

Contory: A Smart Phone Middleware Supporting Multiple Context Provisioning Strategies*

Oriana Riva

Helsinki Institute for Information Technology
P.O. Box 9800, FIN-02015 HUT, Finland
oriana.riva@hiit.fi

Cristiano di Flora

Nokia Research Center
P.O. Box 100, FI-33721 Tampere, Finland
cristiano.di-flora@nokia.com

Abstract

This paper presents Contory, a middleware specifically deployed to support provisioning of context information on mobile devices such as smart phones. Contory integrates multiple strategies for context provisioning, namely internal sensors-based, external infrastructure-based, and distributed provisioning in ad hoc networks. Applications can query Contory about context items of different types, using a declarative query language which features on-demand, periodic, and event-based context queries. Contory allows applications to utilize different provisioning mechanisms depending on resource availability and presence of external infrastructures. This paper illustrates our approach along with its design and implementation on smart phones.

1. Introduction

In the last decade, the use of context in applications running on mobile devices such as PDAs and smart phones has become a crucial requirement for several research areas, including ubiquitous computing, mobile computing, augmented reality, and human-computer interaction. As mobile users move around the physical and social surroundings, they go through several contextual changes which can be exploited to provide adaptive and personalized services, and to make the system accessible in a more effective way.

In principle, mobile devices can acquire context information by using a large variety of sensors, cameras, and microphones, which can be embedded in the device or in the surrounding environment. In practice, context provisioning on mobile devices is a challenging task [10]. First, it is a complex process consisting of many sequential and parallel sub-processes that can lead to significant power consumption, which in turns makes it a critical issue for

energy-constrained devices. Particularly, reasoning algorithms can require large storage space and huge computational power. Second, the integration of sensors in devices should not compromise the portability, usability (e.g., size, weight, design, and aesthetics), lifetime, and cost of everyday devices. Finally, many sensors may not be operative in every environment (e.g., GPS in indoor environments). For all these reasons, defining a suitable strategy for gathering and processing context information on mobile devices is a crucial design issue.

Typically, context-aware applications running on mobile devices directly sense and locally process context data [3] or rely on external context infrastructures [5]. A third alternative is a distributed context provisioning strategy in which devices share context information of different types over mobile ad hoc networks. All three mechanisms are valuable depending on the situation in which they are employed. For example, in the presence of a context infrastructure, the application may exploit it to acquire context information; if such an infrastructure becomes unavailable, the device could rely on its own sensors and computing capabilities or on sensors integrated in neighboring devices.

The focus of our work is to expand the capabilities of mobile devices in supporting context provisioning. We propose the CONTextfactORY middleware for context provisioning on smart phones (Contory). The novelty of Contory is that it integrates multiple strategies for context provisioning (i.e., internal sensors-based, external infrastructure-based, distributed provisioning over ad hoc networks), and make them accessible through a common SQL-like interface. Applications generate context queries in which they can specify type and quality of the desired context items, context sources to be employed, push or pull mode of interaction, and other additional properties. Contory processes context queries by using the most suitable context provisioning strategy, and eventually returns matching results to the application. In this way, such applications do not need to uniquely and continuously rely on their own sensors or on the presence of an external context infrastructure. Ad-

*This work has been supported by the DYNAMOS Project (<http://virtual.vtt.fi/virtual/proj2/dynamos/>).

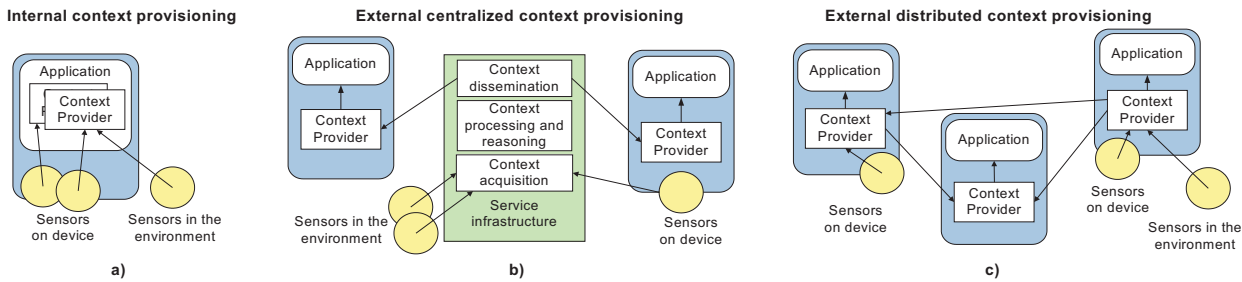


Figure 1. Context-awareness support

ditionally, Contory offers a means to partly relieving the uncertainty of context sources by collecting and combining information from different sources.

The rest of the paper is organized as follows. Section 2 discusses several context provisioning strategies. Section 3 describes how these mechanisms are integrated in Contory along with its design and implementation. Section 4 discussed related work. The paper concludes in Section 5.

2. Context Provisioning

A well-known definition of context [3] states that context represents any information that can be used to characterize the situation of an entity. Context provisioning is the process by which context information is acquired, processed, and made available for usage to the application or other modules. Hereafter, we refer to *context providers* as the software components in charge of performing context provisioning. Raw context data can be collected by monitoring the user's physical and social environment using sensors integrated in the handheld device and in the environment, tags and beacons, positioning systems, biosensors on user. In order to infer higher-level contexts, context providers process such raw data using mechanisms such as feature extraction, aggregation, classification, and clustering. Finally, the extracted context is made accessible to the application and other external components.

A first basic strategy to support context provisioning consists of deploying specialized context providers to be installed on the device. We call this approach, depicted in Figure 1a, **internal context provisioning**. These context providers can be directly integrated into applications, though they can result in increased complexity, loss of generality and reuse, expensive and time-consuming application development. Alternatively, context providers can be organized in libraries, toolkits (e.g., Context Toolkit [3]), frameworks (e.g., TEA framework [9]), middleware (e.g., RCSM [13]), thus providing application developers with uniform context abstractions. However, it is not always possible to assume that individual mobile devices will carry any

type of conceivable sensor or be capable of interacting with any type of sensor embedded in the environment.

A second approach consists of deploying autonomous components that can be used by any application, exist independently of the application, and run on remote devices. These components constitute external service infrastructures [5] (e.g., Confab [6], JCAF [1]) or shared servers (e.g., the Active Badge Location System [11]). We call this approach **external centralized context provisioning** (see Figure 1b). These shared services are in charge of finding suitable context sources and processing, storing, and disseminating gathered context data. Multiple context providers running on different applications can poll or subscribe to these services for context information related to certain context entities. This approach leads to several benefits: *i)* by sharing sensors, processing power, and resources, the computational load on single devices is notably reduced; *ii)* applications turn out to be less tied to a specific sensor platform and potentially more capable of running on any networked device; *iii)* applications can gather context data that they might not be able to collect even if equipped with the necessary sensors (e.g., a region to be monitored is out of the local sensor's range). However, relying on a centralized system presents scalability, extensibility, and fault-tolerance issues.

A third possibility, albeit rarely exploited, is a distributed model as the one depicted in Figure 1c. We call this model **external distributed context provisioning**. The basic idea is to abstract context provisioning as the problem of supporting the access to a distributed database where data are provided by context providers located on the nodes of a Mobile Ad-hoc Network (MANET). Since devices located in proximity of each other are likely to share a similar context view, nodes equipped with the necessary sensors can acquire raw context data, process them, and make them accessible to other neighboring nodes. This approach is enabled by several emerging wireless networking infrastructures which permit to easily build proximity networks. This strategy also enables the collaborative recognition of the context of a group of context-aware devices.

The middleware we propose in this paper integrates

these three provisioning strategies thus enabling context-aware applications to employ the most suitable provisioning mechanism based on current needs and resource availability. For example, let us consider a sailing scenario where all sailing boats are equipped with GPS devices for sensing location and only few boats possess barometers for atmospheric pressure, anemometers for wind speed, hygrometers for humidity, and thermometers for temperature. Each boat continuously stores in a centralized infrastructure sensed context data. A certain boat provided only with GPS device can query at any time the centralized infrastructure to know about weather conditions in a region of interest. If any boat is currently sailing in that region, the infrastructure uses data collected by that boat to satisfy the weather query. Alternatively, boats within the communication range of each other can form an ad hoc network to share context information about the surrounding environment.

3. Contory Design and Implementation

The design of Contory is based on the following guiding principles:

- *Multiple context provisioning mechanisms*: Contory allows the deployment of different kinds of context provisioning mechanisms, namely internal sensor-based, external infrastructure-based, and distributed provisioning over ad hoc networks.
- *Constant availability*: context provisioning in a mobile environment needs to cope with the dynamism of the environment. Ideally, context provisioning should take place without any interruption, e.g., due to hardware faults or temporary disconnections of context sources. Hence, if a provisioning mechanism becomes unusable at certain point, a new strategy should be selected without the need of interrupting the application.
- *Common querying interface*: To formulate requests about heterogenous context items, Contory supports a SQL-like context query language. This common interface allows applications to specify the type and qualifying properties of the required context data.
- *Push and pull access mode*: Context-aware applications can interact with Contory by using either a pull or push mode. They can submit one-time response queries or long-running queries.
- *Modularity and extensibility*: Contory glues several context provider components together, thus making them interoperate in a constantly changing system. Separation of semantic definition of the provided information and availability of modular context providers enhances adaptation to variable configurations. Moreover, as we expect new sources of context data and new processing algorithms to be developed in the forthcoming years, these must be easily integrated.

In the following, we introduce the definition of context item and context query, and present the architecture and implementation of Contory along with its programming API.

3.1 Context Items and Context Metadata

The context associated to a certain situation can be expressed as a collection of *context items*. For instance, the situation *walking outside* can be expressed as the triplet $\langle noise=medium, light=natural, activity=walking \rangle$. Typical context items describe spatial information (location, speed), temporal information (time, duration), user characterization (activity, mood), environmental information (temperature, light, noise), and resource availability (nearby devices, device status). Context items can be further classified by metadata information such as freshness (how recent the value of the context item is), lifetime (for how long the value is considered valid), quality (correctness, precision, accuracy, completeness), level of privacy and trust.

In Contory, context data are exchanged by means of `cxtItem` objects. Each `cxtItem` includes `type`, `value`, `timestamp`, `lifetime`, `source`, and `options`. The `type` is the category of the context item. The set of available context types is called `ContextSpace`. The `value` field carries the current value(s) of the context item. `timestamp` contains the timestamp of the carried value and `lifetime` indicates the validity slot of such a value. `source` specifies the source of the context information such as the sensor type, the infrastructure identifier, or the device's name. `options` contains context metadata describing additional properties of the carried context data. `source` and `options` are optional fields.

3.2 Context Query Language

Contory refers to a novel language to define requests about needed context items, namely the *context query language*. It provides context-aware applications with a simple and intuitive SQL-based interface to define context queries. Our query template has the following format:

```
SELECT {context name} [*]
FROM {source}
WHERE {predicate}
FRESHNESS = time
DURATION = time [*]
EVERY = time
EVENT {predicate}
```

Clauses marked with [*] are mandatory. The `SELECT` clause specifies the type of the required context item. `FROM` permits to specify type and characteristics of the sources from which desired context data should be collected. Sources can be of three types (specified in the `typeSrc` field) according to the three types of context provisioning mechanisms supported. The application can also

explicitly specify the destination (through `destID`)(s) to whom the query must be sent (e.g., the user wants to be notified when a friend is nearby). In case of distributed provisioning, the FROM clause also tells multiplicity and distance of the source nodes to be selected in the ad hoc network. As Table 1 shows, the context collection can involve all nodes that can be discovered or the first (`kNodes`) found in a region of range smaller than `nHops`. Note that if the clause FROM is omitted, Contory selects the most suitable provisioning mechanism based on the resources available. The WHERE clause filters context values according to specific requirements on their associated context metadata. The FRESHNESS clause specifies how recent the context data must be. The DURATION clause specifies the query lifetime. Finally, our query language provides support for one-time response queries and long-running queries. Long-running queries are either periodic through the EVERY clause or event-based through the EVENT clause. The EVERY clause permits to specify the rate at which the query has to be re-processed. The EVENT clause determines the conditions that must be verified at the context provider's node to trigger a new result to be sent.

clause type	values
SELECT	low-level cxt: location, light, noise, temperature higher-level cxt: environment, activity, numOfPersons
FROM	typeSrc, all, first (kNodes), {destID}, range (nHops)
WHERE	accuracy, precision, levelOfTrust

Table 1. Example values for query clauses

3.3 Contory Architecture

The conceptual model of the proposed middleware is depicted in Figure 2. Our driving idea is to provide a specialized and transparent support for retrieving context items. Applications can then further exploit these items to infer more complex context elements. The core of the overall architecture is the *ContextFactory* which is based on the *factory method* design pattern [4]. The *ContextFactory* is also a *singleton* as only one instance of it exists in each application, thus representing the whole context access. The *ContextFactory* is in charge of supporting context provisioning through several *Facade* and *CxtProviders* components, and context storage/dissemination through the *QueryManager*, the *ContextRepository*, and the *ContextPublisher*. In the following, we describe these functionalities and shed some light on the architectural components as well.

i) Context provisioning: Context providers are in charge of processing the submitted context queries and of reporting results to the *QueryManager*. To accomplish this, they have to collect context items of the requested type-/properties, and, optionally, process raw sensed data to infer

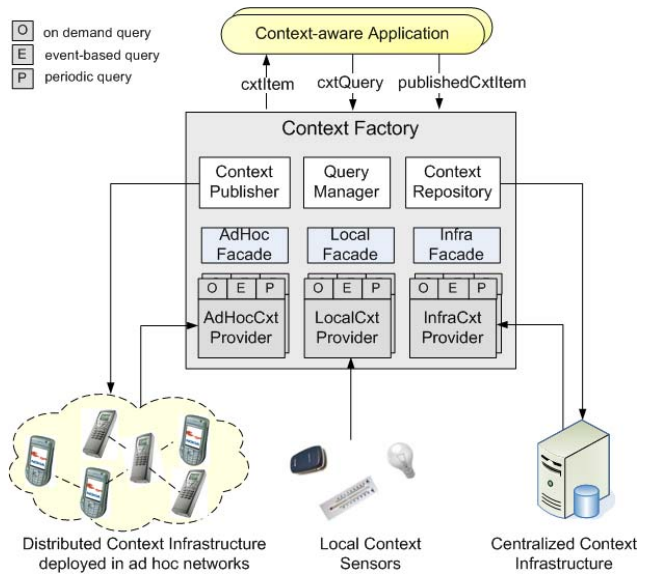


Figure 2. Contory middleware architecture

higher-level context. We assume that, depending on the application semantics, context reasoning mechanisms of different complexity can be implemented in specific context providers of the middleware or directly at application level. Context providers can be of three types: *LocalCxtProviders* for interacting with local sensors, *InfraCxtProviders* for accessing centralized context infrastructures, and *AdHocCxtProviders* for supporting distributed provisioning in ad hoc networks. For each of the three classes of providers a corresponding *facade* module offers a unified interface for accessing the context data collected by such a type of providers. The purpose of utilizing a *facade* is to decouple the access to the collected context data from the actual implementation of the provisioning mechanism. All types of context providers offer three different modes of interaction: *on demand query*, *event-based query*, and *periodic query*.

LocalCxtProviders are responsible for managing the access to local sensors. This is done by periodically polling the sensor devices and reporting values that match the WHERE and FRESHNESS requirements. In the current implementation, mobile phones interact with sensors of different types opening a Bluetooth connection.

InfraCxtProviders are responsible for retrieving context data from available context infrastructures by means of networking technology such as GPRS or WiFi. Since the centralized infrastructure can support more advance reasoning algorithms and maintain long history of past context data, these providers are particularly useful to provide access to more complex context data.

AdHocCxtProviders are responsible for gathering context data from nodes in a MANET. Essentially, any entity in the ad-hoc network can act as context-quierer and/or -

provider. For each issued query, available nodes in the ad hoc network collaborate with each other to forward the query based on the specified range of dissemination. If by the duration time specified in the query, no suitable *AdHocCxtProvider* is found, the query discovery phase ends. If matching *AdHocCxtProviders* are discovered, the requestor *AdHocCxtProvider* collects available data from the discovered context sources and verifies if all the requirements specified in the query are satisfied (e.g., freshness, accuracy, etc.). If the validation is successful, the value of the context item along with additional metadata properties is delivered to the *QueryManager*. In the case of periodic or event-based query, the discovery/collection/delivery process runs till the expiration of the query duration time.

ii) Context storage/dissemination: Three main modules realize this functionality. The *QueryManager* is in charge of interpreting the submitted context queries and providing results accordingly. The *ContextRepository* is in charge of storing pieces of context information related to each registered user. Only few recent context data are stored locally, while complete logs can be stored in remote repositories of external context infrastructures. The *ContextPublisher* allows publishing context items in ad hoc networks. These last two components are essential to provide sharing of context information among middleware platforms running on different nodes, and ultimately to aid the execution of *AdHocCxtProviders* and *InfraProviders*. Optionally, two access modalities can be associated to stored/published context items. Public access allows the access to any other entity. Authenticated access locks the item with a key that must be known by the requestor.

3.4 ContextFactory Interface

The *ContextFactory* provides the following services:

```

1 public interface ContextFactory{
2     String processCxtQuery (CxtQuery q);
3     void cancelCxtQuery (String qID);
4     String publishCxtItem (String cxtItem);
5     void cancelCxtItem (String itemID);
6     void storeCxtItem (String cxtItem);
7     Facade makeFacade (CxtQuery qID);
8     Facade getFacadeByQueryID (String qID);
9     CxtProvider getProviderByQueryID (String qID);
10    CxtPublisher getCxtPublisher ();
11 }

```

It offers services to the application developer for submitting/erasing context queries (*processCxtQuery*, *cancelCxtQuery*), publishing/erasing context items (*publishCxtItem*, *cancelCxtItem*), and storing context items (*storeCxtItem*). The other methods are utilized by internal Contory modules. The *makeFacade* method is the parameterized factory method. This method creates multiple (three in our case) kinds of *Facade* component based on the query requirements. Finally, the

*get** methods are used to get a reference to *Facade*, *CxtProvider*, and *CxtPublisher* objects.

To clarify how a context query is processed, let us consider the query submitted by a sailboater application that wishes to know the temperature in the surroundings by using nodes in the proximity:

```

<query name="temperature"
  select="temperature"
  <struct name="from">
    <String name="typeSrc">"ad-hoc"</String>
    <boolean name="all">true</boolean>
    <int name="range">1</int>
  </struct>
  <struct name="where">
    <int name="confidence">132</int>
  </struct>
  freshness=50
  duration=1000000
  every=100
</query>

```

This query is passed to the *processCxtQuery* method, which triggers the instantiation of an *AdHocTemperatureProvider*. Every 100s this will report collected temperature values not older than 50s.

3.5 Implementation and Evaluation

The middleware has been implemented by using Java 2 Micro-Edition (J2ME), both with the Connected Limited Device Configuration (CLDC) and Connected Device Configuration (CDC). The J2ME platform was selected since it currently represents the most widespread computing platform for personal mobile devices. Contory has been implemented in the context of the DYNAMOS Project. The focus of this project is to develop a system prototype for the support of context-aware services to mobile users. The main DYNAMOS application prototype is deployed to target the needs of a community of recreational sailboaters. We used such a sailing application prototype to derive requirements for the design of Contory and then we re-implemented the sailing application on top of Contory in order to support more context-based functionality, such as sharing of context information among sailboaters for estimating weather conditions or to support community-based functions.

In order to evaluate the performance of the proposed middleware, we deployed the middleware on an experimental testbed consisting of Nokia 6630 phones, Nokia 9500 communicators, Bluetooth GPS Receiver InsSif III, and a Bluetooth prototype sensor box capable of providing 3D accelerations, temperature, humidity, pressure, and light intensity. Within the DYNAMOS project, we built a context infrastructure capable of storing context data and of supporting their sharing among registered users. As for distributed provisioning, the current prototype implements one-hop communication using Bluetooth and multi-

hop communication using the Smart Messages [7] platform over WiFi.

4. Related Work

Most projects researching context support on mobile devices, typically focus only one possible strategy, such as internal sensor-based, centralized infrastructure-based or (rarely) distributed infrastructure-less provisioning. Our middleware can make use at the same time of all three provisioning mechanisms, thus complementing the potential unavailability of one mechanism with another one.

While many infrastructure-based provisioning strategies have been devised (e.g., Confab [6], JCAF [1]), infrastructure-less approaches exploiting the communication support offered by ad hoc networks have been rarely [12] considered to collect dynamic context information. Distributed approaches of this type resembles work done to access data stored in sensor networks (e.g., TinyDB [8]). However, these works consider only stationary sensors, whereas, in our distributed model, there are both stationary and moving context providers. Furthermore, in sensor networks properties/data produced by nodes are known at the deployment time, while in MANETs properties/context data differ over time as nodes of different types move across the physical space. Likewise, the idea of offering access to Contory through a SQL-interface takes inspiration from work done on sensor networks. Declarative queries are one of the preferred ways of interacting with static sensor networks [2]. We specialized our query language to offer support for expressing both type and quality of requested context items.

5. Conclusions and Future Work

This paper presented Contory, a middleware specifically deployed to enable programmers to develop context-aware applications for mobile phones. Given the dynamism of mobile environments and level of resource availability, Contory offers a flexible solution to context provisioning. It integrates different mechanisms for context provisioning (i.e., internal sensors-based provisioning, centralized context infrastructure, distributed context provisioning in ad hoc networks), and make them accessible by means of a simple SQL-based interface supporting context-query specification. Future research includes adding support to make Contory customizable for different devices and interoperable with different external infrastructures and sensor devices. Work is ongoing for enhancing the supported provisioning mechanisms and for developing more complex scenarios to carry out an extensive evaluation of the middleware functionality.

References

- [1] J. E. Bardram. The Java Context Awareness Framework (JCAF) - A Service Infrastructure and Programming Framework for Context-Aware Applications. In *Proceedings of the 3rd International Conference on Pervasive Computing (Pervasive 2005)*, 2005.
- [2] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Proceedings of the Second International Conference on Mobile Data Management (MDM '01)*, pages 3–14, London, UK, 2001.
- [3] A. K. Dey, D. Salber, and G. Abowd. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. *Human-Computer Interaction*, 16(2-4):97–166, 2001.
- [4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [5] I. Hong and J. A. Landay. An Infrastructure approach to context-aware computing. *Human-Computer Interaction*, 16(2-3):287–303, 2001.
- [6] J. Hong and J. Landay. An Architecture for Privacy-Sensitive Ubiquitous Computing. In *Proceedings of The Second International Conference on Mobile Systems, Applications, and Services (Mobisys'04)*, pages 177–189, Boston, MA, 2004.
- [7] P. Kang, C. Borcea, G. Xu, A. Saxena, U. Kremer, and L. Iftode. Smart Messages: A Distributed Computing Platform for Networks of Embedded Systems. *The Computer Journal, Special Focus-Mobile and Pervasive Computing*, pages 475–494, 2004. The British Computer Society. Oxford University Press.
- [8] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proceedings of The 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*, pages 491–502, San Diego, California, 2003. ACM Press.
- [9] A. Schmidt, K. A. Adoo, A. Takaluoma, U. Tuomela, K. V. Laerhoven, and W. V. de Velde. Advanced Interaction in Context. In *Proceedings of the First Symposium on Handheld and Ubiquitous Computing (HUC'99)*, pages 89–101, Karlsruhe, Germany, September 1999.
- [10] A. Schmidt and K. V. Laerhoven. How to Build Smart Appliances? *IEEE Personal Communications, Special Issue on Pervasive Computing*, 8(4):66–71, August 2001.
- [11] R. Want, A. Hopper, V. Falcao, and J. Gibbons. The Active Badge Location System. *ACM TOIS*, 10(1):91–102, 1992.
- [12] O. Wolfson and B. Xu. Data-on-the-Road in Intelligent Transportation Systems. In *Proceedings of the IEEE International Conference on Networking, Sensing, and Control (ICNSC 2004)*, Taipei, Taiwan, March 2004.
- [13] S. Yau and F. Karim. A context-sensitive middleware for dynamic integration of mobile devices with network infrastructures. *Journal Parallel Distributed Computing*, 64(2):301–317, February 2004.