

EMPOWERING MOBILE DEVICES IN DISTRIBUTED SERVICE-ORIENTED ENVIRONMENTS

Ioanna Zidianaki

Thesis submitted in partial fulfillment of the requirements for the
Masters of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes Campus, Heraklion, GR-70013, Greece

Thesis Advisor: Professor Constantine Stephanidis

UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

EMPOWERING MOBILE DEVICES IN DISTRIBUTED SERVICE-ORIENTED ENVIRONMENTS

Thesis submitted by
Ioanna Zidianaki
in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

Author:

Ιωάννα Ζηδιανάκη, Πανεπιστήμιο Κρήτης
Ioanna Zidianaki, University of Crete

Committee
approvals:

Καθηγητής Κωνσταντίνος Στεφανίδης (Επόπτης), Πανεπιστήμιο Κρήτης
Professor Constantine Stephanidis (Advisor), University of Crete

Επίκουρος Καθηγητής Γιώργος Παπαγιαννάκης, Πανεπιστήμιο Κρήτης
Assistant Professor George Papagiannakis, University of Crete

Κύριος Ερευνητής Δρ. Δημήτρης Γραμμένος, Ινστιτούτο Πληροφορικής ΙΤΕ
Principal Researcher Dr. Dimitris Grammenos, ICS-FORTH

Department
approval:

Καθηγητής Αντώνης Αργυρός, Πρόεδρος επιτροπής μεταπτυχιακών σπουδών
Professor Antonis Argyros, Director of Graduate Studies

Heraklion, March 2017

*Αφιερωμένο στους γονείς μου Γιώργο, Καλλιόπη
και στον αδερφό μου Μάνο...*

Declaration of Originality

I declare that this thesis is my own work. Information derived from published or unpublished work of others has been formally acknowledged.

EMPOWERING MOBILE DEVICES IN DISTRIBUTED SERVICE-ORIENTED ENVIRONMENTS

Thesis submitted by
Ioanna Zidianaki
in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

ABSTRACT

Distributed systems are collections of computers that act, work, and appear as a single coherent system. According to the model of Distributed Computing, software services components run in different computers, whereas data are shared among the network. Middleware technologies are used in the context of distributed computing systems in order to facilitate distributed object communication. For example, ICS-FORTH *FAmINE* middleware contributes to the interconnection of distributed services within Aml environments. In particular, it provides a common set of APIs targeting a variety of heterogeneous platforms and different programming languages such as Java, C++, .NET, and Python.

The increasing availability and use of wireless mobile devices brings about opportunities for new types of distributed applications. However, towards this objective, the adopted middleware needs to support mobile devices. The work reported in this thesis aims to build a *FAmINE* middleware extension, called *FAmINE4Android*. The proposed extension library facilitates the development process of distributed Android mobile applications. In details, it provides the required mechanism and tools in order to support remote communication with distributed objects running on both ordinary PCs and Android mobile devices. The proposed library offers to Android developers functionality identical to *FAmINE*'s, in a seamless way based on an intuitive Java API. Using *FAmINE4Android*, Android developers are able to effortlessly create (or re-use existing) distributed real-time applications.

The features of the *FAmINE4Android* middleware have been demonstrated by implementing a case study application in the domain of cultural heritage. This case study refers to a museum guide application, which provides information automatically based on visitors' location. The museum guide application uses the functionality provided by a *FAmINE* tracking service running on Windows OS. The tracking service builds upon advanced computer vision algorithms in order to track multiple persons within exhibition spaces using a network of RGB-D cameras. The case study highlights the contribution of the *FAmINE4Android* middleware towards including mobile devices in distributed computing platforms.

ΑΞΙΟΠΟΙΗΣΗ ΚΙΝΗΤΩΝ ΣΥΣΚΕΥΩΝ ΣΕ ΠΕΡΙΒΑΛΛΟΝΤΑ ΚΑΤΑΝΕΜΗΜΕΝΩΝ ΥΠΗΡΕΣΙΩΝ

Ιωάννα Ζηδιανάκη
Μεταπτυχιακή Εργασία

ΠΕΡΙΛΗΨΗ

Με τον όρο κατανεμημένα συστήματα εννοούμε την συλλογή υπολογιστών οι οποίοι συνεργάζονται για ένα κοινό σκοπό και επικοινωνούν μεταξύ τους μέσω δικτύου. Στο πλαίσιο των κατανεμημένων υπολογιστικών συστημάτων, γίνεται χρήση middleware τεχνολογιών προκειμένου να διευκολυνθεί η εξ αποστάσεως επικοινωνία τους. Για παράδειγμα, η πλατφόρμα *FAmINE*, ως middleware τεχνολογία του ΙΠ-ΙΤΕ, συμβάλλει στη διασύνδεση των κατανεμημένων συστημάτων σε περιβάλλοντα Διάχυτης Νοημοσύνης. Συγκεκριμένα, παρέχει ένα σύνολο από APIs για την υποστήριξη ποικίλων ετερογενών περιβαλλόντων και διαφορετικών γλωσσών προγραμματισμού όπως Java, C ++, .NET, και Python.

Η αυξανόμενη χρήση κινητών συσκευών με δυνατότητες ασύρματης δικτύωσης έχει ως αποτέλεσμα τη δημιουργία μιας νέας κατηγορίας κατανεμημένων συστημάτων. Στόχος της παρούσας εργασίας είναι η επέκταση της πλατφόρμας *FAmINE* για την υποστήριξη κινητών συσκευών. Συγκεκριμένα, προτείνεται η βιβλιοθήκη *FAmINE4Android* η οποία εσωκλείει όλους τους απαραίτητους μηχανισμούς και εργαλεία προκειμένου να υποστηριχθεί η επικοινωνία μεταξύ κατανεμημένων συστημάτων που εκτελούνται σε υπολογιστές αλλά και σε συσκευές τύπου Android. Συγκεκριμένα, μέσω μίας, εύκολης στη χρήση, διεπαφής προγραμματισμού εφαρμογών (API) σε Java, η βιβλιοθήκη *FAmINE4Android* προσφέρει στους προγραμματιστές Android εφαρμογών, πανομοιότυπη λειτουργικότητα όπως αυτή που προσφέρεται από την πλατφόρμα *FAmINE* ΙΠ-ΙΤΕ. Ως εκ τούτου, οι προγραμματιστές μπορούν εύκολα να αναπτύξουν Android εφαρμογές με δυνατότητες επικοινωνίας μεταξύ κατανεμημένων συστημάτων σε πραγματικό χρόνο.

Ένας ψηφιακός ξεναγός για συσκευές τύπου Android αναπτύχθηκε με σκοπό την ανάδειξη της χρηστικότητας της *FAmINE4Android* βιβλιοθήκης. Ο ψηφιακός ξεναγός πληροφορεί αυτόματα τους επισκέπτες ενός μουσείου ανάλογα με το έκθεμα το οποίο επισκέπτονται/πλησιάζουν. Συγκεκριμένα, χρησιμοποιεί τη λειτουργικότητα που παρέχεται από μία υπηρεσία παρακολούθησης ατόμων στον χώρο βάση πολλαπλών RGB-D καμερών. Η υπηρεσία παρακολούθησης εκτελείται σε υπολογιστή Windows και κάνει χρήση της πλατφόρμας *FAmINE* ΙΠ-ΙΤΕ ούτως ώστε να εκθέσει, μέσω δικτύου, πληροφορίες για τα άτομα που εντοπίζει κάνοντας χρήση προηγμένων αλγορίθμων υπολογιστικής όρασης. Η περίπτωση του ψηφιακού ξεναγού αναδεικνύει με σαφήνεια, τη συνεισφορά της βιβλιοθήκης *FAmINE4Android*, περιλαμβάνοντας πλέον τις κινητές συσκευές στο σύνολο των κατανεμημένων υπολογιστικών πλατφόρμων.

Ευχαριστίες (Acknowledgements)

Θα ήθελα να ευχαριστήσω τον επόπτη της μεταπτυχιακής μου εργασίας Καθηγητή Κωνσταντίνο Στεφανίδη για την συνεχή καθοδήγηση και υποστήριξή του τα τελευταία δύο χρόνια στο πλαίσιο του προγράμματος Διάχυτης Νοημοσύνης (Ambient Intelligence) του Ινστιτούτου Πληροφορικής του Ιδρύματος Τεχνολογίας και Έρευνας και ειδικότερα στο πλαίσιο της εκπόνησης της μεταπτυχιακής μου εργασίας.

Θα ήθελα επίσης να ευχαριστήσω τον Δημήτρη Γραμμένο για την ουσιαστική του υποστήριξη στην δομή της μεταπτυχιακής μου εργασίας, όπως επίσης και την Μαργαρίτα Αντόνα για την καθοδήγηση και επιμέλεια αναφορικά με τη συγγραφή της μεταπτυχιακής μου εργασίας.

Τέλος, ένα μεγάλο ευχαριστώ στον αδερφό μου Μάνο για την συνεχή καθοδήγηση και βοήθεια του καθώς επίσης την φίλη μου Ειρήνη για την διόρθωση του κειμένου και την διαρκή υποστήριξή της.

Table of Contents

Abstract.....	ix
Περίληψη.....	xi
1 Introduction	1
1.1 Distributed services in Ambient Intelligence	1
1.2 Middleware technologies in the context of Ambient Intelligence.....	1
1.3 Contribution	2
2 Background and Related Work	3
2.1 Ambient Intelligence Environments.....	3
2.2 Distributed Service Technologies.....	4
2.2.1 Middleware distributed technologies in the context of Aml	4
2.2.2 Middleware approaches	5
2.2.3 FAmINE: A middleware library for Ami environments	8
2.3 Major building blocks	11
2.3.1 Adaptive Communication Environment	11
2.3.2 Interface Definition Language	11
2.3.3 TAO IDL compiler	11
2.3.4 Android Studio dependencies.....	12
2.3.5 Light-weight data exchange formats	12
2.3.6 Run Time Type Reflection library.....	13
3 FAmINE4Android: A FAmINE extension library supporting Android mobile devices	17
3.1 Employing a freely available, open-source, and standards-compliant real-time CORBA implementation in Android	18
3.1.1 Comparison of existing CORBA implementations	19
3.1.2 Porting ACE/TAO to Android.....	20
3.2 FAmINE4Android: Design and Implementation	21
3.2.1 Initialization and Cleanup	21
3.2.2 Service implementation.....	22
3.2.3 Service resolve and usage.....	25
3.3 Addressing interoperability issues between Android Java and JNI.....	27
3.3.1 Calling native functions from Java	28
3.3.2 Calling Java functions from native code	29
3.4 Facilitating interoperability between Java and native code	30
3.4.1 Reification and Reflection in C++	30
3.4.2 Data exchange	32

4	Case study: Museum Guide Application using Android mobile devices.....	41
4.1	Tracking persons using a network of RGB-D cameras.....	41
4.2	Museum Guide application for Android mobile devices.....	43
4.2.1	Design and Usage scenario	44
4.2.2	Implementation details.....	46
4.3	Testing.....	46
5	Conclusion and Future Work	47
5.1	Summary of Achievements	47
5.2	Future Work	47
6	References	49
	APPENDIX A: Step-by-step procedure of porting TAO to Android architecture.....	53
	APPENDIX B: Programming Tutorial	57
	B.1. Generating client/server stubs from .idl file.....	57
	B.2. Service type declaration	57
	B.3. Set up Programming Environment	57
	B.4. Service implementation	63
	B.5. Using a service	67
	B.6. Sending events.....	68
	B.7. Receiving events	69
	B.8. Call Function	70

List of Figures

Figure 1: Communication mechanism	10
Figure 2: High-level architecture of <i>FAMINE4Android</i> middleware	18
Figure 3: Setup consisted of four RGB-D sensors	42
Figure 4: Multiple human tracking based on geometric and color information	42
Figure 5: Main menu.....	44
Figure 6: Using the mobile application to browse the digital exhibits, based on tracking technology.	45
Figure 7: <i>Museum Guide</i> features	45
Figure 8: SDK tools	58
Figure 9: Customize C++ Support.....	59
Figure 10: Build.Gradle configuration.....	59
Figure 11: CMakefile configuration	60
Figure 12: AndroidManifest configuration	60
Figure 13: Assets folder configuration (left), Import AAR package (right), Import Module Dependency (bottom).....	62
Figure 14: Import Library Dependency.....	63
Figure 15: <i>Chaos</i> generated files Imported to Android Studio project	65
Figure 16: Service type declaration for <i>Chaos</i> service.....	65
Figure 17: Configuration of the CMakeList.txt file for the <i>Chaos</i> service.....	65
Figure 18: Implementation of <i>ChaosImplementation</i> Java class	66
Figure 19: Register service <i>Chaos</i> in Java	66
Figure 20: <i>Chaos</i> generated files Imported to Android Studio project (resolve)	67
Figure 21: Service type declaration for <i>Chaos</i> service (resolve).....	67
Figure 22: Configuration of <i>Chaos</i> service in the CMakeList.txt file.....	67
Figure 23: Example of service usage/resolve	68
Figure 24: Send Event	69
Figure 25: <i>earthquake</i> event handler	69
Figure 26: Invocation example of the <i>earthquake</i> method.....	70

List of Tables

Table 1: Communication technologies requirements	8
Table 2: Mapping between the Java types and the native signatures	29
Table 3 : Mapping of Java types to natives types	32
Table 4 : Mapping of Java arrays to native arrays and CORBA arrays	33

List of CodeBlocks

CodeBlock 1: Using <i>toJson</i> method	13
CodeBlock 2 Using <i>fromJson</i> method	13
CodeBlock 3: Registration scope for the “TestStruct” data structure	14
CodeBlock 4: Invocation of unknown type object	14
CodeBlock 5: Construction of unknown type variables	15
CodeBlock 6 : Setter and getter properties in unknown type object	15
CodeBlock 7: Iteration of unknown type objects	15
CodeBlock 8: Initialization of the ORB	21
CodeBlock 9: Service type declaration example	23
CodeBlock 10 : Event dispatching example	24
CodeBlock 11: Process of controlling the event’s elements type	24
CodeBlock 12: Instructions to FamineManager in case of service usage	25
CodeBlock 13: An example of event arguments’ data conversion among various CORBA types	26
CodeBlock 14: Remote procedure call example	27
CodeBlock 15: Load library invocation	28
CodeBlock 16: Auto generated native file	28
CodeBlock 17: Call a Java method from native code using FindClass method	29
CodeBlock 18: Call a Java method from native code using GetObjectClass method	29
CodeBlock 19: Example::Echo; service definition in IDL	31
CodeBlock 20: Injected code of the Example::Echo service	31
CodeBlock 21: Example of primitive data type checking and equivalent conversion to native code ...	32
CodeBlock 22: Example of primitive data type checking and equivalent conversion to native code ...	33
CodeBlock 23 : Conversion of CORBA type variable to jobject type variable	33
CodeBlock 24 : Conversion of CORBA type array to jobject type variable	34
CodeBlock 25: Call <i>toJson()</i> method and <i>fromJson()</i> in Java	35
CodeBlock 26: Implementation of the <i>convert_from_json</i> in the case of the AdvancedMessage example	36
CodeBlock 27 : Dynamic complex data type JSON deserialization	36
CodeBlock 28: Implementation of the <i>convert_from_json</i> in the case of the AdvancedMessageSeq example	37
CodeBlock 29: Type checking logic of the RPC returned value	38
CodeBlock 30: Implementation of the <i>convert_to_json</i> in the case of the AdvancedMessage example	38
CodeBlock 31: Automatically generated injected code in the case of custom sequence of complex data types	39

CodeBlock 32: “Tracking persons using a network of RGB-D cameras” service definition in IDL	43
CodeBlock 33: Handling incoming event in Java	46
CodeBlock 34: ServiceTypes scope	57
CodeBlock 35: Service Interface “Chaos”	64

1 Introduction

Over the past few years, a large number of advances in computing and communication technologies have made it possible for computing to occur anywhere. The increasing availability and use of wireless mobile devices entails opportunities for new types of distributed applications. This work aims at empowering mobile devices in distributed service-oriented environments. The next sections briefly present the role of distributed services within an Ambient Intelligence environment followed by the objectives of middleware technologies for distributed services. The chapter concludes with a discussion on the contribution of the proposed work.

1.1 Distributed services in Ambient Intelligence

Distributed systems are collections of computers that act, work, and appear as a single coherent system. According to the model of Distributed Computing, software services components located on networked computers communicate and coordinate their actions by passing messages [7]. The components interact with each other in order to achieve a common goal. Examples of distributed systems vary from SOA-based systems [31] to massively multiplayer online games [29] to peer-to-peer applications [30].

One of the most typical properties that a distributed system includes is failure tolerance in individual components [17]. Another, according to [41], is that the structure of the system (network topology, network latency, number of computers) is not known in advance. In this context, the system may consist of different kinds of computers and network links, and it may change during the execution of a distributed program.

Ambient Intelligence (Aml) is an emerging research field that aims to make many of the everyday activities of people easier and more efficient [5]. This paradigm gives rise to opportunities for novel, more efficient interactions with computing systems. At a technical level, the vision of Ambient Intelligence is realized by the seamless confluence of diverse computing platforms. In an Aml environment, where interactions are realized by the confluence of different interconnected computing systems, the organization of the overall system architecture in a well-defined set of distributed software entities is crucial [16]. An Aml infrastructure consists of a collection of interconnected distributed services, i.e., a collection of software entities that run on different machines, able to communicate with each other in order to provide to the infrastructure all the required functionality for sensing, drawing inferences, and responding to the needs of its users.

1.2 Middleware technologies in the context of Ambient Intelligence

In the context of Aml, a distributed technology enables: a) the flexible and dynamic extension of the overall system with novel functionality, b) system scalability, by sharing computation demands among different computers, c) enhanced robustness, by isolating potential failures of individual software entities, and d) unambiguous and straightforward modularization of the system's architecture.

The term *Middleware* in the context of distributed computing systems refers to a set of programming libraries and programs (services) that constitute an indivisible platform, which offers a comprehensible abstraction over the complexities and potential heterogeneity of the target problem domain [0]. In an inherently distributed environment such as Aml, the communication middleware should abstract over the intricacies of the underlying communication technologies, machine architectures and operating systems. Moreover, it should hide the distribution of the different parts that comprise the system and

enable programs written in different programming languages to communicate seamlessly. To this end, a software framework (middleware) is essential to enable heterogeneous computing systems to interoperate. Middleware technologies are used in the context of distributed computing systems, in order to facilitate distributed object communication. A middleware technology example, named *FAMINE* (FORTH's Aml Network Environment), is presented in [16]. *FAMINE* provides the necessary functionality for the intercommunication and interoperability of heterogeneous distributed services hosted in Aml environments. It encapsulates mechanisms for service discovery, event driven communication and remote procedure calls. To this end, *FAMINE* provides a common set of APIs targeting a variety of heterogeneous platforms and different programming languages.

1.3 Contribution

Although *FAMINE* middleware facilitates the interoperability of heterogeneous distributed services hosted in diverse platforms, it does not provide support for mobile devices. This work aims at empowering mobile devices in distributed service-oriented environments. To this end, a *FAMINE* middleware extension, called *FAMINE4Android*, is proposed aiming to facilitate the development process of distributed Android mobile applications/services. *FAMINE4Android* provides the required mechanisms and tools in order to support remote communication with distributed objects running on both ordinary PCs and Android mobile devices.

The *FAMINE4Android* library builds upon the *FAMINE* middleware, which caters for the creation of distributed services enabling the exposure of software and hardware resources in Aml environments. The proposed library provides mechanisms for service discovery, event driven communication and remote procedure calls through a seamless and intuitive Java API. *FAMINE4Android* allows Android developers are able to develop applications enabled with distributed computing capabilities in an effortless manner.

2 Background and Related Work

This work aims to build a middleware extension library to facilitate the development process of interconnected distributed objects for Android mobile devices in the context of Ambient Intelligence. This section establishes the foundations of this research work by identifying the state of the art in the targeted application domains.

2.1 Ambient Intelligence Environments

The term Ambient Intelligence (Aml) refers to electronic environments that are sensitive and responsive to the presence of people [40]. According to ISTAG [10], the concept of Aml provides a vision of the Information Society where the emphasis is on greater user-friendliness, more efficient services support, user-empowerment, and support for human interactions. People are surrounded by intelligent intuitive interfaces that are embedded in all kinds of objects and any environment that is capable of recognizing and responding to the presence of different individuals in a seamless, unobtrusive and often invisible way.

Ambient intelligence deals with a new world of ubiquitous computing devices where physical environments interact intelligently with people. These environments should be aware of people's needs, customizing requirements and forecasting behaviors. Aml environments can be diverse, such as homes, offices, meeting rooms, schools, hospitals, museums, control centers, vehicles, tourist attractions, stores, sports facilities, and public spaces. Artificial intelligence research aims to include more intelligence in Aml environments, allowing better support to humans and access to the essential knowledge for making better decisions when interacting with these environments [10]. Furthermore, Aml environments enclose computing interfaces and technologies embedded to the context-awareness requirement on data management strategies and solutions. Aml implies a seamless environment of computing, advanced networking technology and modularized interfaces [36].

The vision of Aml assumes a shift in computing from desktop computers to a multiplicity of computing devices in our everyday lives, whereby computing moves to the background and intelligent ambient interfaces to the foreground. The elaboration of new interaction techniques is becoming the most prominent key to a more natural and intuitive interaction with everyday things [5]. Natural interaction between people and technology can be defined in terms of experience: people naturally communicate through gestures, expressions, movements. To this end, people should be able to interact with technology as they are used to interact with the real world in everyday life [5]. Additionally, Aml systems must be sensitive, responsive, and adaptive to the presence of people.

According to the Institute for the Future [4], emerging technologies are transforming everything that constitutes our notion of "reality"; our ability to sense our surroundings, our capacity to reason, and our perception of the world. In the context of Ambient Intelligence, several challenges emerge in the contributing domains of ubiquitous computing, and Human Computer Interaction (HCI), where many network devices are integrated into the environment. The environment system can judge the situation from the device input, and backend devices share information with the environment system to support users in physical space.

Ubiquitous Computing has as its goal to enhance computer use not only by making many computers available throughout the physical environment, but also by making them effectively invisible to the user [56]. The idea of ubiquitous computing was first thought by Mark Weiser in 1998 at the Computer

Science Lab at Xerox PARC¹. He envisioned computers embedded in walls, tabletops, and everyday objects. A person might interact with hundreds of computers at a time, each invisibly embedded in the environment and wirelessly communicating with each other [55]. A number of researchers around the world are now working in the ubiquitous computing framework. Their work affects all areas of computer science, including hardware components (e.g., chips), network protocols, interaction substrates (e.g., software for screens and pens), applications, privacy, and computational methods. Some researchers say that ubiquitous computing is the Third Wave of Computing [44]. The First Wave was “many people, one computer”, and the Second Wave, the PC, is the era of “one person, one computer”. The Third Wave will be the era of “many computers per person”.

According to [11], ubiquitous or pervasive computing assumes a large number of ‘invisible’ small computers embedded into the environment and interacting with mobile users. Users will experience the world through devices to wear (e.g., medical monitoring systems), to carry (e.g., personal communicators that integrate mobile phones and PDAs), devices that are implanted in the vehicles or the public spaces (e.g., car and public space information systems), and devices integrated in the architectural environment (e.g., interactive walls and furniture). This heterogeneous collection of devices will interact with intelligent sensors and embedded actuators in homes, offices, public spaces and transportation systems, in order to form a mobile ubiquitous computing environment, which aids normal activities related to work, education, entertainment and healthcare. The environment will also provide access to wired backbone computing resources, connected to the Internet.

At a technical level, the confluence of different computing platforms is crucial to accomplish the vision of Ambient Intelligence. In order to achieve this, a software framework (middleware) is essential to enable heterogeneous computing systems to interoperate, supporting human interaction in physical environments in an intuitive and ubiquitous way [16].

2.2 Distributed Service Technologies

Aml environments provide customized interaction through context-aware technologies in order to perceive stimuli from both users and environments [50]. Thus, self-adaptable technologies are important in order to provide an adequate interaction to users and to develop dynamic distributed systems in the context of Aml. This requires the contribution of software entities that are flexible and scalable. According to the model of Distributed Computing, software services components run in different computers and data are shared among the network in order to improve the efficiency and the performance, such as to avoid crashes and enhance the response time of the applications.

2.2.1 Middleware distributed technologies in the context of Aml

Middleware contributes the development of heterogeneous networking environments, supporting programming libraries and services in any programming language. Middleware technologies are used in the context of distributed computing systems [16]. In communication middleware platforms, applications of different programming languages are structured in objects that interact in different programming systems, via transparent method invocation support, reflecting the “request/response” communication protocol. In Aml environments, middleware enables distributed objects to communicate remotely. Synchronous communication is essential for the interaction between services hosted in Aml environments, as it allows the direct transmission of service calls. However, synchronous communication is inefficient to support the modeling of all interactions applied in an Aml environment.

¹ <http://www.parc.xerox.com>

For this reason, asynchronous, event-based communication is required in order to enable Aml services to notify interested parties about changes in the internal state or to communicate the occurrence of an external (expected or unexpected) event. As stated in [16], one of the most important and extensively researched properties of distributed systems is fault tolerance. Fault tolerance refers to the property that enables an Aml infrastructure to continue to function properly even in the event of failures. In the context of an Aml communication middleware, such as the proposed one, the following set of requirements should be met:

- failures isolation
- elimination of single points of failure within the core middleware infrastructure
- restart failing services before the clients that use those services are affected
- provision of mechanisms for notifying higher level entities about the irreparable failure of a specific service.

Additionally, another important requirement of an Aml middleware is security. Apparently, security, in order to be effective, should be considered throughout all the layers of an Aml infrastructure. In this context, however, the security is the ability of the middleware to prevent malicious code from eavesdropping the data exchanged through the network channels that enable services to communicate with each other.

As presented in chapter 0, the proposed middleware extension library builds upon the FORTH's *FAMINE* middleware. As a result, all the aforementioned requirements are satisfied thanks to *FAMINE*, which caters for the creation of distributed services enabling the exposure of software and hardware resources in Aml environments.

2.2.2 Middleware approaches

In this section, existing communication technologies are briefly presented as basic communication tools for an Aml middleware.

2.2.2.1 CORBA

According to [51], the *Common Object Request Broker Architecture* (CORBA) was defined by the *Object Management Group* (OMG), a non-profit organization that promotes the use of object-oriented technologies. Many people refer to CORBA as a middleware or integration software, because CORBA is often used to standalone applications communicating with each other. The phrase common architecture means a technical standard for *Object Request Broker* (ORB), a mechanism for invoking operations on an object in remote process. In particular, it allows applications to talk to each other even if the applications are running on the same or a different computer, different operating systems or different CPU types, implemented with different programming languages. In addition, CORBA is an object-oriented distributed middleware; this means that client does not make calls to a server process. Instead, a CORBA client makes calls to objects. CORBA has many strong points. It allows applications in different programming languages to communicate each other, even if they are running on different operating systems, on different computers, and on different CPU types.

The advantages of the CORBA architecture are numerous. First, in order to define public Application Programming Interfaces (APIs), CORBA makes use of the Interface Definition Language (IDL) [28], which defines a mapping between IDL definitions and constructs of the target programming language. This mapping enables the invocation of attributes and operations between distributed services. In addition, CORBA supports synchronous and asynchronous communication between services, using the standard Notification Service. One more advantage is that CORBA takes account of fault tolerance, obtaining

references to services through the standard Implementation Repository (ImR) service in order to allow infrastructure functions properly, even if a fault has occurred. Moreover, CORBA provides the needed security disincline malicious code, using encrypted communication channels. Even if CORBA is considered difficult to use, there are many open source implementations for each target programming language.

2.2.2.2 ICE

The Internet Communication Engine technology (ICE) defined by the ZeroC company relies on the CORBA architecture [18, 57]. ICE provides an extra functionality, improving the unnecessary complexity of CORBA, purveying protocols to reduce network bandwidth and creating robust security systems. ICE provides many useful standard services, such as IcePatch and Glacier. IcePatch updates software around the distributed infrastructure and sends notifications when services communicate with each other. Whereas Glacier is responsible to provide enhanced security and firewall protection. Moreover, ICE makes use of Slice in order to define the service public API. Slice allows to programmers to define the state of ICE objects, so that they can be stored and loaded automatically. However, an important restriction is that ICE applications and services have to be implemented under the ICE General Public License (GPL) and each extra feature has a corresponding fee.

2.2.2.3 Web Services

Web Services is a new perspective of modern distributed systems. Using an XML-based protocol, Web Services allow applications to publish data across the Web. This middleware uses the Simple Object Access Protocol (SOAP), which defines: a) the format of transferred messages between services, b) the rules for the data format transferred via messages and c) a set of conventions in order to achieve remote procedure calls [48]. While universal firewall traversal is very important for geographically distributed services, it is not essential in the context of an Aml environment where the majority of the deployed services are restricted within a Local Area Network (LAN). Furthermore, Web Services approach supports synchronous and asynchronous communication between services. However, the asynchronous request-response communication works only with HTTP protocol and not with the HTTPS. Moreover, the programming of Web Service abstractions requests libraries and tools, which are not available for many programming languages. As a result, the Web Service approach is an insufficient middleware to support an Aml infrastructure, due to the absence of high level programming idioms and communication guarantees.

2.2.2.4 Thrift

Thrift technology is a software library and set of code-generation tools developed at Facebook to expedite development and implementation of efficient and scalable backend services [49]. Additionally, it allows developers to define datatypes and service interfaces in a single language-neutral file and generate all the necessary code to build RPC clients and servers. Thrift is a communication platform that imports distributed services in many different programming languages. Thrift, like CORBA, uses IDL, which enables the invocation of attributes and operations between distributed services, separating service definition from its actual implementation. Even if this approach is very well structured and efficient, Thrift does not import Naming and Notification Service, something that makes the asynchronous communication impossible.

2.2.2.5 Etch

Etch technology is defined by Cisco and is a framework for building network services [6, 12]. Etch is a cross-platform, language- and transport-independent framework that builds and consumes efficient RPC network services in a resource limited and heterogeneous environment. Etch uses a Network

Service Description Language (NSDL) which, similarly to IDL, separates service definition from the actual implementation. Etch toolset includes a network service description language, a compiler for code generation, and binding libraries for a variety of programming languages, such as C, C++, C# and Java. Etch supports synchronous and asynchronous communication, providing a Naming Service and a Router Service for fault tolerance that supports service replication. However, this approach is not able neither to use a service object as parameter of a method nor as return value. Etch provides the requested functionality for an Aml environment, although it is incomplete.

2.2.2.6 ROS

The Robot Operating System (ROS) is a flexible framework for writing robot software, which runs only on Debian and Ubuntu [45]. ROS makes use of IDL and supports a system tool, which manages the details of distributed synchronous and asynchronous communication between the interacting services. In addition, this approach provides an extensive set of configuring tools and libraries. Although ROS supports asynchronous messaging, it does not support synchronous “request/response” interaction between processes. In general, ROS provides many of the requested functionality of an Aml middleware, but it is very restrictive given that it targets only a restricted number of programming languages, such as C++, Python, LISP and Javascript.

2.2.2.7 RIO

Rio is a dynamic framework able to develop, deploy and manage distributed systems composed of Java services [43]. Rio provides an infrastructure to dynamically instantiate, monitor and manage services, which provide context on service requirements and dependency parameters. In addition, RIO purveys a policy approach based on fault detection and recovery, scalability and dynamic deployment. Key to the architecture are a set of dynamic capabilities and reliance on policy-based mechanisms. RIO turns a network of computing resources into a dynamic service, providing a policy-based approach for fault detection and recovery, scalability and dynamic deployment. To this end, RIO is restrictive enough due to a limited number of running platforms that it supports.

2.2.2.8 Crossbar.io

Crossbar.io is an open source application-networking platform for distributed and micro service applications [8], implementing the open Web Application Messaging Protocol (WAMP), which offers both Publish and Subscribe (PubSub) and Remote Procedure Calls (RPC) [54]. In order to facilitate RPC functionalities in Crossbar.io, a client exposes some code and another asks for its execution. WAMP clients are already implemented for almost twelve different programming languages, covering the necessary communication patterns and functionality within the context of the application. Moreover, Crossbar.io provides tools in order to handle a wide set of standard aspects of distributed applications. In addition, Crossbar.io can be used from any Web application framework that is able to serve (outgoing) HTTP/POST requests, providing many of the requested functionalities for distributed Aml environments. However, the implementation of WAMP does not provide full remote object passing like CORBA, as well as complex data types.

2.2.2.9 Related technologies approaches

Express is a minimal and flexible Node.js web application framework designed to run in Node.js platform [13]. Koa.js is a JavaScript web application framework. It is an evolution on the Express.js framework supporting object-oriented programming [35]. The EMISS (Energy Monitoring via the Internet and Sensors for Sustainable living) implemented with Java, illustrates the idea to provide a modular and simple platform for the design and deployment of a wireless sensor network [9]. OSGi framework implements services in Java [39]. The OSGi technology facilitates the componentization of

software modules and applications and assures remote management and interoperability of applications and services over a broad variety of devices. The .NET Web Services framework and tools for implementing services provide a standard means of interoperating between different software applications, running on various platforms and/or frameworks. In addition, Hydra framework uses a Web Services-based approach, where network technologies create and consume services [52]. The Hydra middleware allows developers to incorporate heterogeneous physical devices into their applications by offering easy-to-use web service interfaces. These efforts are yet very restrictive, since they support only a narrow range of programming languages, such as Java, C++, .NET-based, Python, JavaScript and Node.js.

2.2.2.10 Discussion

Presenting the basic requirements of an Aml middleware, among the aforementioned communication technologies [Table 1], CORBA and ICE Object-oriented middleware approaches are more effective in providing implementation for Aml environments. Both provide specifications and features, such as heterogeneity, which supports multiple programming languages and computing platforms. Additionally, they import synchronous request/response and asynchronous communication, a crucial factor for the development of an Aml environment. Furthermore, both the aforementioned approaches have the ability to use encrypted communication channels, providing essential security to all the entities of an Aml environment. Finally yet importantly, they provide ease of use via the intuitive usage of each target language. These factors make CORBA and ICE technologies independent of target domain, with the exception that ICE applications and services have to be implemented under the ICE's General Public License (GPL) and the extra features are being offered for a fee.

Table 1: Communication technologies requirements

Technology	Multiple Language Support	Synchronous & Asynchronous Communication	Handling of Failures	Secure communication	Ease of Use for target language
CORBA	✓	✓	✓	✓	✓
ICE	✓	✓	✓	✓	✓
Web Services	✓	(✓)	(✓)	✓	✓
Thrift	✓	(✓)	(✓)	✓	✓
Etch	(✓)	(✓)	(✓)	✓	✓
ROS	(✓)	✓	✓	✓	✓
RIO	(✓)	✓	✓	✓	✓
Crossbar.io	✓	(✓)	✓	✓	✓

2.2.3 FAmINE: A middleware library for Ami environments

FAmINE is a software architecture for Ambient Intelligence environments that enables the implementation and deployment of software abstractions [15]. *FAmINE* contributes to the creation of

distributed services that enable the exposure of the software and hardware resources available in an Aml environment. Using the *FAMINE* middleware, the deployment of high-level object-oriented abstractions for programming and using services has been achieved in the context of many Aml development projects. *FAMINE* provides architectural abstractions, interaction capabilities and composability methods. In details, the *FAMINE* middleware supports many different programming languages in order to constitute a viable platform for developing Ambient Intelligence services. In addition, a set of core services is provided for discovery and deployment functionality of the services infrastructure. *FAMINE*'s design is based on high-quality implementation supporting both synchronous and asynchronous communication strategies.

The *FAMINE* middleware builds upon a variety of CORBA libraries according to characteristics of the implementation environment, such as supported platform and programming language. The following subsections describe the service-oriented features that *FAMINE* provides.

2.2.3.1 Service Implementation

The syntax of a service description is based on the Interface Definition Language (IDL), which defines an Application Programming Interface (API), in a way that is independent of any particular programming language. The type of a CORBA object is called an interface, which is similar in concept to a C++ class or a Java interface. The implementation of service description is based on the definition of the methods, events and primitive types. In addition, complex types are defined using *structures*, *enumerations*, *unions* and *sequences*. Primitive and complex types are used as arguments or returned values on methods and events. In addition, the description of service implementation is validated by CORBA IDL, where the service signature is defined from the name of the module and the name of the interface, separated by the scoping operation in IDL “::”. This construct has a similar purpose to a namespace in C++ or to a package in Java. The events are defined as void methods; whose name starts with the prefix *Event_* followed by the event name. The main functionality of the service is provided by the actual implementation of the methods and events. When the service is fully implemented, it can be exposed by instantiating an object that is able to accept and make remote procedure calls.

2.2.3.2 Service discovery and invocation

The CORBA COS (Common Object Services) Naming Service provides a tree-like directory for object references implemented on top of the COS Naming Service specification [1]. A name-to-object association is called a name binding. A name binding is always defined according to the naming context. A naming context is an object that contains a set of name bindings in which each name is unique. Different names can be bound to an object in the same or different context simultaneously. Before the client and server start running, they should both agree on which root-naming context to use. At first, the server invokes the bind or rebind method in order to associate a name with an object reference. Secondly, the Naming Service adds this object reference binding to its namespace database. Thirdly, a client application invokes the resolve method to obtain an object reference with the given name. At the end, the client uses an object reference to invoke methods on the target object. The invocation of methods can be achieved by obtaining a reference to an exposed service. As illustrated in Figure 1, a client makes a remote invocation upon a proxy object. When the client application invokes an operation on a proxy object, the proxy object uses an inter-process communication mechanism to transmit the request to the “real” object as it is shown in the next figure. Then the proxy object waits to receive the reply and passes back this reply to the application-level code in the client [1].

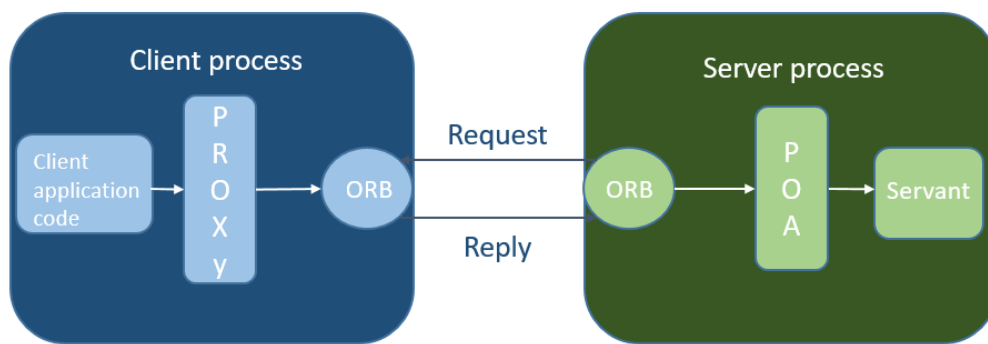


Figure 1: Communication mechanism

The Object Request Broker (ORB) in the client process waits to read a reply message from the server process and then returns the reply buffer back to the proxy object. On the server side, a thread inside the ORB runs in an event loop, waiting for incoming requests. When a request arrives, the ORB reads and dispatches the request to the target servant. When the operation in the servant returns, via the POA to the server-side ORB, it transmits the reply message across the network to the client process. Consequently, through this servant the user program is able to handle the income events by defining a method with the same name of the event.

2.2.3.3 Repository server

Regarding the repository server, CORBA architecture uses the term of Implementation Repository Server (IMR) to describe the repository that is responsible for the storage details of CORBA server applications. The repository server keeps a configuration database that specifies on which machine a specific service should run. The IMR is responsible to redirect the invocation call to the service's actual network location. In case the requested service is not running, due to network failures or in case the server has not start running yet, the resolution process starts again from the beginning. The resolution process supports the resolving of the client service by obtaining a reference to IMR. In case the IMR execution is not responding, the resolution process must be restarted. That means long delays to the client's invocations. The repository server has to import another server in order to be able to either start or restart a service. That server, called Service Activator, aims to configure which machine has the specific service to run and check if the executable program of the specified service is accessible from the local machine. The Service Activator is responsible to check if the executable is available, either at a central file repository as part of the middleware infrastructure or on the local machine. Additionally, it checks if there is an updated version at the central file repository and, in case there is, it downloads and replaces the old version in order the updated version of the service to be deployed.

2.2.3.4 Context and Zones

Concerning the network IP address, on which the IMR is executed, the term context is used to define a runtime, dynamic property of a service that describes and identifies its current instantiation within an Ambient Intelligence environment. Regarding the term zone, it can be seen as an isolated middleware infrastructure where services deployed in a specific zone cannot access and affect the services deployed in another zone. Given that a service can be resolved regardless of its actual network location many times, the term context is necessary for indicating the availability of alternative implementations, by providing access to different sets of resources, or even for implementing application-level redundancy.

2.2.3.5 Asynchronous events

Regarding the *FAMINE4Android* asynchronous communication, the messages that are transferred among the services can be produced and dispatched by any service. Through this mechanism, the producer of an asynchronous message directly propagates it to all its consumers. On the other side, any consumer that had previously declared interest in that specific event is notified asynchronously while the event is delivered to the “event handler” method.

2.3 Major building blocks

In this section, the necessary components contributed for the implementation of the proposed work, are presented. More details are presented in section 3.1.

2.3.1 Adaptive Communication Environment

The presented middleware library is implemented on top of the Common Request Broker Architecture (CORBA) [47], using the Adaptive Communication Environment (ACE), an open-source object-oriented framework, which implements many core patterns for concurrent communication software. Furthermore, ACE enhances portability, meaning that ACE components make it easy to write networked applications on one OS platform and then port them to different OS platforms, providing reusable components and patterns. Additionally, ACE components are designed in such a way to provide flexibility, extensibility, reusability, and modularity of communication software. An additional benefit of ACE components is the support of Quality of Service (QoS), which provides high performance for bandwidth-intensive applications and recurrence for real-time applications. Furthermore, ACE provides a rich set of reusable C++ wrapper facades and framework components that perform common communication software tasks across a range of operating system platforms. TAO simplifies the development of distributed applications by automating and encapsulating object location, connection and memory management, parameter (de)marshaling, event and request de-multiplexing, error handling and fault tolerance, object and server activation, concurrency and security. These capabilities allow applications to interoperate across networks without hard-coding dependencies on their location, programming language, operating system platform, communication protocols and hardware characteristics.

2.3.2 Interface Definition Language

One of the key factors of CORBA is the language independence. Language independence is achieved using a specification meta-language that defines the interfaces of an object. The IDL is used for the description of a service interface. The chosen types, methods and events are meant to highlight many of the service’s capabilities for exchanging messages through the abstraction of method invocations.

2.3.3 TAO IDL compiler

An IDL compiler translates IDL service definitions into similar definitions for the targeted programming language. For each IDL interface, the *tao_idl4Android* compiler generates both *stub* code, a dummy implementation, and *skeleton* code, which describes the server-side code for reading incoming requests and dispatching them to application-level objects. In a distributed system like CORBA, remote calls are implemented by the client making a local call upon a *stub* procedure/object. The *stub* uses an inter-process communication mechanism to transmit the request to a server process and receive back the reply. A CORBA *stub* is a client-side object that acts on behalf of the “real” object in a server process. The so-called *skeleton* provides supporting infrastructure that is required to implement server

applications. The *The ACE ORB (TAO)* provides a compiler which is responsible for the generation of the *stub* and *skeleton* code.

2.3.4 Android Studio dependencies

This section presents the main components that are provided by Android Studio and are contributed to the implementation of the presented work.

2.3.4.1 Java Native Interface

The Java Native Interface (JNI) is a programming framework that enables Java code running in a Java Virtual Machine (JVM) to call and be called by native applications (programs specific to a hardware and operating system platform) and libraries written in other languages such as C or C++ [33]. JNI has the role of moderator, providing the mapping between variables and methods. In addition, JNI supports the loading code from dynamic shared libraries efficiently and this factor is crucial for the implementation of the presented middleware library. The contribution of JNI in the presented middleware implementation is described in sections 3.3.1 and 3.3.2.

2.3.4.2 Native Development Kit

The Native Development Kit (NDK) is a set of tools which allow the use of C++ programming language in Android Studio [38]. NDK provides platform libraries in order to manage native code. In addition, in the last versions of Android Studio, CMake is imported. CMake is an external build tool that works alongside to build native libraries and Java Native Interface (JNI), which is the interface via which the Java and C++ components collaborates. Finally yet importantly, the NDK supports the use of prebuilt libraries, both static and shared. This feature of the NDK assists to the functionality of the presented middleware library as described in the following sections.

2.3.5 Light-weight data exchange formats

According to [25], JSON is a completely language independent text format that uses conventions familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Python, etc. These properties make JSON an ideal data-interchange language, capable to encode objects in string format; this process is called *serialization*. JSON structure is based on the collection of name/value pairs. In detail, as JSON representation components are the following:

- *Array*: square bracket ("[") represents a JSON array
- *Objects*: curly bracket ("{ ") represents a JSON object
- *Key*: a JSON object contains a key that is just a string. Pairs of key/value make up a JSON object
- *Value*: each key has a value that could be string, integer or double e.t.c

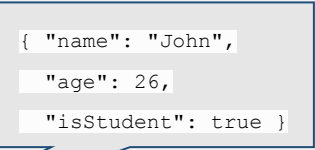
Therefore, an object begins with "{" (left brace) and ends with "}" (right brace), whereas each name is followed by ":" (colon) and the name/value pairs are separated by "," (comma).

For the purpose of the presented work, a Java library that can convert Java Objects into their JSON representation and similarly, a JSON string to an equivalent Java object, is used. This library is called *Gson* and is able to collaborate with arbitrary Java objects, providing several built in serializers and deserializers. A serializer allows the conversion of a JSON string to corresponding Java type while deserializers allows the conversion from Java object to a JSON representation. *Gson* library provides two main methods which are able to convert Java objects to JSON and vice-versa, these methods are

called *toJson()* and *fromJson()*. As depicted in CodeBlock 1, the *toJson()* method takes as argument an instance of a Java object and returns its JSON representation.

```
public class UserProfile
{
    String name;
    int age;
    boolean isStudent;
}

UserProfile userObject = new UserProfile( "John", 26, true);
Gson gson = new Gson();
String userJson = gson.toJson(userObject);
```



CodeBlock 1: Using *toJson* method

Similarly, in CodeBlock 2 is illustrated the call of *fromJson()* method that takes as first argument a JSON string and as second argument the expected Java object.

```
Gson gson = new Gson();
String userJson = " { 'name':'John', 'age':26, 'isStudent':true } " ;
UserProfile userObject = gson.fromJson(userJson, UserProfile.class);
```

CodeBlock 2 Using *fromJson* method

According to the aforementioned data exchange approach, a corresponding data exchange data library in C++ is *RapidJson* [21]. *RapidJson* library can convert C++ Objects into their JSON representation and similarly, a JSON string can convert to an equivalent C++ object. *RapidJson* is based on the Simple API for XML (SAX) style format. A *RapidJson* object is a collection of key-value pairs and it must be a string value. Moreover, the main components of the *RapidJson* library are the *Reader* method, which parses a JSON representation from a stream and the *Writer* method, which converts JSON into the corresponding object. In detail, as *RapidJson* representation components are the following:

- *StartArray*: start a JSON array object using the “[”
- *EndArray*: end a JSON array object using the “]”.
- *StartObject*: start a JSON array object using the “{”
- *EndObject*: end a JSON array object using the “}”

2.3.6 Run Time Type Reflection library

Even if many programming languages (i.e., Java) provide built-in reflection mechanisms, C++ does not. Regarding the requirements of the present work, the application of a generic code approach was required, which could operate with unknown type variables and various instances objects. The proposed work is supported by the *Run Time Type Reflection (RTTR)* library, which is provided by readily available custom and open source implementations that aims to employ Reflection features in C++. The RTTR library describes the ability of a computer program to modify an object at runtime, in an easy and intuitive way. The functionality of the Run Time Type Reflection (RTTR) library requires the

registration procedure of properties, methods, enumeration and constructors to the file where the C++ structure is declared [42]. In detail, the RTTR library provides the following advanced functionality:

- Registration of constructors, properties, methods and enumerations. The registration process is the entry point for the information reflection to the type system (see CodeBlock 3)
- Method invocation process premise that the number of arguments must be provided and the type itself must 100% match the type of the registered function. A method will be successfully invoked when the provided instance can be converted to the declared class type (see CodeBlock 4)
- Constructor invocation, which aims to obtain an instance of a unique type object. The construction of unknown type's variables is able by creating instances of the actual class type, wrapped inside a variant object (see CodeBlock 5)
- Set/Get property process. According to a property set process, it premises that the provided instance can be converted to the declared class type, whereas the get process returns the value of the property (see CodeBlock 6).
- Retrieve object type process. The objective of this process is the retrieve of an object type, in which the type information is hold in the Type class of the RTTR library (see CodeBlock 7).

```
#include <rttr/registration>
using namespace rttr;
struct TestStruct {
    TestStruct() {}
    boolean isValid() {};
    int id;
};

RTTR_REGISTRATION {
    registration::class_<TestStruct>("TestStruct")
        .constructor<>()
        .property("id", &TestStruct::id)
        .method("isValid", &TestStruct::isValid);
}
```

CodeBlock 3: Registration scope for the "TestStruct" data structure

```
TestStruct obj;

method meth = type::get(obj).get_method("isValid");
meth.invoke(obj);

variant var = type::get(obj).create();
meth.invoke(var);
```

CodeBlock 4: Invocation of unknown type object

```
type t = type::get_by_name("TestStruct");  
variant var = t.create();  
  
constructor ctor = t.get_constructor();  
var = ctor.invoke();  
std::cout << var.get_type().get_name();
```

CodeBlock 5: Construction of unknown type variables

```
TestStruct obj;  
property prop = type::get(obj).get_property("id");  
prop.set_value(obj, 3);  
  
variant var_prop = prop.get_value(obj);  
std::cout << var_prop.to_int(); // prints '3'
```

CodeBlock 6 : Setter and getter properties in unknown type object

```
type t = type::get<TestStruct>();  
  
for (auto& prop : t.get_properties())  
    std::cout << "name: " << prop.get_name() << std::endl;  
  
for (auto& meth : t.get_methods())  
    std::cout << "name: " << meth.get_name() << std::endl;
```

CodeBlock 7: Iteration of unknown type objects

3 FAmINE4Android: A FAmINE extension library supporting Android mobile devices

This chapter describes the implementation of the developed middleware library, called *FAmINE4Android*, aiming to facilitate the development process of distributed Android mobile applications. *FAmINE4Android* provides the required mechanism and tools in order to support remote communication with distributed objects running on both ordinary PCs and Android mobile devices. The *FAmINE4Android* library builds upon the ICS-FORTH *FAmINE* middleware, which caters for the creation of distributed services enabling the exposure of software and hardware resources in Aml environments. The ICS-FORTH *FAmINE* middleware provides a common set of APIs for different programming languages, Java, C++, .NET and Python, on a wide range of different types of devices. The core components of the *FAmINE4Android* middleware are directly derived from the source code of the C++ *FAmINE* middleware, ensuring maximum compatibility between the already developed modules and those that are developed for the requirements of the Android architecture. Given that the primary and most popular programming language for the development of Android applications is Java, but no suitable CORBA implementation in Java was readily available, the C++ ACE ORB (TAO) was selected as the optimal choice (see 3.1).

Figure 2 illustrates the high-level architecture of the *FAmINE4Android* middleware extension library. The main components contributing to the overall architecture's synthesis and functionality are the following:

- ICS-FORTH *FAmINE* middleware provides the creation of distributed services, enabling the exposure of software and hardware resources in Aml environments. ICS-FORTH *FAmINE* middleware builds on The ACE ORB (TAO), which is a CORBA middleware framework that allows clients to invoke operations on distributed objects
- Service Reification Manager implements a service type agnostic approach based on the Run Time Type Reflection (RTTR). Reification Manager is able to import generic code and operate with unknown type variables and various instance objects of different C++ classes, encapsulating the heterogeneity between Java and C++ *FAmINE* middleware (see 3.4.1)
- *FAmINE* Manager facilitates the service registration and resolve process. In detail, *FAmINE* Manager keeps in internal data structures information about available service types, JNI environment, etc. Furthermore, it facilitates type agnostic object instantiation in case of service registration (see 3.2.2.1)
- *FAmINE4Android* is the main contribution of the present work aiming to facilitate the development process of distributed Android mobile applications (3.2). *FAmINE4Android* builds on top of ICS-FORTH *FAmINE* middleware
- Data Exchange Controller implements a lightweight data exchange format approach based on the JSON framework in order to facilitate the object passing between Java side and the *FAmINE4Android* middleware (see 3.4.2)
- *FAmINE4Android* Controller encapsulates the functionality provided by the *FAmINE4Android* middleware and through a seamless to use Java API is distributed to the Java developers as a unified AAR library (APPENDIX B presents a complete programming tutorial).

The following sections will further discuss the implementation details and role of the aforementioned components within the proposed middleware extension library.

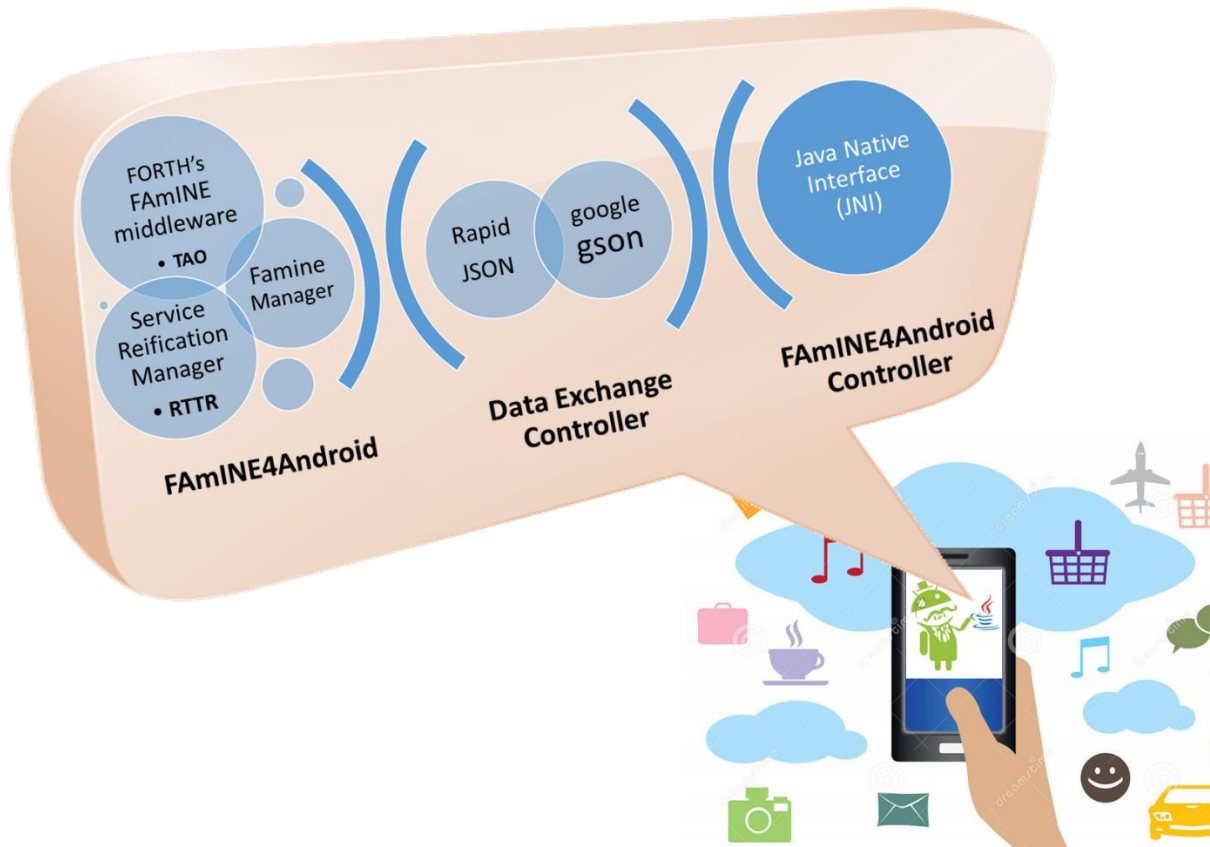


Figure 2: High-level architecture of *FAMINE4Android* middleware

3.1 Employing a freely available, open-source, and standards-compliant real-time CORBA implementation in Android

For the purposes of the present work, a freely available, open-source, and standards-compliant real-time CORBA implementation was preferred. Given that the primary and most popular programming language for the development of Android applications is Java, research was initially focused on Java based CORBA ORB implementations. However, to the best knowledge of the author, there is no readily available such solution. To this end, the implementation of the proposed extension middleware library relies on *TAO* [51]. As presented in 2.3.1, *TAO* [23] is a CORBA middleware framework that allows clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and hardware. In addition, it provides a rich set of reusable C++ framework components that perform common communication software tasks across a range of OS platforms. Some indicative powerful features that *TAO* provides are: a) event handler dispatching, b) heterogeneity, c) service initialization, and d) dynamic (re)configuration of distributed services. The *FAMINE* C++ middleware builds on top of the same CORBA implementation *TAO* (see 2.2.3). This offers the following advantages: a) robustness, b) re-use capability of *FAMINE*'s source code fragments, c) provision of identical functionality to developers as the *FAMINE* C++ middleware does, and d) reduction of the required efforts to maintain overtime.

In the present work, efforts have been focused on the porting process of the *TAO* to Android architecture using the Java Native Interface (JNI) and Android Native Development Kit (NDK) (see 3.1.2). In addition, focus has been given to the design and development of the *FAMINE4Android* extension library in order to encapsulate all the complex functionality provided by *TAO* from Java

developers. To this end, through a seamless use of the provided Java API, Android developers are able to create distributed and embedded real-time applications in an effortless manner.

3.1.1 Comparison of existing CORBA implementations

In this section, a comparison of existing CORBA implementations for the Android architecture is presented. The section begins with the requirements that have to be met during programming on distributed objects. According to [47], CORBA Object Request Brokers (ORBs) allow clients to invoke operations on distributed objects without concern for the following criteria:

- **Object location:** A CORBA object either can be collocated with the client or distributed on a remote server, without affecting its implementation or use,
- **Programming language:** The languages supported by CORBA include C, C++, Java, among others,
- **OS platform:** CORBA runs on many OS platforms, including Win32, UNIX and real-time embedded systems, such as Chorus,
- **Communication protocols:** The communication protocols that CORBA supports is TCP/IP, etc.
- **Hardware:** CORBA shields applications from side effects stemming from hardware diversity, such as different storage layouts and data type sizes/ranges,
- **Client:** A client is a role that obtains references to objects and invokes operations on them to perform application tasks,
- **Object:** Each object is identified by an object reference, which associates one or more paths through which a client can access an object on a server,
- **Servant:** A client never interacts with servants directly, but always through objects identified by object references,
- **ORB Core:** When a client invokes an operation on an object, the ORB Core is responsible for delivering the request to the object and returning a response to the client,
- **ORB Interface:** An ORB is an abstraction that can be implemented in various ways, e.g., one or more processes or a set of libraries. To decouple applications from implementation details, the CORBA specification defines an interface to an ORB,
- **OMG IDL Stubs and Skeletons:** IDL stubs and skeletons serve as a “glue” between the client and servants, respectively, and the ORB. Stubs implement the Proxy pattern and provide a strongly-typed, static invocation interface that marshals application parameters into a common message-level representation,
- **IDL Compiler:** An IDL compiler transforms IDL definitions into stubs and skeletons that are generated automatically,
- **Interface Repository:** The Interface Repository provides run-time information about IDL interfaces. Using this information, it is possible for a program to encounter an object whose interface was not known when the program was compiled, yet be able to determine what operations are valid on the object and make invocations on,
- **Implementation Repository:** The Implementation Repository contains information that allows an ORB to activate servers to process servants. Most of the information in the Implementation Repository is specific to an ORB or OS environment,

- **Dynamic Invocation Interface (DII):** The DII allows clients to generate requests at run-time, which is useful when an application has no compile-time knowledge of the interface it accesses,
- **Dynamic Skeleton Interface (DSI):** The DSI is the server's analogue to the client's DII. The DSI allows an ORB to deliver requests to servants that have no compile-time knowledge of the IDL interface they implementation.

Although there is a large variety of ORBs implementations, the vast majority of them are written in C++, such as *omniORB* [47], *TAO* [22], *MICO* [26] and *ORBit* [20]. To the best knowledge of the author, there are very limited ORBs implementations written in Java, such as *Jacorb* [24] or *ORBexpress for Android* [27]. Although *Jacorb* is a Java based implementation, it is not suitable for the Android architecture. This is due to the fact that *Jacorb* relies on Java JDK 1.6 whereas Android is essentially a Java SE with a different set of libraries. *ORBexpress for Android* provides an easy-to-use communication protocol for distributed systems. In addition, *ORBexpress for Android* provides features beyond CORBA, along with a development environment for building reliable distributed systems. However, the main drawback of *ORBexpress for Android* is the not open-source availability due to the license fee required. As a result, for the purposes of the present work, a C++ ORB implementation was preferred even though such an approach would imply a significant overhead due to the required porting process to the Android architecture. To the best knowledge of the author, the most prevalent C++ ORB implementations are *omniORB* and *ACE ORB (TAO)*. *omniORB* is an Object Request Broker (ORB) that implements the specification of the Common Object Request Broker Architecture, providing a robust high performance CORBA ORB for C++. It is freely available under the terms of the GNU Lesser General Public License (for the libraries) and GNU General Public License (for the tools). Moreover, *omniORB* has been tested and certified as a fast and standards compliant CORBA ORB. It also provides the IDL compiler, which creates C++ definitions into stubs and skeletons. However, *omniORB* is not (yet) a complete implementation of the CORBA core. *omniORB* does not have its own Interface Repository and even if *omniORB* supports interceptors, it does not support the standard Portable Interceptor API [19]. *TAO* satisfies all the design principals for the invocation of remotely distributed objects implemented in different programming languages running in different OS. *TAO* is a highly extensible ORB targeted for applications with real-time QoS requirements, containing network interface, communication protocol, and CORBA-compliant middleware components and services. In addition, *TAO* supports the standard OMG CORBA reference model and Real-time CORBA specification with enhancements designed to ensure efficient, predictable, and scalable QoS behavior for high-performance and real-time applications. To this end, *TAO* was preferred over other C++ CORBA ORBs in the context of the proposed extension middleware library.

3.1.2 Porting ACE/TAO to Android

This section presents the procedure that was followed in order to port the *TAO* to the Android architecture. *TAO* version 6.2.3 was ported using the Android Native Development Kit (NDK) toolset. According to the literature, the port of *TAO* to the Android architecture has already been tested with limited number of Android NDKs versions such as *v6* and *v8e*. For the purposes of the present work, the NDK version *v8e* was selected in order to build and produce the necessary shared *TAO* dynamic libraries (.so). However, it is worth mentioning that newest versions of NDK were tested but due to a plethora of various compilation errors such as mismatch in methods signatures, their use was considered impractical. In order to port *TAO* to the Android architecture, a cross-compilation approach was followed. In details, a cross compiler capable of creating executable code for an Android platform (NDK) was used on a Linux based (x64) host machine. Afterwards, the produced dynamic shared libraries were linked and tested successfully in Android Studio, using the latest stable version (r13b) of the NDK and minimum Android API 14. The step-by-step procedure followed in order to accomplish the porting process is presented in APPENDIX 1.

3.2 FAmINE4Android: Design and Implementation

The design and functionality of the *FAmINE4Android* middleware extension library is inherited from the corresponding C++ *FAmINE* middleware. The source code of the C++ *FAmINE* middleware was imported into an Android Studio project and built using NDK version r13b. The source code of the C++ *FAmINE* middleware was slightly changed to meet the requirements of the Android development's nature, such as variations of file storage policies. *FAmINE4Android* consists of a set of core components, each responsible for a specific functionality. In details, every component implements a specific C++ *FAmINE* middleware module functionality, such as service registration and sending events, by provisioning a connection bridge between the native implementation (JNI) and the Java API provided by the *FAmINE4AndroidController*. Efforts have been focused on the provision of a service type agnostic solution in a way that any *FAmINE* service, either registered or resolved, can be integrated in a seamless and effortless manner. The following sections will discuss the implementation details of the functionality provided to Android developers.

3.2.1 Initialization and Cleanup

3.2.1.1 Initialization

An initialization procedure has to be completed prior to any further service registration or resolve, in order for the *FAmINE4Android* library to be properly initialized. This initialization procedure refers to an internal configuration mechanism responsible to establish some initial connections with the Interoperable Naming Service (INS) and Implementation Repository (ImR), based on configuration data existed in the specific text files *zones.txt* and *options.txt*. That files contain easily modifiable values. The Java developer has to place them specifically under an Android resources dedicated folder named *assets*. The syntax of each text file's configuration data is straightforward and easy to follow. The first text file, *zones.txt*, is used to define the zone, which can be seen as an isolated middleware infrastructure where services deployed in a specific zone cannot access and affect the services deployed in another zone. On the other hand, the second text file, *options.txt*, is used to define the repository network locations of the corresponded zone responsible for the storage details of the deployed services.

```
std::string hostSpec = "-ORBInitRef NameService=corbaloc:iiop:" + namingHost +
"/NameService";

std::string imrSpec = "-ORBInitRef ImplRepoService=corbaloc:iiop:" + s_imrHost +
"/ImplRepoService";

char* argv[] = {
    "FAmINE Application",
    const_cast<char*>(hostSpec.c_str()),
    const_cast<char*>(imrSpec.c_str())
};

s_orb = CORBA::ORB_init(3_, argv);
```

CodeBlock 8: Initialization of the ORB

Regarding the invocation details of the library initialization process, the Java developer has to call the *void Initialize(AppCompatActivity app)* method provided by the *FAmINE4AndroidController* Java API. At first, the *Initialize* method uses the value of the parameter *app* in order to access the aforementioned

configuration files and extract selected execution zone from their contents (i.e., host and port of nameService and imrService endpoints). Afterwards, it calculates a unique identifier based on the device's name and mac address, which is globally unique. That identifier is used later during the Naming service activation process. Subsequently, it uses the C++ *FAMINE* middleware's provided functionality to open a socket connection to the specified 'host and port'. The connection protocol for each endpoint is configured as corbaloc:iiop (see example in CodeBlock 8). The corbaloc URL scheme specifies a text string reference that uniquely identifies an object on a remote server, whereas the iiop specifies the communication protocol, which is used to facilitate network interaction between distributed objects.

3.2.1.2 Run process

The run process refers to the *FAMINE4Android* internal execution mechanism, which creates handling threads running ORB's event loops managing connection requests and handler threads, which serve requests from established connections. In detail, the Java developer calls the initialization procedure (see 3.2.1.1) prior to any service registration or resolve. The initialization procedure, apart from the aforementioned configuration/initialization functionality, creates a separated thread, which calls in infinite loop the TAO's ORB::run method with interval time of 10 milliseconds. During the execution of ORB::run method, the system accepts connections from other services, reads incoming requests and makes remote procedure calls. The reason for having a separated Java thread is to avoid the block of the Android application's UI thread. Finally, when the Java developer calls the native *Cleanup* method (see 3.2.1.3) the shutdown process takes place, the thread is exited and the ORB's event loops stop running.

3.2.1.3 Clean Up

The cleanup method refers to the *FAMINE4Android* internal mechanism which deactivates all the registered or resolved services freeing up the reserved memory. In addition, the thread started by the *Initialization process* (3.2.1.2) exits and the ORB's event loops stop running. Hence, the implemented services are not accessible any more for invocation or for remote procedure calls. In order to enable again the *FAMINE4Android* functionality, the *Initialization process* has to be restarted and the service registration/resolve process must be repeated from scratch.

3.2.2 Service implementation

3.2.2.1 Service registration

The service registration refers to the process followed by the Java developer in order to implement a service oriented Android application. The first required step followed by the Java developer is to describe the functionality of the service using a standardized definition language. The preferred language is the Interface Definition Language (IDL), which defines the public application-programming interface exposed by CORBA objects in a server application [38]. The description of a service in IDL may contain attributes and operations such as complex data types, enumerations, method signatures, etc. The second step followed by the Java developer is to use the TAO IDL compiler (tao_idl) to generate the corresponding *stub* and *skeleton* code of the service in C++ (see 2.3.3).

It is worth mentioning that the two aforementioned steps are identical as those followed in the C++ *FAMINE* middleware. In the case of the C++ *FAMINE* service registration process the C++ developer has to implement within a dedicated C++ class the whole functionality declared in the service's IDL description. That class, called servant, inherits from the service's skeleton generated using the *tao_idl*. The servant's functionality becomes publicly available when its object reference is exposed to the *Interoperable Naming Service* (INS). Regarding the *FAMINE4Android* approach, the Java developer does not need to implement the servant's required functionality in C++. On the contrary, in the third

step, the Java developer provides the service implementation within a dedicated Java class (i.e., Java servant) and registers that Java servant to the INS by using the provided Java API. In order to support the mixing between the Java servant and the C++ (actually required), the following method was adopted. The *tao_idl* compiler was modified in order to be able to generate a slightly changed *stub* code of the dummy C++ servant. The dummy C++ servant contains the initial (dummy) service implementation along with some extra code, called injected code. The injected code aims at bridging the communication between C++ servant and the corresponding Java servant through a set of libraries and functionality provided by the *FAMINE4Android* library and JNI.

To complete the service registration procedure, a service type declaration is needed in terms of updating/extending the gamma of the available services. In details, the Java developer has to write some very simple C++ instructions within a specific block declared in a cpp file compiled by the NDK (e.g., inside the *native.cpp* that is a automatically generated file containing a sample example of a native function). Those instructions constitute the update/notification of the *FamineManager* about the available service definitions. As depicted in CodeBlock 9, the Java developer has to include each generated *stub* header file and place two specific instructions for each service implementation within the *ServiceTypes* block. The first one, *RegisterToMapTypeInfo*, keeps in a C++ Standard Template Library (STL) container (i.e., map) a relation between the service type (i.e., captured via the template argument *service_type*) and a unique identifier of the service (i.e., *id*) given as parameter. To this end, future references of a service type can be easily retrieved using the *id* as a key while searching in *FamineManager*'s internal container. The second one, *RegisterToMapServantPair*, is used to keep internally in a STL container (i.e., map) a pointer to a specific templated function (i.e., *createInstance*) bound to a unique identifier of the service (i.e., *id*) given as argument. The role of the templated function *createInstance* is to create an instance of the dummy C++ servant on every call. To this end, every time a new instance of a dummy C++ servant is required, the *FamineManager* can provide the corresponding function pointer.

```
#include "<service_generated_stub>.h"

ServiceTypes {

    FamineManager::getInstance()->RegisterToMapTypeInfo<service_type>("id");

    FamineManager::getInstance()->RegisterToMapServantPair(
    &(FamineManager::createInstance<dummy_implementation_with_injected_code>), "id");
}
```

CodeBlock 9: Service type declaration example

For the completion of the 3rd step, the Java developer has to call the *void RegisterService(String [] events, String id, String contextName, Object servant)* method of the *FAMINE4Android* library. That method takes as first argument an array of event names, in order to facilitate the subscription of the supported events to the registered clients. The second one is the identifier *id*, which was used earlier in the process of service type declaration. The *id* is used to find the previously stored service type information necessary to the C++ *FAMINE* registration procedure and make the appropriate initializations. The third argument, *contextName*, defines the current instantiation of the service as presented in 2.2.3.4. Finally, the fourth argument is the Java servant providing the actual implementation of the service's functionality.

3.2.2.2 Event dispatching

The *FAMINE4Android* library provides a flexible mechanism for synchronous and asynchronous communication between distributed objects. The objective of that mechanism is to dispatch events, accompanied with a list of arguments to one or more registered clients across the network. The Java

developer has to call the method *void SendEvent(String id, String contextName, String eventName, Object[] param_list)* of the provided Java API as depicted in CodeBlock 10. This method takes as first argument the identifier *id*, which was used earlier within the process of the service type declaration (see 3.2.2.1). The *id* is used by *FamineManager* in order to retrieve the previously stored service type. The second argument, *contextName*, defines the current instantiation of the service as presented in 2.2.3.4. The third argument is the name of event that is going to be propagated. Finally, the fourth argument is a Java array, which provides a list of arguments in order to be propagated with the event.

```
String [] providedEvents = {"time_changed"};
ami.Famine.getInstance().Register(providedEvents,
    "id",
    "mycontext",
    new ImplServiceClass());
ami.Famine.getInstance().SendEvent("id",
    "mycontext",
    "time_changed",
    new Object[]{LocalDateTime.now().toString()});
```

CodeBlock 10 : Event dispatching example

```
jobject myobj = (jobject) (env->GetObjectArrayElement(j_eventArgs, i));
if(env->IsInstanceOf(myobj, stringClass) == JNI_TRUE)
{
    jstring stringVar = (jstring) (env->GetObjectArrayElement(j_eventArgs, i));
    const char *str = env->GetStringUTFChars(stringVar, 0);
    evt.AppendArg(str);
}
//etc...
else if(env->IsInstanceOf(myobj, intClass) == JNI_TRUE)
{
    jmethodID getVal = env->GetMethodID(intClass, "intValue", "()I");
    jint i = env->CallIntMethod(myobj, getVal);
    evt.AppendArg(i);
}
ami::Famine::SendEvent(servant, (const std::string&)eventName, (const
ami::LocalEvent&) evt);
```

CodeBlock 11: Process of controlling the event's elements type

As illustrated in CodeBlock 11, the communication mechanism is responsible to convert the supplied Java arguments into C++ equivalent and, in turn, remote procedure call of the corresponding event handler of each registered client. In details, the following steps take place:

- Instantiation and initialization of a C++ array, which will be the container of the arguments.
- Type checking of every event Java argument in order to create a corresponding temporal C++ variable. The value of that variable will be set with the event argument's value and used in the next steps.
- Append the C++ variable to the C++ array
- Internally call of the *FAMINE*'s library, *SendEvent* method in order to propagate the event to the interested clients across the network

3.2.2.3 Unregister service

The unregister service method refers to the *Famine4Android* internal mechanism which deactivates all the given as parameter registered service freeing up the reserved memory. Hence, the implemented service is not accessible for remote procedure calls any more.

3.2.3 Service resolve and usage

The service resolve refers to the process followed by the Java developer in order to use a distributed service within an Android application. The 1st step is the usage of the TAO IDL compiler (e.g., *tao_idl*) in order to generate the corresponding stub code of the IDL service description in C++. The IDL service description is considered to be readily available as long as the distributed service has previously been implemented. Worth noting that this step is identical to that followed in the C++ *Famine* middleware.

In the C++ *Famine* service resolve process, the C++ developer has to implement in a C++ class an event handler for each event declared in the service's IDL description. However, in *Famine4Android* the Java developer does not deal with C++ at all. Instead, the Java developer has to implement an event handler for each requesting event in a dedicated Java class. The event handler method takes as parameter a Java Object array. The event handler will be called automatically at the time the remote servant sends the corresponding event. Event parameters' values will be propagated to the event handler through the array argument.

At this point, a service type declaration is needed in terms of updating/extending the gamma of the available services. Similarly to 3.2.2.1, the Java developer has to write some very simple C++ instructions within a specific block declared in a cpp file compiled by the NDK. Those instructions constitute to the update/notification of the *FamineManager* about the available service definitions. As depicted in CodeBlock 12, the Java developer has to include each generated *stub* header file and place one specific instruction for each service usage within the *ServiceTypes* block. The instruction *RegisterToMapTypeInfo* makes *FamineManager* to keep in a C++ Standard Template Library (STL) container (i.e., map) a relation between the service type (i.e., captured via the template argument *service_type*) and a unique identifier of the service (i.e., *id*) given as parameter.

```
#include "<service_generated_stub>.h"
ServiceTypes {
    FamineManager::getInstance()->RegisterToMapTypeInfo<service_type>("id");
}
```

CodeBlock 12: Instructions to FamineManager in case of service usage

In the second step, a Java developer has to invoke the *void ResolveService(Object [] events, String id, String contextName, Object eventHandlers)* method of the provided Java API. That method calls internally the equivalent C++ *Famine* middleware's *Resolve* method in order to resolve a reference to the targeted distributed object. That reference is kept internally by the *FamineManager* for future access. This method takes as first argument an array of event names, which are being used for the corresponding event subscription so as the servant to propagate those events to the registered client. The latter is the identifier *id*, which was used earlier in the service type declaration. The *id* is used to find the previously stored service type information necessary to the C++ *Famine* resolve procedure and the appropriate initializations. The third argument, *contextName*, defines the current instantiation of the service as presented in 2.2.3.4. Finally, the fourth argument is the Java class providing the actual implementation of the event handlers' functionality.

3.2.3.1 Events reception

FAMINE4Android library implements a mechanism for the propagation of the incoming events' argument values to the corresponding event handler. The objectives of that mechanism is to retrieve all the arguments, convert them to equivalent JNI variables and pass them to the corresponding Java event handler method. As illustrated in CodeBlock 13, during the execution of that mechanism, the following steps are taking place:

- Type checking of every event argument in order to create a corresponding native variable. The value of that variable will be set temporarily with the event argument's value and be used from the next steps.
- Instantiation of a JNI Object array, which will be returned to the Java side.
- Instantiation and initialization of the corresponding JNI variable.
- Append the JNI variable to the JNI Object array.
- Search the corresponding Java event handler method within the given *eventHandlers* Java class and call with argument the JNI object array.

```
ami::AnySeq args = event.GetAllArgs();
int length = args.length();
jobjectArray returned_array = (jobjectArray)FamineManager::getInstance()->env-
>NewObjectArray(_length, FamineManager::getInstance()->env->FindClass("java/lang/Object"),
NULL);
for(int i =0;i< length;++i) {
    CORBA::Any any = args[i];
    CORBA::TCKind kind = any.type()->kind();
    if(kind == CORBA::tk_short){
        short shortValue;
        any >>= shortValue;
        FamineManager::getInstance()->env->SetObjectArrayElement(returned_array, i,
        ConversionUtils::CreateShortObject(FamineManager::getInstance()->env, shortValue));
    }
    else if(kind == CORBA::tk_boolean) {
        bool boolValue;
        any >>= ACE_InputCDR::to_boolean(boolValue);
        FamineManager::getInstance()->env->SetObjectArrayElement(returned_array, i,
        ConversionUtils::CreateBooleanObject (FamineManager::getInstance()->env, boolValue));
    }
    //etc.
}
jmethodID methodID = FamineManager::getInstance()->env->GetMethodID(hostClass,
event.GetName().c_str(), "([Ljava/lang/Object;)V");
FamineManager::getInstance()->env->CallVoidMethod(hostClass, methodID, returned_array);
```

CodeBlock 13: An example of event arguments' data conversion among various CORBA types

3.2.3.2 Call service process

The call service process refers to the steps followed by the Java developer in order to make a remote procedure call to a distributed service. The call service process consists of two steps. In the first step,

the Java developer has to resolve a distributed service. As mentioned above, a reference to the resolved object is kept by the *FamineManager* for future access. The remote procedure call is executed upon the retrieved reference of the resolved object. Regarding the second step, the Java developer has to call the *Object CallFunction(String id, String contextName, String functionName, Object[] arr)* of the provided Java API (see example in CodeBlock 14). This method takes as first argument the identifier *id*, which was used earlier within the process of the service type declaration. The *id* is used to find the previously stored service type information necessary to the C++ *FAMINE* resolve procedure and the appropriate initializations. The second argument, *contextName*, defines the current instantiation of the service as presented in 2.2.3.4. The third argument is the name of method that is going to be invoked. Finally, the fourth argument is a Java array, which provides a list of arguments to be propagated to the calling method. That method returns a Java object, which contains the returned calling method's returned value. The *CallFunction* converts the given arguments by the Java developer to equivalent C++ values and uses them at the invocation of the corresponding resolved object's method. Further details about the conversion strategies can be found in section 0.

```
Object [] eventsForSubscription = {"event_name_1"};
ami.Famine.getInstance().ResolveService(eventsForSubscription,
                                     "id",
                                     "mycontext",
                                     new EventHandlers());
Object result = ami.Famine.getInstance().CallFunction("id",
                                                    "mycontext",
                                                    "add",
                                                    new Object[]{"param1", 2, 3});
System.out.println("method add returned" + result);
```

CodeBlock 14: Remote procedure call example

3.2.3.3 Dispose service

The dispose service method refers to the *FAMINE4Android* internal mechanism responsible to release the resolved service reference and freeing up the reserved memory. In addition, the process updates the distributed service to unsubscribe the associated event handlers so as dispatched events are no further received by this registered client.

3.3 Addressing interoperability issues between Android Java and JNI

One of the benefits of programming in Java is that the Java Virtual Machine (VM) hides platform differences. The Java Virtual Machine is an abstraction layer software placed between the physical machine and the actual program. Applications can be developed once and run anywhere a Java VM is implemented, in contrast to C++ where there is no guarantee that a C++ program that has written for one platform will perform on another. Java does not support functions that exist outside a class, so code that is written in C++ must be wrapped up in a Java class definition [33]. The methods that are implemented in C++ but are declared in Java are called *native methods* and using the Java Native Interface (JNI), the Java and C++ components are able to talk to each other and call methods among them. The following sections introduce interoperability issues stemming from hosting native code in an Android Java application.

3.3.1 Calling native functions from Java

The procedure of making native methods callable from Java demands the declaration of the native methods' signature in Java code. The native methods' signature consists of the declaration type of the arguments and the return value. The compiled code of native methods, type definitions, and possible constants lies on static or dynamic libraries produced by a NDK suitable compiler. Thus, in order for a native method to be callable, the Java developer has to invoke the *void System.loadLibrary(library-name)* where the parameter *library-name* points to the corresponding library file (.so) containing the compiled code of the method's body. As depicted in the example of CodeBlock 15, the invocation of the *System.loadLibrary(library-name)* has to be placed in a static initializer so as to load the native library just when the Java class is loaded by the classloader.

```
Static {  
    System.loadLibrary("famine_proxy-lib");  
}
```

CodeBlock 15: Load library invocation

Regarding the signature of the native methods, a convention treatment is applied. The native methods' signature consists of the following pattern:

returned_type Java_{package_and_classname}_{function_name}(JNIEnv, jobject, arg1, arg2, ...)*

The first segment *{package_and_classname}* corresponds to the package and class name of the Java class in which the method is declared. The second segment, *{function_name}* represents the name of the native method. Finally, the last one segment *(JNIEnv*, jobject, arg1, arg2, ...)* is a list of arguments in which the first two are required by the JNI programming rules. In detail, the argument *JNIEnv** is a reference to JNI environment, which allows the access to all native system methods and the second *jobject* argument is a reference to the Java class object containing the native declaration. Furthermore, as depicted in the auto-generated file of CodeBlock 16, the keyword *extern* is placed prior the declaration of the native method, allowing the function to be exposed in the shared library at runtime. It is worth mentioning that every native file starts with the inclusion of the header file *jni.h*, which lists all the available JNI functions and datatypes.

```
#include <jni.h>  
#include <string>  
  
extern "C"  
jstring  
Java_com_example_zidian_myapplication_MainActivity_stringFromJNI(  
    JNIEnv *env,  
    jobject /* this */) {  
    std::string hello = "Hello from C++";  
    return env->NewStringUTF(hello.c_str());  
}
```

CodeBlock 16: Auto generated native file

For the purposes of the present work, the *FAMINE4Android* library facilitates also the usage of the implemented native functions through an easy to use Java API. In details, the middleware's functionality is seamlessly integrated into the *FAMINE4Android Controller* and is available to Java developer through a set of functions. The majority of those are directly linked to native JNI functions,

whereas others encapsulate additional functionality based on Java libraries and if necessary they use internally which native JNI functions are in need of usage.

3.3.2 Calling Java functions from native code

Regarding to the communication ability between native and Java code, JNI operates as the medium providing the appropriate mechanism and toolset. JNI provides the necessary functionality in order to enable Java method calls and data conversion from Java to C++ objects and vice-versa. Consider the example of Java method declared in a specific Java class named A. As depicted in CodeBlock 17, the JNI method *FindClass* acts as a native class loader used to retrieve a reference to an appropriate *.class* in the list of directories that was provided at Java Virtual Machine initialization. The *FindClass* method takes as argument the fully qualified class name that consists of the Java package name delimited by *"/"* and followed by the class name (e.g., *"/com/ami/A"*). Thereafter, the class reference is used by the JNI method *GetMethodId* in order to retrieve a method reference within the class. The *GetMethodId* takes three arguments: a) the Java class reference in which the calling method is declared, b) the name of the targeted Java method and c) the method signature regarding the types of the arguments and the returned value. In case of mismatch, the *GetMethodId* returns zero value indicating that not such a method is declared within the Java class. For example, the signature term *"()"* means that the Java method receives no parameter while the term *"V"* means that the return type is void. The available mappings between the Java argument types and the corresponding native signatures (i.e., used in *GetMethodId*) are presented in Table 2.

```
extern "C"
void Java_Test_Call_Java_Method (JNIEnv* env, jobject obj){
    jclass activityClass = env->FindClass ("/com/ami/A");
    jmethodID method = env->GetMethodId(activityClass, "myJavaMethod", "()V");
    env->CallVoidMethod(activityObj, method);
}
```

CodeBlock 17: Call a Java method from native code using FindClass method

```
JNIEXPORT
void Java_Test_Call_Java_Method (JNIEnv* env, jobject obj, jobject classObj){
    jclass activityClass = env->GetObjectClass(classObj);
    jmethodID method = env->GetMethodId(activityClass, "myJavaMethod", "()V");
    env->CallVoidMethod(activityObj, method);
}
```

CodeBlock 18: Call a Java method from native code using GetObjectClass method

Table 2: Mapping between the Java types and the native signatures

Signature	Java Type
V	void
Z	boolean
B	byte
C	char

S	short
I	int
J	long
F	float
D	double
L fully-qualified-class	fully-qualified-class
[type	type[]
(arg-types) ret-type	method type

Depending on the type of Java method's return value (e.g., *Void*), the corresponding JNI function is used to establish the actual invocation (e.g., *CallVoidMethod*, *CallIntMethod*, *CallBooleanMethod*, etc.). An alternative approach for retrieving a Java class reference is the JNI method *GetObjectClass* that searches using an object instance instead of the class name as required in the case of *FindClass* method. An example of the JNI *GetObjectClass* is depicted in CodeBlock 18.

3.4 Facilitating interoperability between Java and native code

The proposed middleware library embeds advanced communication mechanisms to support interconnection between Java and native code in order to provide seamless functionality to the Android developer. The main goal of the *FAMINE4Android* library is to support the development process of distributed and embedded real-time applications. To this end, the *FAMINE4Android* library provides the Java developer with an appropriate and intuitive to use Java API, hiding the heterogeneity issues between Java and native code. The following sections present the developed mechanisms facilitating the data exchange process between Java and internal native functionality provided by the *FAMINE* middleware and TAO.

3.4.1 Reification and Reflection in C++

The reification and reflection process refers to the collaboration ability of unknown type variables and object instances of different classes, through a mechanism able to operate with generic type code. This mechanism is called *Reflection*, and even if many programming languages provide built-in reflection mechanism, C++ does not support it. In details, *Reflection* is able to: a) investigate the object type of class at runtime, b) has access to every object's fields/properties and c) method invocation. Given that the core components of the *FAMINE4Android* library are directly derived from the source code of the C++ *FAMINE* middleware, the contribution of the *Reflection* mechanism was crucial. In addition, the *Reflection* mechanism stands as a prerequisite in order to support types of not known distributed services as they are produced dynamically through the *tao_idl* compiler. This is supported using the *Run Time Type Reflection (RTTR)* library. The RTTR library facilitates Reflection in a semi-automated manner in which pre-defined information regarding method signatures, types, etc., are required by the developer. The RTTR's provided functionality is further presented in section 2.3.6.

In order to employ the functionality of the *RTTR* library, the (RTTR) registration of each distributed service's methods and defined data types is required. Normally, the (RTTR) registration is performed from the user/developer of the RTTR library. For the purposes of this work, a fully automated approach was adopted. Taking into account that each service is declared into the corresponding generated *stub* and *skeleton* files (see 2.3.3), the need of automatically placing the necessary (RTTR) registration code

within those files was evolved. The term *injected code* is used to describe the aforementioned (RTTR) registration code. To this end, the *tao_idl* compiler was modified accordingly (*tao_idl_4Android*). The *tao_idl_4Android* compiler contributes towards the auto-enrichment of a service *stub* with the *injected code* required for the proper operation of RTTR library. The additional functionality of the *tao_idl_4Android* compiler is illustrated below by indicating some injected code examples in the case of a sample service named `Example::Echo` (see CodeBlock 19).

```
#include <ami.idl>

module Example {

    interface Echo {

        enum Priority {PR_INFO, PR_WARNING, PR_ERROR};

        struct AdvancedMessage {

            Priority priority;

            string msg;

            long number;

        };

        string EchoString (in string text);

        void Event_NewMessage (in AdvancedMessage msg);

    };

};
```

CodeBlock 19: `Example::Echo`; service definition in IDL

	using namespace rttr;
	RTTR_REGISTRATION {
A	<pre>.enumeration<Example::Echo::Priority>("Example:: Echo::Priority") (value("PR_INFO", Example::Echo::Priority::PR_INFO), value("PR_WARNING", Example::Echo::Priority:: PR_WARNING), value("PR_ERROR", Example::Echo::Priority::PR_ERROR))</pre>
B	<pre>.method("EchoString ", &Example::Echo::EchoString)</pre>
C	<pre>registration::class_<Example::Echo::AdvancedMessage>("Example::Echo::AdvancedMessage") .property("pr", &Example::Echo::AdvancedMessage:: priority) .property("msg", &Example::Echo::AdvancedMessage::get_msg, &Example::Echo::AdvancedMessage::set_msg) .property("number", &Example::Echo::AdvancedMessage::number)</pre>
D	<pre>registration::class_<Example::Echo>("Example::Echo")</pre>
	}

CodeBlock 20: Injected code of the `Example::Echo` service

The injected code generated by the *tao_idl_4Android* compiler, taking as input the definition of the `Example::Echo` service, consists of the following declaration categories: a) enumerations, b) complex data types (e.g., structs), and c) methods (or event methods). In the case of the *Priority* enumeration,

the equivalent injected code is depicted in CodeBlock 20 (A). The declaration of the *EchoString* method is responsible for the injected code illustrated in CodeBlock 20 (B). It is worth mentioning that the event's declaration does not create any injected code, as it is not necessary for reflection. That happens because the event dispatching process is implemented with a service type agnostic approach. Similarly, the struct *AdvanceMessage* creates the injected code as figured in CodeBlock 20 (C). Finally, the service type information has to be registered in the RTTR as depicted in CodeBlock 20 (D). All the aforementioned injected code snippets constitute the additional outcome of the *tao_idl_4Android* placed at the end of the corresponding *stub* file.

3.4.2 Data exchange

A mechanism responsible for data exchange between Java and native code was implemented in the context of the present work. Such mechanism is able to exchange the format of primitive or complex data types in order to facilitate the data type differences between Java and native code. In detail, JNI native functions are used to smoothly accomplish the conversion of primitive types (e.g., such as integer, float, string, etc.) between the two languages. Complex types are also supported as presented in the next sections.

3.4.2.1 Primitive types

The conversion approach of Java primitive types such as *int*, *float*, *string*, etc. to native equivalents is facilitated by the underlying functionality of the JNI. The data types' names remain identical between Java and C++. The only difference is that the JNI data type name has as prefix the character 'j' as illustrated in the first two columns of Table 3. Regarding the conversion of Java string variable to a native one, a special approach is required. In detail, the JNI native function *const char* GetStringUTFChars(jstring, jboolean)* is required to convert a *jstring* to a sequence of characters (e.g., *char []*). As depicted in CodeBlock 21, that function takes two arguments; the first one is the *jstring* and the second one is a *boolean* variable (in case of *JNI_TRUE* (1) the returned string is a copy of the original *java.lang.String* instance, otherwise, in case of *JNI_FALSE* (0), the returned string is a direct pointer to the original *String* instance).

Table 3 : Mapping of Java types to natives types

Java Type	JNI Type	Native Type
boolean	jboolean	bool
byte	jbyte	unsigned char
char	jchar	char
short	jshort	short
int	jint	int
long	jlong	long
string	jstring	char*
float	jfloat	float
double	jdouble	double
void	Void	void

```
const char * str_ = env->GetStringUTFChars(jstring_m, 0);
args.push_back(*(new variant(str_)));
```

CodeBlock 21: Example of primitive data type checking and equivalent conversion to native code

Regarding the arrays of primitive types, a special treatment is also required because there is no direct mapping supported by JNI. For example, an array of *Short* variables in Java (e.g., *Short[]*) is mapped to a *jshortArray* instead of *jshort[]* as should be expected. In addition, for the purposes of this work, the

conversion of arrays (primitive types) into the specific arrays supported by CORBA specification was necessary (see the last two columns of Table 4). As a result, special JNI functions are used to retrieve each specified short element from a *jshortArray* as illustrated in CodeBlock 22.

Table 4 : Mapping of Java arrays to native arrays and CORBA arrays

Java Type	JNI Type	CORBA types
Object[]	jobjectArray	Not Applicable
Boolean[]	jbooleanArray	::CORBA::BooleanSeq BooleanSeq;
Byte[]	jbyteArray	::CORBA::OctetSeq OctetSeq;
Char[]	jcharArray	::CORBA::OctetSeq OctetSeq;
Short[]	jshortArray	::CORBA::ShortSeq ShortSeq;
Int[]	jintArray	::CORBA::LongSeq LongSeq;
Long[]	jlongArray	::CORBA::LongLongSeq LongLongSeq;
Float[]	jfloatArray	::CORBA::FloatSeq FloatSeq;
Double[]	jdoubleArray	::CORBA::DoubleSeq DoubleSeq;
String[]	jobjectArray	::CORBA::StringSeq StringSeq;

```
int length = env->GetArrayLength(m_jshortarray);
jshort *it = env->GetShortArrayElements((jshortArray)m_jshortarray, NULL);

ami::ShortSeq *numbers = new ami::ShortSeq();
numbers->length(length);

for(int j=0;j<length;j++) {
    (*numbers)[j] = it[j];
}
```

CodeBlock 22: Example of primitive data type checking and equivalent conversion to native code

Additionally, a mechanism was implemented in order to facilitate the data transportation from C++ to Java code. Primitive type variables as well as CORBA's specifics are converted into the JNI's generic data type (i.e., *jobject*) which subsequently is passed to the Java-side. Specific JNI functions are used in the conversion process according to the type of each native variable. According to the conversion approach followed in the *FAMINE4Android* library, every native variable is converted to a generic JNI object, which encloses its type for further type retrospection from Java side. Regarding the specific types provided by CORBA specification (e.g., *::CORBA::Short*), they are also converted to *jobject*. In details, the conversion process requires the construction of a new *jobject* variable, using the corresponding type-based class constructor. Afterwards, the *jobject* variable is passed to the Java-side along with the corresponding value of the native variable as depicted in CodeBlock 23.

```
::CORBA::Short extracted_value = val.get_value<::CORBA::Short>();
jclass shortClass = (env)->FindClass("java/lang/Short");
jmethodID shortConstructor = (env)->GetMethodID(shortClass, "<init>", "(S)V");
jobject return_value = (env)->NewObject(shortClass, shortConstructor, extracted_value);
```

CodeBlock 23 : Conversion of CORBA type variable to jobject type variable

Concerning C++ primitive arrays, as well as the special primitive array of types provided by the CORBA specification, a similar conversion process is required. As illustrated in CodeBlock 24, the steps required for the conversion of primitive array types to JNI Object type variables are the following:

- Initialization of a JNI wrapper definition for a type-based array (e.g., *jshortArray jnumbers*),
- Initialization of a JNI primitive array (e.g., *jshort cArray[length];*),

- Iteration through the actual data (e.g., *ami::ShortSeq *numbers*) and insertion to the primitive,
- Copy the whole region of the primitive array to the JNI wrapper array (e.g., *SetShortArrayRegion(jnumbers, 0, length, cArray)*).

Finally, a *jobject* variable with value the wrapper array object is returned to the Java side.

```
ami::ShortSeq *numbers = ret.get_value<ami::ShortSeq*>();
int length = numbers->length();
jshortArray jnumbers = env->NewShortArray(numbers->length());
jshort cArray[length];

for(int i =0; i<length; ++i ) {
    cArray[i] = (*numbers)[i];
}
env->SetShortArrayRegion(jnumbers, 0, length, cArray);

jobject return_value = jnumbers;
return return_value;
```

CodeBlock 24 : Conversion of CORBA type array to jobject type variable

3.4.2.2 Complex data types

For the purposes of the present work, focus has been given to support not only the aforementioned primitive types but also complex data types. In this direction, a common ground had to be established due to the lack of support provided by JNI native functions to convert C++ structs/class instances to Java objects and vice-versa. *FAMINE4Android* implements a common ground mechanism by employing a popular data description language such as XML or JSON. That approach facilitates the variables' values storage and transportation between the two programming languages in an efficient manner. JSON was preferred thanks to its increased processing speed over XML [2.3.5]. To this end, the values of a C++ instances are stored in a JSON format (e.g., as pure text), which subsequently are transported to Java-side. Transported values can be similarly translated back to a corresponding Java object thanks to the functionality provided by a plethora available Java serialization/ deserialization libraries. The aforementioned procedure operates similarly in the case of values transportation from Java to C++.

The exchange process of complex data from Java to native code is facilitated by the Gson Java library [32]. Gson can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object. Gson can work with arbitrary Java objects without requiring the placement of specific Java annotations in the classes. It also fully supports the use of Java Generics leading to be the most prevalent choice for the purposes of the proposed work.

Gson provides through an easy to use Java API the *toJson()* and *fromJson()* methods to convert Java objects to JSON and vice-versa. The example illustrated in the CodeBlock 25 consists of the clock::alarm service definition (see CodeBlock 25 (A)), the equivalent class definition of the struct Time in Java (see CodeBlock 25 (B)), as well as an invocation example of the *FAMINE4Android* library's *CallFunction* based on *Time* references (see CodeBlock 25 (C)). In detail, the *toJson()* method is used to convert each object instance of the *Time* class (e.g., *currentTime*, *snoozeTime*) to JSON format. Consequently, the JSON values are given as parameters to the invocation of the *CallFunction*. Within the internal functionality of the *FAMINE4Android* library, the input parameters are used in order to generate the equivalent C++ struct instances declared in the service's definition *stub* files. At this point, the *fromJson()* method is used in order to deserialize the JSON value and construct the *Time* instance appropriately. The returned *Time* value of the service function *add* is converted back to JSON format and transported to the Java side through a *jobject* variable (details are presented in the following paragraph).

A	<pre>#include<ami.idl> module clock { interface alarm { struct Time { long hours; long minutes; }; Time add(in Time a, in Time b); } }</pre>
B	<pre>public class Time { int hours; int minutes; Time(int hours, int minutes){ this.hours = hours; this.minutes = minutes; } }</pre>
C	<pre>Time currentTime = new Time(LocalDateTime.now().getHour(), LocalDateTime.now().getMinute()); Time snoozeTime = new Time(0, 9); String currentTimeJSON = Famine.getInstance().gson.toJson(currentTime); String snoozeTimeJSON = Famine.getInstance().gson.toJson(snoozeTime); Object retObj = Famine.getInstance().CallFunction("id", mycontext, "add", new Object[]{currentTimeJSON, snoozeTimeJSON}); Time alarmTime = ami.Famine.getInstance().gson.fromJson(retObj.toString(), Time.class);</pre>

CodeBlock 25: Call *toJson()* method and *fromJson()* in Java

Regarding the deserialization process of JSON within native code, an ad hoc approach is followed. As briefly presented in 3.2.2.2, a type checking approach is followed in order to convert a Java variable to the corresponding C++ variable. The type checking begins with the checking of the available primitive types as well as with those declared by CORBA (e.g., sequences of data types such as `::CORBA::LongSeq`). Given that the incoming data type is different from the aforementioned, the procedure continues with the assumption that the type of the checking variable corresponds to a complex data type. Thus, the actual value of the passed *object* argument should be equal to the JSON representation of a complex data type. At this point, *RTTR* functionality is employed in order to retrospect the complex data type based on information already registered to *RTTR* by the corresponding *injected code*. As partially presented in 3.4.1, the *injected code* declares information about service's types and methods to the *RTTR* facilitating the reflection underlying mechanism (see CodeBlock 20). The *injected code* is extended with the declaration of two JSON-oriented static methods

in the case of complex data types. As depicted in CodeBlock 26 (A), *tao_idl4android* compiler automatically generates an appropriate *convert_from_json* method for each complex data type. In addition, *tao_idl4android* declares the existence of that method to the *RTTR* library as depicted in CodeBlock 26 (B). As its name describes, the *convert_from_json* method takes as an argument the JSON representation of a complex data type value and constructs the corresponding native instance using the *RTTR*'s provided functionality for JSON serialization and deserialization. To sum up, the retrospection as well as the invocation of the corresponding *convert_from_json* static method in order to deserialize a native complex data type is illustrated in CodeBlock 27.

A	<pre>const Example::Echo::AdvancedMessage Example::Echo::AdvancedMessage::convert_from_json(const std::string& json){ Example::Echo::AdvancedMessage item; io::from_json(json, item); return item; }</pre>
B	<pre>.method("convert_from_json", &Example::Echo::AdvancedMessage::convert_from_json);</pre>

CodeBlock 26: Implementation of the *convert_from_json* in the case of the *AdvancedMessage* example

<pre>jstring str = (jstring) (env->GetObjectArrayElement(j_arguments, argument_index)); std::string json(env->GetStringUTFChars(str, 0)); rttr::string_view name = info.get_type().get_name(); method meth0 = type::get_by_name(name).get_method("convert_from_json"); if(meth0.is_valid()){ variant res = meth0.invoke(instance(), json); args.push_back(*(new variant(res))); }</pre>

CodeBlock 27 : Dynamic complex data type JSON deserialization

A similar approach is followed for the serialization/deserialization needs of custom sequences of complex data types. The *convert_from_json* static method is also automatically injected by *tao_idl4Adnroid* compiler within the scope of a custom sequence definition. CodeBlock 28 illustrates the implementation of the *convert_from_json* method (see CodeBlock 28 (B)) triggered by the user's custom sequence definition in IDL (see CodeBlock 28 (A)).

A	<pre>typedef sequence<AdvancedMessage> AdvancedMessageSeq;</pre>
B	<pre>const Example::Echo::AdvancedMessageSeq Example::Echo::AdvancedMessageSeq::convert_from_json(const std::string& json) { Example::Echo::AdvancedMessageSeq seq; rapidjson::Document document; if (document.Parse(json.c_str()).HasParseError()) return seq; assert(document.IsArray());</pre>


```

        seq.length(document.Size());
        for (rapidjson::SizeType i = 0; i < document.Size(); ++i) {
            Example::Echo::AdvancedMessage item;
            io::fromjson_recursively(item, document[i]);
            seq[i] = item;
        }
        return seq;
    }
}

```

CodeBlock 28: Implementation of the *convert_from_json* in the case of the AdvancedMessageSeq example

A similar approach is followed in order to transport complex type values from native code to Java-side. As briefly presented in the example of CodeBlock 25 (C), JSON functionality is employed in order to serialize *CallFunction*'s returned complex value. In addition, JSON serialization functionality is also used in the case of event's arguments management. In detail, within the internal functionality of the *CallFunction* method, a type checking procedure is followed in order to determine the type of the value returned by the remote procedure call. That value is temporarily stored in a generic holder provided by RTTR library (e.g., *rttr::variant* class that allows storing data of any type). As depicted in CodeBlock 29, the procedure starts checking against the known primitive types and then continues with the available CORBA types and sequences as well.

```

if (return_value_type == type::get<void>()) { //the remote procedure call returns VOID
    return env->NewByteArray(0);
}
//handle primitive (CORBA) types
if (return_value_type == type::get<char *>()) {
    ret = env->NewStringUTF(return_value.get_value<char *>());
}
else if (return_value_type == type::get<int>()) {
    int extracted_value = return_value.get_value<int>();
    ret = ConversionUtils::CreateIntegerObject(env, extracted_value);
}
else if (return_value_type == type::get<::CORBA::Short>()) {
    ::CORBA::Short extracted_value = return_value.get_value<::CORBA::Short>();
    ret = ConversionUtils::CreateShortObject(env, extracted_value);
}
//etc... now continue checking against CORBA sequences
else if (return_value_type == type::get<ami::FloatSeq*>()) {

    ami::FloatSeq *numbers = return_value.get_value<ami::FloatSeq*>();
    jfloatArray jnumbers = env->NewFloatArray(numbers->length());
    jfloat cArray[numbers->length()];

    for(int i = 0; i < numbers->length(); ++i ) {
        cArray[i] = (*numbers)[i];
    }
    env->SetFloatArrayRegion(jnumbers, 0, numbers->length(), cArray);
    ret = jnumbers;
}
else if (return_value_type == type::get<ami::ShortSeq*>()) {

    ami::ShortSeq *numbers = return_value.get_value<ami::ShortSeq*>();
    jshortArray jnumbers = env->NewShortArray(numbers->length());
    jshort cArray[numbers->length()];
    for(int i = 0; i < numbers->length(); ++i ) {
        cArray[i] = (*numbers)[i];
    }
    env->SetShortArrayRegion(jnumbers, 0, numbers->length(), cArray);
    ret = jnumbers;
}
//etc...

```

```

else { //assumption: it's a complex type! try serialize it to json
    ret = env->NewStringUTF(io::to_json(return value).c_str());
}

```

CodeBlock 29: Type checking logic of the RPC returned value

If the type of the returned value is not a primitive one or equivalent to a CORBA sequence, the procedure continues with the assumption that the type of the returned variable is equivalent to a complex data type. As a result, the RTTR's embedded functionality is employed to serialize the returned value to JSON format and subsequently, to transport the serialized object to the Java side. In order to support this process, the RTTR's JSON capabilities have been enhanced as follows: a) complex data types, and b) CORBA defined or custom-defined sequences declared within the scope of complex data types. Furthermore, event handling functionality has also been enhanced. Given that each event argument type is a generic type container (e.g., CORBA::Any) some extra-injected code is required to automate the serialization process as illustrated in CodeBlock 30. In details, *tao_idl4android* compiler automatically generates an appropriate *convert_to_json* method for each complex data type. In addition, *tao_idl4android* declares the existence of that method to the RTTR library as depicted in CodeBlock 30 (B). As its name describes, the *convert_to_json* method takes as an argument the CORBA wrapper container of a complex data type and returns its corresponding JSON representation (see CodeBlock 30 (A)).

A	<pre> std::string Example::Echo::AdvancedMessage::convert_to_json(const ::CORBA::Any &_tao_any) { const Example::Echo::AdvancedMessage *msg; _tao_any >>= msg; return io::to_json(msg); } </pre>
B	<pre> .method("convert_to_json", &Example::Echo::AdvancedMessage::convert_to_json) </pre>

CodeBlock 30: Implementation of the *convert_to_json* in the case of the *AdvancedMessage* example

A similar approach is followed for the serialization needs of custom sequences of complex data types (e.g., CodeBlock 31 (A)). The *convert_to_json* static method is also automatically injected by *tao_idl4Adnroid* compiler within the scope of a custom sequence definition. CodeBlock 31 (B) depicts the implementation of the *convert_to_json* static method facilitating the event handling process for a custom sequence argument type (e.g., *AdvancedMessageSeq*) wrapped in a generic type container (e.g., *::CORBA::Any*). CodeBlock 31 (C) illustrates the *convert_to_json* static method facilitating the serialization process of the RPC returned value in case of custom sequence of complex data types. CodeBlock 31 (D) presents the necessary injected code aiming to RTTR-register the custom sequence type (e.g., *AdvanceMessageSeq*), as well as the aforementioned overloaded functions.

A	<pre> typedef sequence<AdvancedMessage> AdvancedMessageSeq; </pre>
B	<pre> std::string Example::Echo::AdvancedMessageSeq::convert_to_json(const ::CORBA::Any &_tao_any) { const Example::Echo::AdvancedMessageSeq *seq; _tao_any >>= seq; std::stringstream resu; resu << "["; </pre>

	<pre> for (int i = 0; i < seq->length(); ++i) { const Example::Echo::AdvancedMessage &item = (*seq)[i]; resu << io::to_json(item); if (i + 1 < seq->length()) resu << ","; } resu << "]; return resu.str(); } </pre>
C	<pre> std::string Example::Echo::AdvancedMessageSeq::convert_to_json(Example::Echo::AdvancedMessageSeq* seq) { std::stringstream resu; resu << "["; for (int i = 0; i < seq->length(); ++i) { const Example::Echo::AdvancedMessage &item = (*seq)[i]; resu << io::to_json(item); if (i + 1 < seq->length()) resu << ","; } resu << "]; return resu.str(); } </pre>
D	<pre> registration::class_<Example::Echo::AdvancedMessageSeq>("Example::Echo::AdvancedMessage Seq") .method("convert_to_json", select_overload<std::string(const ::CORBA::Any &)>(& Example::Echo::AdvancedMessageSeq::convert_to_json)) .method("convert to json", select_overload<std::string(Example::Echo::AdvancedMessageSeq*)>(& Example::Echo::AdvancedMessageSeq::convert_to_json)) </pre>

CodeBlock 31: Automatically generated injected code in the case of custom sequence of complex data types

4 Case study: Museum Guide Application using Android mobile devices

The last few decades, interactive technologies have been applied to museums in order to deliver interactive and immersive user experience through on-site Virtual Exhibitions and augmented reality technologies. In this context, the proposed work has been demonstrated by implementing an example application in the domain of cultural heritage, named *Museum Guide*, suitable for android mobile devices. That application builds upon the proposed *FAMINE4Android* library and aims to deliver guidance support in an automatic manner based on visitors' location. In details, the proposed case study delivers rich museum touring guidance that escorts users during their visit while using their mobile phones. The *Museum Guide* provides always-available information and multimedia, such as images, videos and descriptive text, regarding all exhibits and other points of interest of the exhibition area. Moreover, visitors are essentially accompanied by a comprehensive, intelligent guidance functionality that enhances the museum experience by visualizing, interacting and navigating into the available digital museum collections.

In order to deliver implicit location-based guidance within the exhibition spaces, a human body recognition and localization approach was considered crucial. After conducting thorough research in the field of human tracking and sensing technologies, it was decided to employ the "*Tracking persons using a network of RGBD cameras*" [14] approach. This approach builds upon advanced computer vision algorithms that achieve the detection and localization of humans in indoor environments. According to [14], the underlying infrastructure consists of an RGB-D camera network aiming to track multiple humans in real-time. The case study highlights the contribution of the *FAMINE4Android* library in a distributed system where heterogeneous devices, such as PCs and Android devices, can communicate in real-time.

The following sections present the employed infrastructure within a simulation space located in the FORTH-ICS Ambient Intelligence Programme^{2,3} Facility, along with some implementation details about the used tracking service. Afterwards, the design and implementation of the *Museum Guide* are elucidated followed by the presentation of the preliminary evaluation results.

4.1 Tracking persons using a network of RGB-D cameras

The implemented setup employs multiple visual sensors in order to enumerate, localize and track individual persons. As depicted in Figure 3, a network of four *Microsoft® Kinect (XBOX one)* RGB-D cameras have been placed to the corners of a rectangular area, evenly and high, surrounding a volume in which persons can be localized and tracked. Multiview human localization methods perform a 3D representation of the imaged users to be registered to a map of the oriented area, in order to provide accurate information about the number and location of individuals.

The localization of humans is based solely on reconstructed volumes, which are projected on a representation of the floor plane, whereas the detected individuals are tracked based on both geometric and color information. The vision system assigns a unique identification number to each person entering the room (Person ID) and remains same even in the case that two or more users are situated very near each other, as illustrated in Figure 4.

² http://www.ics.forth.gr/index_main.php?l=g&c=4

³ <http://ami.ics.forth.gr/>

The computer vision algorithm declares a Person ID, i.e., a unique ID for each individual that discriminates from others. The Person ID is registered as long a visitor enters the cameras' observed area and corresponds to his/her visual representation. Since users are associated with a unique ID, the tracking service stores geometric and color related information in order to re-recognize users when, after a period of absence, they re-enter the room.

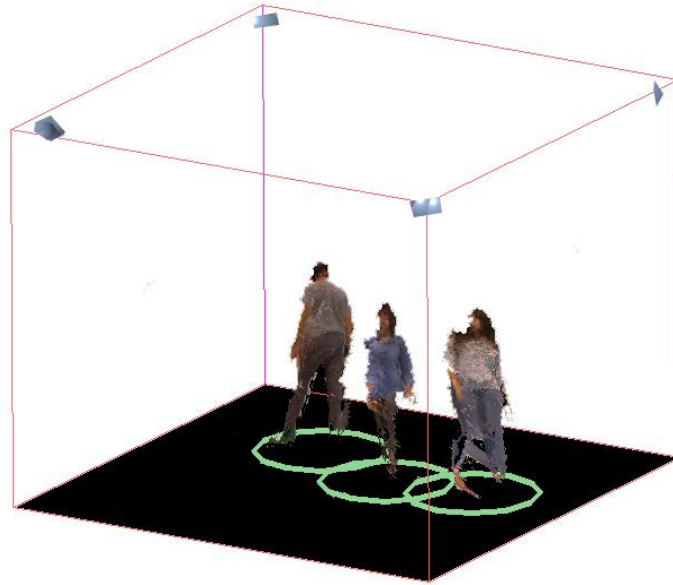


Figure 3: Setup consisted of four RGB-D sensors



Figure 4: Multiple human tracking based on geometric and color information

Except the aforementioned advanced computer vision algorithms, the tracking service builds upon the *FAMINE* middleware in order to expose tracking results to any registered third party client applications. Such a third party client applications can use a public API in order to receive that data (i.e., tracking results) and proceed on them. The public API has been defined in the interface *PeopleTrackingService* using IDL as illustrated in CodeBlock 32. In detail, the interface consists of four data structures, in which the *Person* data type contains the necessary information about the tracked individuals. The *Person* data type consists of: a) a short variable that represents the identification *id* of the tracked person, b) a float variable that describes the ellipse axes ratio, and c) a variable named *Point2f*. The *Point2f* data type contains the accurate position of the person in the room in (x, y) coordinates. In addition, the service

defines a specific sequence as a container of *Person* instances. Finally, *PeopleTrackingService* dispatches the *NotifyTracked* event to any registered client application with a frequency of 20fps. The event's arguments are the identification *id* of the tracked person as well as the relative position within the observed area.

```
#include <ami.idl>

module Test {

    interface PeopleTrackingService {

        struct Point2f{

            float x;

            float y;

        };

        struct Person{

            short id;

            float ellipse_axes_ratio;

            Point2f position;

        };

        struct MatF{

            ami::FloatSeq data;

            long cols;

            long rows;

            long channels;

        };

        struct MatU{

            ami::OctetSeq data;

            long cols;

            long rows;

            long channels;

        };

        typedef sequence<Person> PersonSeq;

        void Event_NotifyTracked (in PersonSeq actors);

    };

};
```

CodeBlock 32: “Tracking persons using a network of RGB-D cameras” service definition in IDL

For the purposes of the present case study, the *FAMINE4Android* library is used as a communication medium with the “Tracking persons using a network of RGBD cameras” service, in order to develop a mobile museum guide application that automatically presents information related to the visitor’s current position.

4.2 Museum Guide application for Android mobile devices

The Museum Guide application for Android mobile devices, aims to facilitate the visit of individuals within a museum and provide information related to their adjacent artefacts in an intuitive and user-friendly way. Actually, the main goal of the case study is to assess the integration of the proposed *FAMINE4Android* library in environments of distributed services. The following sections present the

design and explain its applicability through a usage scenario. Moreover, some implementation details are presented at the end of this chapter.

4.2.1 Design and Usage scenario

The Museum Guide application allows users to use their mobile phones in order to get access to information regarding the museum exhibits. The following usage scenario exemplifies its applicability. Firstly, the museum visitor launches the application while entering the exhibition hall. The application starts and four functionalities are presented as depicted in Figure 5. Those functionalities are: a) *navigation mode*, b) *points of interest*, c) *map*, and d) *about*.

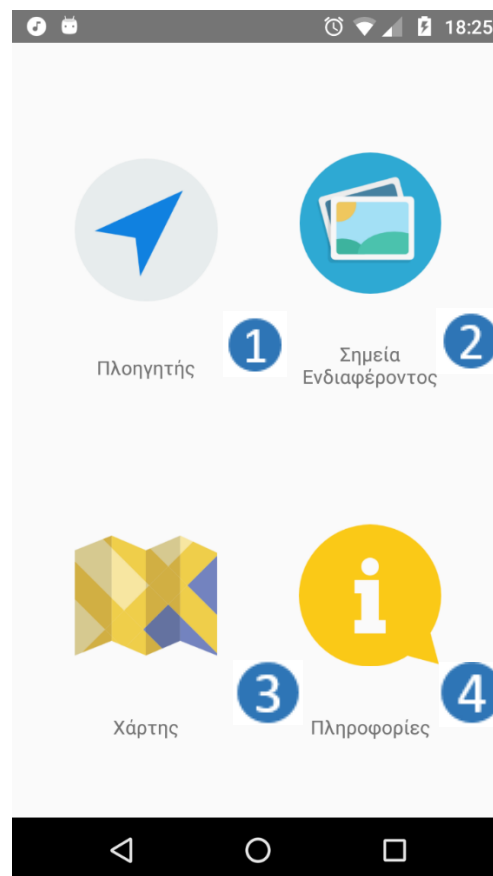


Figure 5: Main menu

Regarding the first one, *navigation mode*, when a visitor is located adjacent to an artefact, a notification on his mobile device will be received and information related to the artifact will be presented on the screen (see Figure 6). As the user moves around the exhibition area, the presented information will be updated automatically according to the distance from the nearby exhibits.

The second option, *points of interest*, presents the full list of available exhibits within the museum (see Figure 7 (2)). The user is able to interact by selecting each artefact of his interest and view further information about it. As presented in Figure 7 (1), the user has access to a representative image of the selected exhibit as well as to relevant multimedia content.

The third option, *map*, depicts the floor plan where all exhibits are digitally illustrated according to their actual position. As shown in Figure 7 (3), a moving green dot is dynamically depicted in order to reflect the tracked position of the visitor, which is calculated by the tracking system in real-time. Lastly,

the option *about* provides information about the development of the Android application and its copyrights.



Figure 6: Using the mobile application to browse the digital exhibits, based on tracking technology

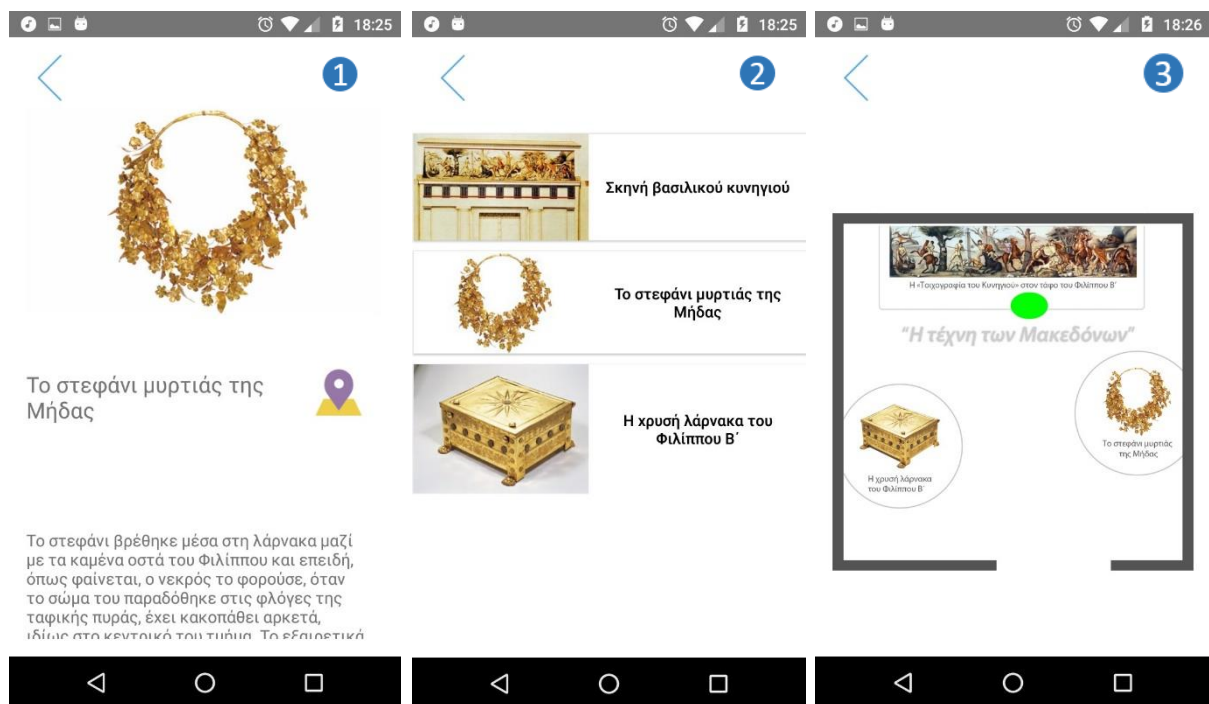


Figure 7: Museum Guide features

4.2.2 Implementation details

Within the mobile application's Java code, a dispatched event is captured by the corresponding event handler *NotifyTracked*, as it is described in CodeBlock 33. Subsequently, the event handler propagates the received information for each tracked person to the *map* and *guidance*. As depicted in Figure 7 (3), the first module, *map*, is used to update the graphical representation of the position of the tracked person within the observed area. The second module, *guidance*, checks the position of the person tracked against the position of each exhibit/point of interest. In case of intersection, the content manager receives an appropriate notification in order to present information relative to the exhibit near the user. The exhibit's position within the observed area is predefined into the application for the purposes of this case study.

```
public void NotifyTracked(final Object [] arr)
{
    Test.PeopleTrackingServicePackage.Person[] persons =
    ami.Famine.getInstance().gson.fromJson(arr[0].toString() ,

    Test.PeopleTrackingServicePackage.Person[].class);
    for (Test.PeopleTrackingServicePackage.Person p : persons)
    {
        Map.NotifyHandler(p.position.x, p.position.y);
        Guidance_Details.NotifyHandler(p.position.x, p.position.y);
    }
}
```

CodeBlock 33: Handling incoming event in Java

The implementation of this case study was seamlessly achieved through the provided, intuitive to use, Java API. Distributed programming on Android devices can effortlessly facilitated with the support of the proposed *FAMINE4Android* middleware.

4.3 Testing

In the context of assessing the reliability of the mobile Museum Guide application, a multiuser informal testing took place in the simulation space located in the Ambient Intelligence Facility of FORTH-ICS. The simulation space was customized to accommodate some copy artifacts of the “*Art of Macedonians*” collection. In details, two golden artefacts and a fresco of ancient Greece were placed around the observed area acting as real exhibits within a museum exhibition area. Four users, aged between 25 and 30 years old, were invited to participate in the preliminary user-based evaluation. They were each asked by the evaluator to download and install the application to their mobile phones. Afterwards, they were asked to launch the Museum Guide application and experiment with the four provided options. Thereafter, the evaluator asked them to walk freely within the exhibition area and show their interest to any of the three installed exhibits. All four of them were able to access information related to the nearby exhibit and iterate through the description and multimedia content that was automatically presented on their mobile screens. Additionally, they all were able to familiarize their selves with the *map* presentation and navigate easily within the exhibition area. To conclude, the case study showed that the proposed *FAMINE4Android* library contributes to the development process of mobile user-friendly applications equipped with advanced functionality in the context of distributed computing. Additionally, the case study proved that the *FAMINE4Android* extension library facilitates real-time remote communication among distributed objects running on both ordinary PCs and Android mobile devices.

5 Conclusion and Future Work

This chapter summarizes the achievements of the work reported in this thesis, discusses its findings and contributions, and outlines directions for future research.

5.1 Summary of Achievements

This thesis has presented the development of a *FAMINE* middleware extension, named *FAMINE4Android*, which aims at empowering Android mobile devices in distributed service-oriented environments. *FAMINE4Android* builds upon the *FAMINE* middleware, which caters for the creation of distributed services enabling the exposure of software and hardware resources in Aml environments. *FAMINE4Android* provides the required mechanisms and tools in order to support service discovery, event driven communication and remote procedure calls, through a seamless and intuitive Java API. Therefore, Android developers are able to develop effortlessly applications enabled with distributed computing capabilities.

FAMINE4Android employs TAO, a freely available, open-source, and standards-compliant real-time CORBA implementation in C++. Efforts have been focused on the provision of a service type agnostic solution so that any *FAMINE* service, either registered or resolved, can be integrated in a seamless and effortless manner. To this end, interoperability issues stemming from hosting native code in an Android Java application have been addressed successfully. In addition, a mechanism responsible for data exchange between Java and native code has been implemented. Such mechanism is able to exchange the format of primitive or complex data types in order to address the data type variations between Java and native code. In detail, JNI native functions were employed in order to accomplish smoothly conversion of primitive types (e.g., such as integer, float, string, etc.) between the two languages. Data transitions of complex data types as well as user or CORBA defined data sequences is also supported. The intuitive usage of the *FAMINE4Android* library was exemplified thoroughly through a presented programming tutorial and indicative examples.

The features of the *FAMINE4Android* library were demonstrated by implementing a case study application in the domain of cultural heritage. This case study refers to a *Museum Guide* application, which provides information automatically based on visitors' location. In details, the museum guide application uses the functionality provided by a *FamineTrackingService* running on Windows OS. The tracking service builds upon advanced computer vision algorithms in order to track multiple persons within exhibition spaces using a network of RGB-D cameras. The case study highlights the contribution of the *FAMINE4Android* middleware towards increasing the number of distributed computing platforms. An informal testing of the application took place in the simulation space located in Ambient Intelligence Facility of FORTH-ICS in order to assess the reliability of the proposed work. The preliminary evaluation results showed that the proposed *FAMINE4Android* library contributes to the development process of mobile user-friendly applications equipped with advanced distributed computing capabilities. Furthermore, the case study highlighted the exploitation capabilities of distributed computing hosted in heterogeneous services in Ambient Intelligence Environments.

5.2 Future Work

Regarding future work, completing the development of *tao_idl4Android* compiler is critical along with the development and integration of a mechanism able to generate Java bindings from a given IDL containing the service definition. Such functionality seems to minimize the effort required for the development of application based on distributed services that define complex data types. Moreover, it is considered crucial to extend the support for more platforms, such as Microsoft Windows devices,

to holistically address the mobile computing market. Finally, it is considered essential to design an evaluation strategy with the active contribution of Android developers in order to measure FAmINE4Android applicability and usability from an end-user perspective.

6 References

1. A simple C Client-Server in CORBA, <https://www.codeproject.com/Articles/24863/A-Simple-C-Client-Server-in-CORBA>
2. Anastasopoulos, Michalis, et al. "Towards a reference middleware architecture for ambient intelligence systems." ACM conference on object-oriented programming, systems, languages, and applications. 2005.
3. Application Binary Interface, <https://developer.android.com/ndk/guides/abis.html>
4. Blended Reality report. Technology Horizons Program (2009). Institute for the Future, Palo Alto: <http://www.iftf.org/uploads/media/SR-122~2.PDF>
5. Bouma, H. "True visions—The emergence of ambient intelligence, by E. Aarts, JL Encarnação; 2006." *Gerontechnology* 6, no. 1 (2007): 58-60. Valli, A. (2008). The design of natural interaction. *Multimedia Tools and Applications*. 38(3): 295 - 305.
6. Cisco Unified Application Environment, <http://www.cisco.com/web/developer/cuae>
7. Coulouris, G. F., Dollimore, J., & Kindberg, T. (2005). *Distributed systems: concepts and design*. pearson education.
8. Crossbar.io, <http://crossbar.io>
9. Dommel, P., Wagner, R., Edwards, R., and Doran, A. (2005). A Middleware Framework for the Adaptive Home, in S. Geros and H. Pigot (Eds). *From Smart Homes to Smart Care*, IOS Press, pp 167-173, <http://www.cse.scu.edu/~hpdommel/publications/hpd.icost05.pdf>
10. Ducatel, Ken, Marc Bogdanowicz, Fabiana Scapolo, Jos Leijten, and Jean-Claude Burgelman. "Scenarios for ambient intelligence in 2010." *Office for official publications of the European Communities* (2001). Ramos, C., Augusto, J. C., & Shapiro, D. (2008). Ambient intelligence—the next step for artificial intelligence. *IEEE Intelligent Systems*, 23(2), 15-18.
11. Erickson, T.D. (1990). *Working with Interface Metaphors*. In B. Laurel (Ed.), *The Art of Human-Computer Interface Design* (pp. 65-73). Reading, MA: Addison-Wesley Publishing Company, Inc
12. Etch, <https://cwiki.apache.org/confluence/display/ETCH>
13. Express, <http://expressjs.com/>
14. Galanakis, G., Zabulis, X., Koutlemanis, P., Paparoulis, S., & Kouroumalis, V. (2014). Tracking persons using a network of RGBD cameras. In the Proceedings of the 7th ACM International Conference on Pervasive Technologies Related to Assistive Environments (PETRA 2014), Rhodes, Greece, 27-30 May.
15. Georgalis, I. (2013). *Architectures, Methods and Tools for Creating Ambient Intelligence Environments*. PhD Thesis, University of Crete, Computer Science Department, June 2013, <https://www.didaktorika.gr/eadd/handle/10442/29423>
16. Georgalis, Y., Grammenos, D., & Stephanidis, C. (2009). Middleware for Ambient Intelligence Environments: Reviewing Requirements and Communication Technologies. In C. Stephanidis, (Ed.), *Universal Access in Human-Computer Interaction - Intelligent and Ubiquitous Interaction Environments*. - Volume 6 of the Proceedings of the 13th International Conference on Human-Computer Interaction (HCI International 2009), San Diego, CA, USA, 19-24 July, pp. 168-177. Berlin Heidelberg: Lecture Notes in Computer Science Series of Springer. Lim, A. (2001). Distributed services for information dissemination in self-organizing sensor networks. *Journal*

of the Franklin Institute, Volume 338, Issue 6, September 2001, Pages 707-727,
[http://dx.doi.org/10.1016/S0016-0032\(01\)00020-5](http://dx.doi.org/10.1016/S0016-0032(01)00020-5).

17. Ghosh, S. (2014). Distributed systems: an algorithmic approach. CRC press.
18. Henning, M. (2204). A new approach to object-oriented middleware, in IEEE Internet Computing, vol. 8, no. 1, pp. 66-75, Jan-Feb 2004.
<http://ieeexplore.ieee.org/document/1260706/>
19. <http://omniorb.sourceforge.net/omni41/omniORB/omniORB001.html>
20. <http://orbit-resource.sourceforge.net/>
21. <http://rapidjson.org/index.html>
22. <http://www.cs.wustl.edu/~schmidt/TAO.html>
23. <http://www.cs.wustl.edu/~schmidt/TAO-overview.html>
24. <http://www.jacorb.org/>
25. <http://www.json.org/>
26. <http://www.mico.org/>
27. <http://www.ois.com/Products/communications-middleware.html>
28. [https://en.wikipedia.org/wiki/IDL_\(programming_language\)](https://en.wikipedia.org/wiki/IDL_(programming_language))
29. https://en.wikipedia.org/wiki/Massively_multiplayer_online_game
30. <https://en.wikipedia.org/wiki/Peer-to-peer>
31. https://en.wikipedia.org/wiki/Service-oriented_architecture
32. <https://github.com/google/gson>
33. Java Native Interface, <https://developer.android.com/training/articles/perf-jni.html>
34. Json Framework, <https://developer.android.com/reference/android/util/JsonReader.html>
35. Koa.js: A Future-Proof JavaScript Middleware Framework,
<https://www.upwork.com/hiring/development/koa-js-a-future-proof-javascript-middleware-framework/#sm.0002ct9xk12sjfoyq3u1rtrw765ce>
36. Ling, F., Apers, P. MG., and Jonker, W. (2004). Towards context-aware data management for ambient intelligence. In: Galindo F., Takizawa M., Traunmüller R. (eds) Database and Expert Systems Applications. DEXA 2004. Lecture Notes in Computer Science, vol 3180, pp-422-431. Springer, Berlin, Heidelberg
37. McHale, C. (2007). Corba explained simply. <http://www.ciaranmchale.com/corba-explained-simply/>
38. Native Development Kit, <https://developer.android.com/ndk/index.html>
39. OSGi Alliance, <http://www.osgi.org>
40. Partarakis N., (2005). Using Ambient Intelligence Technologies for Producing and Disseminating Art. PhD Thesis. University of Crete, Computer Science Department.
41. Peleg, D. (2000). Distributed computing: a locality-sensitive approach. Society for Industrial and Applied Mathematics.
42. Rapidjson, <https://codeflu.blog/2015/02/26/understanding-rapidjson/>

43. Rio Dynamic Distributed Services, <http://www.rio-project.org/>
44. Riva G, Vatalaro F., Davide F., Alcaniz M. (2005). *New Technologies for Ambient Intelligence*; IOS Press.
45. ROS, <http://www.ros.org/core-components/>
46. Run Time Type Reflection, <http://www.rttr.org/>
47. Schmidt, D. C. and Cleeland, C. (1999). Applying patterns to develop extensible ORB middleware. *Comm. Mag.* 37, 4 (April 1999), 54-63. <http://omniorb.sourceforge.net/>
48. Simple Object Access Protocol (SOAP), <http://www.w3.org/TR/soap>
49. Slee, M., Agarwal, A., and Kwiatkowski, M. (2007). Thrift: Scalable Cross-Language Services Implementation. Technical Report, Facebook, Palo Alto, CA, USA, April 2007. <https://thrift.apache.org/static/files/thrift-20070401.pdf>
50. Tapia, D. I., Abraham, A., Corchado, J. M., & Alonso, R. S. (2010). Agents and ambient intelligence: case studies. *Journal of Ambient Intelligence and Humanized Computing*, 1(2), 85-93.
51. The ACE ORB (TAO), <http://www.cs.wustl.edu/~schmidt/TAO.html>
52. The Hydra project, <http://www.hydramiddleware.eu>
53. The Object Management Group (OMG), <http://www.omg.org>
54. WAMP, <http://www.wampserver.com/en/>
55. Weiser, M. (1993). Some computer science issues in ubiquitous computing. *Communications of the ACM* 36 7, pp. 75-84
56. Weiser, Mark. "Hot topics-ubiquitous computing." *Computer* 26, no. 10 (1993): 71-72.
57. ZeroC, <http://www.zeroc.com>
58. Zhang, C., and Jacobsen, H.-A. (2004). Requirements Analysis for Middleware Aspects. Technical Report, Middleware Systems Research Group, University of Toronto, July 2004 <https://pdfs.semanticscholar.org/12b6/45e6a65592b9aeffd39a03117a9c886ff961.pdf>

APPENDIX A: Step-by-step procedure of porting TAO to Android architecture

The implementation of the presented middleware library relies on *TAO*. This section presents the step-by-step procedure that was followed in order to port the *TAO* to the Android architecture using the Android Native Development Kit (NDK) toolset (see 3.1.2).

1	We have installed GNU make 3.79. When using ACE's per-platform configuration method we must use GNU make otherwise ACE will not compile successfully.
2	Afterwards we have downloaded the ACE+TAO package <pre>tar -xvf ACE+TAO-6.1.8.tar</pre>
3	Then create a clone directory for the host <pre>cd ACE_wrappers mkdir -p build/HOST ./bin/create_ace_build build/HOST</pre>
4	In addition we have created a configuration file, <code>\$ACE_ROOT/ace/config.h</code> that includes the appropriate platform/compiler-specific header configurations from the ACE source directory. <pre>echo '#include "ace/config-linux.h"' > build/HOST/ace/config.h echo 'include \$(ACE_ROOT)/include/makeinclude/platform_linux.GNU' > build/HOST/include/makeinclude/platform_macros.GNU</pre>
5	Then we have changed to the build/HOST directory. After we have set the ACE_ROOT environment variable to point to build/HOST. Bear in mind that we should also build the gperf perfect hash function generator application and the host tools. <pre>cd build/HOST export ACE_ROOT=\$PWD make -C ace make -C apps/gperf/src</pre>
6	Define the TAO root, in order to build TAO_IDL tool <pre>export TAO_ROOT=\$PWD/TAO make -C TAO/TAO_IDL</pre>
7	The cross compilation method requires the definition of the TARGET platform. Consequently, we have created a clone directory for the target <pre>cd ../../ mkdir -p build/TARGET ./bin/create_ace_build build/TARGET</pre>
8	We configured the target build <pre>echo '#include "ace/config-android.h"' > build/TARGET/ace/config.h</pre>
9	Then we created a build configuration file, <code>\$ACE_ROOT/include/makeinclude/platform_macros.GNU</code> , that contained the appropriate platform/compiler-specific Makefile configurations.

	<pre>echo 'include \$(ACE_ROOT)/include/makeinclude/platform android.GNU'> build/TARGET/include/makeinclude/platform_macros.GNU</pre>
10	<p>We can point out that we can override the default values by adding several lines to our platform_macros.GNU file. Assuming \$(HOST_ROOT) is set to the location of our host build where we previously built gperf and tao_idl, we can change the target build by adding the following lines in order TARGET build to use the HOST IDL compiler and gperf tools:</p> <pre>nano build/TARGET/include/makeinclude/platform_macros.GNU TAO_IDL := \$(HOST_ROOT)/bin/tao_idl TAO_IDLFLAGS += -g \$(HOST_ROOT)/bin/gperf TAO_IDL_DEP := \$(HOST_ROOT)/bin/tao_idl INSTALL_PREFIX = \$(ACE_ROOT)/output static_libs_only=1</pre>
11	<p>Then we set the ACE and TAO root paths</p> <pre>cd build/TARGET export ACE_ROOT=\$PWD export TAO_ROOT=\$PWD/TAO</pre>
12	<p>Then we have created the folder which included the outcome dynamic shared libraries of the building. Headers had been installed to \$INSTALL_PREFIX/include, executables to \$INSTALL_PREFIX/bin, documentation and build system files to \$INSTALL_PREFIX/share and libraries to \$INSTALL_PREFIX/lib</p> <pre>mkdir output</pre>
13	<p>Set Android architecture to arm.</p> <pre>export ANDROID_ARCH=arm</pre>
14	<p>Export the path of Android Native Document Kit</p> <pre>export NDK=/path/android-ndk-r8e</pre>
15	<p>Exportation of the platform arch directory in order to set the standard alone toolchain. The preferred standard alone version was 4.4.3 and the android API level was 14.</p> <pre>export SYSROOT=\$NDK/platforms/android-14/arch-\$ANDROID_ARCH cd \$NDK ./build/tools/make-standalone-toolchain.sh --toolchain=arm-linux-androideabi-4.4.3 -- arch=arm --platform=android-14 --install-dir=./arm_tools --system=linux-x86_64</pre>
16	<p>Set the tools and the bin folder of the Android Native Development Kit</p> <pre>export NDK_TOOLS=/path/arm_tools export PATH=\$PATH:\$NDK_TOOLS/bin</pre>
17	<p>Afterwards we have changed to the HOST directory and set the HOST root</p> <pre>cd \$ACE_ROOT/../../build/HOST export HOST_ROOT=\$PWD</pre>
18	<p>Then we run the perl script \$ACE_ROOT/bin/mwc.pl in the TAO_ROOT directory to try all the tests</p>

	<pre>cd \$TAO_ROOT perl \$ACE_ROOT/bin/mwc.pl TAO_ACE.mwc -type gnuace</pre>
19	<p>Then we have exported the library folder, that is included in the HOST folder, to the LD_LIBRARY_PATH. Because ACE builds shared libraries, LD_LIBRARY_PATH has to be set to the directory where binary version of the ACE library is built into</p> <pre>export LD_LIBRARY_PATH=\$LD_LIBRARY_PATH:\$HOST_ROOT/lib</pre>
20	<p>After concluded all the above steps we run 'make' and then 'make install'.</p> <p>This had built the ACE library, tests, the examples, and the sample applications.</p>

APPENDIX B: Programming Tutorial

This section presents a step-by-step programming tutorial for developers who would like to use the *Famine4Android* library in order to develop mobile applications equipped with distributed computing capabilities. The tutorial provides indicative to exemplify *Famine4Android* usage.

B.1. Generating client/server stubs from .idl file

Firstly, a service definition in IDL is required in order to implement or use a distributed service. Secondly, the *tao_idl4Android* compiler is used to generate the corresponding *stub* and *skeleton* files for either service implementation or use respectively. The files generated by the *tao_idl4Android* compiler taking as input the *.idl* file are imported in the Android project as depicted in the following section.

B.2. Service type declaration

When the Android developer creates a new Android project, in which the support of C++ standards is selected, the Android Studio auto-generates the native debug configuration file, which is called *native-lib.cpp*. This file is located under the `<project path>/app /src/main/cpp` directory. In this file, the Java developer should write the proper code for the service type declaration. As depicted in CodeBlock 34, the service type is declared within the *ServiceTypes* static block as described in 3.2.2.1 section. Thorough examples are presented in the following sub-sections B.4 and B.5.

```
#include "ServiceTypes.h"
#include "FamineManager.h"

ServiceTypes {
    Service type registration instructions
}
```

CodeBlock 34: ServiceTypes scope

B.3. Set up Programming Environment

B.3.1. Configure Android Studio: NDK and Build Tools

The following components are prerequisites to the proper functionality of the proposed library:

- *The Android Native Development Kit (NDK)*: a toolset that allows the use of C and C++ code with Android, providing platform libraries that allow the management of native activities and access physical device components.
- *CMake*: an external build tool that works alongside *Gradle*, an advanced build toolkit that automate and manage the build process. There is no need for this component unless there is plan for *ndk-build* use.
- *LLDB*: the debugger Android Studio uses to debug native code.

The Android Software and Development Kit (SDK) Manager can be used for the installation of SDK tools, platforms, and other components that help for the development of Android app. Figure 8 depicts the selection of the aforementioned components within the Android Studio SDK tools.

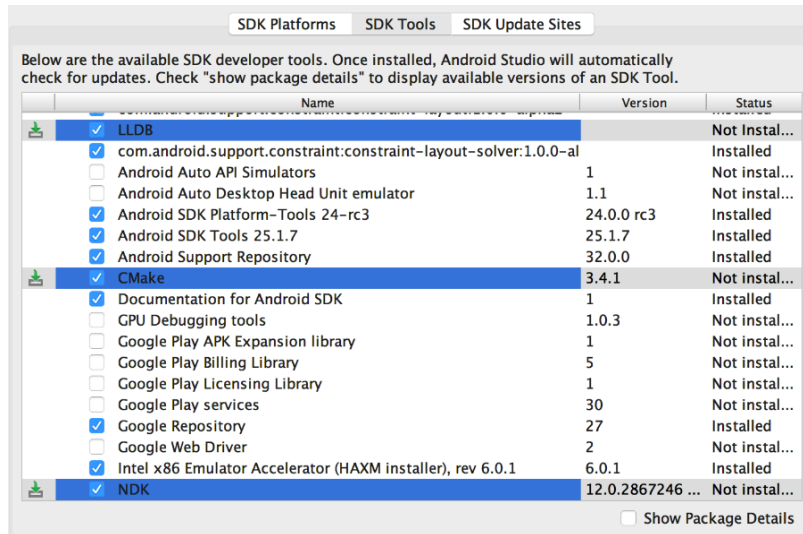


Figure 8: SDK tools

B.3.2. Creation of a new Android Project with C/C++ Support

According to the Android Studio documentation, creating of a new project with support for native code is similar to creating any other Android Studio project. However, there are a few additional steps:

- C++ Support checkbox has to be selected within the configuration wizard,
- Minimum SDK has to be set to API 14: Android 4.0 (IceCreamSandwich),
- As shown in the Figure 9, the following options has to be enabled in the C++ Support customization section:
 - **C++ Standard**, which enables C++11 features,
 - **Exceptions Support**, which enables the C++ exception handling,
 - **Runtime Type Information Support**, which enables support for RTTI.

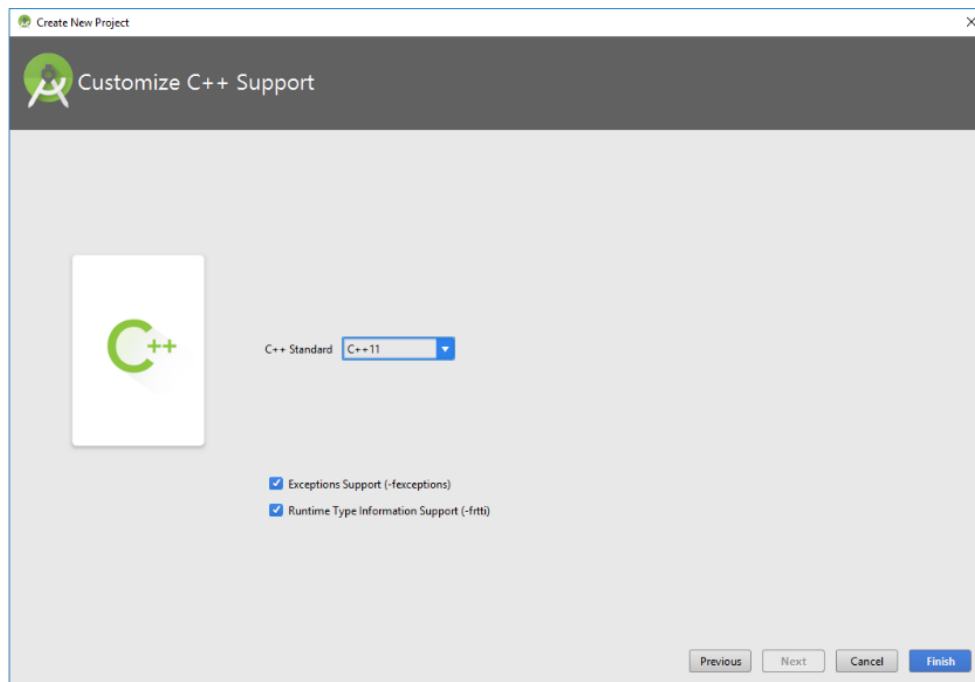


Figure 9: Customize C++ Support

B.3.3. Configure Build.Gradle

As depicted in Figure 10, some effort is needed in order to configure the NDK, CMake, and linking properties to the *Famine4Android* library. In detail, the following sections has to be set appropriately as listed below:

- As illustrated in Figure 10 (1), the GNU STL (shared library) property should be set to *gnustl_shared*. In addition, gcc compiler flags should be set to *-std=c++11 -frtti -fexceptions*. Moreover, the supported toolchain should be set to *gcc*
- The Application Binary Interface (ABI) should be set to *armeabi-v7a* as depicted in As illustrated in Figure 10 (2). ABI defines how an application's machine code is supposed to interact with the system at runtime. *Famine4Android* is only compatible with *armeabi-v7a* architecture.
- The configuration of CMake in order to import the *Famine4Android*'s required libraries (.so) should be as shown in Figure 10 (3).

```

android {
    compileSdkVersion 25
    buildToolsVersion "25.0.2"
    defaultConfig {
        applicationId "com.example.amidemo.test"
        minSdkVersion 14
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
        externalNativeBuild {
            cmake {
                cppFlags "-std=c++11 -frtti -fexceptions"
                arguments "-DANDROID_STL=gnustl_shared", "-DANDROID_TOOLCHAIN=gcc"
            }
        }
        ndk {
            abiFilters 'armeabi-v7a'
        }
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
    externalNativeBuild {
        cmake {
            path "CMakeLists.txt"
        }
    }
    sourceSets.main {
        jniLibs.srcDirs = [ System.getenv('FAMINE_ROOT') + '\\Android\\famine\\lib',
                           System.getenv('FAMINE_ROOT') + '\\Android\\famine_proxy\\lib',
                           System.getenv('FAMINE_ROOT') + '\\Android\\json_serialize\\lib',
                           System.getenv('FAMINE_ROOT') + '\\Android\\rttr\\lib',
                           System.getenv('FAMINE_ROOT') + '\\Android\\tao\\lib' ]
    }
}

dependencies {
    compile fileTree(include: ['*.jar'], dir: 'libs')
    androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
}

```

Figure 10: Build.Gradle configuration

B.3.4. Configure CMakeLists.txt

The update of the CMakeLists.txt file, which is located under the `<project path>/app/src` directory is required (see Figure 11). The CMakeLists.txt, located in the *FAMiNE4Android* installation folder should be included in the CMakeLists.txt of the Android project. This CMakeLists.txt is responsible for the specification of the necessary header files and shared libraries that are required in order to develop distributed Android mobile applications. In addition, the configuration of the folder that includes the service's generated *skeleton* or *stub* files is required as depicted at the end of the marked area (see Figure 11).

```
cmake_minimum_required(VERSION 3.4.1)

file(TO_CMAKE_PATH $ENV{FAMINE_ROOT} FAMINE_ROOT)
include(${FAMINE_ROOT}/Android/CMakeLists.txt)

include_directories(src/main/cpp/ChaosService/include )

# Creates and names a library, sets it as either STATIC
# or SHARED, and provides the relative paths to its source code.
# You can define multiple libraries, and CMake builds it for you.
# Gradle automatically packages shared libraries with your APK.

add_library( # Sets the name of the library.
            native-lib

            # Sets the library as a shared library.
            SHARED
```

Figure 11: CMakefile configuration

B.3.5. Configuration AndroidManifest.xml

Regarding the AndroidManifest.xml, which is located under the `<project path>/app/src/main/` directory, the internet permission should be enabled, as illustrated in Figure 12.



```
manifest uses-permission
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.amidemo.test">

    <uses-permission android:name="android.permission.INTERNET" />

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="Test"
        android:supportRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

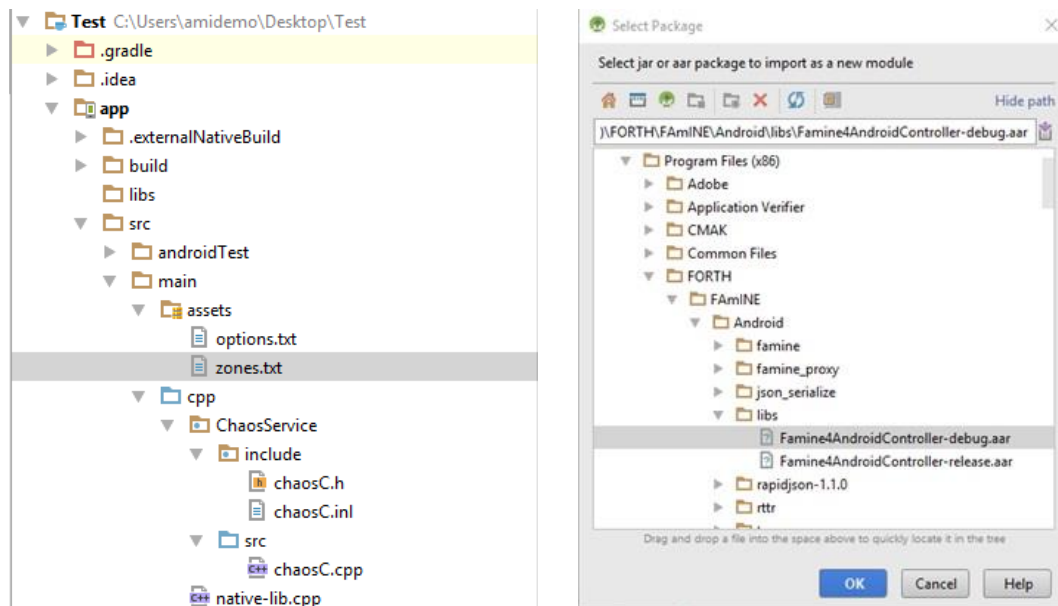
Figure 12: AndroidManifest configuration

B.3.6. Configuration of assets folder in the Android project

The creation of a specific folder, named *assets*, is required in order to define the network locations (e.g., endpoints) of the Implementation Repository Server and the Naming Service. Java developer is responsible to create that folder and add two text files, the *zones.txt* and the *options.txt* (see Figure 13 top-left).

B.3.7. Import FAmiNE4Android Controller

The developer has to import *Famine4AndroidController* AAR package in order to employ the middleware functionality provided by the *Famine4Android* library. To this end, the Java developer has to select from the Android project the “New” option and then the “Import Module” option. Afterwards, the Java developer selects from the *FAmiNE4Android* installation folder where the ARR package as illustrated in Figure 13 (top-right). The next step is the update of the current (e.g., *app*) module’s dependencies in order to add the *Famine4AndroidController*. In detail, the Java developer select the tab Dependencies from the Project Structure menu and picks the imported *Famine4AndroidController* as depicted in Figure 13 (bottom).



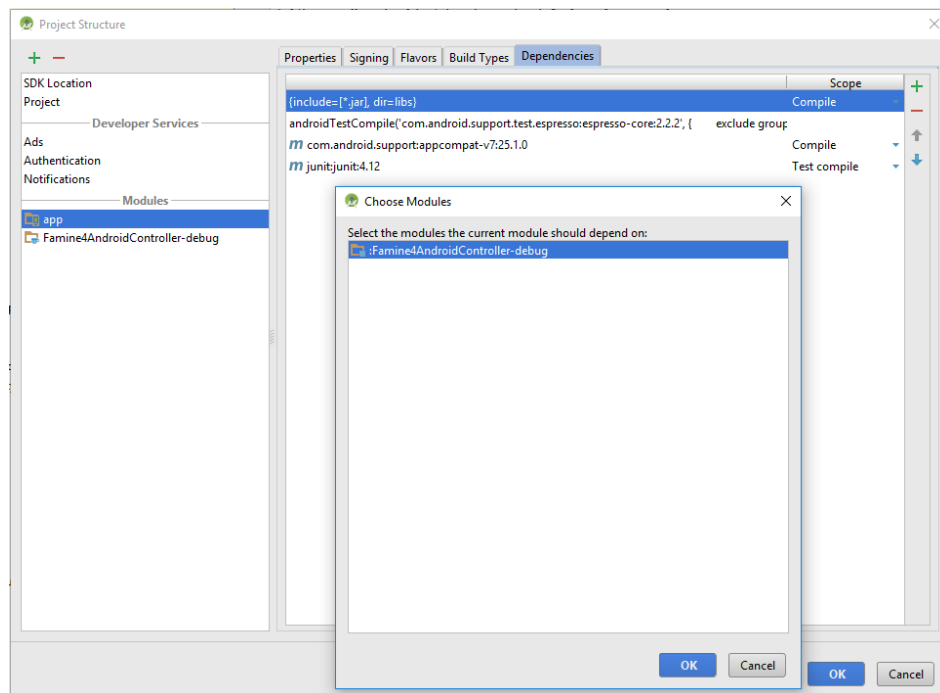


Figure 13: Assets folder configuration (left), Import AAR package (right), Import Module Dependency (bottom)

B.3.8. Import Gson 2.8.0 library

Java developer has to import the *Gson 2.8.0* or newer Java library (required for the conversion of Java Objects into their JSON representations and vice-versa) in the Android Studio project. In detail, Java developer navigates to the project structure window, selects the tab Dependencies and then select Library dependency. In the window opened, Java developer searches for the “com.google.code.gson:gson:2.8.0” library, picks the corresponding result and then clicks “OK” (see Figure 14).

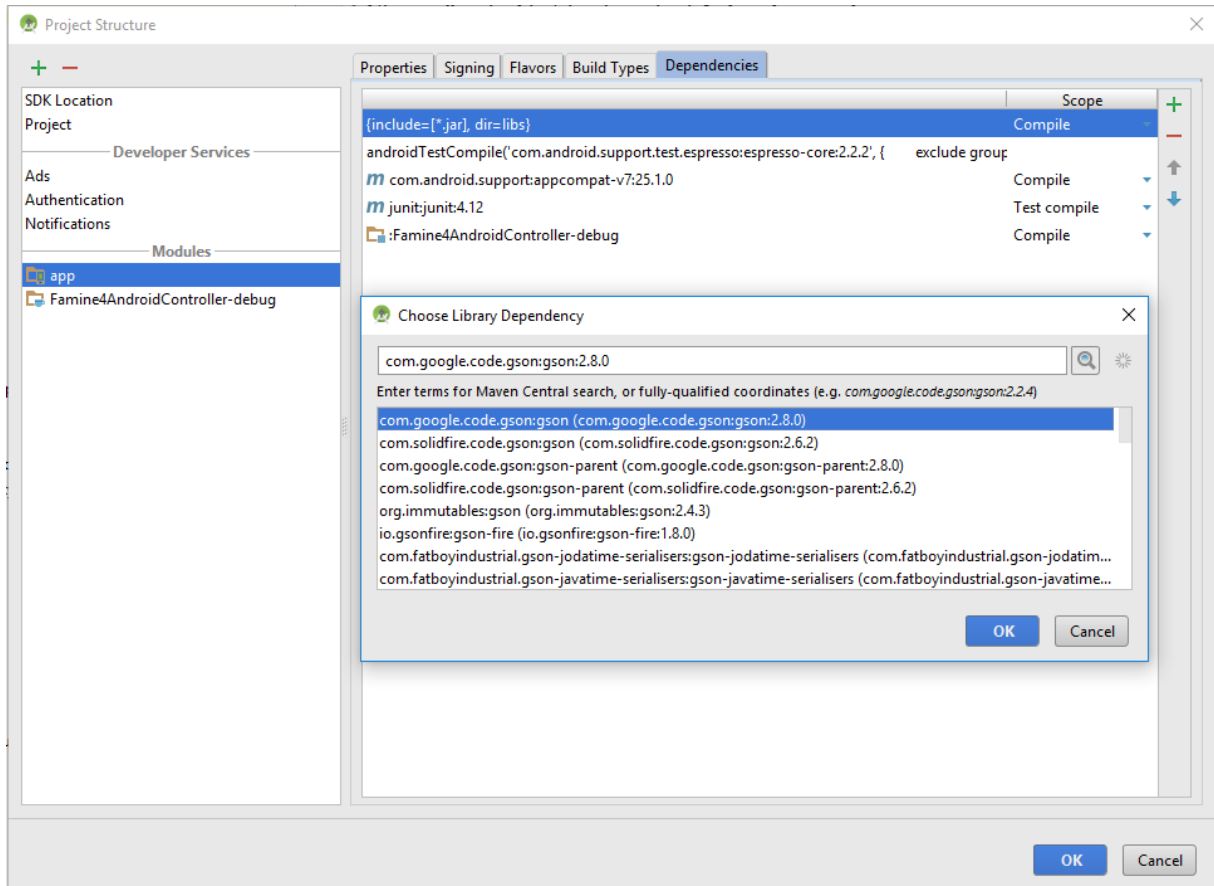


Figure 14: Import Library Dependency

B.4. Service implementation

This section presents the service implementation procedure. The *Chaos* service interface is illustrated in CodeBlock 35 and is used as an example. The *Chaos* IDL file is compiled with the *tao_idl4Android* compiler in order to generate corresponding *stub* and *skeleton* files. These generated files are: a) *chaosC.h*, b) *chaosC.cpp*, c) *chaosC.h*, d) *chaosS.cpp*, e) *chaosI.h*, f) *chaosI.cpp* and g) *chaosC.inl*. These files are imported into the Android Studio project by the Java developer as depicted in Figure 15.

```
#include <ami.idl>

module demo {
    interface chaos {
        enum Priority {PR_INFO, PR_WARNING, PR_ERROR};

        struct A {
            Priority pr;
            string a_msg;
            long number;
        };

        typedef sequence<A> ASeq;
    };
};
```

```

    struct B {
        A msg;
        string b_msg;
        float humidity;
        long temperature;
        boolean pressed;
    };
    typedef sequence<B> BSeq;

    struct C {
        string c_msg;
        BSeq bs;
        ami::LongSeq numbers;
    };
    typedef sequence<C> CSeq;

    struct D {
        string d_msg;
        C ch;
    };

    //method
    D method_chaos_all (in D d);

    //events
    void Event_earthquake (in D d);
};

```

CodeBlock 35: Service Interface “Chaos”

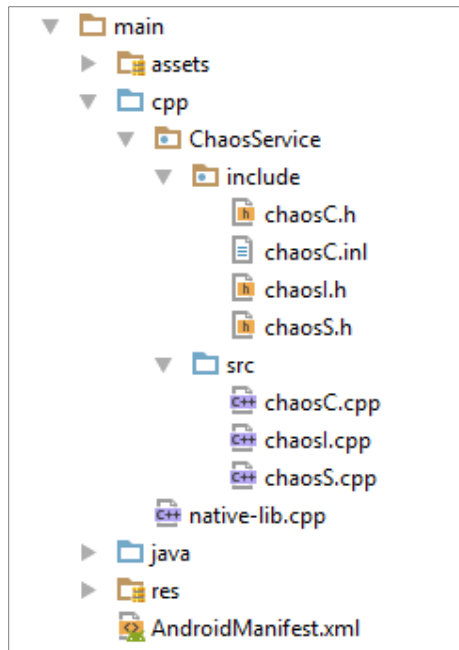


Figure 15: *Chaos* generated files Imported to Android Studio project

Afterwards, the configuration of the class service's type file is required. The declaration of the service type in *ServiceTypes*, based on the *Chaos* service example, is illustrated in Figure 16.

```
#include "ServiceTypes.h"
#include "FamineManager.h"
#include "ChaosC.h"
#include "ChaosI.h"

ServiceTypes {
    FamineManager::getInstance()->RegisterToMapTypeInfo<demo::chaos>("demo::chaos");

    FamineManager::getInstance()->RegisterToMapServantPair(&(FamineManager::createInstance<demo_chaos_i>), "demo::chaos");
}
```

Figure 16: Service type declaration for *Chaos* service

In addition, the corresponding service configuration in the CMakeList.txt is needed. CMakeList.txt is located under the <project path>/app directory of the Android Studio project (see Figure 17).

```
add_library( ChaosService-lib
    SHARED
    src/main/cpp/ChaosService/src/ChaosC.cpp
    src/main/cpp/ChaosService/src/ChaosI.cpp
    src/main/cpp/ChaosService/src/ChaosS.cpp)

target_link_libraries( ChaosService-lib ${famine-libs})

target_link_libraries( # Specifies the target library.
    native-lib

    # Links the target library to the log library
    # included in the NDK.
    ${log-lib}
    ${famine-libs}
    ChaosService-lib )
```

Figure 17: Configuration of the CMakeList.txt file for the *Chaos* service

In order to register a service, a Java class has to be created in order to provide the actual implementation (e.g., functionality) of the methods defined in the service interface. The example service *Chaos* declares one method called *method_chaos_all*, which takes as argument a complex type *D* and returns a complex type *D* as well. The *ChaosImplementation* class provides the implementation of the given method *method_chaos_all*, as illustrated in Figure 18.

```
public class ChaosImplementation {
    public D method_chaos_all(D d) {
        Log.d(" ", " "+d.d_msg);
        for(int i =0; i< d.ch.bs.length; i++) {
            Log.d(" ", "c_msg: " + d.ch.c_msg);
            Log.d(" ", "numbers: " + d.ch.numbers[i]);
        }

        for(int i =0; i< d.ch.bs.length; i++) {
            Log.d(" ", "B pressed:" + d.ch.bs[i].pressed+" B temperature:"+d.ch.bs[i].temperature+" B humidity:"+d.ch.bs[i].humidity+" B b_msg:"+d.ch.bs[i].b_msg);
            Log.d(" ", " A a_msg:"+d.ch.bs[i].msg.a_msg+" A number:"+d.ch.bs[i].msg.number+" A pr:"+d.ch.bs[i].msg.pr);
        }

        A _a = new A();
        _a.pr = Priority.PR_ERROR;
        _a.number = 23;
        _a.a_msg = "Hello";

        B _b = new B();
        _b.msg = _a;
        _b.b_msg = "World";
        _b.humidity = (float) 23;
        _b.pressed = true;
        _b.temperature = 12;

        C _c = new C();
        _c.bs = new B[1];

        int arr [] = {3,4};
        _c.bs[0] = _b;
        _c.numbers = arr;
        _c.c_msg = "!";

        D _d = new D();
        _d.ch = _c;
        _d.d_msg = "See you!";

        return _d;
    }
}
```

Figure 18: Implementation of *ChaosImplementation* Java class

Finally, the native *RegisterService* method is called taking the following arguments (see Figure 19):

- an array which contains the event names that will be exposed to the registered clients
- the service identifier *id*
- the context name of the current execution environment
- an instance of a Java class which implements the methods provided from the service interface

```
public class MainActivity extends AppCompatActivity {
    static {
        ami.Famine.loadLibraries();
        System.loadLibrary("native-lib");
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        ami.Famine.getInstance().Initialize(this);

        String [] eventsChaos = {
            "earthquake"
        };

        ami.Famine.getInstance().RegisterService(eventsChaos, "demo:chaos", "Chaos", new ChaosImplementation());
    }
}
```

Figure 19: Register service *Chaos* in Java

B.5. Using a service

This section describes the procedure of resolving a service. Taking as example the previous service interface *Chaos*, only the generated files *chaosC.h*, *chaosC.cpp*, *chaosC.h*, and *chaosC.inl* are required. Accordingly, the generated files must be imported to the Android Studio project under the *<project path>/app/main/cpp* directory as illustrated in Figure 20. The corresponding service type declaration within the *ServiceTypes* scope is illustrated in Figure 21.

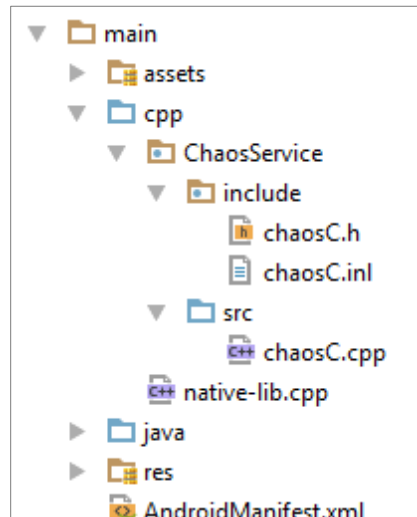


Figure 20: *Chaos* generated files Imported to Android Studio project (resolve)

```
#include "ServiceTypes.h"
#include "FamineManager.h"
#include "ChaosC.h"

ServiceTypes {
    FamineManager::getInstance()->RegisterToMapTypeInfo<demo::chaos>("demo::chaos");
}
```

Figure 21: Service type declaration for *Chaos* service (resolve)

In addition, the corresponding service configuration in the *CMakeList.txt* is needed. *CMakeList.txt* is located under the *<project path>/app* directory of the Android Studio project (see Figure 22)

```
add_library( ChaosService-lib
    SHARED
    src/main/cpp/ChaosService/src/ChaosC.cpp

target_link_libraries( ChaosService-lib ${famine-libs})

target_link_libraries( # Specifies the target library.
    native-lib

    # Links the target library to the log library
    # included in the NDK.
    ${log-lib}
    ${famine-libs}
    ChaosService-lib )
```

Figure 22: Configuration of *Chaos* service in the *CMakeList.txt* file

Finally, the native *Resolve* method is called taking the following arguments (see Figure 23):

- an array with the event names of interest
- the service identifier *id*
- the context name in which the resolved service executes
- an instance of a Java class in which the Java developer has implemented the corresponding event handlers

```
package com.example.amidemo.test;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

import com.example.amidemo.test.demo.ChaosEventHandler;

public class MainActivity extends AppCompatActivity {

    static {
        ami.Famine.loadLibraries();
        System.loadLibrary("native-lib");
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        ami.Famine.getInstance().Initialize(this);

        Object [] eventsChaos = {
            "earthquake"
        };

        ami.Famine.getInstance().ResolveService(eventsChaos,"demo::chaos","chaos", new ChaosEventHandler());
    }
}
```

Figure 23: Example of service usage/resolve

B.6. Sending events

This section presents the procedure of sending events facilitated by the *SendEvent* method. The *SendEvent* method take the following arguments:

- the service identifier *id*
- the context name
- the name of the event that is going to be sent
- an array type of Object which contains the event arguments

For example, the *Chaos* service interface provides the *earthquake* event, which takes as argument a struct type of D. Hence, the appropriate usage of the *SentEvent* is illustrated in Figure 24.


```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    ami.Famine.getInstance().Initialize(this);

    String [] eventsChaos = {
        "earthquake"
    };

    ami.Famine.getInstance().RegisterService(eventsChaos,"demo::chaos","Chaos",new ChaosEventHandler());

    A _a = new A();
    _a.pr = Priority.PR_ERROR;
    _a.number = 23;
    _a.a_msg = "Good";

    B _b = new B();
    _b.msg = _a;
    _b.b_msg = "Morning";
    _b.humidity = (float) 80;
    _b.pressed = true;
    _b.temperature = 17;

    C _c = new C();
    _c.bs = new B[1];

    int arr [] = {32,1,7,344};
    _c.bs[0] = _b;
    _c.numbers = arr;
    _c.c_msg = "!";

    D _d = new D();
    _d.ch = _c;
    _d.d_msg = "Bye!";

    ami.Famine.getInstance().SendEvent("demo::chaos","Chaos","earthquake", new Object[]{_d});
}

```

Figure 24: Send Event

B.7. Receiving events

This section presents the procedure of receiving events. For this purpose, a Java class is needed in order to handle the received event. For example, the *Chaos* service interface provides the *earthquake* event. The declaration of the corresponding event handler is illustrated in Figure 25.

```

public class ChaosEventHandler
{
    public void earthquake(Object [] arr) {

        Log.d("", "*****earthquake*****");

        D d = ami.Famine.getInstance().gson.fromJson(arr[0].toString(), D.class);

        Log.d("", "d.msg: "+d.d_msg+" d.ch.c_msg: "+d.ch.c_msg);

    }
}

```

Figure 25: *earthquake* event handler

B.8. Call Function

Regarding the process of remote procedure call, the *CallFunction* method is required. The *CallFunction* method takes the following arguments:

- the service identifier *id*
- the context name in which the resolved service executes
- the name of the function that is going to be invoked
- a Java Object array, which contains the parameter list

Concerning the *Chaos* service interface, the *earthquake* method is providing. An example *CallFunction* method invocation is illustrated in Figure 26.

```
ami.Famine.getInstance().ResolveService(eventsChaos, "demo::chaos", "Chaos", new ChaosImplementation());

A _a = new A();
_a.pr = Priority.PR_ERROR;
_a.number = 23;
_a.a_msg = "Good";

B _b = new B();
_b.msg = _a;
_b.b_msg = "Morning";
_b.humidity = (float) 80;
_b.pressed = true;
_b.temperature = 17;

C _c = new C();
_c.bs = new B[1];

int arr [] = {32,1,7,344};
_c.bs[0] = _b;
_c.numbers = arr;
_c.c_msg = "!";

D _d = new D();
_d.ch = _c;
_d.d_msg = "Bye!";

Object returned_value = ami.Famine.getInstance().CallFunction("demo::chaos", "Chaos", "earthquake", new Object[] {_d});
D obj = ami.Famine.getInstance().gson.fromJson(returned_value.toString(), D.class);
Log.e("", "d_msg = " + obj.d_msg + " c_msg = " + obj.ch.c_msg + " arr.length = " + obj.ch.bs.length);
```

Figure 26: Invocation example of the *earthquake* method