ContextDroid: an Expression-Based Context Framework for Android

Bart van Wissen, Nicholas Palmer, Roelof Kemp, Thilo Kielmann and Henri Bal
Vrije Universiteit
De Boelelaan 1081A
Amsterdam, The Netherlands
{bvwissen, palmer, rkemp, kielmann, bal}@cs.vu.nl

Application	Purpose	Context
Always Prompt	notify when to leave	calendar, location
SongDNA	song recognition	sound
Hoccer	information exchange	location, movement
Screebl	power saving	movement, orientation
PhotoShoot [3]	duel game	movement, orientation
iNap	notify when close to destination	location

Table 1: Context Aware Applications Seen in ADC2

ABSTRACT

In this paper we describe our work on ContextDroid, a framework for Context Aware applications on Android powered smartphones. This framework is designed to provide application developers with the services required to easily build Context Aware applications with an eye towards reducing energy consumption of context monitoring, especially as multiple context aware applications are run on the same device.

1. INTRODUCTION

We are currently living in the Information Age, an era characterized by an abundance of information and the presence of many technological devices interacting with this information. One of these devices, the *smartphone*, is of particular interest for personal context information.

Where the first generation phones, were only equipped with radios for communication, modern smartphones have many sensors. They can, for instance, sense location, nearby devices, movement of the phone using an accelerometer and/or gyroscope, proximity to the users skin, and even compass orientation. These devices are often used to store personal information such as contacts and calendar information, but they can also be used to retrieve information from web services such as weather, traffic, and news services as well as run advanced applications.

Due to the opening of centralized markets, we have seen an explosion in the number of third party applications for smartphones. A similar rise on smaller scale occurred for applications that take advantage of context information available on the phone in order to make the phone behave in a smarter way. In Table 1 we list applications that use context information which participated in the recent Android Developers Challenge for innovative smartphone applications.

Although we see an increase in the number of context aware applications, writing such applications can be a complex task, especially when the information comes from many different types of sensors, each with their own unique programming interface.

Furthermore, when multiple context aware applications are used in combination, it is likely that more than one will monitor the same sensors. This can result in highly inefficient use of the device's resources as duplicate analysis of sensor data can not be combined. If programmers would use an easy to program centralized framework for this, they would not only save development time and be able to develop more interesting applications, but in addition the phone's resources could be used more efficiently, resulting in longer battery life for the user.

In this paper we discuss which properties are required for such a framework, and present our ongoing work on the design and implementation of *ContextDroid* a Context Framework running on the Android platform.

The contributions of this paper are:
- We show that there is a need for a Context Framework to ease programming of context aware applications and reduce inefficient use of resources
- We present the requirements for a Context Framework for smartphones
- We present the design and implementation of ContextDroid, a Context Framework that fulfills the necessary requirements.
- We demonstrate this framework with a simple baby-monitor application

The remainder of this paper is organized as follows. In Section 2, we describe the requirements for a Context Framework derived from our analysis of the problem. In Section 3 we describe the design of ContextDroid in a bottom-up fashion, starting from the sensor and ending at context expressions. Section 4 goes into detail about important aspects or our implementation. We then demonstrate ContextDroid with an application that can be used to monitor a sleeping baby in Section 5 and conclude by giving an overview of related work in Section 6, future work in Section 7 and conclusions in Section 8.

2. REQUIREMENTS

In our research project, called Interdroid, we strive towards creating an environment for smartphones in which truly smart (possibly distributed) applications can easily be constructed. We believe that smart usage of context information is of key importance for next generation smart applications. Ideally such a smart application will automatically adapt its behavior towards the actual context of the smartphone and its user. For instance, during an important meeting incoming phone calls should be directly sent to voicemail without ringing, but if a user's partner calls three times in a row, there might be something even more

important and then the phone should notify the user. As another example, when close to the supermarket during opening hours, the shopping list application should inform you that you have to buy milk. However, it should not do so if you have a meeting scheduled for which you, according to your location, have to hurry to be on time.

Today, when application programmers write such an application, they have to use the available programming interfaces of the platform to access several context sensors. Accessing the position information from the GPS is totally different from looking into the calendar, or getting the current sound level. For each and every situation the programmer has to write a new – possibly complex – piece of code that will get the desired context information. They also have to write the logic which will allow them to combine these pieces of information into smart behaviors.

We believe that a generic framework for the gathering and evaluation of context information will greatly reduce development time for such applications, but only if it meets the following requirements:

- *Usable*: The system has to provide a framework that enables application developers to very easily make their applications context-aware, even if context-awareness is not the main point of their application. Access to different context elements should be uniform.
- *Efficient*: Running on portable devices with limited battery-life means that attention to energy consumption has to be paid from the beginning.
- *Extensible*: Writing intelligent context-knowledge gathering routines is a complex task that is best left to programmers that have specific knowledge of the sensors. Thus the system should enable third party programmers to easily add components to the system.
- *Portable*: Software built using the framework has to be usable on a large number of devices.

3. DESIGN

In this section we describe the different elements of our context framework, *ContextDroid*, which meets the aforementioned requirements. We explain the design of our context framework beginning with our model of context information, and continuing with how we gather this information, and then how expressions can be formed around them and finally to how those expressions will be evaluated. We also detail how ContextDroid deals with energy usage and Quality of Service through service level requests and how these mechanisms can be extended.

3.1 Context Entities

The devices context, as known to ContextDroid, consists of *context entities*. A context entity is a collection of information. For example, the user's current location, expressed in latitude, longitude, and altitude, is a context entity. The current state of that entity is determined by context sensors. Context sensors are programs that gather information from a particular source (hardware sensors, internal memory, files, the internet) and deliver it as a context entity to ContextDroid.

We split context entities into two categories:

- *synchronous entities*, the state of which can be read in real time. Example: the current time.
- *asynchronous entities*, the state of which is 'pushed' into the system. The service then saves this state in

a history so that applications can obtain the state of such an entity at a certain point in time.

The state of an entity may be unknown at any time, at which point its value will be `null`.

3.2 Context Entity Readings

We call the state of a context entity during a certain period in time a *context entity reading*. A reading consists of a value that represents the state of the entity, a timestamp, and a time of expiration. The reading is valid from its timestamp up to its time of expiration. A new reading effectively invalidates and replaces the previous reading from the timestamp of the new reading, even though the previous reading's expiration time may indicate that it is still valid.

This approach was chosen because the system may not always be able to provide new readings in time. Readings must thus have an expiry time, which can be chosen by the sensor providing the reading. Highly dynamic context entities will in general have a shorter time of validity than context entities that hardly ever change.

3.3 Context Conditions

Typically an application designer does not want to use the raw context entity readings, but rather wants to evaluate them according to a function with application specific parameters. For instance the raw location is of much less interest to a developer than the evaluation of whether the distance to the supermarket is less than 50 meters. Therefore, in order to make programming easy, our framework provides *context conditions*.

A context condition is defined by a boolean evaluator function, a number of constant parameters for the evaluator function and a description of what context information should be used as input.

Evaluator functions are functions which take one variable and a number of constants as arguments and return a boolean value. To formalize this, we designed an object called EvaluatorInputSelector. This object contains settings that together define which value should be passed to the evaluator function. Those settings are:

- a Context Entity identifier
- a "value path" describing which part of the entity to take in case it is a complex value (see below)
- a time span
- a selection mode

The context entity identifier uniquely identifies the context entity.

3.3.1 Value Path

Some context entities have values that are actually a set of key/value-pairs, or a list of values, or even a list of sets of key/value pairs. We may want an evaluator function to use a part of that complex data structure as input to the evaluation, for instance we may only be interested in the altitude from the GPS sensor, so we need a way to describe which part of a given entity an expression is needed. For this, we use a path-string, which allows regular expression like selections of parts of values. For brevity we don't go into details of the selection mechanism.

3.3.2 Time Span

The time span specifies a window in the history of the state of the context entity. The effect of this window depends on the mode, but in general it determines how much history to take into account when evaluating the condition.

3.3.3 Selection Mode

The selection mode can be one of the following:

- ANY: The condition is true if the evaluator returns true for any of the values inside the window.
- MEAN: The condition is true if the evaluator returns true for the mean value calculated over the values inside the window. This can only be used for numeric values.
- MAXIMUM: The condition is true if the evaluator returns true for the highest value inside the window. This can only be used for values that implement the Comparable interface.
- MINIMUM: Like the former, but with the lowest value instead of the highest.
- ALL: The condition is true if the evaluator returns true for all values currently inside the window.

3.4 Evaluating a Context Condition

Evaluating a context condition consists of the following steps. First a part is selected out of a context entity's history. Then, from every reading inside this history, one or more atomic values are selected. Depending on the mode, in case of ANY, all values are passed to a *evaluator* in reverse order (latest first) until one value is found for which it returns true, or in case of MINIMUM, MAXIMUM or MEAN, first a value is calculated and then it is passed to the evaluator.

The settings of the EvaluatorInputSelector can have a big impact on performance. When no history is used, only one value has to be evaluated, and it only has to be done whenever a new value comes in, making the condition very cheap to evaluate. On the other hand, when the time span is very long, the number of readings inside it is high, and a mode like MINIMUM or MEAN is used, evaluating the condition can become a time- and energy consuming operation.

We apply several evaluation strategies to optimize the evaluation. When mode ANY is used, if the evaluator returns true for a value that just entered the window, we can be sure that it will be true for as long as the value is inside the window. We can calculate at what moment it will move out of the window, and make sure that we do not evaluate before that moment.

When mode MINIMUM is used, we only have to re-evaluate when the latest minimum value moves out of the window, or when a new value enters the window that is lower than the last one.

A similar optimization can be done for MAXIMUM. The most problematic mode is MEAN, because it changes continuously, even when no new values come in. Since we cannot evaluate it continuously, we chose to automatically re-evaluate it when no new value has come in for a certain period.

3.5 Evaluators

The *evaluator* is a simple interface with one method named evaluate, which takes a variable number of parameters. ContextDroid includes a number of predefined evaluators. A few of these are:

- ==, >=, >, <, <=: These evaluators act as their names imply. They all take two arguments and compare them.
- regexp: This evaluator takes a string and a pattern as arguments and returns true if the string matches the pattern.

- distance within: This evaluator takes two pairs of coordinates and a radius as arguments, and returns true if the distance between the two pairs of coordinates is less than the specified radius. This can be used to create "proximity alert" style conditions.

3.6 Expressions

In order to provide a way for applications to react to certain conditions in the context, as mentioned in the requirements, applications should be able to describe that set of conditions to the service in a formal way. For this, our framework uses *context expressions*. A context expression is a boolean expression in which the axioms are the context conditions on context entities.

Since most conditions are a combination of several subconditions, we need a way to combine conditions into expressions. We have chosen to use a tree structure because every logical formula can be expressed as a tree. In such a tree, a conjunction is an AND node with two children, a disjunction is an OR node with two children, and a NOT node has only one child whose result is inverted. The leaf nodes are the atoms, which are specific context conditions.

In addition to being expressive and intuitive to use the tree structure also adds the possibility of short-circuit evaluation. In an expression A OR B, evaluation of B can be skipped as long as A is true. Future work with our framework will include optimizing such short circuiting based on energy consumption to minimize the energy consumption of evaluating the total framework, for instance by turning off high energy consumption sensors such as the gps when in conjunction with a low energy sensor such as time.

## 3.7 Quality of Service: Service Level Requests

Different applications may have different demands in terms of information that has to be available. One application may be dependent on an entity which is highly dynamic, such as the microphone level, and may want to react to changes quickly. Another application may only want to check periodically if some WiFi network is available.

Because battery life is limited, mobile operating systems are designed to put the phone into a sleep mode whenever possible. To perform any readings for our context service however, the phone has to be awake. Our background service should thus ideally not perform any readings when they are not really necessary.

Thus, we decided to have the applications tell the service the minimum service level that they require to operate correctly. They do this by issuing a service level request. In principle, an application can request any service level, but no guarantees are given by the service. The service will deliver a best effort that matches the requested service level as closely as possible. A service level consists of a list of context entities, each with a number of parameters. These parameters include:

- whether or not the entity's sensor has to be active
- the minimum frequency at which readings of that entity should be performed
- the amount of history that should be kept
- entity-specific requirements

Upon receipt of a service level request, the service adjusts its settings according to the request. When multiple applications issue service level requests, for each setting the "highest" setting is chosen. All those highest settings together form the composite service level. Consequently, whenever
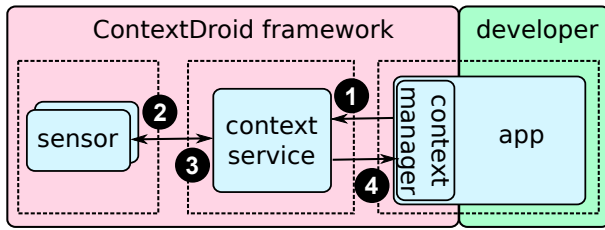
Figure 1: Overall ContextDroid Structure. Each dashed box is a separate process. Service requests are passed from the application using the manager to the service (1) and forwarded to the subsequent sensor (2) via IPC. Sensors push readings back to the service (3), which evaluates them and notifies the application if needed via a broadcast (4).

a request is cancelled, the service level is restored to the highest level without that request.

3.8 Extensibility

It is very important that the default set of available context information made available by our system is easy to extend with new entities. To create a new context entity, basically two conditions must be fulfilled:

- The entity itself must be declared, giving it an identifier and a data type.
- A sensor must be implemented to provide readings for that entity.

Note that a single sensor can (and in most cases will) actually provide readings for multiple related entities. Implementing a new entity may thus also mean adding its implementation to an existing sensor, provided the developer has access to its source code.

4. IMPLEMENTATION

Now that we have seen the design of the expression based context framework ContextDroid, we turn our attention to the implementation. We selected the open source Android mobile platform as target platform for our implementation. Android has several application components that match well with the requirements and design of ContextDroid. Android allows for long running background processes (*services*) which fit particularly well for both the Sensors and the ContextManager, a feature unavailable on the popular iPhone platform.

Figure 1 shows the different components of ContextDroid's implementation. A client-service architecture was chosen to enable efficient sharing of knowledge between multiple applications that run simultaneously.

Android's AIDL interfaces have been used to enable transportation of objects in so-called *parcels* between client and server. The application is linked with *ContextManager*, a helper class that facilitates communication between the application and the ContextDroid service. The ContextManager sets up a connection to the service interface, sends service level requests and installs listeners to receive broadcasts from the server.

The service launches and maintains connections to the Sensors, which are in themselves Android services with their own interfaces. Sensors connect back to the ContextDroid service to 'push' context knowledge, which effectively makes a two-way binding between a sensor and the ContextDroid service.

The ContextDroid service provides the applications with context knowledge by means of *Broadcast Intents*. Broadcast intents are Android's way of broadcasting information to applications system-wide. New readings as well as expressions of state-transitions are broadcast this way.

4.1 The ContextService

The ContextService is the heart of the Context Framework. Its main responsibilities include:

- maintain a shared knowledge base of context information
- process updates to the context information provided by Context Sensors
- provide an interface and act as a mediator between applications, the context knowledge base and the service level manager

The context service maintains a history of readings for a specific context entity. The amount of history is set by the Service Level and can be changed dynamically.

The service level manager is also included in the context service and creates a composite service level request out of a list of service level requests. New requests can be put into the data structure, and they can be canceled using their unique id.

4.2 Expression Engine

The evaluation of a context expression can be triggered as a result of two types of events:

- One of the expression's context entities changes value
- The evaluation is triggered by the *scheduler*

The first type is implemented as a simple *observer* pattern. When an expression is added, a list is made of all the entities it depends on, and the root of the expression tree is subscribed to all those entities. Whenever a new reading comes in for any of those entities, re-evaluation of the entire expression tree is requested asynchronously.

The second type is implemented by using Android's AlarmManager. The alarm manager makes sure that the device wakes up whenever an alarm is scheduled, if it is in sleep mode. The evaluation scheduling system also uses asynchronous evaluation requests.

The asynchronous approach was chosen because it prevents blocking in situations where one event triggers the evaluation of an expression that is already being evaluated at that moment.

We use a top down approach for evaluating tree expressions. That is, whenever any of the Context Entities that any of the tree's leaf nodes depend on change, the whole tree is re-evaluated in a top-down order. We use this approach because it is relatively simple to implement short-circuit evaluation. When evaluating an AND node, for example, the right operand only has to be evaluated if the left operand is true. Even though "A AND B" is the same as "B AND A" from a logical point of view, the programmer can optimize the energy- and time consumption of the expression by considering the order of the operands.

5. EXAMPLE: BABY MONITOR

To illustrate the use of ContextDroid we created a context aware application, which allows a smartphone to be used as a baby monitor. The phone will monitor the sound level and when a certain threshold is passed the application can notify another phone by calling, texting or even mailing a small video clip (see Figure 2).

The code excerpt in Figure 3 shows the lines in the application that deal with the context.
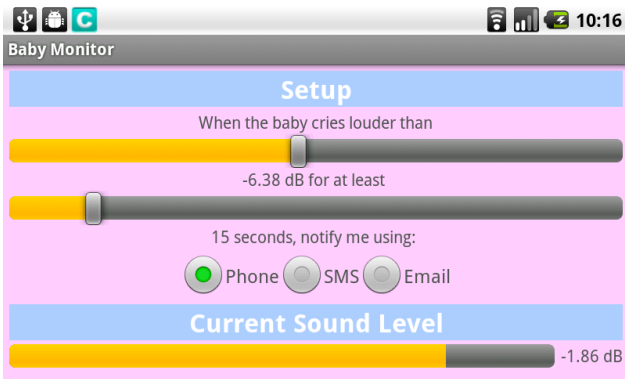
Figure 2: Screenshot of the Baby Monitor App

```
manager = new ContextManager(this, new ContextManagerListener() {
  // connected with service
  public void onConnected() {
    // look at the minimum sound level over the last time period (15 s)
    selector = new EvaluatorInputSelector("sound.level_rms", MINIMUM,
      timeThreshold);
    // check the sound level against this threshold (−6.38 dB)
    parameters.putDouble("value", soundLevelTreshold);
    // and evaluate whether it's greater than or equal to it
    condition = new ContextCondition(">=", parameters, selector);
    // add the condition to the manager
    manager.addContextExpression(condition, "baby", serviceLevelRequest);
    // and if it evaluates to true, notify another phone
    manager.registerContextListener("baby", new ContextListener() {
      public void onTrue(String expressionId) {
        notifyAnotherPhone();
      }
    });
  }
});
```
Figure 3: Code Example of Baby Monitor App

6. RELATED WORK

A project similar to ContextDroid is Context Weaver [1], which was developed at IBM in 2004. It is a platform that simplifies writing of context-aware applications. It lets applications access context information through a simple, uniform interface. Applications access data not by naming the provider of the data, but by describing the kind of data they need, after which the system will respond with a suitable provider. An important aspect of Context Weaver is that when a provider fails, Context Weaver automatically tries to find another provider of the same kind of data.

Context Weaver only considers current values of context, whereas ContextDroid includes historic information and adds expiration times to values, which results in more accurate context data. Furthermore, ContextDroid has specifically been designed for mobile platforms and takes energy usage into account, while to our knowledge there are no reports of Context Weaver running on mobile platforms.

WildCAT [2] is a Java toolkit/framework whose goal is the same as ContextDroid's: to ease the creation of context-aware applications for application-programmers. WildCAT too offers an API for programmers to access context information both synchronously and asynchronously. WildCAT uses a string based expression model, which is largely equivalent to ContextDroid's expression model.

WildCAT does not offer any means of service level management such as ContextDroid does. And although Wild-CAT is written in Java and in theory could be easily ported to mobile devices, it has not been designed especially for mobile platforms and for instance the lack of service level man-

agement makes it less suitable for mobile platforms, since efficiently handling the devices resources is of key importance on mobile platforms.

FRAP [4] is another context framework targeted at the construction of pervasive (multi-player) games. In FRAP, a central server keeps track of all context information of the clients, which have to be connected to the server. FRAP uses *WildCAT2* [2] to store context information and thus is also not appropriate for mobile platforms.

7. FUTURE WORK

Our future work with the framework will involve further evaluating and optimizing the energy consumption of the framework. We also intend to look further at usability and extensibility through the construction of more context aware applications. We will also add support for distributed context expressions which run over multiple devices in order to enable distributed context applications. For instance a user may request to be notified to initiate a call when both they and their partner are not in meetings. Finally, we intend to explore context policy enforcement with our framework.

8. CONCLUSIONS

In this paper we have presented ContextDroid, a framework that eases the development of context aware applications for smartphones. We designed and implemented ContextDroid based on the requirements for a context framework targeted at smartphones: usable, efficient, extensible and portable.

ContextDroid offers a simple, uniform and intuitive way for applications to register context expressions. Due to the centralized setup ContextDroid integrates multiple context expressions and computes a composite service level, such that multiple application requirements are met with the lowest pressure on the device's resources.

We have evaluated the ContextDroid framework with a real world smartphone application.

9. REFERENCES

[1] N. Cohen et al. Building Context-Aware Applications with Context Weaver. *IBM Research Division, TJ Watson Research Center*, 2004.
[2] P.-C. David and T. Ledoux. Wildcat: a generic framework for context-aware applications. In *MPAC '05: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–7, 2005.
[3] R. Kemp, N. Palmer, T. Kielmann, and H. Bal. Opportunistic Communication for Multiplayer Mobile Gaming: Lessons Learned from PhotoShoot. In *MobiOpp '10: Proceedings of the Second International Workshop on Mobile Opportunistic Networking*, pages 182–184, 2010.
[4] J.-P. Tutzschke and O. Zukunft. Frap: a framework for pervasive games. In *EICS '09: Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*, pages 133–142, 2009.