This enables the discovery of CCes in the network and can be used in complex scenarios where, for instance, intent for composing an application out of specific CCes occurs and there are multiple in the network with similar functionality. An event message of type registry query can be published to the service tag *component-registry*, requesting of the list of CCes running and based on this information, the proper service binding can be decided. A concrete example is a user-composable application, where the users can choose what service and widget components want to use and bind them together, i.e., the user decides to use the service component *university-party-photos* and bind it to the widget component *awesome-gallery-view* instead of the widget *simple-gallery-view*. In addition, it provides local pub/sub interfaces, enabled by the Local Bus, to allow Component Controller to register and manage CCes.

3.3.7 Component Controller

The Component Controller undergoes the management of the CCes on a node. It runs in its own process and handles the lifecycle of a CC, it is able to install, uninstall, stop, start and replace a CC. It provides a pub/sub interface where other CCes in the network can publish their services and widgets, including their metadata. Then it decides whether the received CC should be installed on the node or not. It queries the Component Registry through the Local Bus to look up, register and unregister CCes.

3.4 Security

In this work, we discuss an architecture in OppNets, which are characterized by long communication delays, high mobility of nodes and disruptive links. Those features lead to difficulties in designing applications and services as well as authentication and access control mechanisms. In OppNets, centralized key servers and central certificate authority is not possible to be used, since end-to-end communication cannot be established. Such networks focus on disseminating the content to interested nodes, rather end-to-end communication. In this section, we provide a brief overview of the potential security challenges in the discussed scenarios and solutions that have been proposed and can be taken into consideration for future research on this work.

The major security threats in such scenarios are the following: (i) any unauthorized access and utilization of resources can cause system performance degradation, since the in-network resources are limited. (ii) the architecture can allow code injection attack since it is easy for the attacker to inject executable source code directed to participate in a composable application in the network. Code Injection occurs as the result of including an external resource and it is commonly referred to as a Remote File Inclusion (RFI). (iii) Event messages are being stored and forwarded to intermediary nodes to reach the destination node, that can lead to confidentiality disclosure by copying the message on one of the intermediate nodes. (iv) In addition, privacy invasion is feasible since the intermediary nodes that store the event messages can steal destination node location data, identity or other sensitive user data.

To deal with the above-mentioned threats, requirements such as privacy protection, authentication, authorization and access control, data confidentiality and integrity must be taken into account. In [Wu 15], the authors propose a general security architecture built on five modules: authentication, secure routing, access control, trust management and cooperation, and application/user-specific privacy protection. They state that generally, one or more security modules can be selected and implemented, depending on different threats and specific user or network requirements. In [Lili 07], the authors provided a similar security framework, but did not provide concrete security solutions.

3.5 Summary

This chapter discussed the proposed architecture of this work. Initially, it provided the motivation behind this architecture, real-life scenarios that this architecture is applicable and the requirements needed to be fulfilled to realize those scenarios. Subsequently, we dived into the key concepts that enable the discussed framework and how these concepts help in solving particular issues in designing and developing mobile opportunistic applications. A detailed high-level description of the framework structure is following, while at the end we discuss one of the most significant challenges in such scenarios, security. Implementation of security mechanisms is out of the scope of the thesis, thus, we only discuss crucial security challenges in OppNets and solutions concerning the particular scenarios.

Chapter 4

Designing Composable Opportunistic Applications

The goal of this section is to initiate the evaluation part of the thesis, which is entirely presented in the chapters 4, 5 and 6. In this chapter, we present a set of composable applications and services designs to demonstrate the applicability of the discussed architecture presented in Chapter 3 in various contexts and network topologies and show that it is feasible to design and develop applications that are partitioned into multiple composable components running on different devices, the intercommunication of which is enabled by opportunistic pair-wise contacts. We begin by presenting the polling application scenario described earlier, for which we additionally provide the implementation and its evaluation in the following chapters, and we continue with two more system designs, a course rating system and an event organization application, both described in Chapter 3.

In all scenarios, the systems comprise static (or infrastructure) nodes, which have stronger capabilities in terms of storage and computing power. Those might be a Liberouter [Kark 14] store-carry-forward router, a laptop, desktop or any other powerful device running an instance of the SCAMPI. For the mobile nodes, we assume that they are smartphones, tablets or other modern mobile devices with moderate capabilities, and they run an instance of SCAMPI as well as the framework presented in section 3.3. The application and service CCes are instantiated on the network nodes by the framework instances, based on information that accompanies the CCes e.g. the type of the node that is designated to be run, configuration before their execution and more.

4.1 Polling Application

As described in section 3.1.1.1, the polling application enables the creation of a polling service in a neighborhood network consisted of an arbitrary number of mobile devices that might be smartphones, tablets or laptops. Figure 4.1 presents the system decomposition using a component UML diagram. In this scenario, there are three different roles for the mobile devices: (i) the PollCreator, which represents the service creator, since it publishes the service's CCes in binary files in the network to be installed on the network nodes, and allows the user to create and publish a new poll, (ii) the PollManager, which is responsible for storing and publishing the newly created Poll and processing the users' poll answers to generate and publish the poll results, and (iii) the PollParticipant, which is enabled to offer a UI to the user to respond to the polls and display the poll results. On each node, the framework is running and provides the required pub/sub interfaces to enable the communication among the system's CCes. The CCes of this application publish and subscribe to event messages under the service tag *polls*.

PollCreator comprises two CCes, the Poll Creator Widget (PCW) and the New Poll Service (NPS). PCW provides a UI to the users to allow them to create a poll and publish it as well as it maintains in its resources storage all the binaries to create the polling service in the network and publishes them as soon as it is started. As mentioned in Chapter 3, the widget components have only dependency to the local bus, thus, the PCW publishes the binaries and the new polls as local events to the local bus. The NPS, since it is a service component, has dependencies to both local and remote buses. It has subscribed to the local events NewPollLocal for receiving a new poll and InstantiateCCLocal for receiving a CC binary and metadata which includes information about the configuration and installation of the CC. As soon as it receives the local events, it stores the NewPollLocal in its local database and prepares the remote events, NewPollRemote and InstantiateCCRemote, including the user content and CCes binaries respectively and publishes them to the network. The PollManager node is configured to act as an infrastructure node, which means it is able to store big amount of data and has stronger computing capabilities, and receives the InstantiateCCRemote event that includes the binary for the Poll Management Service CC (PMS), which is designated to be run on infrastructure nodes. As soon as it receives it, the framework instantiates the component on the node. In addition, it receives two more InstantiateCCRemote events, each of those includes a widget component, the Published Polls Widget (PPW) and the Poll Results Widget (PRW), able to run on the specific PollManager device. The PMS is responsible for maintaining the published polls and calculate the poll results. As soon as, it receives a NewPollRemote event, it publishes to the network the PollPublishedRemote event to notify the interested CCes on the network about the published poll.

The PollParticipant node is responsible for providing a user application for responding to the published poll. Initially, no CCes is running on the node, only the framework. As soon as the InstantiateCCRemote events that includes the participant application binaries

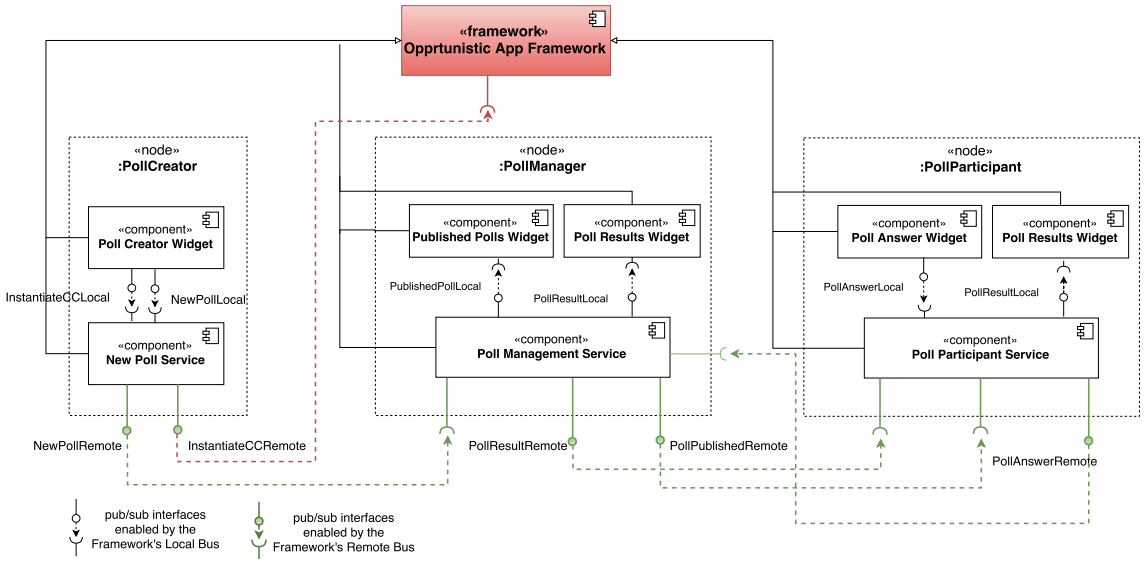


Figure 4.1: UML component diagram of the polling application

published by the PollCreator node, arrives at the PollParticipant, they get instantiated. Those binaries include the Poll Participant Service (PPS), the Poll Answer Widget (PAW) and the Poll Results Widget (PRW). The PPS has subscribed to the PollPublishedRemote event in order to receive the new poll and publish it locally to the PAW which is responsible to provide the poll form to the user. As soon as the user responds to the poll, the PAW creates the PollAnswerLocal event and publishes it to the local bus. The PPS which has subscribed to the PollAnswerLocal receives the user answer, stores it and prepares the PollAnswerRemote to publish it to the remote bus.

The PMS component running on PollManager node, has subscribed to the PollAnswerRemote event and thus, it receives the user answer and updates the poll results based on the new value. Subsequently, it publishes the PollResultRemote and PollResultLocal events so that the interested local and remote PRW CCes receive the new poll result. According to the scenario in section 3.1.1.1 on PollManager, the PPW and PRW display the published polls and the poll results respectively on the laptop of the speaker which is connected to the projector so that everyone can view the content. Additionally, the users' mobile devices which act as PollParticipant nodes get the content on their screens.

4.2 Course Rating System

In this section, we present the design of the scenario described in Section 3.1.1.2 using the proposed architecture. In this scenario, the system comprises a number of mobile nodes and one static node, called throwbox. There are four types of nodes in this scenario: (i)

the QuestionnaireDistributor, which is responsible for publishing the questionnaire to be filled in and the system's binaries, (ii) ReviewsInfopoint, which is the throwbox and is responsible for maintaining all the reviews for a long period of time and calculating the final course ratings and publishing the information at the end of each semester. (iii) The Respondent, which is responsible for providing the questionnaire form to the user and publishing the user input to the network, and (iv) the FinalReviewsReceiver, which is in charge of receiving, storing and displaying the final reviews and ratings information. The CCes of this system publish and subscribe to event messages under the service tag *reviews*.

The QuestionnaireDistributor comprises two CCes, the Questionnaire Selector Widget (QSW) and the New Questionnaire Service (NQS). The QSW displays a list of questionnaires associated with the university courses and allows user to select a questionnaire. As soon as the selection is done, the QSW publishes it within the NewQuestionnaireLocal event message to the local bus. In addition, the QSW is in charge of retrieving the service's binaries from its resources and publishing them to the local bus using an InstantiateCCLocal event message. The NQS has subscribed to both NewQuestionnaireLocal and InstantiateCCLocal, thus, it gets notified by the local bus and publishes the user content and the binaries to the network nodes using the NewQuestionnaireRemote and InstantiateCCRemote respectively.

The ReviewsInfopoint is an infrastructure node, thus it handles the InstantiateCCRemote messages that include binaries designated to this kind of nodes. As soon as, it receives these messages, the framework instantiates the Reviews Monitor Widget (RMW) and the Reviews Service (RS). The Respondent node, is a mobile node, thus it handles the InstantiateCCRemote messages that include binaries running on this kind of nodes. As soon as, it receives these events, it extracts the binaries and instantiates the Questionnaire Widget (QW) and the Respondent Service (RS). As soon as the RS is instantiated, it can handle messages published to the remote bus and since it has subscribed to NewQuestRemote, it receives and stores the questionnaire which then publishes to the local bus, so that the QW receives it and displays it to the user. When the user fills in the questionnaire and submits it, validation is being triggered and if it is successful the QW creates the FilledQuestLocal event which includes the questionnaire answers and publishes it to the local bus. The RS which has subscribed to this events, receives the questionnaire and publishes it to the remote bus by including it into the FilledQuestRemote.

The RS on the ReviewsInfopoint node, receives the FilledQuestRemote and extracts the user's responses and updates the courses ratings and stores the specific review. At the end of the semester, the RS calculates the final ratings and prepares the FinalReviewsResultLocal and FinalReviewsResultRemote events which include the final ratings and all the reviews for each course and publishes them to the local and remote bus respectively. The Review Monitor Widget (RMW) on the ReviewsInfopoint node and Final Reviews Widget (FRW) on the FinalReviewsReceiver node receive the events and display the information. The FRW receive the information through the local bus on the

FinalReviewsReceiver node which has been published within FinalReviewsResultLocal event message by the Reviews Receiver Service (RRS).

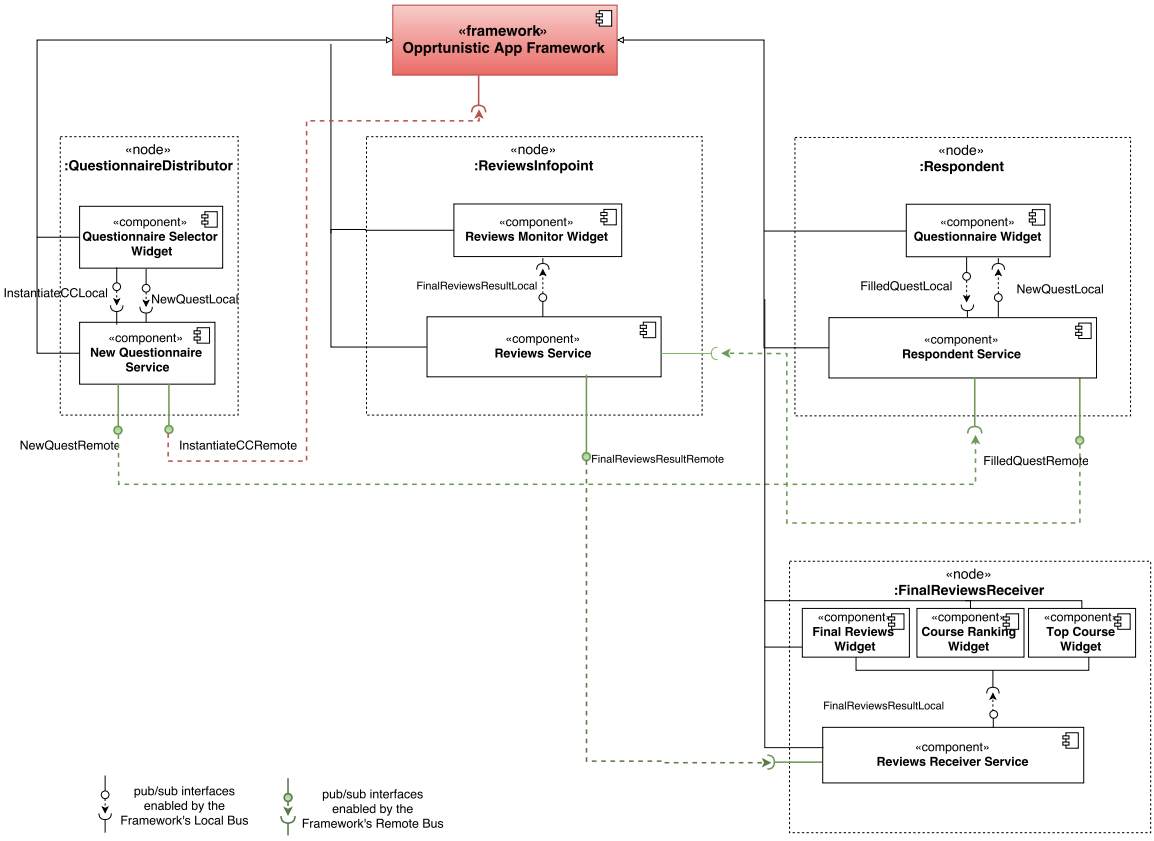


Figure 4.2: UML component diagram of the course rating system

The Ranking Widget (RW) and Top Course Widget (TCW) display a subset of the information published within the FinalReviewsResultLocal, i.e., only the courses rating ordered by ranking and the top course as home screen card. The latter shows one of the benefits of the local bus, which is the ability to efficiently design and develop applications which enable user to choose which view they would like to have running on their devices depending on the amount of the content and their individual preferences. This can be done without additional development effort during the entire life of the system since all widgets are completely decoupled and independently deployable. They can be selected by the user, replaced with a newer version and still listen to the intended content.

4.3 Event Registration and Check-in Application

Section 3.1.1.3 presented the last university scenario, the design of which demonstrates the applicability of the discussed architecture as well. In this scenario, there are four types

of nodes: (i) the MobileEventDistributor, which is responsible for distributing the event poster to other in-network nodes as well as the system's binaries to be installed on the nodes. It is a node characterized by high mobility since it changes from one neighborhood network to the other to offload the binaries and the event poster. (ii) the EventInfopoint, which is an infrastructure node and is in charge of maintaining and publishing the system's binaries by installing the event poster distributor service CC to publish the event poster CCes to the newly connected nodes and creating the event guest list associated with the particular neighborhood. (iii) The Attendant, which is a mobile node and responsible for displaying the event poster to the user and store and publish their response to the event invitation, and (iv) the GuestListHolder, is a mobile node which is responsible for merging all neighborhood guest lists and provide a finalized list that can be used to check in the attendants. The CCes of this application publish and subscribe to event messages under the service tag *events*.

The MobileEventDistributor offers a UI to the user to allow them to pick the event poster out of a list of university events and publish them to the network. It comprises two CCes, the Event Poster Widget (EPW), which provides the event selector UI and the Event Poster Distributor Service (EPDS) which publishes the event poster application to the network. As soon as the user selects the event poster, the EPW creates a NewInvitationLocal event which includes the particular event poster binaries that provide all the functionality to display the event information, accept the invitation and publish the response to the interested subscribers. The EPDS receives the local event and prepares and publishes a NewInvitationRemote event with the binaries and event information.

In this design, we decided to bind the event content with the application binaries, designated to be run on the Attendant node, in one event message so that we reduce the steps of the instantiation of the services. More specifically, in polling application scenario, the PollCreator publishes initially the binaries and then the poll. In the current case, those two steps are combined to one and it seems more reasonable here since we want to start a service for one particular event, while in the context of a polling application, it may happen that the user wants to start multiple polls for a specific topic.

For the remaining CCes need to be run on the network nodes, i.e., Guest List Service (GLS) and EPDS on the EventInfopoint node, the MobileEventDistributor publishes InstantiateCCRemote events which include the binaries and metadata for the description and configuration of the CCes. The EPDS that is being instantiated on the EventInfopoint node is in charge of publishing the particular event poster to the connected to the network nodes and is used instead of the EPDS running on the EventDistributor node, since the latter moves from one neighborhood to the other and may never return to the same neighborhood, thus the EventInfopoint takes charge of the distribution.

The Attendant node receives and instantiates the Event Invitation Widget (EIW) and Event Invitation Service (EIS) CCes extracted by the NewInvitationRemote event. As soon as the user of Attendant device accepts the invitation using the EIW, it publishes the response to the local bus using the InvitationRespLocal and the EIS stores the

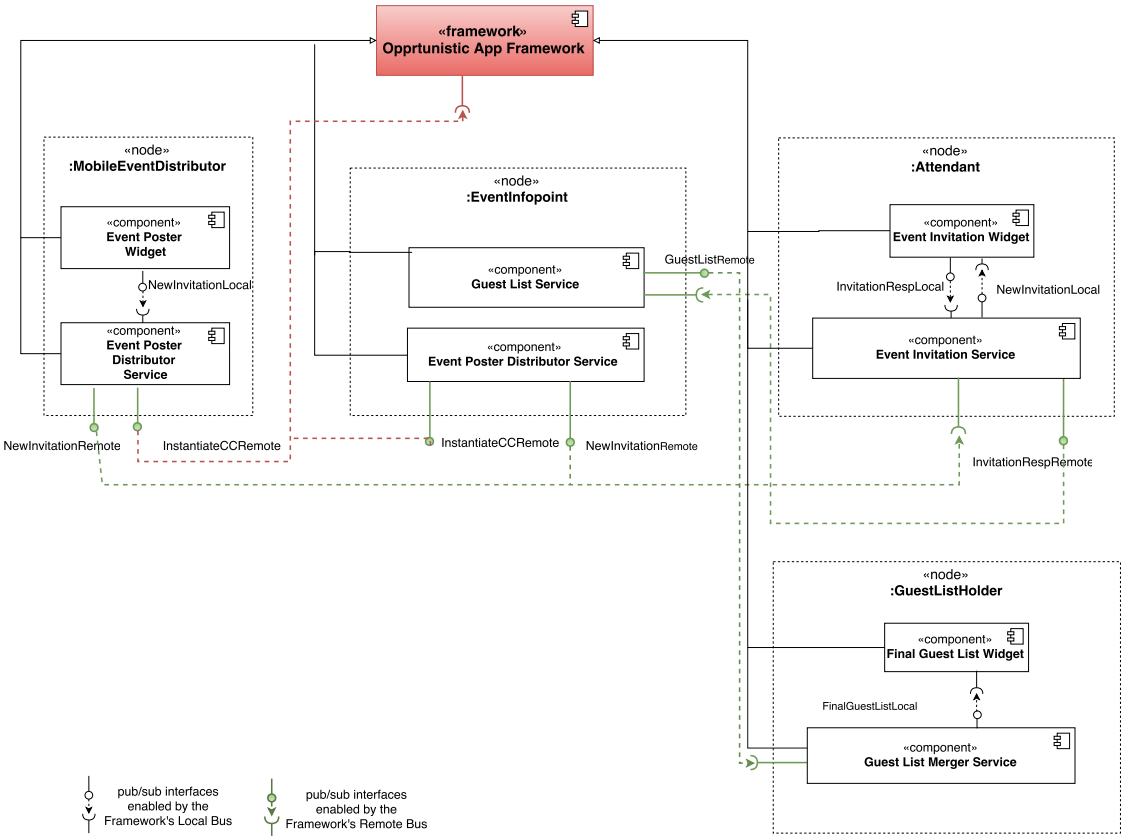


Figure 4.3: UML component diagram of the event registration and check-in application

response and publishes the InvitationRespRemote event to remote bus. The Guest List Service (GLS), having being instantiated on the EventInfopoint and has subscribed to the InvitationRespRemote, receives the Attendant invitation response and updates the guest list associated with the particular event and publishes the list to the remote bus within the GuestListRemote event message. The GuestListHolder node moves to the neighborhood networks and picks up the guest lists. The Guest List Merger Service (GLMS) has subscribed to the GuestListRemote event and thus, it queries the GLS running on the EventInfopoint by publishing a remote event GuestListQueryRemote with the identifier for the specific event and it receives the GuestListRemote. Subsequently, it merges the received list with the locally stored one and thus, it updates the final guest list. It prepares the FinalGuestListLocal event and publishes it to the local bus. The Final Guest List Widget (FGLW) receives the local event, extracts the list and renders it. Additionally, the FGLW offers the capability of user check off the guests from the list and thus check them in the event as soon as the attendants arrive at the event location.

4.4 Summary

This chapter provided the first part of the evaluation of this work. It presented a set of designs structured by applying the architecture discussed in Chapter 3 with the purpose of demonstrating the feasibility of a system being created dynamically out of autonomous components running multiple nodes that communicate leveraging opportunistic, peer-wise contacts. Chapter 5 presents the second part of the evaluation, which is the implementation of the framework and the polling application introduced in chapters 3 and 4 respectively and lastly, chapter 6 demonstrates the viability of the system presenting the results of the experiments run on real devices using the implementations of chapter 5.

Chapter 5

Implementation

In Chapter 3, we presented the major contribution of this work which is an architecture that enables us to build opportunistic applications and services out of self-contained components utilizing an arbitrary number of consumer mobile devices. Subsequently, in Chapter 4, we provided a set of designs based on this architecture to evaluate the feasibility of this sort of composable systems. However, undoubtedly, it is necessary to prove this concept by providing a real implementation and show how this architecture works in practice. Therefore, we proceeded with the generation of a proof-of-concept by implementing two versions of the framework, one in pure Java and one for the Android platform as well as the CCes of the polling application scenario, the decomposition of which is presented in section 4.1. In Chapter 6, we deploy those implementations on real devices and run a set of experiments to study the functionalities of the framework and the interactions of different CCes in detail.

5.1 Framework

In this section, we describe in detail the implementation of the discussed framework. Initially, we provide the description of a framework library with abstract classes and interfaces, which is used to implement concretely the local bus, remote bus, component registry, component controller and the composable components subsystems, used for both Android and pure Java framework versions. The design of the framework is depicted in Figure 3.4 and the implementation presented in this section tends to map this design. Afterwards, we proceed with the individual implementations for each version and in the following section, we move with the description of each CC of the polling application. While we are describing the implementations, we provide information about the external libraries used as well as the design patterns that have been applied to create a clean and modular codebase.

5.1.1 Common Library

The implementation of a common library is necessary to provide the generic structure of the framework that is used to build the Android-specific and pure Java-based concrete implementations. The library is written in Java and consists of the local bus interface and its event messages, a generic composable component that has dependencies to both local bus and remote bus, the generic component storage and concrete implementation which is used in both framework versions, the generic service component model which corresponds to the domain model in the Figure 3.5, the component registry implementation and the framework-level logging subsystem. The Common library is provided as a JAR file.

5.1.1.1 Service Composable Component and Remote Bus

Figure 5.1 shows the class diagram of the AbstractComponent which maps to the generic implementation of the service composable component, the ComponentRole enum class that is used to define the role of the component, and the MessageField class which includes a set of constants used as keys for the metadata fields to the instantiation event messages (`ConfigMessage`). The important attributes of the AbstractComponent are the following:

- *componentRole*: the role of the AbstractComponent is defined using the ComponentRole enum class shown in the figure. A component can be widget or service and can be designated to run either on an infrastructure or mobile node.
- *serviceTag*: defines the service identifier.
- *events*: is a collection of events that the AbstractComponent needs to subscribe to.
- *ComponentStorage*: is the storage implementation of the service composable component and is presented in section 5.1.1.4.
- *ILocalBus*: is the interface of the local bus implementation provided by the framework common library and needed to implemented to provide a concrete local bus implementation. It is presented in section 5.1.1.2.
- *AppLib*: is provided by the Scampi middleware and used to enable the remote bus discussed in the previous chapters.
- *IEventLogger*: is part of the logging mechanism provided by the framework and used in AbstractComponent to log the remote events published and received.

The Remote Bus implementation corresponds to the Scampi middleware which provides services to publish and receive messages through the Application Library (AppLib). AppLib exposes an API, which is used by the AbstractComponent, to communicate with the Scampi middleware instance running on the device. In addition, the AbstractComponent needs to get notified about the status of the connection to the Scampi instance, to do so, we attach an AppLibLifecycleListener to the

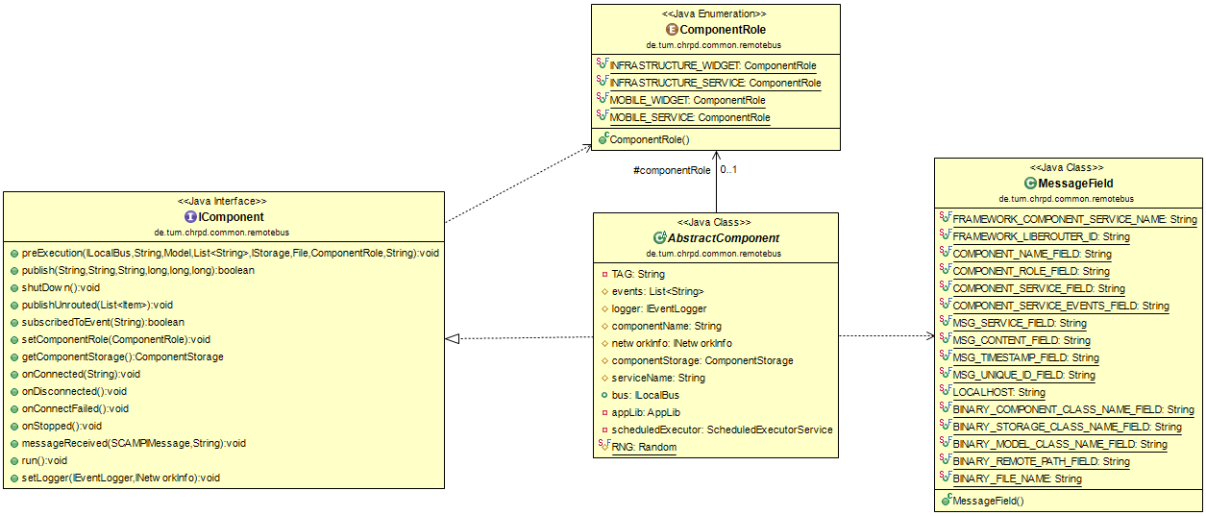


Figure 5.1: UML class diagram of the abstract component in common library

AppLib. The interface defines methods invoked during transitions of the AppLib lifecycle: `onConnected`, `onDisconnected`, `onConnectFailed` and `onStopped`. In case the connection has been established successfully, the ComponentStorage instance of the AbstractComponent asynchronously retrieves the unrouted messages in order to be published. To receive messages, the AbstractComponent can `subscribe()` to a service tag and a set of service events, if required. In order to do that, it must implement the MessageReceivedCallback to be executed when AppLib receives a new message from the local middleware instance, and subscribe to a service tag so that the middleware instance starts delivering messages to the AbstractComponent with the callback function `messageReceived(SCAMPIMessage scampiMessage, String service)`. In order to publish a message the AbstractComponent initially constructs a message of type SCAMPIMessage provided by the AppLib and uses the `publish()` method to post it to the local middleware. The necessary parameters to publish a remote message are: (i) serviceTag, to define the service/application, (ii) content, the message content, (iii) timestamp, that defines the timestamp the message is generated, (iv) dbid, the event message database identifier that is supposed to be globally unique in the scope of the service. The MessageField class is used to set the keys for those values and extract them when a SCAMPIMessage is received.

The AbstractComponent provides the method `preExecution()` in order to offer the ability to configure the component before its instantiation by the Component Controller (see 5.1.2 and 5.1.3). In practice, when a CC is sent over the network in a binary file, the message that includes it contains additional information on how to configure it. This information is being exported using the constants of the MessageField class, i.e., `BINARY_COMPONENT_CLASS_NAME_FIELD`, `BINARY_STORAGE_CLASS_NAME_FIELD`, `BINARY_MODEL_CLASS_NAME_FIELD`, `COMPONENT_ROLE_FIELD`, `COMPONENT_SERVICE_FIELD`, and `COMPONENT_SERVICE_EVENTS_FIELD`. As soon as the Component Controller has

extracted those field values, it invokes the `preExecution()` of the AbstractComponent and passes them as parameters. Then it instantiates it. In addition, the IComponent interface is used for the class loading in the instantiation process. Regarding the latter, we provide more details later in this chapter.

5.1.1.2 Local Bus

The local bus, as described in section 3.3.5, represents an abstraction layer between the service and widget composable components. Therefore, it exposes an interface to subscribe, unsubscribe and publish a message to the bus to enable the communication models Local RPC and Local PubSub (see section 3.2.3) among the components running locally. The interface is shown in Figure 5.2. Apart from the Java version, we also support an Android version for the framework, thus this interface provides various methods to support both. In order to accelerate the development of the framework, we use the Cooking Fox EventBus Adapter [Cook] that wraps various EventBus implementations for Java and Android using a uniform interface (com.cookingfox.eventbus.EventBus). The encapsulated implementations are Google Gueva EventBus [Goog] for Java and GreenRobot EventBus [Gree] for Android. The `post()` method is invoked in the Java version, while the `publishXprocess()` is invoked in the Android version which is result of the combination of the Android BroadcastReceiver to support inter-process communication on Android devices and the GreenRobot EventBus responsible for publishing events within an Android application. The BusMessage and ConfigMessage classes represent the messages posted

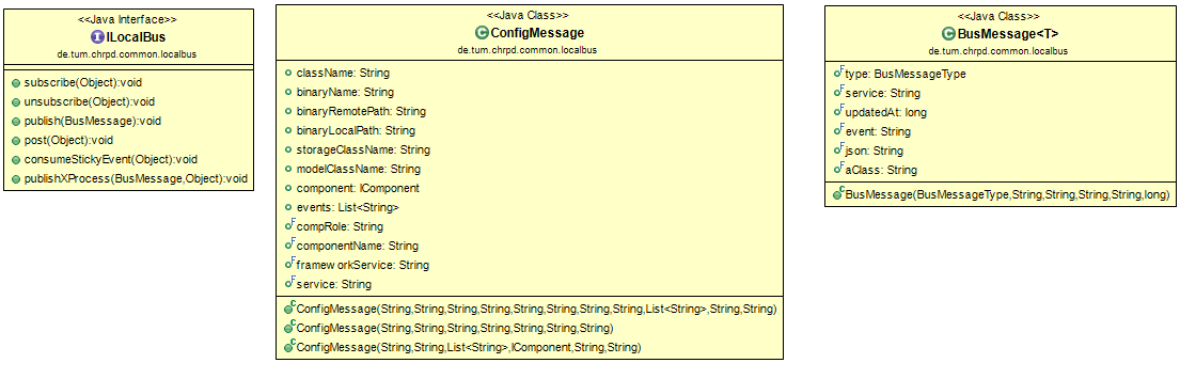


Figure 5.2: UML class diagram of the local bus in common library

to the local bus. The first one is used for publishing an event message and the second for publishing the instantiation event messages, called ConfigMessage. The ConfigMessage supports three different constructors to support different requirements; configuration for remote mobile or infrastructure service, remote mobile widget, local instantiation (not publishing to remote bus). We discuss this in detail later in this chapter.

5.1.1.3 Component Registry

The Component Registry subsystem is responsible for keeping track of the running CCes within the node and provide this information to Component Controller subsystem or other systems, which might be running on different nodes and wish to gather information on the locally running services. To realize those functionalities, the Component Registry is designed in this work as an independent module that uses the Local Bus and the Remote Bus respectively, as described in 3.3.6. However, the focus of the implementation is to achieve the decoupling of the widget and service components utilizing the buses abstractions, thus, in this chapter, we present the Component Registry coupled with the Component Controller, having a reference of the IComponentRegistry in the ComponentController class. We provide more details on this in the sections 5.1.2 and 5.1.3. For the implementation of the databse used in the Component Registry module, we used MapDB Data Engine [MapD], which provides Java Maps, Sets, Queues and other collections backed by disk storage or off-heap-memory. Figure 5.3 presents the structure of the component_registry package. The IDisk interface facilitates the creation of the necessary registry directories on the node's disk and needs to be injected using the `init()` method which is the initialization method of the ComponentRegistry that implements the IComponentRegistry interface. There are two different implementations of the IDisk in order to support both the pure Java and Android-based frameworks. Each implementation is included in the corresponding framework version.

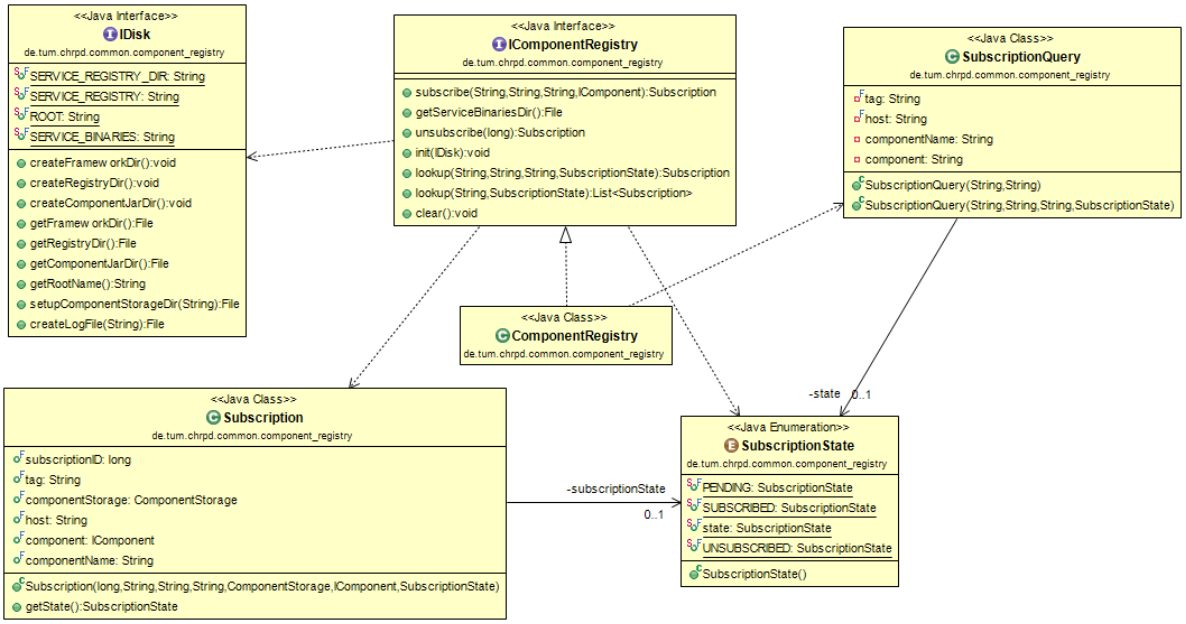


Figure 5.3: UML class diagram of the component registry in common library

The ComponentRegistry supports on-disk storage and indexing and the queries are realized using a SubscriptionQuery class instance. The storage organization is as follows:

- `BTreeMap<Long, Subscription> primary`: is the main collection declared as a BTreeMap, which stores a Subscription object under long identifier.
- `NavigableSet<Object[]> tagIndex`: defines a secondary index on the primary collection based on the serviceTag attribute of the Subscription class.
- `NavigableSet<Object[]> hostIndex`: defines a secondary index on the primary collection based on the host attribute of the Subscription class.
- `NavigableSet<Object[]> stateIndex`: defines a secondary index on the primary collection based on the state attribute of the Subscription class.
- `NavigableSet<Object[]> hostTagIndex`: defines a secondary index on the primary collection based on the host and serviceTag attributes of the Subscription class.
- `NavigableSet<Object[]> hostTagComponentStateIndex`: defines a secondary index on the primary collection based on the host, serviceTag, state and component attributes of the Subscription class.

The important methods of IComponentRegistry are the following:

- `lookup(String host, String tag, String componentName, SubscriptionState state)`: when a CC is sent using the remote event ConfigMessage, the Component Controller must decide whether this CC should be installed on the node or not. Thus, it invokes the lookup method of the IComponentRegistry and gets a Subscription object as response which includes the info of the CC stored in the database, otherwise it returns `null`.
- `subscribe(String host, String serviceTag, String componentName, IComponent component)`: in case the lookup() returns null, the ComponentController invokes the subscribe() method to register the CC by providing information about the service tag, local host, the CC name and the IComponent.
- `unsubscribe(long id)`: it is used to unsubscribe a CC by passing the identifier of the CC Subscription.
- `lookup(String host, SubscriptionState state)`: it is used to lookup all CCes on a specific node using the SubscriptionState: PENDING, SUBSCRIBED, UNSUBSCRIBED.

5.1.1.4 Component Storage

The Component Storage subsystem is used by stateful service CCes to persist their domain data. In order to build a cross-platform storage module, we used the MapDB Java engine. The structure of this subsystem is illustrated by Figure 5.4 and corresponds

to the Strategy Design Pattern [Stra], which assists in interchanging between different database implementations that might be developed for specific purposes for particular service CCes. The IStorage interface represents the template code in the owner class of the ComponentStorage. The Component Storage subsystem provides the storage implementation represented by the ItemStorageImpl, however a different implementation that realizes the IStorage interface can be used and injected using the `setStorage()` of the ComponentStorage class.

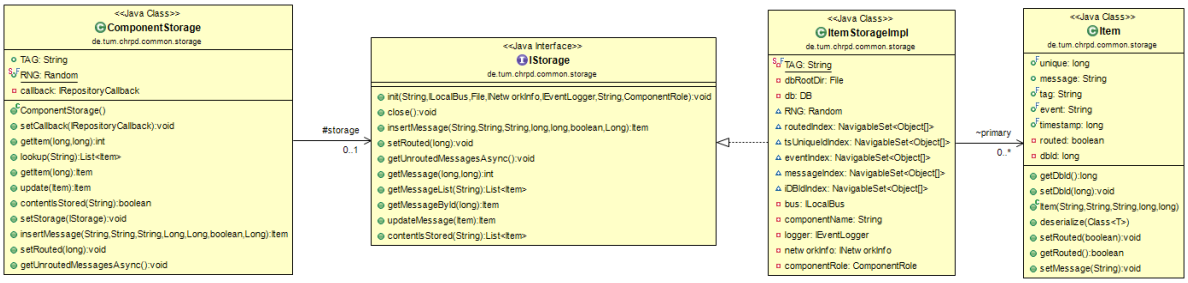


Figure 5.4: UML class diagram of the component storage in common library

ItemStorageImpl class, as mentioned previously, uses the MapDB data engine to realize a database for storing Item objects. Item class represents the wrapper class of the message content published and received using the Remote Bus. Its attributes are: (i) unique, which is a unique identifier of the message (ii) message, which is the content of the event message in JSON format (iii) tag, that defines the service tag (iv) event, the event identifier that is unique within the scope of the composable application/service, (v) routed, a flag that shows whether the event has been published by the Scampi instance, (vi) timestamp, the event creation timestamp which along with the unique field can ensure the unique occurrence of the event message in the database and (vii) dbid, the database identifier which is globally unique in the scope of the composable application/service. The same dbid is used in every node for the specific event message in the particular application. The Item class contains the method `deserialize()`, which uses the GsonUtil helper class to convert a JSON string to a Java object. That allows the run-time loading of the Java objects needed in the Component Model classes used by the concrete service CC implementations.

The important methods of IStorage interface are the following:

- `insertMessage(String tag, String event, String message, long timestamp, long uniqueid, boolean routed, Long dbId)`: is used to insert a message into the database along with all the necessary fields described above. It returns the stored Item associated with a database id which is created inside the method only in case the dbid parameter is `null`. As soon as an item is stored in the database, the `messageReceived()` method of the IStorageItemInsertionListener is being invoked in order to notify the observer classes about the newly stored item. Those classes inherit from the Model class (see 5.1.1.5) that implements the IStorageItemInsertionListener. The listener is set in the `preExecute()` method

of the IComponent, which is responsible for configuring the component and its associated classes before it gets instantiated by the ComponentController.

- `updateMessage(Item i)`: it allows the updating of a specific item stored in the database by passing the modified item along with the dbid.
- `getMessage(long timestamp, long uniqueid)`: getter that queries the database using an ItemQuery object given the timestamp and uniqueid fields of the Item.
- `getMessageById(long dbid)`: getter that queries the database using an ItemQuery object given the dbid field of the Item.
- `getUnroutedMessagesAsync()`: getter to asynchronously retrieve the unrouted to the Scampi instance messages. The class that has subscribed to the local event *ItemUnroutedRetrieved* receives the even through the local bus.

5.1.1.5 Component Model

The Model class is inherited by the domain model classes implemented for the service CCes. It provides all the required dependencies to publish model updates to the local widget CCes through the Local Bus and update the model locally using the ComponentStorage instance. Figure 5.5 presents the UML class diagram of the Model class which contains the setters for the ComponentStorage and ILocalBus objects and the `onEvent(event: UnregisterEvent)` event subscription method that receives the UnregisterEvent local event, in case the client of the Model class requests to unregister the Model from the local bus. In addition, Model implements the `IStorageItemInsertionListener` which allows its child classes to get notified about new Item insertions to the Component Storage. Logging is supported for the Component Model.

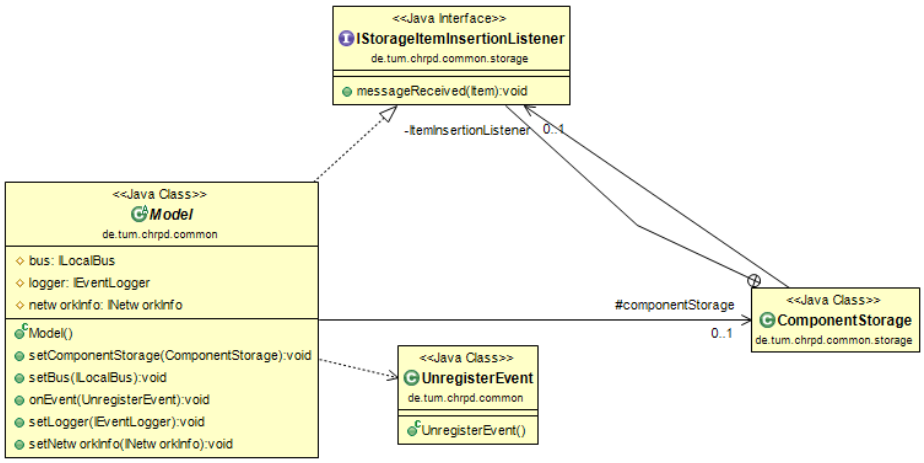


Figure 5.5: UML class diagram of the component model in common library

5.1.2 Java-based Implementation

In this section, we present the implementation of the Java version of the framework presented in Figure 3.4. The main classes of this implementation are the ComponentController and the LocalBus. The DiskImpl and EventLogger classes are presented to describe the creation of the framework directories and the logging mechanism respectively. The remaining classes and interfaces are included in the Common library described above. Figure 5.6 shows the UML class diagram of the Java-based framework including the dependencies to the Common library. The ComponentController class

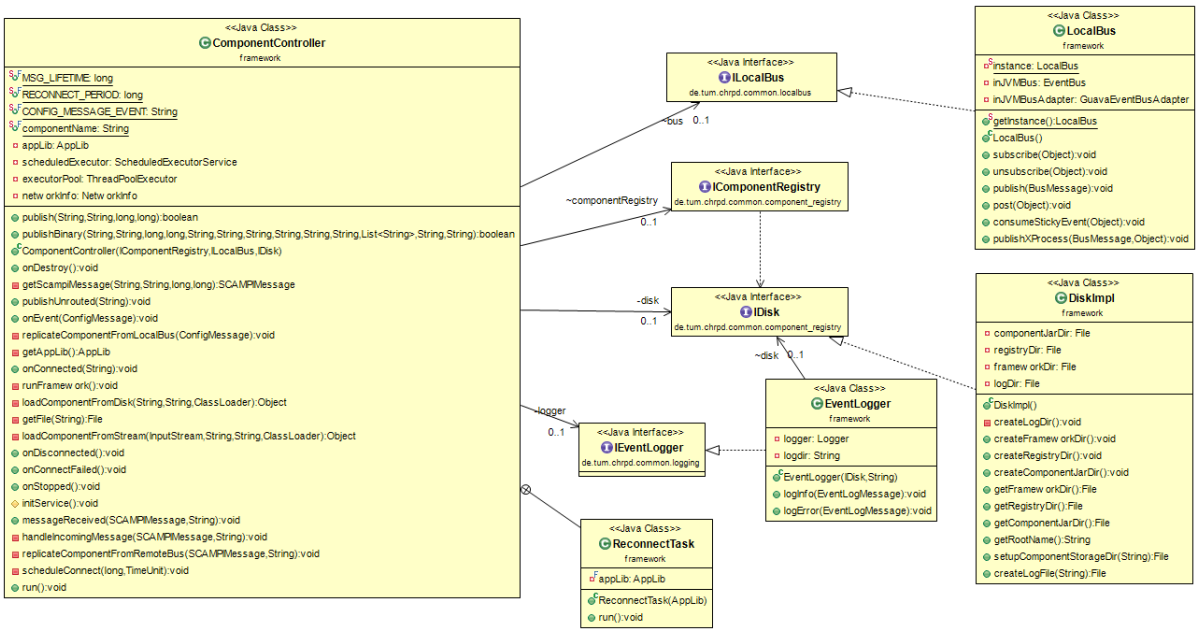


Figure 5.6: UML class diagram of the Java-based framework

implements the following interfaces: (i) `Runnable`, for allowing the class's instances to be run by a thread, (ii) `MessageReceivedCallback`, as mentioned in Section 5.1.1.1, provides the callback functions to receive a message from the local Scampi instance, which is used to publish and receive event messages of type ConfigMessage posted to the service tag *InstantiateCCRemote*, this can be realized by invoking the method `subscribe(InstantiateCCRemote: String)` of the AppLib in order to subscribe to the particular service, and (iii) `AppLibLifecycleListener`, which provides the methods to monitor the state of the connection to the local Scampi instance. The ReconnectTask inner class is being executed by a thread as soon as the ComponentController is being instantiated in order to attempt to connect to the local Scampi instance at a fixed interval until the connection is established. In order to instantiate the ComponentController, the client class must pass as parameters to the constructor objects of IDisk, ILocalBus and IComponentRegistry. As mentioned earlier, in Chapter 3, we present the Component Registry subsystem as a self-contained, independent module that communicates with

the framework elements through the Local Bus applying the publish-subscribe models described in section 3.2.3. However, in the implementation we focus on how to achieve the decoupling of the CCes, thus, the ComponentRegistry is part of the ComponentController implementation and it is being referenced in the ComponentController class.

The ComponentController owns the attribute executorPool of type `ThreadPoolExecutor`, with which it can instantiate the receiving CCes to the service tag *InstantiateCCRemote*. More specifically, as soon as the remote event message of type ConfigMessage (see section 5.2) is received, the ComponentController invokes the `lookup()` method of the ComponentRegistry in order to check whether the received component is already running. If not and in case the contained component is of type service, i.e. `componentRole = SERVICE_INFRASTRUCTURE`, the ComponentController `replicateComponentFromRemoteBus()` locally by loading the class that implements the IComponent interface and the complementary classes that are needed to instantiate the component from the JAR file contained in the event message. The additional classes are: (i) a class that inherits from the Model (see section 5.1.1.5) and (ii) a class the implements the IStorage interface (see section 5.1.1.4). The class loading is provided by the `loadComponentFromStream()` method. In addition, in the SCAMPIMessage that contains the remote ConfigMessage event, further necessary information is included, i.e., a list of strings that represent a collection of remote events that the component needs to subscribe to, and which actually represents the dependencies to the remote CCes, the component role, the component name and the IStorage, Model and IComponent classes names. Subsequently, it invokes the `preExecute()` method of the IComponent and configures it based on this configuration and thus, it injects the instances of the classes, the event list, the component name and role as well as the path to the component disk partition, where it can persist its own data and files. After the instantiation of the `Runnable` IComponent, it is submitted to the thread pool held by the ComponentController instance and at last a component subscription to the Component Registry subsystem is realized by the `subscribe()` method of the ComponentRegistry being invoked.

Using an instance of the class LocalBus, which implements the ILocalBus interface, the ComponentController can subscribe to the local event message of type ConfigMessage. This is realized by the callback method `onEvent(configMessage: ConfigMessage)` of the `com.cookingfox.eventbus.adapter.GuavaEventBusAdapter`, the occurrence of which in the code defines the subscription to the event. As soon as a local ConfigMessage event is being received, the ComponentController similarly to the reception of a remote ConfigMessage event, checks whether the component is registered in the Component Registry subsystem and if not, it instantiates it by invoking the method `replicateComponentFromLocalBus`. The latter occurs only in case `componentRole = SERVICE_INFRASTRUCTURE`. Subsequently, it prepares a remote ConfigMessage event and publishes it to the remote bus using the `publishBinary()` method that invokes the `publish()` method of the AppLib.

Logging is supported in the ComponentController by using the `logInfo()` and `logError`

methods of the IEventLogger. The EventLogger class implementation is slightly different from the Android one since an IDisk instance is required and that needs to be implemented differently to support both versions.

5.1.3 Android-based Implementation

The architecture of the Android-based framework is similar to the Java version, as illustrated by the UML class diagram in Figure 5.6. Thus, in this section, we focus on the main differences between the two versions. The Android-based framework is developed as an Android Service running in a separate process on Android-enabled devices that starts at boot time. It comes as a deployment variant in APK (Android Application Package), framework.apk, and a development variant framework.jar which provides an Android Framework Library (FrameworkLib) to develop android opportunistic applications. The FrameworkLib uses the Common Library, described earlier, to utilize the Remote Bus, Component Registry and Abstract Composable Component subsystems. Regarding the utilized external libraries, AndroidAnnotations open source framework [Andr] is used to realize the Dependency Injection pattern in order to eliminate the boilerplate code and speed development and GreenRobot EventBus for the implementation of the local bus.

The major implementation differences between the two framework versions are related to the Component Controller and the Local Bus implementations which for Android are provided by the FrameworkLib. The ComponentController class inherits from the Android Service in order to provide a long-running thread running in the background with own lifecycle decoupled from the Android Activities. Simiralry to the Java-based ComponentController class, it implements the `MessageReceivedCallback, AppLibLifecycleListener` interfaces to receive remote event messages from the local Scampi instance and monitor the connection to it and holds a thread pool which uses to instantiate the received CCes. The Android-based ComponentController handles the instantiaton of service CCes that come as JAR files and widget CCes that come as APK. Similarly to the Java-based ComponentController, as soon as the callback method `messageReceived()` of the AppLib is invoked, a remote Config message arrives and the ComponentController checks whether the `componentRole = SERVICE_MOBILE` or `componentRole = WIDGET_MOBILE`, if so, it then invokes the `handleIncomingMessage()` which checks whether the CC is already running using the ComponentRegistry provided by the Common Library and if so, it replicates it locally based on its role. In case it is a widget, it stores the APK on the disk and then uses a `new Intent(Action.View)` for which it sets the targeted APK file and its MIME type `intent.setDataAndType(Uri.fromFile(apkFile), "application/vnd.android.package-archive")` and starts a new Android Activity using the intent. This results in showing to the user a new activity that prompts them to install the received widget. In case the type of the CC is service, the same implementation with the Java-based version is supported.

The purpose of the Local Bus subsystem is to enable communication among widget and service CCes. To achieve that, we need to provide a mechanism in the FrameworkLib that allows widgets that are autonomous Android applications residing in their own process to interact with `Runnables` executed by threads running in the same process with the ComponentController Android Service. The inter-process communication (IPC) is realized by using the GreenRobot EventBus to publish local event messages within the same process and the `XProcessSender` and `XProcessReceiver` classes to allow passing event messages among different processes on the device. The GreenRobot EventBus works similarly to the Guava EventBus described earlier and enables to publish and receive messages among all the classes that are contained in one APK. The method `publishXProcess()` of the ILocalBus interface described in 5.1.1.2 is implemented by the IPCBus class within the FrameworkLib and is responsible for sending the messages to different processes. Using AndroidAnnotations, the field injection of the `XProcessSender` and `XProcessReceiver` is done and they are used by the IPCBus to delegate the IPC tasks. When the `publishXProcess()` is invoked the execution control goes to the `publish(message: XProcessMessage)` method of `XProcessSender` class which by invoking the `context.sendStickyBroadcast(intent)` method sends broadcast intent objects, having firstly set the action string to the BroadcastReceiver project package. The BroadcastReceiver in this case is the `XProcessReceiver` class which inherits from the Android BroadcastReceiver. As soon as the callback method `onReceive()` of the `XProcessReceiver` is invoked, the deserialization of the event message is realized and using the GreenRobot EventBus, the event message is being forwarded to the interested subscribers within the same process.

At last, the FrameworkLib provides an abstract Android Activity that should be inherited by the Activities of the widget implementations in order to allow them to register to the IPCBus. Using AndroidAnnotations field injection, the IPCBus is being injected in the `BaseActivity` class.

5.2 Polling Application

In this section, we describe the implementation of the polling application the design of which is provided in section 4.1. In the following subsections, we present in detail the components that compose the polling application, i.e., the Poll Creator Widget (PCW), the Poll Creator Service (a.k.a. New Poll Service, NPS), the Poll Participant Widget (PPW), the Poll Participant Service (PPS), the Poll Management Service (PMS) and the Poll Monitor Widget (a.k.a. Poll Results Widget, PRW).

Before we proceed with the description of each CC, it is important to gain an understanding of the set of remote and local events used in this application for the interaction of the components. Figures 5.7 and 5.8 present the UML class diagrams of the classes that represent the remote and local events respectively. These objects are being

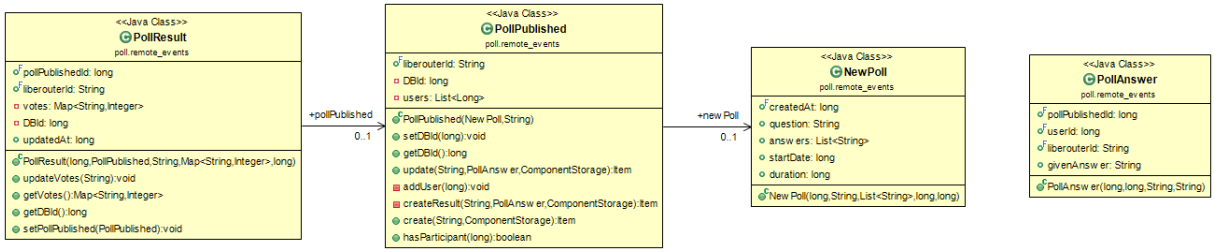


Figure 5.7: UML class diagram of the remote events used in the polling application

converted to JSON strings to be transfered and at the reception end are being converted back to Java classes. Some of those classes encapsulate business logic that is being executed by the responsible CC. We provide more details in the following subsections.

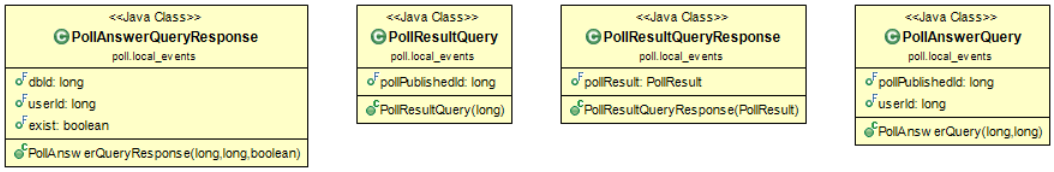


Figure 5.8: UML class diagram of the local events used in the polling application

It is important to mention that all service CCes subscribe to the local event *ItemUnroutedRetrived* in order to get notified about messages that have not been published by the Scampi instance and thus, an additional attempt is required and realized as soon as the event has been received.

5.2.1 Poll Creator Widget

The PCW is responsible for providing a UI to the user to create and publish a new poll. In addition, it is in charge of publishing the application binaries to dynamically instantiate it in the network. The PCW comes in a deployment variant pollCreatorWidget.apk and has dependency to the FrameworkLib. It consists of one Android Activity called CreatePollActivity that inherits from the FrameworkLib BaseActivity and the view resources. The CreatePollActivity acts as a view controller that handles the user input by validating the poll form and posting the new poll to the local bus. AndroidAnnotations is used to inject the views at instantiation time. As soon as the activity is started the registration to the bus is realized by invoking the `ipcBus.registerIPCBus(this)` method of the BaseActivity. When user inputs the poll question and options and they tap on the publish button, the validation of the input is being invoked. if the input is correct, the creation of NewPoll object is done with the following parameters: `long createdAt, String question, List<String> answers, long startDate, long duration`. Subsequently, the NewPoll object is converted to json string using the GsonUtil helper class and is passed as parameter to the constructor of the

BusMessage class to create a BusMessage object required to publish to the IPCBus. The additional required parameters for the creation of the BusMessage object are: the service tag, the event class name, the event identifier and the BusMessageType that should be of type `EVENT_NEW_MESSAGE`. At last, the invocation of the `ipcBus.publishXProcess()` method is done and message is published to the local bus and the form views are being reseted.

With respect the ConfigMessage events published by the CreatePollActivity to local bus in order to achieve the application instantiation in the network, the following methods are being invoked at the start time of the CreatePollActivity; (i) `initAdminPollMobileServiceComp()` which loads the NPS binary file from the resources of the widget and adds the necessary parameters required for the configuration of the CC e.g. componentName, componentClassName, componentStorageName, events list. (ii) `initInfrastructureServiceComp()` which loads the PMS binary file and adds the necessary parameters required for the configuration of the CC, (iii) `initParticipantPollServiceComp()` which loads the PPS binary file and adds the necessary parameters required for the configuration of the CC and (iv) `initParticipantPollWidgetComp()` which loads the PPW binary file and adds the necessary parameters required for its configuration. After defining the required parameters in those methods, a ConfigMessage is prepared by the `publishConfigMessage()` of BaseActivity which is then published to the local bus. In fact, the classes for all the service CCes that are needed to be published to the network are included in one JAR file called pollService.jar and is used instead of having a different binary for each component. However, this is implemented to speed up development and the intended way to achieve the instantiation is mentioned previously.

5.2.2 Poll Creator Service

The Poll Creator Service or New Poll Service (NPS) as presented in section 4.1 is responsible for receiving the local *NewPoll* events and transform them to remote events which are then published to the remote bus. The NPS consists of three classes presented in Figure 5.9: (i) NewPollModel, which inherits from the Model class of the Common Library (see section 5.1.1.5) in order to get the dependency to the Local Bus subsystem, (ii) NewPollComponent, which inherits from the AbstractComponent of Common Library to get the dependencies to the Local Bus and Remote Bus (see section 5.1.1.1), and (iii) NewPollStorage, inherits from ItemStorageImpl of Common Library and is used for database management based on the Item objects as described in section 5.1.1.4. NewPollComponent subscribes to the local event `NewPoll` using the method `onEvent(newPoll: NewPoll)` and as soon as it receives an event message it checks if the particular message has been published again and if not, it proceeds with storing it locally using a ComponentStorage object that has been configured to be of type NewPollStorage in the instantiation of the NewPollComponent by the

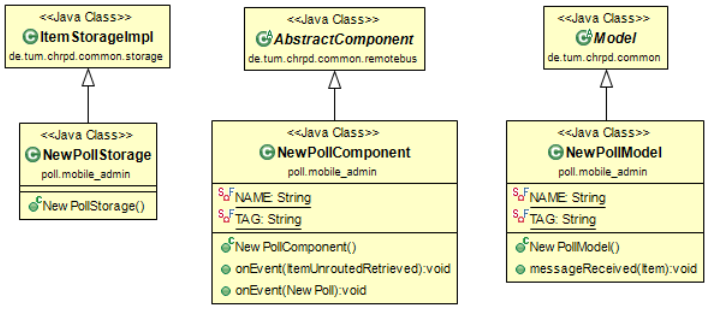


Figure 5.9: UML class diagram of New Poll Service on the PollCreator node used in the polling application

ComponentController and subsequently, it publishes it to the locally running Scampi instance. NewPollModel subscribes to the local event `NewPollEntry` which is used for confirming the storing and publishing of the poll.

5.2.3 Poll Participant Widget

The PPW corresponds to the widgets running on the PollParticipant node in the design provided in section 4.1. For the implementation, we proceeded with realizing the Poll Answer Widget (PAW) and Poll Results Widget (PRW) as different Android Activities instead of different applications that come in separate APKs to speed up development and evaluation, however the separation of those is feasible. The PAW maps to the `PublishedPollDetailActivity` which provides a detailed view of the poll and allows user to vote for the preferred answer. The PRW maps to the `PublishedPollListActivity` which provides a list of published polls and as soon as the user taps on one of the list items, they get redirected to the `PublishedPollDetailActivity` which shows the poll results. The user interface of the implemented activities are presented in section 5.2.7. AndroidAnnotations framework is used to develop a clear codebase by separating the handling of the views realized by the `PollDetailLayout` and `PollListLayout` respectively and the local event message handling realized by the activities themselves. Both activities inherit from the BaseActivity of FrameworkLib.

`PublishedPollListActivity` subscribes to the local event *PublishedPoll* using the `onEventMainThread(publishedPoll: PublishedPoll)` method and as soon as it receives a new published poll it adds it to the list. To allow user to respond to a poll, interactions between the `PublishedPollDetailActivity` and the Poll Participant Service (PPS) described in the following section are required. As soon as `PublishedPollDetailActivity` is started, a query event message *PollAnswerQuery* to the PPS is sent through the local bus by invoking the `ipcBus.publishXProcess()` method in order to check whether the user has answered the question. The PPS sends back the response event message *PollAnswerQueryResponse* and in case the user response has not been found, the form is enabled and allows user to vote. In case the user has responded

to the poll, a second query local event *PollResultQuery* is sent to the retrieve the poll results. The activity subscribes to the event message *PollResultQueryResponse* and as soon as it receives it, it displays the results. The classes of those events are presented in Figure 5.8.

5.2.4 Poll Participant Service

The Poll Participant Service (PPS) as presented in section 4.1 is responsible for receiving the remote *PublishedPoll* and *PollResult* events and transform them to local events which are then published to the local bus. In addition, it subscribes to the local event *PollAnswer* in order to receive the user's response to the poll by the PPW and publish it to the remote bus as well as to the local events *PollAnswerQuery* and *PollResultQuery* for allowing PPW to query the local storage as described in the previous section. The PPS consists of three classes presented in Figure 5.10: (i) PollParticipantModel, which inherits from the Model class of the Common Library (see section 5.1.1.5) in order to get the dependency to the Local Bus subsystem and realize the above-mentioned interactions, (ii) PollParticipantComponent, which inherits from the AbstractComponent of Common Library to get the dependencies to the Local Bus and Remote Bus (see section 5.1.1.1), and (iii) PollParticipantStorage, inherits from ItemStorageImpl of Common Library and is used for database management based on the Item objects as described in section 5.1.1.4.

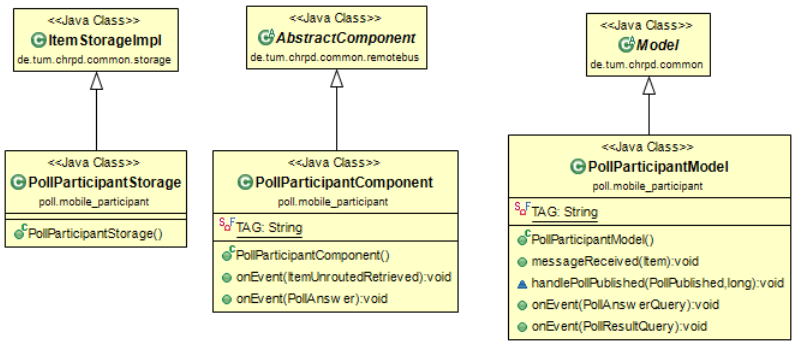


Figure 5.10: UML class diagram of Poll Participant Service on the PollParticipant node used in the polling application

5.2.5 Poll Management Service

The Poll Management Service (PMS) as presented in section 4.1 is responsible for receiving the remote *NewPoll* and *PollAnswer* events and process them in order to execute its logic for publishing a new poll and the updates on the poll results to remote and local CCes. The PMS consists of three classes presented in Figure 5.11: (i)

PollManagementModel, which inherits from the Model class of the Common Library (see section 5.1.1.5) in order to get the dependency to the Local Bus subsystem and trigger the above-mentioned interactions, (ii) PollManagementComponent, which inherits from the AbstractComponent of Common Library to get the dependencies to the Local Bus and Remote Bus (see section 5.1.1.1) and realizes the above-mentioned logic and network interactions, and (iii) PollManagementStorage, inherits from ItemStorageImpl of Common Library and is used for database management based on the Item objects as described in section 5.1.1.4. As soon as the PollManagementModel gets notified that a message

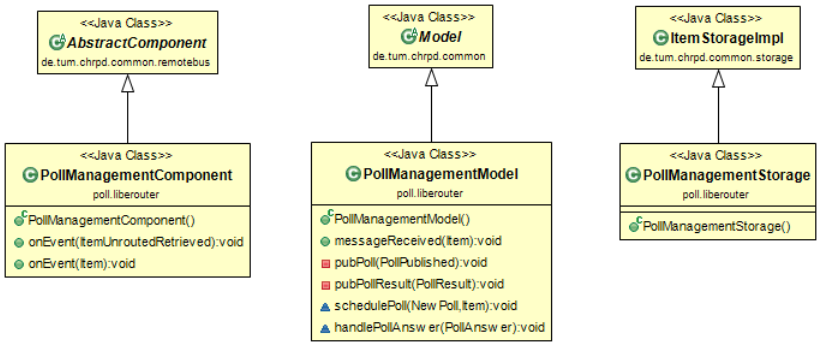


Figure 5.11: UML class diagram of Poll Management Service on the PollManager node used in the polling application

has been received by the invocation of the callback method `messageReceived(item: Item)` of the ComponentStorage.IStorageItemInsertionListener (see section 5.1.1.4), it checks the type of the event and handles it properly. In case, the item is of event type *NewPoll*, the `schedulePoll()` method is being invoked and creates a PollPublished object which contains the information included in NewPoll and converts it to Item object by invoking the `create(serviceTag, componentStorage)` method of the PollPublished class and publishes it to the local bus by invoking the `publishXProcess()`. The PollManagementComponent subscribes to the local event messages *Item* and thus it receives the local event published in the `schedulePoll()` method and publishes it to the remote bus. In case, the item delivered to the PollManagementModel is of type *PollAnswer*, the `handlePollAnswer()` is being invoked which encapsulates the logic for updating the results of the particular poll. At last, in case the item is of type *PollPublished* or *PollResult*, the PollManagementModel prepares the respective local events and publishes them to the local bus so that the interested widgets running on the node can receive the messages.

5.2.6 Poll Results Widget

The Poll Results Widget maps to the widget running on the PollManager node as described in section 4.1. It is a widget that provides a view suitable for screens of high resolutions. It is developed as a JavaFX application and the views are implemented

in HTML and CSS. The architecture of this widget implementation is presented in Figure 5.12 which shows the UML class diagram of the widget. The Observer pattern is

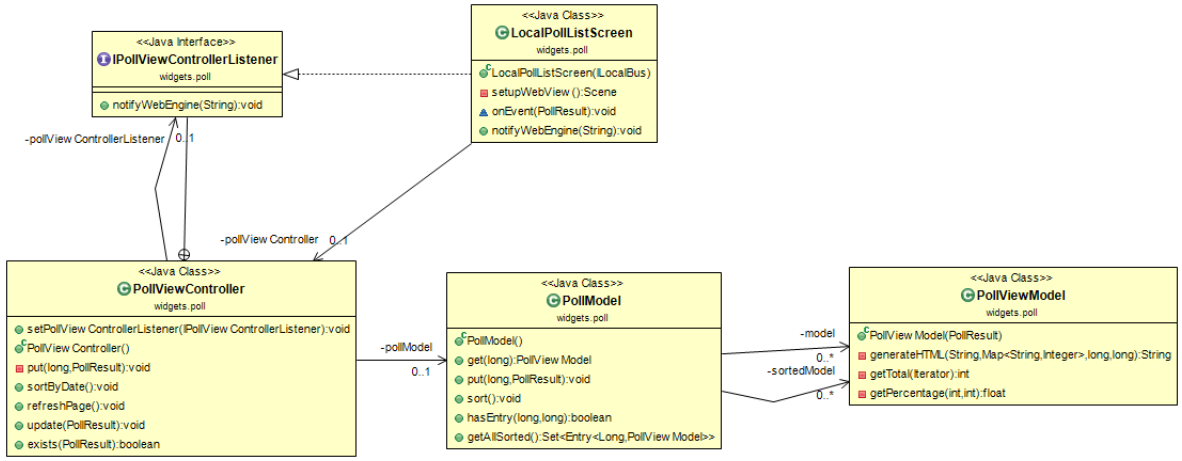


Figure 5.12: UML class diagram of Poll Results Widget on PollManager node used in the polling application

realized in this design in a simplified way. The observer class LocalPollListScreen, which is responsible for updating the `javafx.scene.web.WebView` with the real-time results of all the local polls, implements the `PollViewController.IPollViewControllerListener` and subscribes to the local event `PollResult` with the method `onEvent(pollResult: PollResult)`. As soon as it receives a new event, it invokes firstly the `pollViewController.update(pollResult)` method in order to update the cache with the new results and the responsive HTML views which are both combined using the PollViewModel class, then it calls the `pollViewController.sortByDate()` to sort the cached PollViewModel objects by date and finally it invokes the `pollViewController.refreshPage()` which concatenates all the generated HTML views on which the model is attached and notifies the Webview for the updated HTML by invoking `pollViewControllerListener.notifyWebEngine(concatenatedHtml)`.

5.2.7 User Interface

The user interface of the Android and JavaFX widget components is presented in Figure 5.13 and Figure 5.14 respectively. The first screenshot presented in Figure 5.13 corresponds to PCW, which provides a form view to create and publish a poll with multiple options. The remaining screenshots correspond to PPW and illustrate a list view displaying a collection of published polls showing the question, the author and the timestamp of the poll. As soon as the user selects the poll from the list can view either the result of the poll, in case they have already responded, or the poll responding form (third screenshot). When the user provides an answer to the poll, real time statistics based on the poll results are shown (forth screenshot). At last, Figure 5.14 illustrates the user interface of PRW, which

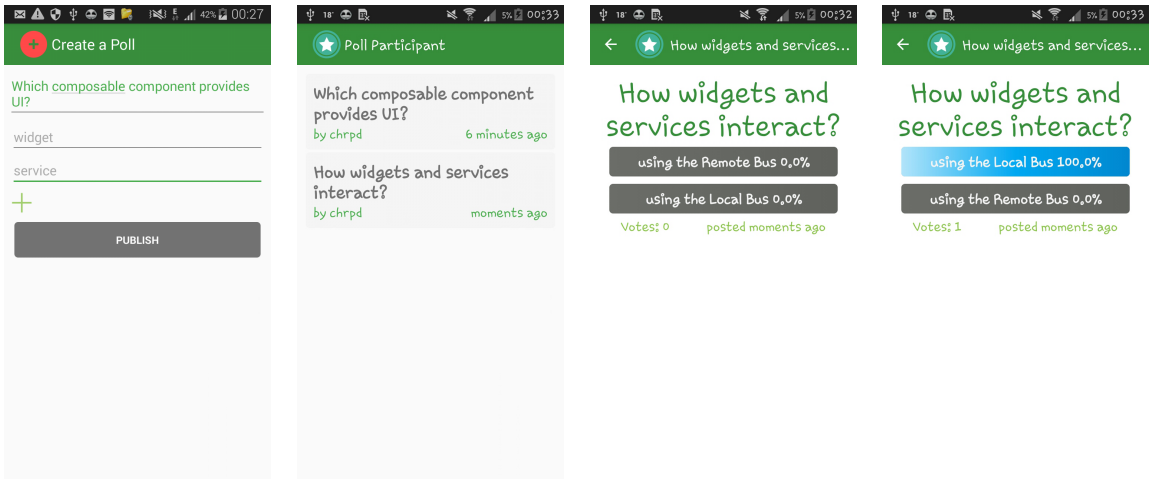


Figure 5.13: Mobile user interface of polling application

shows in real time the results of all published polls, including additional information, i.e., author, total votes, timestamp, in a view designated for screens of high resolution.

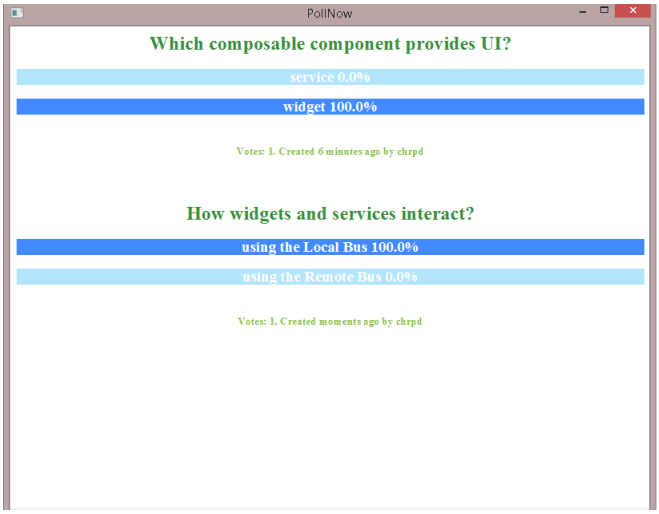


Figure 5.14: User interface of Poll Results Widget on PollManager node

5.3 Summary

In this chapter, we described the implementation of the framework presented in Chapter 3 and the polling application design provided in Chapter 4. We provide a framework for both Java and Android and a polling application that can run on Android-powered devices. Initially, we described the Java Common Library which realizes the Component

Registry subsystem, the Remote Bus subsystem, the Component Storage subsystem and provides the necessary abstract classes and interfaces to be used by the concrete framework implementations, i.e., Android-based and Java-based versions. Subsequently, we described those two framework implementations in detail and proceeded with the description of each CC participating in the polling application as well as the remote and local events used as the communication protocol among the CCes. At last, we provided screenshots of the user interfaces developed for the polling application. In the following chapter, we utilize those implementations to experiment with on Android-powered devices and examine the interactions among the participating CCes.

Chapter 6

Implementation Evaluation

This section presents the evaluation of an experimental deployment of the framework implementation and the polling application, as described in Chapter 5. It represents the last part of the evaluation of the discussed architecture and aims at establishing the viability of the architecture in practice. In order to achieve this, we demonstrate important interactions and functionalities of the system by running a set of experiments on real devices where the implementations described above are being deployed. The focus is not on evaluating the performance of the implemented system, but rather on validating the framework and proving that the implemented functionalities work as designed.

We begin with presenting the testbed devices and general assumptions and setup that hold for all described experiments. Subsequently, we proceed with describing each experiment individually and providing the respective results analysis. The experiments fall into two categories: (i) those being carried out in a static, minimal topology where the CCes and devices act as intended without failures and network topology changes and (ii) those demonstrating dynamic behavior, proving the capability of the system to maintain its state while long delays and network changes take place.

6.1 Evaluation Testbed and Data Collection

The goal of the experiments is to validate the implementation and protocol design, not study the performance of the system in production deployments. To this end, we run a set of experiments by deploying real implementations on real devices. The evaluation testbed, as presented in Table 6.1, consists of four consumer Android smartphones that will act as clients and two Windows laptops that will act as Liberouters.

We consider the following assumptions during the experiments: (i) SCAMPI must be running on all testbed devices during the experiment sessions. The LibeRouter-1.1.1.apk is used for the Android-enabled devices and the SCAMPI.jar for the Windows laptop. (ii)

Table 6.1: Evaluation Testbed

Samsung Galaxy Note II N7100	Android 4.4.2 Quad-core 1.6 GHz Cortex-A9 2 GB RAM Wi-Fi 802.11 a/b/g/n
Samsung Galaxy S Mini 5	Android 4.4.2 Quad-core 1.4 GHz Cortex-A7 1.5 GB RAM Wi-Fi 802.11 a/b/g/n
Sony Xperia M2	Android 4.4.4 Quad-core 1.2 GHz Qualcomm Snapdragon 1 GB RAM Wi-Fi 802.11 a/b/g/n
Fujitsu Lifebook UH572 Ultrabook	Windows 8.1 `Java 1.8.0_66` Intel Core i5 3317U Prozessor 2x 1,70 GHz 8 GB RAM

The framework software enables the interactions among the composable components of the system and the instatiation of the services and must be pre-installed and trunning on all testbed devices. The microframe.apk is used for the Android-enabled devices and the microframe.jar for the Windows laptops. (iii) The PollCreatorWidget.apk is pre-installed on the nodes of type PollCreator, before each experiment. PollCreatorWidget is designed as a widget CC and is responsible for publishing the remote instantiation event messages that include the CC binaries and metadata and creating a poll.

The experiments are automated using background tasks that run on the mobile devices and generate `NewPoll` and `PollAnswer` local events which in turn trigger the execution of the examined interactions. Furthermore, an Android Service is developed for automatically switching on and off the Wi-Fi on the Android-enabled devices in the *temporary node absence* test case presented in section 6.3.2.1. A logging mechanism is developed and used to record the local and remote events that occur during the CCes interactions. The logging provide the following information for each event:

- `eventAction`: it defines the action of the event, i.e., publish, subscribe, save.
- `eventType`: it defines whether the event is local or remote.
- `eventTag`: the tag of the event, e.g., *NewPoll*.
- `serviceTag`: the tag of the service, e.g., *Polls*.
- `eventContent`: the content of the event in JSON format.
- `eventID`: the event identifier.

- `eventCreatedAt`: the timestamp that the event is created.
- `componentName`: the name of the CC.
- `componentRole`: it defines the role of the CC, i.e., widget or service.
- `componentChild`: it represents the class of the CC binary code where the logger recorded the event.
- `IP`: the IP address of the device, on which the event takes place.
- `MAC`: the MAC address of the device, on which the event takes place.

6.2 Static Topology Test Cases

This section presents the results of the experiments that take place in a static topology, where only the participating devices are connected to a stable access point during the entire experiment session. The selected scenarios to be run on top of this topology are (i) the *service instantiation phase*, where we examine how an application consisted of a set of autonomous components can be instantiated in the network, originating from only one device. (ii) The second scenario refers to the *functional phase* of the running polling application where we examine the interactions and functionalities of the system.

6.2.1 Experiments Setup

In this section we present two test cases taking place in a static topology that demonstrate the basic scenarios of the application instantiation and operation, i.e., *instantiation phase* and *functional phase*. Figure 4.1 illustrates the interactions among the specified nodes used in the design of the experiments in this chapter.

In *instantiation phase*, as shown in Table 6.2, in the static topology experiments, Client A (PollCreator) acts as the poll application initiator, which publishes the required binaries (PollParticipantWidget.apk, PollParticipantService.jar and PollManagementService.jar) in the network, as well as the poll creator, which is responsible for creating and publishing new polls. Client B (PollParticipant) acts as the poll participant, which at a certain point receives the PollParticipantWidget.apk and the PollParticipantService.jar files and handles them in order to load the proper views and classes needed to instantiate the CCes within the node instance. At last, the Liberouter, which acts as the infrastructure node (PollManager), able to persist and process large sets of data for a long period of time, receives the PollManagementService.jar and loads the classes required for the instantiation of the service CC.

After the completion of the application creation on multiple nodes in the network, the *functional phase* takes place, where interactions occur in order to realize all the features

described in the polling application scenario in section 4.1. More specifically, an Android background task, called *publishDummyPoll*, is implemented and triggered by the PCW as soon as it is started. This task is responsible for creating a new poll and publishing it to the Local Bus within a *NewPoll* local event message every 30 seconds. The initiation of this task denotes the beginning of the *functional phase* experiment. The creation and publishing of all the remote and local events described in section 4.1 is automated for the evaluation purposes.

The interactions of the participating CCes are described in section 4.1. In the following sections, we present in detail the evaluation of the aforementioned functionalities and interactions.

Table 6.2: Experiment Testbed - Static Topology

Node Name	**Node Role**	**Software**	**Test Device**
Client A	PollCreator	PollCreatorWidget.apk (PCW.apk, 5.6 MB) PollCreatorService.jar (PCS.jar, 24 KB)	Samsung Galaxy Note II N7100
Client B	PollParticipant	PollParticipantWidget.apk (PPW.apk, 5.4 MB) PollParticipantService.jar (PPS.jar, 24 KB) framework.jar (pre-deployed, 5.69 MB)	Sony Xperia M2
Liberouter	PollManager	PollManagementService.jar (PMS.jar, 24 KB) PollResultsWidget.jar (PRW.jar, 24 KB) framework.jar (pre-deployed, 4.21 MB)	Fujitsu Lifebook UH572 Ultrabook

6.2.2 Results and Analysis

In this section, we provide the results of the *service instantiation phase* and *functional phase* experiments carried out in the above-mentioned setting. Furthermore we analyze the results proving the viability and functionalities of the opportunistic polling application described in section 4.1.

6.2.2.1 Service Instantiation Phase

In this experiment, we measure the dynamic instantiation of the polling application on multiple node instances. As described in section 4.1, the polling application consists of several CCes that run on different nodes and together compose the final application. We prove the viability of the instantiation of the discussed CCes passed as executable files in the network from one of the nodes. Figure 6.1 illustrates the *ConfigMessage* and *Instantiate* remote event messages that lead to the instantiation and configuration of the resulting service and the corresponding mean delays. More specifically, at time t_0,

Table 6.3: The mean delays in milliseconds recorded for the instantiation remote events and the confidence interval on each mean

	$\bar{\Delta t_2} = \bar{t_2} - \bar{t_0}$	$\bar{\Delta t_4} = \bar{t_4} - \bar{t_1}$	$\bar{\Delta t_5} = \bar{t_5} - \bar{t_3}$
Mean Delay	14575.9 ms	11203.5 ms	54053.7 ms
CI 95%	[7976.36, 21175.44]	[6210.43, 16196.57]	[37458.58, 70648.82]
95th Percentile	34202.2 ms	25508.9	92320.2 ms
Binary Size	24 KB	24 KB	5.4 MB

Client A (PollCreator) publishes the event *ConfigMessage* (1: publish ConfigMessage with parameters: PMS.jar, config) which includes the PollManagementService.jar and configuration metadata for proper instantiation by the receiving node. The configuration information includes the CC role which is *SERVICE_INFRASTRUCTURE* and thus, it is handled and instantiated by the Liberouter node (1.1 Instantiate PMS.jar). In the meantime, at time t_1, Client A (PollCreator) publishes an other *ConfigMessage* (2: publish ConfigMessage with parameters: PPS.jar, config) with the PollParticipantService.jar the role of which is *SERVICE_MOBILE* and handled and instantiated by Client B (2.1 Instantiate PPS.jar) which acts as the PollParticipant node, as described in section 4.1. At last, Client A, at time t_3 publishes the last *ConfigMessage* for the PollParticipantWidget CC (3: publish ConfigMessage with parameters: PPW.jar, config). The role of this CC is *WIDGET_MOBILE* and is handled and instantiated by nodes of type *PollParticipant*. In this experiment the PollParticipant node is Client B. Each of the timestamps t_2, t_4 and t_5 denote the completion of the CC instantiation on the corresponding nodes. In Table 6.3, we present the mean delays recorded during the instantiation phase as shown in Figure 6.1. For the completion of the steps 1 and 1.1 that lead to the instantiation of the PMS CC of size 24 KB in the Liberouter node, the mean delay $\bar{\Delta t_2}$ equals to 14575.9 ms with confidence interval in 95% [7976.36, 21175.44]. For the instantiation of PPS of size 24 KB on Client B, the mean delay $\bar{\Delta t_4}$ equals to 11203.5 ms with confidence interval in 95% [6210.43, 16196.57]. At last, for the steps 3 and 3.1 that refer to the transmission and instantiation of PPW of size 5.4 MB, for which user permission required to be given manually, the mean delay $\bar{\Delta t_5}$ is 54053.7 ms with confidence interval in 95% [37458.58,

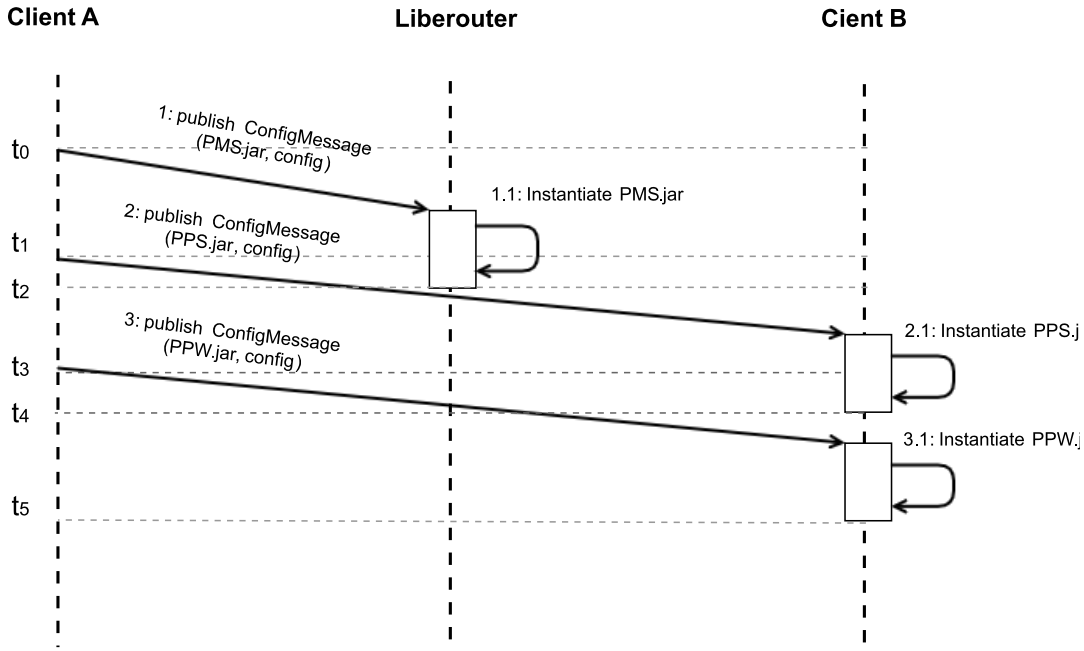


Figure 6.1: UML sequence diagram of instantiation phase for remote events ConfigMessage and the respective delays.

70648.82]. $\bar{\Delta t_5}$ is greater than the other two mean delays because the size of the binary code transfered and installed is bigger and the installation process includes a step where human input is required in order to provide installation permission.

6.2.2.2 Functional Phase

With this experiment we want to validate the viability of the framework by studying the interactions among the composable components of a polling opportunistic application. The interaction model consists of local and remote event messages. In the examined polling application, the local messages correspond to the communication protocol that enables composable components residing on the same machine to interact with each other via the Local Bus. The remote events are published to the Remote Bus, an abstraction that encapsulates the opportunistic networking infrastructure, and consumed by remote composable components. The aim of this experiment is to present all the important interactions after the instantiation of the application. We focus first on the remote events and then we go deeper and evaluate the local events and the internal execution flows.

Figure 6.2 illustrates all the remote events among Client A, Client B and Liberouter nodes for publishing a poll, responding to it and calculating, publishing, displaying poll results as a result of a new poll answer issued by the poll participant. The Table 6.4 presents

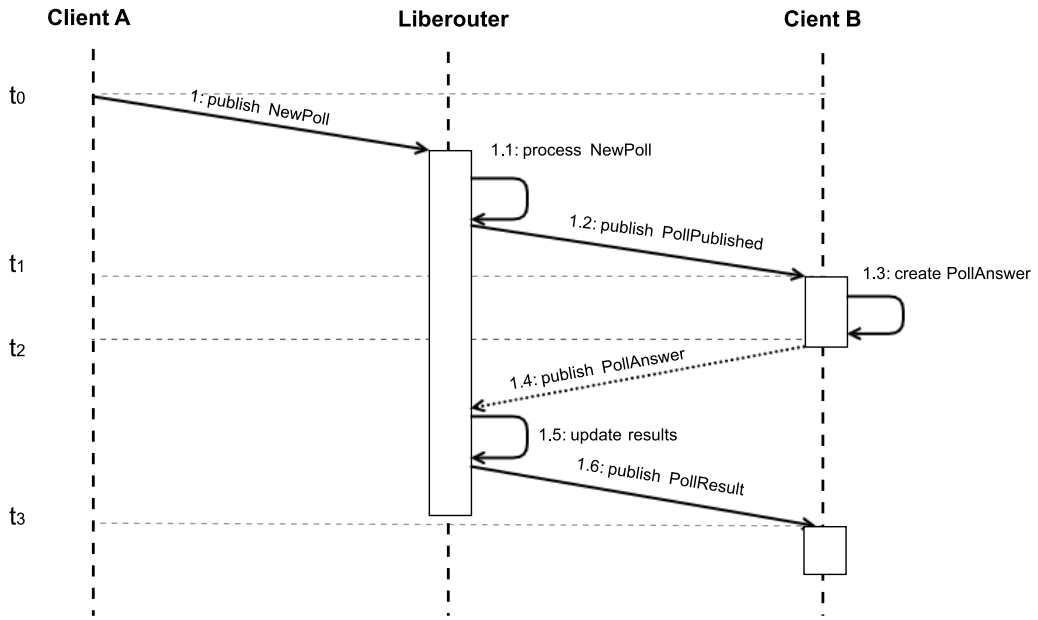


Figure 6.2: UML sequence diagram of functional phase for remote events and the respective delays.

the means of the events processing and transmission delays that prove the realization of those interactions.

More specifically, Client A publishes the event message *NewPoll* at time t_0 which contains the poll question and answer options of the new poll, a globally unique user identifier and the required metadata (1. publish NewPoll). As soon as the event arrives on Liberouter, it processes the new poll message by extracting the new poll information, checking if the poll already exists in the database and if not, it stores it locally and creates a new object called *PublishedPoll* that contains the information of the new poll as well as a globally unique poll identifier (1.1: process NewPoll). Subsequently, it publishes the event message *PollPublished* (1.2: publish PollPublished). At time t_1 the *PollPublished* arrives at Client B, the participant processes the event and automatically selects the first poll answer option (1.3: create PollAnswer, more details are presented later in this section). At time t_2, the participant publishes the *PollAnswer* event to the Remote Bus (1.4: publish PollAnswer). As soon as, it arrives at Liberouter, the PollService composable component that is running on that node, updates the poll results and creates a new *PollResult* event (1.5: update results) that publishes to the Remote Bus provided by the framework (1.6: publish PollResult). At t_3, Client B receives the *PollResult*. As shown in Table 6.4,the mean delay $\bar{\Delta t_1}$ equals to 1498.06 ms (with confidence interval [1485.24,1510.87] and 95th percentile equals to 1559 ms) and includes the steps 1 and 1.2 that correspond to the transmission times from Client A to Liberouter and Liberouter to Client B respectively as well as the step 1.1 that corresponds to the processing time of *NewPoll* on Liberouter. It is logical that the $\bar{\Delta t_2}$, which is equal to 777.96 ms (with confidence interval [745.15,

810.76] and 95th percentile equals to 948.5 ms), is smaller than $\bar{\Delta t}_1$ since it only includes local processing delay. Concerning the $\bar{\Delta t}_3$, it equals to 610.55 ms (with confidence interval [598.58, 622.52] and 95th percentile equals to 732 ms) and is smaller than the previously-mentioned, it includes the steps 1.4 and 1.6 that correspond to transmission delays and 1.5 that is refers to processing tasks on the Liberouter. A possible reason that $\bar{\Delta t}_2$ is greater than $\bar{\Delta t}_3$, even though $\bar{\Delta t}_2$ only includes local processing, is the realization of the Local RPC (see section 3.2.3) within Client B node, which is a resource-constraint device, during the step 1.3.

Table 6.4: The mean delays in milliseconds recorded in the functional phase for remote events and the confidence interval on each mean

	$\bar{\Delta t}_1 = \bar{t}_1 - \bar{t}_0$	$\bar{\Delta t}_2 = \bar{t}_2 - \bar{t}_1$	$\bar{\Delta t}_3 = \bar{t}_3 - \bar{t}_2$
Mean Delay	1498.06 ms	777.96 ms	610.55 ms
CI 95% (ms)	[1485.24, 1510.87]	[745.15, 810.76]	[598.58, 622.52]
95th Percentile	1559 ms	948.5 ms	732 ms

Figure 6.3 presents a sequence diagram for the local events taking place during the step 1.3 of the sequence diagram 6.2 on Client B. This diagram shows that the mean time required from the time point that Client B has received the remote event *PollPublished* message till the time Client B publishes the remote event message *PollAnswer* that corresponds to the sum of the processing delays of the interactions of the local composable components on Client B. More specifically, we analyze the step 1.3 presented in Figure 6.2. The overall time needed for the local interactions in Figure 6.3 equals to 777.96 ms, which approaches $\bar{\Delta t}_2$ mean delay that corresponds to the aforementioned step 1.3.

As soon as the remote event *PollPublished* is received on Client B, the PollParticipantService composable component processes the remote message and stores the object PollPublished (1.3.1: process PollPublished) and publishes it to the Local Bus (1.3.2: publish PollPublished), so that the widgets that have subscribed to this event can receive the notification. The Local Bus notifies the subscriber PollParticipantWidget (1.3.3: notify PollPublished), which processes the local event and responds to the poll (1.3.4: respond to PollPublished). Subsequently, it publishes the *PollAnswer* to Local Bus that notifies the subscriber PollParticipantService (1.3.5 and 1.3.6 respectively). PollParticipantService component processes the event and prepares to publish it to Remote Bus (1.3.7 and 1.4 respectively).

As shown in Table 6.5,the mean delay $\bar{\Delta t}'_1$ equals to 78.85 ms (with confidence interval [71.82,85.89] and 95th percentile equals to 126.75 ms) and includes the steps 1.3.1, 1.3.2 and 1.3.3 that correspond to the processing time of the remote event *PollPublished* received on Client B from the Liberouter and the time required for notifying the widget components subscribers running on Client B node via the Local Bus. The mean delay $\bar{\Delta t}''_1$, recorded for the step 1.3.4, equals to 542.27 ms (with confidence interval [513.51,

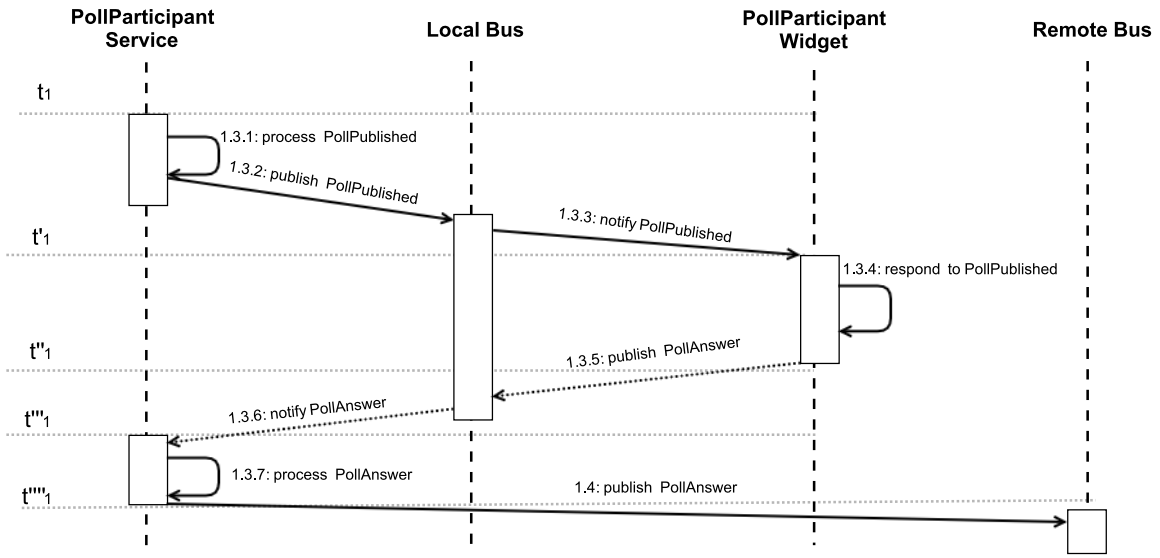


Figure 6.3: UML sequence diagram of functional phase for local events on Client B and the respective delays.

571.03] and 95th percentile equals to 723.85 ms) and is the greatest local delay recorded on Client B. The reason is the realization of the local RPC model, i.e., in step 1.3.4, the PollParticipantWidget queries the PollParticipantService in order to check whether the particular user has already responded to the poll and whether there are available poll results to be shown. We implemented this functionality to show that the framework enables the *client pull* communication model on top of opportunistic networks. The mean delay $\Delta\bar{t}'''_1$, which equals to 72.15 ms (with confidence interval [60.15, 84.14] and 95th percentile equals to 169.35 ms), includes the time required to publish the local event *PollAnswer* via the Local Bus (1.3.5 and 1.3.6). At last, the mean delay $\Delta\bar{t}''''_1$, which equals to 81.17 ms (with confidence interval [75.54, 86.80] and 95th percentile equals to 113.75 ms), corresponds to the processing of the local event *PollAnswer* on PollParticipantService that processes and encapsulates the local event into the remote event by attaching a globally unique identifier and then, it publishes it to the Remote Bus (1.3.7 and 1.4 respectively).

Considering the above, we conclude that an application or a service designed by applying the discussed architecture on top of challenged networks is able to show behavior similar to centralized infrastructure networks. We observe low delays which lead to real-time messaging experience that is taking place in standard Internet-based mobile applications as well as we prove that an application composed of self-contained components, with the

Table 6.5: The mean delays in milliseconds recorded in the functional phase for local events and the confidence interval on each mean

	$\Delta\bar{t}'_1 = \bar{t}'_1 - \bar{t}_1$	$\Delta\bar{t}''_1 = \bar{t}''_1 - \bar{t}'_1$	$\Delta\bar{t}'''_1 = \bar{t}'''_1 - \bar{t}''_1$	$\Delta\bar{t}''''_1 = \bar{t}''''_1 - \bar{t}'''_1$
Mean Delay	78.85 ms	542.27 ms	72.15 ms	81.17 ms
CI 95% (ms)	[71.82, 85.89]	[513.51, 571.03]	[60.15, 84.14]	[75.54, 86.80]
95th Percentile	126.75 ms	723.85 ms	169.35 ms	113.75 ms

discussed structure can provide sufficient orchestration for its proper operation needed in standard application without the need of a centralized coordinator, i.e., a centralized entity commonly used in Internet-based systems, e.g., Kubernetes Master.

6.3 Dynamic Behavior Test Cases

In this section, we present two experiments that demonstrate dynamic behavior within the polling application system. The goal is to show that the state can be maintained after long delays and system failures. In the first experiment, the PollParticipant node disconnects shortly from the network while the application is running and when it returns, it reconstructs the state of the old and current polls. In the second experiment, we evaluate the performance of the framework itself by interchanging between two framework instances at a fixed time interval in order to show that a temporary framework failure does not affect the reliability of the system. It affects the availability since part of the service is down, however the state gets reconstructed as soon as the whole system is up and running again.

6.3.1 Experiments Setup

Table 6.6 presents the evaluation testbed for the dynamic test cases. As previously in the static topology setup, Client A acts as a PollCreator node and Client B as PollParticipant node. The differences in the setup of this set of experiments compared to the static topology are: (i) in both dynamic cases, i.e., *temporary node absence* and *switching framework instance*, the PollParticipant node is a more powerful smartphone (Samsung Note II instead of Sony Xperia M2, see Table 6.1) and (ii) in the second test case two Framework instances (F and F') are created on the laptop and used to evaluate the framework performance. Similarly to the static topology setup, framework is installed and running on all devices apart from the PollCreator device where it is included in the PCW.apk and started as soon as the widget is being initialized.

In the *temporary node absence* test case, the PollCreator disseminates the *ConfigMessage* events which include the binary code for each application CC as described in section

6.2.2.1. As soon as the PPW CC is instantiated on the PollParticipant node, an Android Service, which is implemented within PPW, is triggered and starts running in the background with the task of switching on and off the Wi-Fi every, forcing the framework and SCAMPI instances running on the node get disconnected from the network and reconnect again. In the *switching framework instance* test case, after the application instantiation, given the framework instance F running on the PollManager node, the automated poll posting task, *publishDummyPoll*, on the PollCreator starts publishing a new poll every 30 seconds and every 2 minutes the F and F' instances interchange. The interactions of the participating CCes are described in section 4.1.

Table 6.6: Experiment Testbed - Dynamic behavior

Node Name	**Node Role**	**Software**	**Test Device**
Client A	PollCreator	PollCreatorWidget.apk (pre-deployed, PCW.apk, 5.6 MB) PollCreatorService.jar (PCS.jar, 24 KB)	Samsung Galaxy S Mini 5
Client B	PollParticipant	PollParticipantWidget.apk (PPW.apk, 5.4 MB) PollParticipantService.jar (PPS.jar, 24 KB) framework.apk (pre-deployed, 5.69 MB)	Samsung Galaxy Note II N7100
Liberouter (Framework instances *F* and *F'*)	PollManager	PollManagementService.jar (PMS.jar, 24 KB) PollResultsWidget.jar (PRW.jar, 24 KB) framework.jar (pre-deployed, 4.21 MB)	Fujitsu Lifebook UH572 Ultrabook

6.3.2 Results and Analysis

In this section, we provide the results and analysis of the *temporary node absence* and *switching framework instance* experiments carried out in the above-mentioned setting in order to evaluate the framework performance.

6.3.2.1 Temporary Node Absence

In this experiment, we demonstrate the capability of the PollParticipant node to reconstruct the current and previously published polls state after a brief disconnection from the network. As mentioned previously, the Wi-Fi on Client B is being switched on

and off every 30 seconds. We call the periods that the Wi-Fi is switched on and off, connected period and disconnected period respectively. The delays for the instantiation phase, mapping to the timestamps displayed in the sequence diagram 6.1 are the following: (i) Δt_2 equals 31180 ms, (ii) Δt_4 equals 25157 ms and (iii) Δt_5 equals 90705 ms. Figure

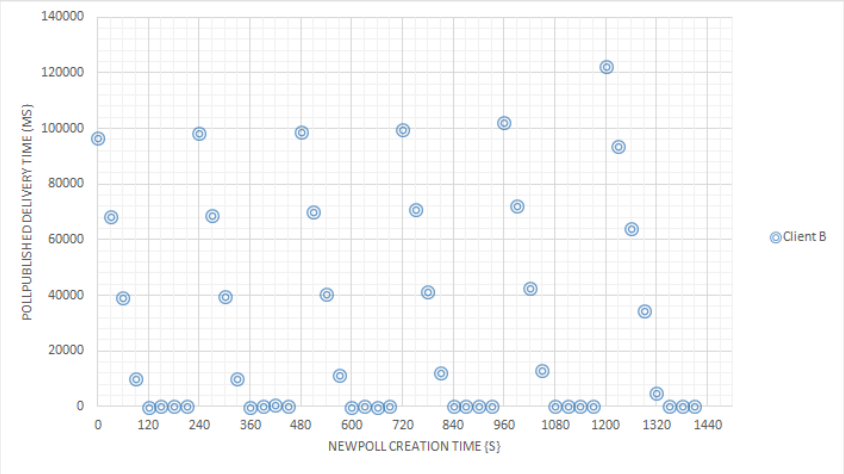


Figure 6.4: PollPublished remote event delivery delay in milliseconds on Client B in function of NewPoll local event creation time in seconds on Client A

6.4 illustrates the end-to-end delays for the *NewPoll* and *PollPublished* local and remote event messages from the perspective of Client B as shown in the sequence diagram 6.2 as a function of the creation time of the local event *NewPoll* created within the PCW CC on Client A. Those interactions correspond to the first part of the polling application functional phase, i.e., $\Delta\bar{t}_1 = \bar{t}_1 - \bar{t}_0$ of Figure 6.2, where event messages published by Client A, received, transformed and republished by Liberouter to be received by Client B. A sawtooth pattern is formed while generated event messages are buffered in the SCAMPI instance cache waiting for the next contact to be published. The delay reaches its peak just after the end of the of each connected period and approaches its minimum values during the connected period. The maximum delay shown in Figure 6.4 for the first part of the functional phase is approximately 122 seconds, which corresponds to the disconnected periods due to Wi-Fi switching off. During the connected periods, it is observed, similarly to the *functional phase* experiment, that the values are close to zero which leads to real-time experience that resembles centralized infrastructure network behavior.

In this experiment we observe that the mean delays follow the same pattern noticed in *functional phase* test case results with the difference of the additional delay of the diconnected periods of PollParticipant node. As shown in Table 6.7, the mean delay $\Delta\bar{t}_1$ equals to 29827.9 ms (with confidence interval [18967.7, 40688.1] and 95th percentile equals to 101118.3 ms) and includes the steps 1 and 1.2 of Figure 6.2 that correspond to the transmission times from Client A to Liberouter and Liberouter to Client B respectively as well as the step 1.1 that corresponds to the processing time of *NewPoll* on Liberouter. Mean delay $\Delta\bar{t}_2$, which is equal to 29105.8 ms (with confidence interval [0.0, 78416.7]

Table 6.7: The mean delays in milliseconds recorded in the temporary node absence scenario for remote events and the confidence interval on each mean

	$\bar{\Delta t_1} = \bar{t_1} - \bar{t_0}$	$\bar{\Delta t_2} = \bar{t_2} - \bar{t_1}$	$\bar{\Delta t_3} = \bar{t_3} - \bar{t_2}$
Mean Delay	29827.9 ms	29105.8 ms	13937.1 ms
Max Delay	122469 ms	1207934 ms	130586 ms
CI 95%	[18967.7, 40688.1]	[0.0, 78416.7]	[2897.1, 24977.1]
95th Percentile	101118.3 ms	56569.8 ms	129546.4 ms

and 95th percentile equals to 56569.8 ms), is smaller than $\bar{\Delta t_1}$ since it only includes local processing delay. The CI lower level equals to 0.0 due to the occurrence of extreme values. $\bar{\Delta t_3}$ equals to 13937.1 ms (with confidence interval [2897.1, 24977.1] and 95th percentile equals to 129546.4 ms) and is smaller than the previously-mentioned delays, since it includes the steps 1.4 and 1.6 that correspond to transmission delays and 1.5 that is refers to processing tasks on the Liberouter. We conclude that the system behavior resembles the interactions observed in *functional phase* and eventually the PollParticipant node is capable of resynchronizing with the current system state and retrieve the old and current polls, respond to them and retrieve the poll results.

6.3.2.2 Switching Framework Instance

With this experiment, we evaluate the framework performance and aim at proving that the framework is capable of rebuilding its state. As described in the setup section 6.3.1, two framework instances are running in turn on Liberouter, F and F'. The period that a framework instance is down is called disconnected period and the time it is up is called connected period. Similarly to the previous test cases, Client A and Client B act as PollCreator and PollParticipant respectively and Liberouter as PollManager. We perform a variation of the *functional phase* test case, illustrated in Figure 6.2, with the difference of switching framework instance on Liberourer at a fixed time interval, i.e., 2 minutes, and we focus on the delays of the event messages delivered to Liberouter. The *NewPoll* local event message is being create every 30 seconds similarly to previous test cases. The discussed messages correspond to the steps *1: publish NewPoll* and *1.4: publish PollAnswer* of Figure 6.2.

Tables 6.8 and 6.9 present the mean delays of the aforementioned messages for F and F' respectively. We observe that the event messages reach both framework instances with different delays for each one. For F, which is the instances that is created first during the experiment session, the mean delay for step 1 equals to 40356.44 ms (with confidence interval [24799.46, 55913.42] and 95th percentile equals to 132033.30 ms). Concerning the same delay on F', which is being created on the expiration of the first two

Table 6.8: The mean delays in milliseconds recorded in switching framework instance scenario for remote events delivered and published from framework instance F on Liberouter as illustrated in Figure 6.2 and the confidence interval on each mean

	1: publish NewPoll	**1.4: publish PollAnswer**
Mean Delay	40356.44 ms	36954.22 ms
CI 95%	[24799.46, 55913.42]	[21980.92, 51927.52]
95th Percentile	132033.30 ms	120520.80 ms

Table 6.9: The mean delays in milliseconds recorded in switching framework instance scenario for remote events delivered and published from framework instance F' on Liberouter as illustrated in Figure 6.2 and the confidence interval on each mean

	1: publish NewPoll	**1.4: publish PollAnswer**
Mean Delay	38110.69 ms	35496.16 ms
CI 95%	[23175.36, 53046.01]	[20671.20, 50321.11]
95th Percentile	122727.05 ms	114483.05 ms

minutes of the experiment, it is equal to 38110.69 ms (with confidence interval [23175.36, 53046.01] and 95th percentile equals to 122727.05 ms and slightly smaller than the mean delay on F, which proves the pattern illustrated in Figure 6.5. In this figure, we notice that a sawtooth pattern is formed, based on the *NewPoll* remote event message delay perceived on Liberouter as a function of the creation time of the local event *NewPoll* created within the PCW CC on Client A, demonstrating the buffering of the application state, its dissemination at the next contact from the SCAMPI local instance and the capability of the system to continue performing as intended. Similarly, we observe the same phenomenon for the step *1.4: publish PollAnswer*, the occurrence of which also confirms that the framework is capable of reconstructing the system state and perform as designed.

Table 6.10 presents the mean delays Δt_1, Δt_2, and Δt_3 for the remote events as illustrated in Figure 6.2, which demonstrate that the PollParticipant node receives the intended remote events and interacts with the PollManager node instance, on which either the F or F' is running, as designed with an additional delay due to the time needed by the framework instances to get synchronized after the disconnected periods.

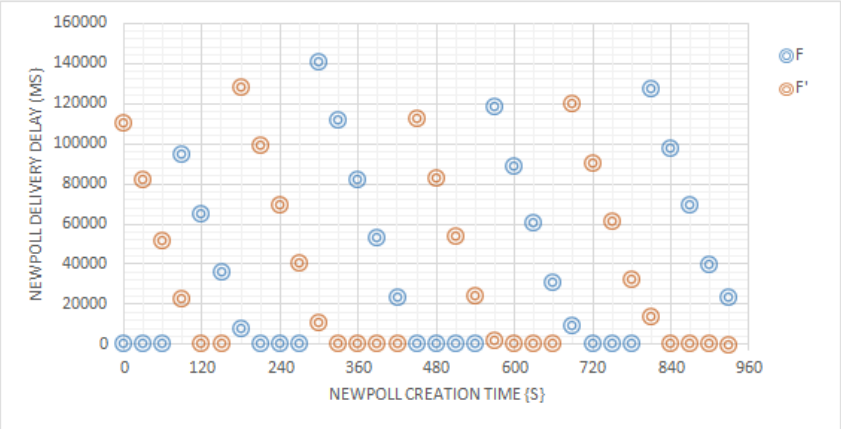


Figure 6.5: NewPoll remote event delivery delay in milliseconds on Liberouter in function of NewPoll local event creation time in seconds on Client A

Table 6.10: The mean delays in milliseconds recorded in the switching framework instance scenario for remote events as illustrated in Figure 6.2 and the confidence interval on each mean

	$\bar{\Delta t_1} = \bar{t_1} - \bar{t_0}$	$\bar{\Delta t_2} = \bar{t_2} - \bar{t_1}$	$\bar{\Delta t_3} = \bar{t_3} - \bar{t_2}$
Mean Delay	3696.03 ms	391.19 ms	459390.41 ms
CI 95%	[1543.61, 5848.46]	[355.23, 427.14]	[366194.72, 552586.09]
95th Percentile	23611.25 ms	641.15 ms	894623.65 ms

6.4 Summary

In this chapter, we presented the last part of the evaluation of the architecture proposed in this work. Initially, we described the evaluation testbed and general assumptions applied to the examined test cases. Subsequently, we proceeded with the description of the setup for a set of static topology scenarios used to carry out experiments with the aim to prove the viability of the discussed framework in a static, minimal setting and provided the results of the experiments and their analysis. At last, we presented the evaluation of the system considering a set of scenarios triggering dynamic behavior in order to evaluate the capability of the framework to rebuild its state and act as intended after a dynamic change in the system.

Chapter 7

Conclusion

This thesis presents an architecture for building composable applications and services on top of challenged networks. Initially, it describes the background areas that frame this work and proceeds with the requirement analysis of concrete scenarios taking place in OppNets and observes key concepts that needed to be adopted in order to achieve their realization. Taking those requirements and concepts into account, it elaborates on the architecture that can enable those scenarios and describes an application framework that can enforce the discussed structure to opportunistic applications and frameworks. Subsequently, in order to evaluate the viability of the architecture, it provides the design of the scenarios by applying the examined architecture, utilizing the application framework, the real implementation of the framework for both Java and Android and one of the discussed application scenarios, i.e., the polling application. At last, it evaluates the implementations by experimenting on real devices.

The thesis concludes that building opportunistic applications and services by composing them out of self-contained components and enforcing the proposed structure is viable and it does not induce additional complexity commonly seen in Internet-based systems enabled by Microservices architecture due to coordination overhead. In the implementation evaluation, we observe system behaviors noticed in standard Internet-based applications such as real-time experience as well as the capability of the system to reconstruct its state after long delays and system changes.

The major contribution of this work is the application framework that (i) enables modular system designs by breaking complex applications and services into smaller manageable self-contained components, (ii) enforces self-coordination without the need of a central orchestration entity usually needed in standard web-based applications, (iii) supports the realization of systems on top of OppNets providing the ability to implement both *client pull* and *server push* communication models, (iv) enables the expansion of the applications in remote neighborhood networks by buffering the current state and disseminate as soon as being connected to the next network without the use of Internet and (v) allows the distribution and instantiation of the applications without the need of a centralized

distribution mechanism.

Concerning future research efforts on this work, we consider the following improvements:

- Evaluation of a set of complex test cases considering dynamic topologies. For instance, multiple neighborhood networks are established utilizing infrastructure nodes, i.e., Liberouter routers, and the participating mobile nodes physically switch networks with the result of transferring the current applications and services state from one network to the other.
- The development and evaluation of an independent Component Registry framework subsystem that is able to communicate with the Component Controller framework subsystem using the Local Bus and provide public interfaces to local and remote CCes for providing information on the in-network running CCes. This will allow the realization of scenarios where users can decide on the CCes of their applications and thus, build their own composable user interfaces and gain the freedom to generate an application based on their preferences.
- The design, implementation and evaluation of a security layer that will provide protection against unauthorized access and utilization of resources, privacy invasion and confidentiality disclosure.

Bibliography

[Ahlg 11] B. Ahlgren, C. Dannewitz, C. Imbrenda, and D. Kutscher. "A Survey of Information-Centric Networking (Draft)". 2011.

[Andr] "AndroidAnnotations Framework". http://androidannotations.org/. Accessed: 15.07.2016.

[Bria 97] L. C. Briand, J. W. Daly, and J. Wust. "A unified framework for cohesion measurement in object-oriented systems". In: *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pp. 43–53, Nov 1997.

[Burg 06] J. Burgess, B. Gallagher, D. Jensen, and B. Levine. "MaxProp: Routing for vehicle-based delay-tolerant networks". IEEE, Spain, 4 2006.

[Burn 05] B. Burns, O. Brock, and B. N. Levine. "MV routing and capacity building in disruption tolerant networks". IEEE, 3 2005.

[Cerf 07] V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall, and H. Weiss. "Delay-tolerant networking architecture.". Tech. Rep., SRI International, Menlo Park, California, 4 2007. RFC 4838.

[Cock 05] A. Cockburn. "Hexagonal architecture". April 2005. http://alistair.cockburn.us/Hexagonal+architecture.

[Cont 10] M. Conti and M. Kumar. "Opportunities in Opportunistic Computing". IEEE, 1 2010.

[Cook] "Cooking Fox EventBus Adapter". https://github.com/cookingfox/eventbus-adapter-java. Accessed: 15.07.2016.

[Dela] "Delay-tolerant Networking at Aalto University Comnet". http://www.netlab.tkk.fi/ jo/dtn/. Accessed: 29.07.2016.

[Dock] "Docker". https://www.docker.com/. Accessed: 05.07.2016.

[Four] "Foursquare". https://foursquare.com/. Accessed: 05.07.2016.

[Goog] "Google Guava EventBus". https://github.com/google/guava/wiki/EventBusExplained. Accessed: 15.07.2016.

[Gree] "Greenrobot EventBus". http://greenrobot.org/eventbus/. Accessed: 15.07.2016.

[Grou] "Groupon". https://www.groupon.com/. Accessed: 05.07.2016.

[Hyyt 11] E. Hyytiä, J. Virtamo, P. Lassila, J. Kangasharju, and J. Ott. "When does content float? Characterizing availability of anchored information in opportunistic content sharing". In: *INFOCOM, 2011 Proceedings IEEE*, pp. 3137–3145, April 2011.

[Kark 12] T. Kärkkäinen, M. Pitkänen, P. Houghton, and J. Ott. "SCAMPI Application Platform". In: *Proceedings of the Seventh ACM International Workshop on Challenged Networks*, pp. 83–86, ACM, New York, NY, USA, 2012.

[Kark 14] T. Kärkkäinen and J. Ott. "Towards Autonomous Neighborhood Networking". *IEEE WONS*, 2014.

[Kube] "Kubernetes Production-Grade Container Orchestration". http://kubernetes.io/. Accessed: 05.07.2016.

[Lili 07] L. Lilien, Z. H. Kamal, V. Bhuse, and A. Gupta. *The Concept of Opportunistic Networks and their Research Challenges in Privacy and Security*, pp. 85–117. Springer US, Boston, MA, 2007.

[MapD] "MapDB Data Engine". http://www.mapdb.org/. Accessed: 15.07.2016.

[Neum 95] P. G. Neumann. "Architectures and formal representations for secure systems.". Tech. Rep., SRI International, Menlo Park, California, 10 1995.

[Next] "Nextdoor". https://nextdoor.com/. Accessed: 05.07.2016.

[Pitk 12] M. Pitkänen, T. Kärkkäinen, J. Ott, M. Conti, A. Passarella, S. Giordano, D. Puccinelli, F. Legendre, S. Trifunovic, K. Hummel, M. May, N. Hegde, and T. Spyropoulos. "SCAMPI: Service Platform for Social Aware Mobile and Pervasive Computing". In: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, pp. 7–12, ACM, New York, NY, USA, 2012.

[Powe] "Power Systems and SOA Synergy". http://www.redbooks.ibm.com/redbooks/pdfs/sg247607.p Accessed: 15.07.2016.

[Rich 15] M. Richards. *Software Architecture Patterns*. O'Reilly Media, 2015.

[Scot 07] K. Scott and S. Burleigh. "Bundle Protocol Specification". Tech. Rep., SRI International, Menlo Park, California, 11 2007. RFC 5050.

[Stra] "Strategy Design Pattern". https://sourcemaking.com/design_patterns/strategy. Accessed: 15.07.2016.

[Tros 10] D. Trossen, M. Sarela, and K. Sollins. "Arguments for an Information-centric Internetworking Architecture". *SIGCOMM Comput. Commun. Rev.*, Vol. 40, No. 2, pp. 26–33, Apr. 2010.

[Tros 12] D. Trossen and G. Parissis. "Designing and Realizing an Information-Centric Internet". *IEEE Communications Magazine*, pp. 60–67, 7 2012.

[Vlie 08] H. v. Vliet. *Software Engineering: Principles and Practice*. Wiley Publishing, 3rd Ed., 2008.

[Wu 15] Y. Wu, Y. Zhao, M. Riguidel, G. Wang, and P. Yi. "Security and Trust Management in Opportunistic Networks: A Survey". *Sec. and Commun. Netw.*, Vol. 8, No. 9, pp. 1812–1827, June 2015.

[Zhao 06] W. Zhao, Y. Chen, M. Ammar, M. Corner, B. Levine, and E. Zegura. "Capacity enhancement using throwboxes in DTNs". IEEE, 10 2006.