Micro-agents on Android: Interfacing Agents with Mobile Applications

Christopher Frantz, Mariusz Nowostawski, and Martin K. Purvis

Department of Information Science, University of Otago, New Zealand
{cfrantz,mnowostawski,mpurvis}@infoscience.otago.ac.nz

Abstract. The comparatively recent move towards smartphones, and along with this new operating systems, such as Android, opens up new opportunities and offers potential to build new mobile agent-based applications.

Android gives applications access to a wide-ranging set of sensors and different communication channels – realizing the notion of nomadic computing – and supports a concurrent application layout based on loosely coupled components. This combination makes it particularly suitable for agent-based applications. Yet, it has several limitations: Android is not a multi-agent system on its own behalf and does not consistently employ loose coupling to give access to its capabilities.

To address those concerns we have ported our lightweight μ-agent framework μ^2 to the Android platform and directly interfaced it with Android platform facilities. This offers mutual benefits: agent-based applications can access Android functionality in a loosely coupled and unified fashion, while at the same time allowing the developer to consistently think in an agent-oriented manner. Android can use the μ-agent platform as a lightweight middleware module to build distributed applications in a hybrid fashion.

We present our system architecture, called *Micro-agents on Android* or *MOA*, and describe an example application using this approach as well as a performance benchmark. We further outline potential application areas and contrast it to existing approaches to build multi-agent applications on Android.

Keywords: multi-agent systems, mobile applications, μ-agents, micro-agents, agent organisation, android, MOA, benchmark.

1 Introduction

The consideration of mobile devices in multi-agent systems has often been limited to the provision of a downsized derivate of the full multi-agent system implementation (for Java implementations typically targeting Java 2 MicroEdition (J2ME)), resulting in limited performance and a reduced feature set. Examples for this are JADE-LEAP [9] and 3APL-M [8].

The current transition from feature phones (with closed software platforms) to the increasingly popular smart phones (open platforms, 3rd party developers and applications as a central feature) shows a significant change of the potential to use of agent-based applications on mobile devices. Smartphones come with a capability set that is foreign to regular stationary systems. This includes multi-modal interfaces, a wide-ranging sensory inputs (e.g. accelerometers, gyroscope, camera(s), GPS) and various communication channels (such as Internet, SMS, Bluetooth) which makes *nomadic computing* a realizable possibility. The 'smartness' of applications on those devices doesn't typically derive from sophisticated intelligent features, but instead, from a meaningful combination of those new capabilities in a both flexible and efficient manner. Consequently, many applications on those devices have a mash-up character (for example in the context of location-based services) with a stress on reusable application components.

A promising approach to facilitate the smooth composition of those application elements is provided by the increasingly deployed mobile application platform Android [5][1] which enforces a modeling paradigm of asynchronously communicating loosely coupled application components.

Android's platform and organisation shows some similarity to the principles of multi-agent systems. However, Android does not fully relieve application developers from low-level aspects such as interacting with actual sensors or communication handling, and demands for an explicit handling of threads to avoid applications with poor responsiveness or performance problems. Thus, apart from the wide-ranging functionality and increased computing power available, smartphones still demand careful software engineering and cannot afford straightforward translation of heavyweight agent concepts directly to them – even to those running Android.

We think that lightweight efficiently communicating μ-agents are a useful approach to provide a symbiotic advantage for mobile technologies and allow an efficient implementation of agent-based applications on Android-based mobile devices by

- providing an organisational model to structure application functionality,
- transparently interfacing with Android application components,
- offering better performance than Android's builtin inter-component communication mechanism (which is to be shown in this work), and
- serving as a light-weight middleware towards Android application components to facilitate distributed applications.

To show this potential we first introduce Android's concept of application components and the interaction mechanisms. Then we introduce our μ-agent architecture and show its potential to interoperate with Android. Later, we describe the architecture of 'Micro-agents on Android' (MOA). We describe an actual application based on MOA to demonstrate its use for application development. We point out potential application areas of MOA based on its flexibility and interfacing qualities. Finally we relate it to existing approaches to use agents on Android.

[1] Android has increased its market share (for smartphone operating systems) to about 33 percent in Q4 2010 – from about 8.7 percent in Q4 2009 [2].

2 Android and μ-Agents

2.1 Android Architecture and Developmental Principles

Android [5] is a Linux-based software stack and application execution environment for use on mobile devices. It comes with a comprehensive set of libraries for a wide range of aspects such as security and GUI development.

Applications themselves, including the built-in ones such as the phone application, are composed of a dynamically linked combination of *application components*. Android defines four basic types of application components, namely: activities, services, broadcast receivers and content providers [4]. *Activities* run in the foreground, are rather short-running, present a user interface, and can directly interact with the user. Multiple activities can be composed to create more comprehensive applications (e.g. wizards). *Services* complement activities, as they run in the background and are relatively long-running. *Broadcast receivers* are started and run upon announced broadcasts (e.g. indicating system start or received SMS). Broadcast receivers can then start activities or services and are destroyed immediately after execution. *Content providers* maintain storage for specific data sets (e.g. the phone contacts) and allow access by other components.

Activation and communication between those components is done asynchronously via messages, called *intents*[2]. Android interprets intents as an abstract request specification. As intents themselves represent a generic, dynamically typed data structure, they can hold arbitrary application-defined content and allow late runtime binding. This mechanism ensures loose coupling of the application components.

Intents can be sent either in an explicit manner, using the target component's class name, or in a more elaborate implicit manner. Implicitly resolved intents can contain either of the following elements which allows to match intents against so-called *Intent Filters* linked to individual application applications registered with the Android instance. Those can include *Actions*, which the target component needs to perform (such as calling (CALL)); *Data*, which are uniform resource identifiers (such as tel://7843982); and finally *Categories*, which indicate an alternative for target component resolution and describe characteristics of the target application (e.g. BROWSABLE indicates that the target activity can be invoked by a browser). Along with this, intents can encapsulate arbitrary data (so-called *extras*) passed as key/value-pairs.

Comprehensive information on Android's architecture and details on the applications components can be found under [5].

2.2 The μ-Agent Concept and Its Implementation

μ-agents are goal-directed, autonomously acting entities without a particular prescribed internal architecture. However, we would expect the architecture to support the notion of hierarchical agent levels of abstraction – i.e. μ-agents may contain within their internal architecture other, more elementary μ-agents – and we

[2] The only exceptions are content providers. Intents are not used for the activation of content providers; in this case a *content resolver* is used.

expect interactions to be based on efficient asynchronous and/or synchronous message passing. Although the mentioned aspects find, in stronger or weaker sense, consideration in conventional agent systems, one key objective of μ-agents is to allow a consistent 'modeling in agents', even when 'drilling down' to the lowest level of implementation (e.g. primitive μ-agents wrapping external resources).

We see the multi-level modeling with the combined use of μ-agents and eventual heavier notions as the computationally rational approach to satisfy the key characteristics of Agent-Oriented Software Engineering (AOSE) [7], namely

- *Decomposition* of functionality down to an appropriately fine level of granularity,
- *Abstraction* by selective hiding of lower levels of the agent organisation, and
- *Organisation* which consistently describes the overall structure.

Inasmuch resource constraints (such as memory and battery capacity) are of particular concern in the context of mobile computing (and continue to be a concern with the more powerful smartphones) we see μ-agents as a low-threshold and easy-entry approach to allow the contemporary use of agent-based technology on mobile devices.

Although μ-agents do not commit to a particular internal architecture type, they are goal-directed, engage in multiple conversations, are computationally cheap and put strong focus on efficient execution and interaction so as not to harm overall system performance; with μ-agents the choice to instantiate yet another agent should have limited impact on system resources but be a matter of modeling pragmatism. Functionality is composed of and embodied by a larger number of functionally small entities at different levels of granularity. Because the efficiency of communication is paramount, the architecture affords two levels of communication: synchronous communication via direct method calls bound at runtime and asynchronous message passing.

In order to clarify the organisational aspects of the μ-agent metamodel (shown in Figure 1), it is discussed before giving a brief description of the overall platform architecture of our platform which we call μ^2 [3].

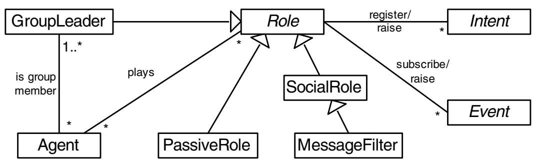

Fig. 1. Core Relationships in μ^2

The μ-agent model is based on the KEA model [10] and identifies *roles* as first-order entities with various specializations. μ-agents play an arbitrary number of roles. Roles, however, come in three first-level specializations, namely *Group Leader roles*, *Passive Roles* and *Social Roles*. Passive roles allow the most simplistic agent implementations only providing synchronous inter-agent communication. Their execution is extremely efficient, and programmers can use these to implement low-level functionality, in places where more coarse-grained agent implementations would fall back to embedded object-oriented or structured programming approaches. Social roles, in contrast, communicate via asynchronous messages and allow long-running concurrent conversations. Those are the main concern of this paper. Group leader roles are specializations in connection with the organisation of μ-agents. By default, each agent is associated with at least one group, its *primary group*. The group leader role allows any μ-agent to create a group itself and manage so-called sub-agents. The only exception is the predefined SystemOwner agent, which is the only agent to be its owning group's owner – a recursion termination condition for the emerging agent hierarchy. In consequence, a consistent hierarchical organisation of arbitrary depth can be modelled using agent-oriented abstractions, not only allowing the decomposition into agents but also the definition of abstraction layers (by hiding sub-agents beyond a given level). The modeling of functionality can then be handled to any degree of granularity.

MessageFilters which are a specialization of social roles, and are a helper construct to support the organisational modeling and functional decomposition into sub-agents by means of message-based delegation. Incoming messages on the super-agent are dispatched to registered message filters (which are sub-agents playing the Message filter role). Message filters then match messages against individual patterns[3] and eventually process those.

However, message filters are just one helper mechanism to handle the functional decomposition in a semi-automated manner; the application developer is free to model the decomposition by other means (e.g. explicit definition of agent/sub-agent relationships and in-code handling of functionality delegation). For all cases, though, agents will at least be sub-agents of the SystemOwner agent to allow consistent platform management (e.g. shutdown).

Role implementations themselves register *applicable intents* which allow the resolution of roles to dynamically bind requests. In μ^2 intents resemble the notion of intentions and include the information necessary for fulfillment. Intents in μ^2 have a similar function as Android's intents, but instead of a fixed method set (as with Android), intent type implementations are entirely left to the application developer. As a consequence, μ^2 intents can have arbitrary structure (potentially including both properties and operations), which is not problematic, since only the requester and the executing agent need to know the semantics of the intent internals.

Any role can register applicable intents (which its implementation needs to be able to handle) and request the execution of intents, hence, agent functionality can be composed across the entire agent organisation. Any execution request (sent via *send(intent)*) will receive a response either by the fulfilling agent or the platform agent in case of failed automated lookup of a potential target agent.

[3] Patterns can be of a simple kind and merely test for particular message fields, or they can be more complex by taking the individual agent state into account.

The event mechanism in μ^2 follows the Publish-Subscribe pattern and is similar to the intent approach, but it requires the definition of the source of an event apart from the otherwise free implementation by the application developer. Once the event is raised all agents (respectively their roles) subscribed to the event are notified. Intents, in contrast, are only delivered to one agent which is capable of fulfilling the intent (as determined by the platform).

The event subscription mechanism equally serves as a hook to react to system events (such as newly created agents, connecting platforms etc.). Both the intent-based dynamic binding and the raising of events works fully distributed across connected platform instances.

2.3 Comparison of μ^2 and Android

The notions of *intents* highlighted in the previous sections differ in Android and μ^2. While intents themselves are message containers in Android, μ^2 introduces a separate message container (the *MicroMessage*) which encapsulates intents. μ-agents do not necessarily rely on intents (if not using dynamic binding), and they can send any payload to an arbitrary agent (e.g. by addressing messages via agent name). Moreover, unlike Android's intents MicroMessages allow for the specification of a sender.

One of the key facilities for a MAS is the ability to dynamically bind communication destinations. Android's approach uses implicit intents which allow the lookup of registered *intent filters* in order to invoke an application component. μ^2, in contrast, looks up μ-agents playing roles which have registered their *applicable intents.*

Overall, *μ-agents* themselves can be seen as an equivalent to Android's *services.* Both have a lifecycle management and persist for longer periods of time (unlike the rather short-running *activities* in Android). However, Android's services do not show autonomous capabilities and are externally activated to provide their service. Additionally they do not engage in actual conversations. A key difference from the application modeling perspective is Android's lack of an organisational model. Apart from the different application components Android does not provide modeling means for a structured application organisation.

Activities in Android are similar to *agent operations* which are not explicitly modelled in μ^2 but are the result of interactions or are initiated to perform a specific task instance. In Android, activities have similar functionality but additionally provide a user interface.

The final concept to discuss is that of *events.* Android reacts to external events via *broadcast receivers* whose functionality can be implemented by the developer (e.g. sending intents to other activities). μ^2 provides an explicit subscription mechanism. Roles can subscribe to specific event types and will receive an event notification once raised. Both broadcast receivers and events can provide payload along with the notification.

Table 1 provides an overview on the similarities discussed to this point.

Table 1. Comparison of Components in Android and μ^2

Component	*Android*	μ^2
Message structure/container	Intent	Intent encapsulated in MicroMessage
Dynamic binding mechanism	Intent filter	Applicable intent
Persistent active entity	Service	μ-agent
Operation	Activity	μ-agent operation
Events	Broadcast receiver	Event subscription

3 μ-Agents on Android

3.1 Design

The notion of loose binding between μ-agents and Android's application components motivates an integrated approach for mutual benefit: μ-agents enable an effective organisational modeling of agent-based applications on Android with unrestricted access to the device capabilities (sending SMS/MMS, retrieving GPS coordinates, accessing the phone's address book). In addition to this, μ-agents can potentially directly and spontaneously interact with existing applications or the user (e.g. to pick an address book entry) by raising according intents in the Android subsystem – supporting the principle and benefits of open systems.

Legacy Android applications, in on the other hand, can access μ-agent capabilities, respectively delegate functionality to μ-agents, or use the entire μ-agent framework as a middleware for distributed applications spanning across mobile devices as well as stationary devices running the desktop version of μ^2.

In order to realize this potential, μ^2 has been ported to Android (as a first step) and then (as a second step) the infrastructure of Android and μ^2 have been mapped against each other, constituting MOA. Its full schema is visualized in Figure 2 and explained in the following section.

The interaction between Android and the μ-agents is mediated via a mutually linked μ-agent/service entity. Each of those two linked components (the AndroidInterfaceAgent and the MicroAgentInterfaceService in Figure 2) represents the interface to the according technological counterpart, i.e. for Android applications MOA appears as 'yet another service' whose lifecycle can be controlled from Android; μ-agents perceive Android as 'yet another agent' providing services to other agents (indicated by registered applicable intents). The purpose of this combined agent/service is the dynamic conversion of requests/events raised from either side. Detailed differences handled by the conversion mechanisms are:

- The respective intent class structures of μ^2 and Android differ considerably.
- Android's intent invocation mechanisms require a specification of the application component type to be invoked (i.e. Activity (via *startActivity*()) or Service (via *startService*()). In μ^2 this is not of concern, as addressed entities are always agents respectively roles.

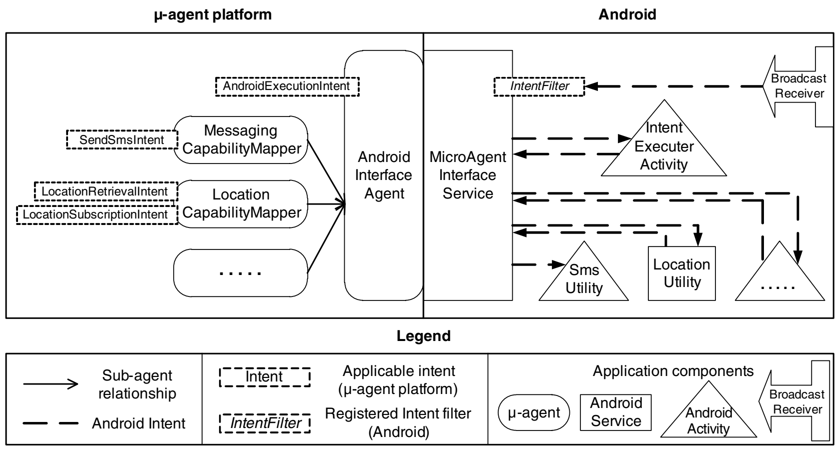

Fig. 2. Architectural Schema of MOA

– Android's intents (hereafter, for clarity, called *AndroidIntents*) do not maintain an explicit sender reference but invoke callbacks upon processing in the calling application component.

To match its more limited dynamic binding capabilities, an intent specialization rebuilding the AndroidIntent class method signature is provided with the μ-agent framework, which, in addition, allows the specification of the application component type. This particular intent type (AndroidExecutionIntent in Figure 2) is registered as an applicable intent with the interfacing μ-agent. Thus any intent to be raised in Android will consistently be resolved to this μ-agent. The Android service tied to this μ-agent additionally adds a custom sender field to any AndroidIntent and uses a dedicated activity (see IntentExecuterActivity in Figure 2) to raise all intents in Android. By implementing this Android activity's *onActivityResult*() method, this IntentExecuterActivity it will be notified upon the processing of an intent. Maintaining the reference of the sender agent then allows eventual responses from an invoked application component to be forwarded to the original sender μ-agent (e.g. response to request for user input), thus overcoming the absence of a sender specification in Android. This allows μ-agents to formulate Android intents and thus directly interact with any other installed Android applications.

Since not all Android functionality – in particular hardware capabilities – can be accessed through Intent-based interactions, additional *Utility components* (which are either activities or services)[4] are provided with MOA and are called from the interfacing service. Capabilities necessitating encapsulation in utility components are the different 'managers' which control Android capabilities (such as LocationManager to provide location information or SMSManager which allows the sending of SMS messages). Those utility components are comple-

[4] Figure 2 shows the SmsUtility and LocationUtility activity/service as examples.

mented with *Capability mapper μ-agents* (e.g. MessagingCapabilityMapper, LocationCapabilityMapper) which register according applicable intents (e.g. SendSmsIntent, LocationRetrievalIntent, LocationSubscriptionIntent) and ensure the proper conversion to AndroidIntents as well as the interaction with the according utility component. Those capability mappers are modelled as sub-agents of the interfacing μ-agent, thereby exploiting the modeling capabilities provided by the μ-agent concept. Their registered applicable intents can be relatively simple, thus breaking down the Android capability into numerous specific intents or combining those into more coarse-grained intents.

From the Android side the InterfaceService can either be directly addressed via explicit intents (using the class name) or implicit intents which are resolved via according intent filters (represented as IntentFilter in Figure 2).

Another noteworthy aspect – and of particular relevance when considering the context of nomadic computing – is the handling of system and/or application events. In order to receive events originated from the mobile device, such as receiving an SMS, the InterfaceService registers broadcast receivers for corresponding events (see Broadcast Receiver in Figure 2). The service itself forwards the message (via its μ-agent counterpart) to the corresponding capability mapper which raises an event in the μ-agent framework. Any subscribing agent will then receive the message (e.g. SMS message) via the event subscription mechanism of the μ-agent framework. This way μ-agents can both address and receive intents/events from Android.

Expanding further on the potential symbiosis of MOA and Android, it is worth noting that Android applications can use MOA as a lightweight middleware for distributed Android applications, since MOA will dynamically dispatch application-dependent intents using its addressing mechanisms by sending an intent to the service and interpreting it as a distributed μ-agent event which ensures the dispatch to subscribing agents on *any* connected platform instance running on a mobile device or a desktop machine. This offers a significant infrastructural extension to Android, whose serialization per se is not fully compatible with the one of desktop JVM instances.

In addition to the functional benefits realized by the interaction-centric dynamic linking associated with this approach is the nearly complete platform-independence of μ-agent implementations; they run both with the desktop as well as the Android version of μ^2. Portability limitations (e.g. when relying on JVM languages not running on Android itself) can be overcome using the cross-platform dynamic linking mechanism.

3.2 Context-Aware Assistant Application

To get a better impression on designing MOA-backed Android applications, we describe one example application using this approach. In this scenario an application receives an incoming message and must decide how to change the phone's preferences in a location-sensitive manner and react to incoming calls or messages depending on the sender's importance. A schematic overview is provided in Figure 3 and discussed in the following.

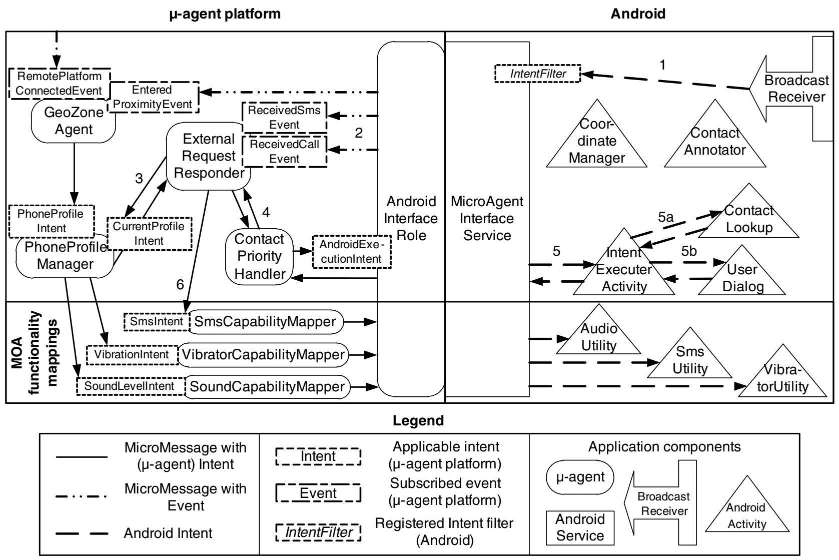

Fig. 3. Schematic Overview on Location-Aware Mobile Application

The application consists of a legacy Android activity used to maintain coordinates for locations of concern, such as: *working place*, *main street*, *home* and associated phone profiles. Additionally, another activity (*ContactAnnotator*) annotates contacts saved in the phone with priorities indicating their relevance to the phone owner (e.g. high priority for family members or boss). The storage locations for both data sets remain in the Android realm but are maintained by so-called content providers. Upon the start of the *InterfaceService*, MOA is started along with application-related agent implementations. In order to maintain actual agent implementations together with the related Android application components, an *AgentLoader* instance is passed to the InterfaceService which specifies the initialization of the μ-agents along with the Android application.

The InterfaceService subscribes to proximity alerts for the entered coordinates. Once reached an according event is raised (*EnteredProximityEvent*), upon which the μ-agent managing phone profiles (*PhoneProfileManager*) activates the according phone profile (e.g. disabling phone sounds and activating vibration in workplace environment). This includes the handling of potentially overlapping locations, in which case the more restrictive phone setting is chosen. An event which could also activate a particular profile (as an alternative to GPS-based location determination) could be a connecting μ-agent platform associated with a particular environment (e.g. workplace).

When receiving a phone call or text message, the InterfaceService is notified (Arc 1 in Figure 3), and the μ-agent receiving the incoming request decides – depending on the current profile (Interaction 3) and priority of the sender

(Interaction 4/5a) – how to react, e.g. if sounds are disabled in the workplace environment, call can be ignored, answered with automatic SMS message promising a return call, or even override phone profile (for very important calls).

In cases where the sender is not yet annotated (e.g. recently added), μ-agents can generate an Android intent to open an application-specific dialog (5b) in order to allow the annotation in real-time and handle the external request accordingly. This simple example shown here exemplifies the potential to delegate functionality to embedded μ-agents. Doing so provides structured agent-oriented modeling along with a flexible extensibility using elements such as synchronization of contact annotations with other connected phones, and consideration of calendar entries (on local phone and remote machines) to allow more precise responses where appropriate (”Am currently in a meeting with XY, ...”).
Key advantages of using μ-agents in conjunction with Android are:

- Consistent loose coupling – μ-agents can address all Android capabilities in a unified loosely coupled manner; modelling of functionality is reduced to the mere composition of these intents. At no point do agent names need to (but can) be involved.
- Agent-oriented modeling – modeling applications using agent organisations with multiple levels of functionality granularity for maximum reuse. The flexible definition of intents and association with appropriate μ-agents allows mobile application developers to effectively specify their intent-based functionality repository while using the built-in organisation mechanisms to structure their applications.
- Distributed applications – MOA can be seamlessly distributed; developers do not need to deal explicitly with any network-related aspects.
- Performance – As briefly elaborated in the following subsection μ-agent interaction outperforms Android's internal communication mechanism considerably, allowing effective decomposition into agent societies without performance loss.

A further consideration is the more extended realization of the open system principle, since agents can proactively interact with Android components which – in the context of MOA – represent ’their’ environment.

3.3 Performance

To demonstrate the performance of MOA's interaction mechanisms, we constructed a benchmark scenario for both AndroidIntents and μ-agent intents. The scenario is loosely based on the previously described context-aware application and was built as both a native Android application and as an application using the MOA approach. The internal functionality of agents and services is normalized to isolate the relevant comparative interaction performance. The scenario is shown in Figure 4 and described with references to the right section of the figure (Android scenario).

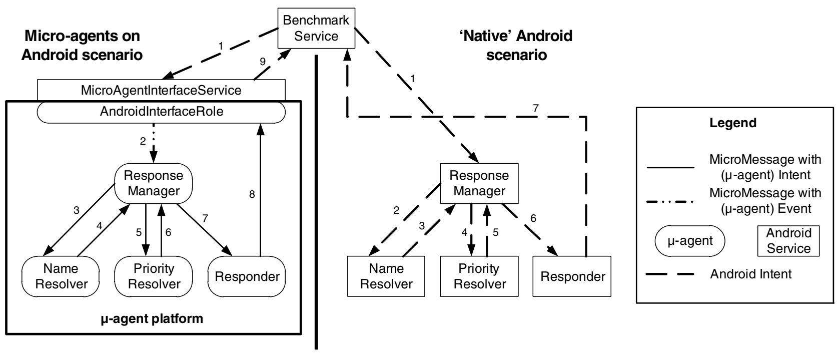

Fig. 4. Benchmark Scenario for Interaction Performance Comparison

The *BenchmarkService* (started via an Android activity) runs the given scenario for a specified number of rounds with the according benchmark variant (MOA or native Android).

The system simulates the arrival of a new SMS message by generating the initial intent which is dispatched to the ResponseManager (arc 1). The service identifies the SMS message and requests the resolution of the received telephone number to a name (2) via the lookup in the phone's contact database. Upon response (3) the SMS message sender's relevance is determined (arcs 4, 5). If the sender is of relevance (which is always the case in the benchmark), an SMS response message is sent via the Responder (arcs 6, 7).

The μ-agent version is functionally equivalent. It includes two additional messages enforced by the necessary conversion between Android intents and associated MicroMessages (with intent/event payload). As such, the interaction includes seven MicroMessages and two Android intents. In both cases the coupling of components is loose; intent filters (in the Android example) respectively applicable intents (with μ-agents) resolve target components/agents. Table 2 shows selected results of the benchmark[5].

Table 2. Selected Performance Benchmark Results per Scenario Rounds (with factors indicating relative performance of Android intents to μ-agent intents)

Rounds	MOA (ms)	Native Android (ms)	Factor
5	257	614	2.39
50	1834	4328	2.36
500	17757	42446	2.39
5000	156384	465606	2.98

[5] The benchmark has been executed on an Intel Core2 Quad-Core CPU at 2.66 GHz and 3.25 GB RAM running on Windows XP Professional SP3 using the Android 2.1 Emulator. Each run has been undertaken 10 times and the results represent the average over the eight central results, ignoring highest and lowest values. In all the cases, we have allowed warm-up runs of 5 rounds before the actual timing.

The results indicate a significant performance advantage of μ-agents over Android intents. Given those results, the use of μ-agents for interaction-intensive applications is likely to be beneficial even if the multi-agent nature of the application is not of primary concern for the application developer.

Android's application components principle has potential support for low level system features (such as Inter-Process Communication (IPC) – which is not used in this scenario). We consider the processing overhead associated with this as one aspect introducing inefficiencies when executing intents. Conceptually, μ-agents sit below the application component level, directly built from the provided libraries, and thus do not share these performance penalties unless they actively interact with Android application components.

3.4 Potential Application Areas

For a better impression of the potential of MOA, we outline areas where we deem the presented approach promising, beyond the consideration of μ-agents as lightweight middleware for distributed desktop/mobile applications.

Agent-based mobile applications
The use of MOA allows for the modelling and implementation of agent-based applications that are compatible with desktop versions of the multi-agent platform. This is of interest when considering the increasing computing power of mobile devices. Furthermore, agent-based applications are not only restricted to *run on* Android but to directly access device functionality and show the unique capability to natively *interact with* Android application components. μ-agents are particularly suitable for dealing more generically with heterogeneous application environments and representing a low-threshold approach to the development of smart mobile applications using agent-based principles without confining the developer to a particular technology.

Intelligent agents
The original motivation for the μ-agent platform, i.e. to support the wider software engineering advantages of agent-oriented decomposition, is equally available on mobile devices. The architecture-independent approach does not only provide support for agent-based organisational decomposition and abstraction levels, but also the ability to embed particular agent architectures (e.g. cognitive architectures) which can then directly access phone capabilities (e.g. reading and manipulation of contacts, calendar, SMS, etc.), serving as a base for seamless integration of agent-based with mobile technologies, to enable intelligent, agent-based, applications. This can be realized *on* the devices or delegated to more powerful desktop machines using MOA's intent-based dynamic linking.

Robotics
Another area where the combined use of Android and agents is of practical concern is the area of robotics. General-purpose robots with resource-limited computing capacity often only support a subset of necessary functionality (e.g. J2ME) and are relatively expensive. The use of MOA facilitates the embedding of the agent functionality in a standard Android device connected to the mechanical robot

using wireless technologies (e.g. bluetooth). This "externalization" of the core agent capabilities could allow more economical use of robots and make upgrades a matter of replacing the phone. Additionally, MOA enables robots to communicate via SMS 'out-of-the-box' as a fallback mechanism if no other network connection is available (e.g. IP-based network) – or as general means of communication.

4 Related Work

The use of Android in the context of agent-based applications is a novel and relatively recent development. Not surprisingly, some approaches to build agent-based systems in conjunction with Android have been undertaken by other researchers.

Agüero et al. [1] present an approach to implement their Agent Platform Independent Model (APIM) in Android. The implementation is entirely based on the Android application components. Given that the elements of the APIM largely concentrate on agent internals, the modeling of agent organisations is not considered. Direct interaction with Android applications is not part of this concept.

JaCa-Android [11] is an implementation of the Agents and Artifacts model on Android. It identifies agents and artifacts as first-order entities to model agent-based systems. In order to implement agents, the approach embeds the Jason reasoning engine, thereby welding itself to a specific internal agent architecture. Relevant Android capabilities are encapsulated as artifacts which can be used by agents (e.g. GpsArtifact, SmsArtifact). This implementation provides an expressive means to handle Android capabilities using AgentSpeak, including the ability to act in remote workspaces. Agents in JaCa-Android cannot directly formulate Android-compatible intents to interoperate in a spontaneous manner.

A last notable approach to run agents on Android is the ported version of the multi-agent platform JADE [6]. It is conceptually weaker than any of the other approaches and represents the consequent move to provide a JADE version for Android. Its limitations notably include the requirement of a connected desktop version of JADE to allow distributed operation. Apart from this only one agent can be run on a mobile device.

5 Conclusion

Android's infrastructure shows characteristics which make it attractive for the development of multi-agent systems. However, its loose coupling does not extend to actual capabilities such as sensors. The integration of Android with μ-agents brings mutual advantages:

Loose coupling and Agent-based Modeling – μ-agents can address Android capabilities and components in a consistently loosely coupled manner. Coarse-grained Android services can be decomposed into μ-agents, which increases the reusability across different applications and remains computationally affordable as a consequence of the better interaction performance.

Distributed applications – Android can rely on MOA's infrastructure to build distributed applications, across both mobile and desktop devices, allowing automated delegation of functionality.

Our μ-agents further enable a hybrid approach by harmonizing with legacy application components; MOA agents' unique ability to directly interact with other application components not only follows the spirit of open systems but also supports the development of mobile applications in a cross-paradigmatic manner.

High performance and distributedness make MOA a viable extension to the legacy mechanisms for general application development on Android, thus promoting agent-based concepts for the wider realm of software engineering.

Plugging in specific agent architectures selectively will allow the integration of more intelligent features – beyond contemporary 'smart' applications.

References

1. Agüero, J., Rebollo, M., Carrascosa, C., Julián, V.: Does Android Dream with Intelligent Agents? In: Corchado, J., Rodríguez, S., Llinas, J., Molina, J. (eds.) International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI 2008). Advances in Soft Computing, vol. 50, pp. 194–204. Springer, Heidelberg (2009)
2. Arthur, C.: Android overtakes Symbian in smartphone sales (January 2011), `http://www.guardian.co.uk/technology/2011/jan/31/android-symbian-smartphone-sales` (accessed on: September 15, 2011)
3. Frantz, C.: Micro-agent platform μ^2, `http://www.micro-agents.net` (accessed on: September 15, 2011)
4. Google. Application Fundamentals – Android Developers, `http://developer.android.com/guide/topics/fundamentals.html` (accessed on: September 15, 2011)
5. Google. What is Android?, `http://developer.android.com/guide/basics/what-is-android.html` (accessed on: September 15, 2011)
6. Gotta, D., Trucco, T., Ughetti, M., Semeria, S., Cucè, C., Porcino, A.M.: JADE Android Add-on Guide, `http://jade.tilab.com/doc/tutorials/JADE_ANDROID_Guide.pdf` (accessed on: September 15, 2011)
7. Jennings, N.R., Wooldridge, M.: Agent-Oriented Software Engineering. Artificial Intelligence 117, 277–296 (2000)
8. Koch, F., Meyer, J.-J.C., Dignum, F., Rahwan, I.: Programming Deliberative Agents for Mobile Services: The 3APL-M Platform. In: Bordini, R.H., Dastani, M.M., Dix, J., El Fallah Seghrouchni, A. (eds.) PROMAS 2005. LNCS (LNAI), vol. 3862, pp. 222–235. Springer, Heidelberg (2006)
9. Moreno, A., Valls, A., Viejo, A.: Using JADE-LEAP to implement agents in mobile devices. EXP - in search of innovation (Special Issue on JADE) 3(3) (2003)
10. Nowostawski, M., Purvis, M., Cranefield, S.: KEA - Multi-Level Agent Architecture. In: Dunin-Keplicz, B., Nawarecki, E. (eds.) CEEMAS 2001. LNCS (LNAI), vol. 2296, pp. 355–362. Springer, Heidelberg (2002)
11. Santi, A., Guidi, M., Ricci, A.: JaCa-Android: An Agent-Based Platform for Building Smart Mobile Applications. In: Dastani, M., El Fallah Seghrouchni, A., Hübner, J., Leite, J. (eds.) LADS 2010. LNCS, vol. 6822, pp. 95–114. Springer, Heidelberg (2011)