Contory: A Middleware for the Provisioning of Context Information on Smart Phones

Oriana Riva

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

Abstract. Context-awareness can serve to make ubiquitous applications deployed for mobile devices adaptive, personalized, and accessible in dynamically changing environments. Unfortunately, existing approaches for the provisioning of context information in ubiquitous computing environments rarely take into consideration the resource constraints of mobile devices and the uncertain availability of sensors and service infrastructures. This paper presents the design, prototype implementation, and experimental evaluation of Contory, a middleware specifically designed to accomplish efficient context provisioning on mobile devices. To make context provisioning flexible and adaptive based on dynamic operating conditions, Contory integrates multiple context provisioning strategies, namely internal sensors-based, external infrastructure-based, and distributed provisioning in ad hoc networks. Applications can request context information provided by Contory using a declarative query language which features on-demand, periodic, and event-based context queries. Experimental results obtained in a testbed of smart phones demonstrate the feasibility of our approach and quantify the cost of supporting context provisioning in terms of energy consumption.

Key Words: Context-awareness, middleware, smart phones, energy consumption

1 Introduction

Context-awareness is emerging as a promising enabler of various ubiquitous applications deployed for usage on mobile devices. In principle, mobile devices can acquire context data through a large variety of sensors embedded in the device and in the surrounding environment. In practice, making context information available for usage to applications running on such devices often turns out to be an ambitious demand [1]. Mobile devices are typically resource-constrained, while context provisioning is often a complex process consisting of several sequential and parallel sub-processes, which can lead to significant power consumption and memory utilization; for example, reasoning algorithms can require large storage space and complex computations. The integration of sensors in mobile devices should not compromise portability, usability (e.g., size, weight, design, and aesthetics), cost, and lifetime of everyday mobile devices. Some sensors may not be operative in every environment (e.g., GPS in indoor environments).

Typically, context-aware applications either directly sense and locally process context data (e.g., Context Toolkit [2]) or rely on external context infrastructures [3], which collect, process, and disseminate context data of multiple entities. Additionally, the increasing availability of ubiquitous connectivity, such as Bluetooth and WiFi, on mobile devices makes feasible a distributed provisioning approach, in which devices share context information of different types in mobile ad hoc networks. These three strategies for context provisioning are all valuable, but they build upon specific assumptions which might not be always and constantly verified. In ubiquitous environments, operating conditions of mobile clients can vary widely over time and space. For instance, in resource-rich environments, powerful context infrastructures can provide applications with required context data, thus reducing the computational load on single devices. Conversely, in resource-impoverished environments, devices can rely either on their own sensors and processing capabilities or on neighboring devices. In order to cope with the dynamism and heterogeneity of such environments, more flexibility is required in accomplishing context provisioning.

This paper proposes the CONTextfactORY (Contory) middleware for context provisioning on smart phones. Contory offers an SQL-like interface to generate context queries, in which applications can specify type and quality of the desired context items, context sources, push or pull mode of interaction, and other properties. Contory processes context queries and collects context data by employing multiple strategies for context provisioning, namely internal sensors-based, external infrastructure-based, and distributed provisioning in ad hoc networks. This approach presents two advantages. First, arranging different context strategies permits compensating for the temporary unavailability of one mechanism and coping with dynamic resource availability. Second, combining results collected through different context mechanisms allows applications to partly relieve the uncertainty of single context sources and to more accurately infer higher-level context information. Since smart phones are becoming increasingly interesting to academia and industry as platforms for realizing the ubiquitous computing vision, the smart phone was selected as development platform. To assess system performance and quantify the energy consumption on smart phones, we ran experiments in a testbed of Nokia Series 60 and Nokia Series 80 phones. Moreover, to evaluate the practical feasibility of the proposed approach, we built a prototype application for a sailing scenario.

Core concepts and design principles for the deployment of Contory have been previously presented in [4]. This paper makes the following contributions: *(i)* it presents full design and implementation of a middleware supporting multiple strategies for context provisioning; *(ii)* among these strategies, it offers an infrastructureless approach to collect context data over mobile ad hoc networks; *(iii)* it describes a middleware and real-world applications implemented on a smart phone platform; *(iv)* it provides experimental results that give insights into the performance of smart phones in terms of energy consumption.

The rest of the paper is organized as follows. Section 2 discusses several existing context provisioning strategies. Section 3 presents requirements and design principles at the basis of Contory. Section 4 presents query model, software architecture, and programming interface of Contory, while implementation details are given in

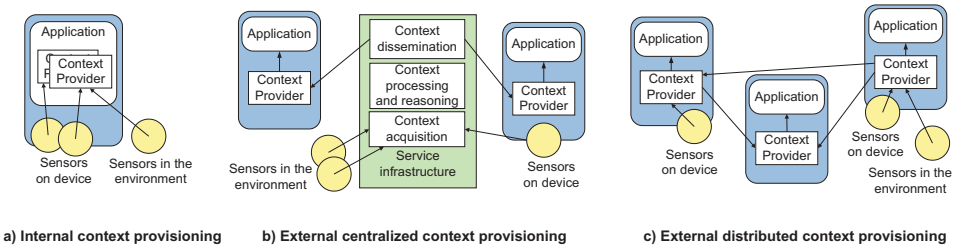

Fig. 1. Context provisioning strategies

Section 5. Section 6 describes experimental results and a prototype application using Contory. Section 7 discusses related work. The paper concludes in Section 8.

2 Context Provisioning

Context can be defined as any information that can be used to characterize the situation of an entity [2]. Context provisioning is the process by which context information is acquired, processed, and made available for usage. Hereafter, we refer to *context providers* as the software components in charge of performing context provisioning. Sensors integrated in the handheld device and in the environment, tags and beacons, positioning systems, biosensors on user can be used to acquire raw context data about the user's physical and social environment. Context providers process raw data using mechanisms such as feature extraction, aggregation, classification, and clustering in order to infer the user's context. Finally, the extracted context is made accessible to the application and other external components.

A large number of approaches have been proposed to support context provisioning. As Fig. 1a shows, a first basic strategy, called **internal context provisioning**, consists of deploying specialized context providers to be installed on the device. The integration of these context providers into applications can lead to increased complexity, loss of generality and reuse, and expensive and time-consuming application development. Alternatively, context providers can be organized in libraries, toolkits (e.g., Context Toolkit [2]), frameworks (e.g., TEA framework [5]), middleware (e.g., RCSM [6]), thus providing application developers with uniform context abstractions. However, in many situations, it is unrealistic to assume that individual mobile devices will constantly carry any type of conceivable sensor or will be capable of interacting with any type of sensor embedded in the environment.

A second strategy consists of deploying autonomous context-service components, running on remote devices, accessible by multiple applications, and independent of the application logic. Existing examples are external service infrastructures [3] (e.g., Confab [7] and JCAF [8]), and shared servers (e.g., the Trivial Context System (TCoS) [9]). We call this approach, depicted in Fig. 1b, **external centralized context provisioning**. These shared context services are in charge

of discovering suitable context sources and processing, storing, and disseminating gathered context data. Multiple context providers on different applications can pull or subscribe to these services to retrieve context information related to certain context entities. On the one hand, by sharing sensors and computing resources, this approach reduces the computational load on single devices and makes applications less tied to a specific sensor platform. On the other hand, relying on a centralized system presents scalability, extensibility, and fault-tolerance issues.

A third possibility, albeit rarely considered in this field, is a distributed model as the one depicted in Fig. 1c. We call this approach **external distributed context provisioning**. The key 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). Nodes equipped with the necessary sensors can acquire raw context data, process them, and make them accessible to neighboring nodes. Ubiquitous connectivity already available on commercial mobile devices enables proximity networks of this type.

3 Contory Requirements and Design

The deployment of a middleware for context provisioning stems from the necessity to move a number of core data and services for context sensing, management, and distribution from their multiple instances into a centralized provision of services. Requirements for the deployment of Contory were gathered from experiences with a context-based application developed in the DYNAMOS[1] Project. The DYNAMOS application, described in [10], aims to proactively provide mobile users with nearby services that are of interest based on the user's current context and needs. The application prototype runs on smart phones and was specifically designed to target the needs of a community of recreational sailboaters. In June and August 2005, we conducted two field trials with sailboats in which such an application was used by about 30 persons equipped with Nokia 6630 phones and GPS devices.

During the regatta, location-awareness was accomplished by means of GPS devices connected through Bluetooth (BT) to the phone. Location updates were encapsulated in events and constantly transmitted over GPRS/UMTS for storage in a remote repository. Collected location traces were fairly discontinuous due to several disconnection problems. First, several BT disconnections between the phone and the GPS device occurred (typically one disconnection per hour). Second, when a UMTS connection was active and the phone went through 2G/3G handover, the phone switched off (this did not occur if the phone was set to operate only in 2G mode). Additionally, the traffic of events carrying context updates and going from the phone to the remote repository had to be optimized and largely reduced, in order to avoid the phone to switch off due to high memory consumption and network connectivity problems.

Besides these phone-specific issues, these experiences in the real field of action helped discover technical problems regarding context sensing and context management. We found that *(i)* context provisioning based exclusively on local sensors is

[1] Dynamic Composition and Sharing of Context-Aware Mobile Services. URL: http://virtual.vtt.fi/virtual/proj2/dynamos/

often not reliable enough; *(ii)* sharing of context information owned by multiple users can provide useful services to the end-user, enlarge the spatial range of context monitoring, reduce global resource utilization, and permit coping with sensor unreliability; *(iii)* external infrastructures should be ready to cope with frequent user disconnections (e.g., by incorporating prediction or learning algorithms); and *(iv)* the client application should be ready to cope with frequent disconnections from remote repositories.

The design of Contory followed four main guiding principles:

- *Flexible and reliable context provisioning*: Ideally, context provisioning should take place without any interruption, e.g., due to hardware faults or temporary disconnections from context sources. In Contory, multiple context provisioning strategies are made available and can be dynamically and transparently interchanged based on sensor availability and resource consumption.
- *Common querying interface*: To formulate requests about heterogenous context items, Contory supports an SQL-like context query language. This common interface allows applications to specify 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 a push mode; they can submit on-demand queries or long-running queries (periodic or event-based queries).
- *Modularity and extensibility*: Contory glues several context provider components together. Separation of semantic definition of the provided information and availability of modular context providers enhances adaptation to variable configurations. New sources of context information and processing algorithms, which will be developed in the forthcoming years, will need to be easily accommodated in the existing architecture.

4 Contory Middleware Architecture

Contory aims to provide specialized and transparent support for retrieving context items of different types and quality. This section describes core concepts for the design of Contory, its software architecture, and programming interface.

4.1 Context Items and Context Metadata

The context associated with a certain situation can be expressed as a set of *context items*, each describing a specific element of the situation. For instance, the situation *walking outside* could be represented by the triplet <*noise=medium, light=natural, activity=walking*>. Context items can describe spatial information (location, speed), temporal information (time, duration), user status (activity, mood), environmental information (temperature, light, noise), and resource availability (nearby devices, device power). In Contory, context data are exchanged by means of `cxtItem` objects. Each `cxtItem` consists of `type` (context category), `value` (current value(s) of the item), and `timestamp` (the time at which the context item had such a value). Optionally, it can have a `lifetime` (validity duration), a `source` identifier (e.g., sensor, infrastructure, and device addresses), and

other `metadata` information. Types of metadata information include correctness (i.e, closeness to the true state), precision, accuracy, completeness (if any or no part of the described information remains unknown), and level of privacy and trust.

4.2 Context Query Language

From an application's point of view, Contory mostly acts as a data-retrieval system to which context-aware applications submit context queries. Although similar to a database system, the dynamism and fuzziness of context data lead to important differences. Context sources can provide large amounts of context data, hence some aggregation and filtering functions are required. Context monitoring is a continuous process, hence not only on-demand queries but also long-running queries have to be supported. Although different applications have usually different requirements, rather than deploying application-specific interfaces, we abstracted the functionality of several applications into one common SQL-like *context query language*. The query template has the following format.

```
SELECT <context name> [*]
FROM <source>
WHERE <predicate clause>
FRESHNESS <time>
DURATION <duration> [*]
EVERY <time> | EVENT <predicate clause>
```

The SELECT and DURATION clauses (marked with [*]) are mandatory. SELECT specifies the `type` of the requested context item. DURATION specifies the query lifetime as time (e.g, `1 hour`) or as the number of samples that must be collected in each round (e.g., `50 samples`).

Contory aims to offer different levels of transparency to the application developer. The maximum transparency is achieved when the FROM clause is unspecified and the middleware autonomously and dynamically selects the context provisioning mechanism to be employed. Alternatively, the FROM clause offers to the programmer several ways to control type and characteristics of the context sources to be employed. Context sources can be of three kinds according to the three context provisioning mechanisms supported: internal sensor-based (`intSensor`), external infrastructure-based (`extInfra`), and distributed context provisioning in ad hoc networks (`adHocNetwork`). In the case of `adHocNetwork` provisioning, the FROM clause also tells multiplicity (`numNodes`) and distance (`numHops`) of the context source nodes. For example, the search for suitable context items can involve all nodes that can be discovered (`numNodes=all`) or the first k nodes found within a distance lower than j hops (`numNodes=k`, `numHops=j`). Alternatively, the programmer can also specify the `destination` to which the query has to be sent. This destination can be the identifier of an entity (e.g., to know when a friend is nearby) or the coordinates of a region to be monitored (e.g., next exit on the highway).

WHERE contains filtering predicates expressed using the context item's `metadata`. FRESHNESS specifies how recent the context data must be. Finally, our query language provides support for long running queries by means of EVERY and EVENT

clauses. These clauses are mutually exclusive. The EVERY clause allows the application to specify the rate at which context data should be collected (periodic query). The EVENT clause determines the set of conditions that must be met at the context provider's node before a new result is returned (event-based query).

In the following example, the query returns, for one hour, temperature values collected from the first 10 nodes found in an ad hoc network within a distance of at most 3 hops; data are not older than 30 seconds, have accuracy of 0.2 ^{o}C, and are sent every time the average temperature exceeds 25 ^{o}C.

```
SELECT temperature
FROM adHocNetwork(10,3)
WHERE accuracy=0.2
FRESHNESS 30 sec
DURATION 1 hour
EVENT AVG(temperature)>25
```

4.3 Contory Software Architecture

Fig. 2 depicts the conceptual architecture of Contory. *ContextFactory* is the core component of the overall architecture. One *ContextFactory* is instantiated on each device and made accessible to multiple applications. Based on the *Factory Method* design pattern [11], this design model aims to define an interface for creating objects, but let subclasses decide which class to instantiate. In our case, the *ContextFactory* offers an interface to submit context queries, but lets *Facade* components (subclasses) decide which *ContextProvider* components (classes) to instantiate. The *ContextFactory* provides support for *(i) context sensing and communication*, *(ii) context provisioning and sharing*, and *(iii) queries and providers management*. In the following, we describe each functionality along with their core architectural components.

Context Sensing and Communication Context data can be sensed from a large variety of *CxtSources* such as external sensors (e.g., a GPS device), integrated monitors (e.g., a power management framework), external servers (e.g., a weather station). To provide discovery of *CxtSources* as well as to support communication with them, different types of *Reference* modules can be available on the device. Typically, a *Reference* mediates the access to a certain communication module by offering useful programming abstractions. As shown in Fig. 2, Contory includes four types of *References*. The *InternalReference* is specialized to support communication with sensors integrated in the device. The *BTReference* provides support to discover BT devices and services, and to communicate with them. The *WiFiReference* manages communication in WiFi networks, but also provides abstractions for content-based routing, geographical routing, and multi-hop communication in ad hoc networks. The *2G/3GReference* manages communications with remote entities over the corresponding network standards and offers an event-based interface.

Mobile systems can undergo unexpected changes in the level of resource availability, for example, when a new application is started or when the host moves to a different network domain. Moreover, in wireless environments, disconnections

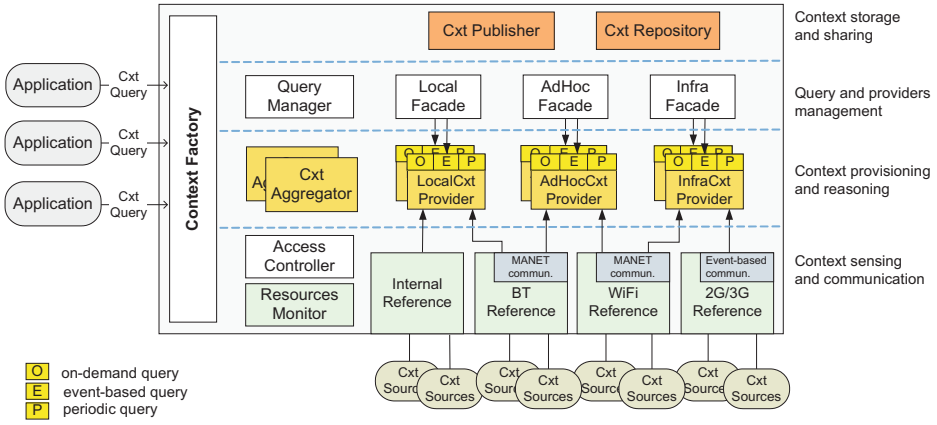

Fig. 2. Contory middleware architecture

and bandwidth fluctuations are common. These issues make necessary the adoption of dynamic resource allocation mechanisms. The *ResourcesMonitor* component is in charge of maintaining an updated view on the status of several hardware items (e.g., device drivers), on the device's overall power state, and on the available memory space. Each time, network, sensors, or device failures affect the functioning of a communication module, the corresponding *Reference* notifies the *ResourcesMonitor* module. This, in turn, will inform the *ContextFactory* which will enforce a reconfiguration strategy to take over. For example, if a BT-GPS device suddenly disconnects, the location provisioning task can be moved from a *LocalLocationProvider* (using the *BTReference*) to an *AdHocLocationProvider* (using the *WiFiReference*).

The *AccessController* module is responsible for controlling the interaction with external sources and requesters of context items. The *AccessController* keeps track of previously connected context sources (such as sensors or devices) and also of blocked context sources. This list is continuously refreshed so that only the most recent and the most often accessed sources are kept in memory. If the application requires high-security operating mode, every time a new context source is encountered, it is blocked or admitted based on explicit validation by the application. In low-security mode, every new entity is trusted.

Context Provisioning and Sharing *CxtProviders* are responsible for accomplishing context provisioning. Optionally, they can also incorporate reasoning mechanisms for inferring higher-level context data. A *CxtAggregator* can be used to combine context items collected from single or multiple *CxtProviders*. Alternatively, advanced context processing mechanisms can be performed by external context infrastructures, distributed across remote components or implemented at the application level.

CxtProviders are of three types. *LocalCxtProviders* manage the access to local sensors which can be integrated in the device or be accessible via BT. These providers periodically pull sensor devices and report values that match WHERE and FRESHNESS requirements. *InfraCxtProviders* are responsible for retrieving context data from remote context infrastructures. *AdHocCxtProviders* are responsible for supporting distributed context provisioning in ad hoc networks; to gather context data from nodes in a MANET, these providers utilize the *BTReference* (only for one-hop routing) or the *WiFiReference* (also for multi-hop routing). Based on the EVENT and EVERY clauses specification, context providers offer three modes of interaction: *on-demand query*, *event-based query*, and *periodic query*.

The *CxtRepository* module is responsible for storing gathered context information, locally or remotely. Only a few recent context data are stored locally, while complete logs can be stored in remote repositories of context infrastructures. The *CxtPublisher* allows publishing context information in ad hoc networks by means of the *BTReference* or the *WiFiReference*. Each time a context item has to be published, two access modalities can be applied: public access allows any external entity to access the item, and authenticated access locks the item with a key that must be known by the requester.

Queries and Providers Management The *QueryManager* is responsible for maintaining an updated list of all active queries and for assigning queries to suitable *Facade* components. For each of the three types of context provisioning mechanisms supported, a corresponding *Facade* module offers a unified interface for managing *CxtProviders* of that specific type. The purpose of utilizing the *Facade* design pattern [11] is to abstract the subsystem of *CxtProviders* to offer a more convenient (unidirectional) interface to the *ContextFactory*. The *Facade* knows which subsystem classes (i.e., *CxtProviders*) are responsible for a certain query and can direct actions or requests of the *ContextFactory* to the correct component.

The *QueryManager* invokes the factory method `processCxtQuery(CxtQuery q)` of the *ContextFactory* to assign the query to one or multiple *Facades*. The assignment is done base on the requirements specified in the query's FROM clause, based on sensor availability, and in the respect of the active control policies. For instance, a control policy can specify the maximum level of memory and power consumption that should be tolerated at runtime. Control policies are formulated as `contextRules` consisting of a condition and an action statements. Conditions are articulated as Boolean expressions, and the operators currently supported are `equal`, `notEqual`, `moreThan`, and `lessThan`. An example of condition is `<batteryLevel,equal,low>`. Through `and` and `or` operators, elementary conditions can be combined to form more complex ones. Whenever a condition is positively verified at runtime, the associated action becomes active and it is enforced by the *ContextFactory*. Actions currently supported are `reducePower`, `reduceMemory`, and `reduceLoad`. The enforcement of these actions can have different effects such as the switch from a certain provisioning mechanism to another one or the interruption of a query execution. For example, the activation of the `reducePower` action can cause the suspension or termination of high energy-consuming queries

(e.g., those using the *2G/3GReference*) or the replacement of WiFi-based multi-hop provisioning with BT-based one-hop provisioning.

Once the query has been assigned to a *Facade*, in order to avoid redundancy and keep the number of active queries minimal, the *Facade* performs query aggregation. This process consists of two sub-processes: query merging and post-extraction. The *Facade* first checks whether the new submitted query `q1` can be merged with any other active query `q2`. If `q3 = merge(q1,q2)` can be found, `q3` is the new query to be processed. The post-extraction sub-process is applied to the received results for `q3` in order to extract the data matching the original queries `q1` and `q2`. The `merge` function implements a simplified version of the clustering algorithm defined in [12]. This algorithm builds on the definition of a "distance" metric between queries. The algorithm computes the distance between each pair of queries and if it is below a certain threshold, the two queries are put in the same cluster. In our design, for simplicity, we put in the same cluster queries with the same `SELECT` clause. Once clusters are formed, the merging is performed by applying clause-specific merging rules, as exemplified below:

```
q1:                          q2:                          q3:
SELECT temperature           SELECT temperature           SELECT temperature
FROM adHocNetwork(all,3)     FROM adHocNetwork(all,1)     FROM adHocNetwork(all,3)
FRESHNESS 10sec              FRESHNESS 20sec              FRESHNESS 20sec
DURATION 1hour               DURATION 2hour               DURATION 2hour
EVERY 15sec                  EVERY 30sec                  EVERY 15sec
```

Upon the aggregation process has completed, the *Facade* module either instantiates a new *CxtProvider* or updates the query parameters of an existing *CxtProvider* (e,g., in case the new query has been merged with an already active query). *CxtProviders* of different *Facades* can be assigned to the same query, but each *CxtProvider* is assigned only to one (single or merged) query at time.

4.4 Contory Programming Interface

The Contory API shields the programmer from the underlying communication platforms and context provisioning aspects. To interact with Contory, an application needs to implement a `Client` interface and implements the following methods:

- `receiveCxtItem(CxtItem cxtItem)` in order to handle the reception of collected context items;
- `informError(String msg)` to be called by several Contory modules in case of malfunctioning or failure;
- `makeDecision(String msg)` to be invoked by the *AccessController* to grant or block the interaction with external entities.

The application can access Contory services through the `ContextFactory` interface. As shown below, this interface offers methods for submitting and erasing context queries (line 2 and 3), for publishing or erasing context items (line 4), and for remotely storing context items (line 5). In order to be eligible to publish context items and made them accessible to other clients, the publisher must register and be authenticated (line 6). Likewise, the client can deregister (line 7).

```
1 public interface ContextFactory{
2   boolean processCxtQuery (CxtQuery query);
3   void cancelCxtQuery(String queryID);
4   boolean publishCxtItem(String cxtItem, boolean published);
5   void storeCxtItem(CxtItem cxtItem);
6   void registerCxtServer(CxtServer client);
7   void deregisterCxtServer(CxtServer client);
8 }
```

Different vocabularies are made available to the application developer: *(i)* the `CxtVocabulary` contains context types, context values, and metadata types for specifying context items and device resources; *(ii)* the `QueryVocabulary` contains parameters for specifying context queries; and *(iii)* the `CxtRulesVocabulary` contains operators and actions for specifying control policies.

5 Implementation

Contory has been implemented using Java 2 Platform Micro Edition (J2ME). Currently, two separate implementations exist: one for Connected Limited Device Configuration (CLDC) 1.0 and Mobile Information Device Profile (MIDP) 2.0 APIs, and one for Connected Device Configuration (CDC) 1.0. The J2ME platform was selected since it currently represents the most widespread computing platform for personal mobile devices. All software development was done using Nokia Series 60 and Nokia Series 80 phones. In the following, we provide specific insights into the implementation of *References* and distributed context provisioning. The *InternalReference* module has not been implemented yet because no sensors integrated in the phone platform used for the development were available at deployment time.

5.1 References Implementation

The *BTReference* utilizes the Java Specification Request 82 (JSR-82) available for CLDC. This specification defines a standard set of APIs for BT wireless technology and specifically targets devices that are limited in processing power and memory. The specification includes support for *(i)* discovery (device discovery, service discovery, and service registration), *(ii)* communication (establishing connections between BT devices and using those connections for BT communication), and *(iii)* device management (managing and controlling these BT connections).

Since no standardized support exists to program ad hoc networks, the *WiFiReference* provides device and service discovery, content-based routing, multi-hop communications in ad hoc networks by means of the Smart Messages (SM) [13] distributed computing platform. This was specifically designed for highly volatile networks such as MANETs. We utilize the portable version of SM [14] implemented for the J2ME CDC platform. An SM is a user-defined application, similar to a mobile agent, whose execution is sequentially distributed over a series of nodes using execution migration. The nodes on which SMs execute are named by properties, called *tags*, and discovered dynamically using application-controlled routing. Tags have a name, similar to a file name in a file system, which is used for content-based

naming of nodes. To move between two nodes of interest, an SM explicitly calls for execution migration. An SM consists of *code bricks*, *data bricks* (mobile data explicitly identified in the program), and execution control state. To support SM execution, the SM runtime system runs inside a Java virtual machine and consists of: *(i) admission manager* that performs admission control and prevents excessive use of resources by incoming SMs, *(ii) code cache* that stores frequently executed code bricks, *(iii) scheduler* that dispatches ready SMs for execution on the Java virtual machine, and *(iv) tag space* that provides a shared memory addressable by names for inter SM communication and synchronization. The tag space offers a uniform view of the network resources in terms of naming and access to resources. We use SM tags to publish context items in the ad hoc network.

The *2G/3GReference* offers support for event-based communication by using the Fuego middleware [15]. This middleware is implemented in Java and provides a scalable distributed event framework and XML-based messaging service. This middleware also runs on mobile phones supporting Java MIDP 1.0.

5.2 Distributed Context Provisioning using BT and SM

Distributed context provisioning has been implemented using the *BTReference* in one-hop ad hoc networks and using the *WiFiReference* in multi-hop ad hoc networks. Distributed context provisioning is accomplished in three phases: initialization, publishing, and execution.

In the BT-based implementation, the initialization phase places the BT device into inquiry mode and specifies an event listener that will respond to inquiry-related events. A context item can be published by advertising a context service on the BT server (service registration). The server creates a service record describing the offered context service and adds it to the server's Service Discovery Database (SDDB). This is visible and available to external BT entities. The *AdHocCxtProvider* first discovers accessible BT devices (in some cases a list of pre-known devices is used) and then looks for available services on the discovered devices.

In the WiFi-based implementation, the *WiFiReference* expresses its willingness to participate in the Contory ad hoc network by exposing the tag "contory". In such a way, every time an SM needs to be routed from a certain source to a certain destination, all nodes in the ad hoc network exposing the "contory" tag will collaborate with each other to forward the SM towards the destination. To publish a context item in the ad hoc network, the *AdHocCxtPublisher* exposes on the local node a tag whose name contains the type and whose value contains the value and metadata of the context item (e.g., $(temperatureTag :< name = temperature >, < value = 14^oC, 1^oC, trusted >)$). To discover context items of interest, the context query is encapsulated in an `SM-FINDER` that is routed towards nodes exposing the desired context tag (i.e., the tag whose name matches the SELECT clause of the carried query). To disambiguate between multiple messages, a unique identifier is associated with each query and with each result. If no valid result is received within a certain timeout, the query is cancelled. If nodes exposing context items of the type of interest are discovered, WHERE, FRESHNESS and EVENTS requirements specified in the query are evaluated. If positively verified,

Table 1. Latency times of basic Contory operations

Entity acts as:	Operation	Elapsed time (msec) Avg [90% Conf interval]
ContextProvider	createCxtItem	0.078 [0.001]
	adHocNetwork, BT-based: publishCxtItem	140.359 [0.337]
	adHocNetwork, WiFi-based: publishCxtItem	0.130 [0.006]
	extInfra, UMTS-based: publishCxtItem	772.728 [158.924]
ContextRequester	createCxtQuery	0.219 [0.001]
	adHocNetwork, BT-based, one hop: getCxtItem	31.830 [0.151]
	adHocNetwork, WiFi-based, one hop: getCxtItem	761.280 [28.940]
	adHocNetwork, WiFi-based, two hops: getCxtItem	1422.500 [60.001]
	extInfra, UMTS-based: getCxtItem	1473.000 [275.000]

the value of the context item along with additional metadata properties are saved in the `SM-FINDER` which is routed back to the query issuer. In order to cope with nodes mobility, the `SM-FINDER` maintains a `hopCnt` that indicates how many hops the message has traversed until that moment. When the `SM-FINDER` is delivered to the *AdHocCxtProvider* issuer, if `hopCnt`>`numHops` the receiver discards the result because the *CxtPublisher* that provided such a result is out of the range of interest.

6 Evaluation

We evaluated Contory in two phases. We built an experimental testbed of smart phones and measured response times and energy consumption for different context operations. We then evaluated Contory by building a real-world application using it. This section presents experimental results and prototype application.

6.1 Experimental Results

The objective of this experimental analysis was to demonstrate the practical feasibility of the proposed approach, give an insight on the performance of our prototype implementation, and quantify its cost mostly in terms of energy consumption. Our experimental testbed consisted of a Nokia 6630 phone (Symbian OS 8.0a, 220 MHz processor, WCDMA/EDGE, 9 MB of RAM), a Nokia 7610 phone (Symbian OS 7.0s, 123 MHz processor, GPRS, 9 MB of RAM), 3 Nokia 9500 communicators (Symbian OS 7.0s, 150 MHz processor, WLAN 802.11b/EDGE, 64 MB of RAM), and a Bluetooth GPS Receiver InsSirf III.

Latency Experiments Table 1 reports latency times for four main Contory operations: `createCxtItem`, `publishCxtItem`, `createCxtQuery`, and `getCxtItem`. The size of a context query object is 205 bytes, while the size of a context item varies from 53 bytes (e.g., a wind item) to 136 bytes (e.g., a location item). For these experiments, we used a `lightItem` whose size is 136 bytes. `CxtItem` and `cxtQuery` objects that are transmitted over UMTS using the event-based platform are encapsulated in event notifications whose size is 1696 bytes.

Fig. 3. Power measurements testbed setup

On the context publisher side, publishing a context item with the BT-based mechanism takes much longer than with the WiFi-based mechanism. The reason for this stems from the BT registering process. With BT, to make an item accessible, this needs to be encapsulated in a `DataElement` and registered into the BT `ServiceRecord`. With SM, this operation corresponds to simply creating a new SM tag and storing its name and value in the `TagSpace` hashtable. The variability of latency times for publishing a context item in the remote infrastructure is quite extreme and is due to the high delay variability in UMTS networks.

On the context provider side, adHocNetwork provisioning can be BT-based or WiFi-based. For the BT case, the latency time reported in the table represents the time needed to receive a context item, once device and service discovery has occurred (BT device discovery takes approximately 13 sec and BT service discovery takes approximately 1.12 sec). For the WiFi case, we ran experiments using a 2-hops topology with three communicators arranged in a line. The two latency times reported in the table represent the time needed to retrieve one context item located at a distance of one or two hops, once the route has been built. The additional time required to build the route is approximately twice the corresponding latency value in the table. The break-up analysis for SM experiments shows that connection establishment accounts for 4-5% of the total latency time, serialization for 26-33%, thread switching for 12-14%, and transfer time for 51-54%. The SM overhead is negligible. Finally, measured latency times for extInfra vary enormously, ranging from 703 msec up to 2766 msec.

Energy Consumption Experiments Energy consumption remains one of the most critical issues that needs to be addressed in application development on mobile phones. While CPU speed and storage capacity have increased over the last 10 years, battery energy shows the slowest trend in mobile computing [16]. To measure energy consumption on phones, we inserted a multimeter in series between the phone and its battery. The testbed setup is shown in Fig. 3. We used a Fluke 189

multimeter, which was connected to a PC to record the readings. The meter read current inputs approximately every 500 ms. The precision of our measurements depends mostly on the precision of the multimeter and the stability of the voltage on the phone battery. The resistance of the wires was found to be negligible. The multimeter has an accuracy of 0.75% and precision of 0.15%. The stability of the voltage is important since this is used to compute the power consumption based on Ohm's law. We did some preliminary experiments to measure the voltage on the phone while performing different operations; we found out that under high load the battery deviated less than 2% from 4.0965 V for the first hour at least. To minimize the impact of the voltage variance, we ran short experiments and always with a full battery. Given that the shunt voltage of the meter is 1.8 mV/mA, we calculated that the maximum inaccuracy of our experiments was approximately 8%. We ran the experiments in an office environment with background noise due to other mobile phones, wireless LANs, BT, etc. Even though a noise-free environment would have been desirable, we ran all experiments in the same spot, thus emulating a daily life scenario with an almost constant level of background noise.

All experiments were performed from five to ten times. High energy consuming experiments were set to last no longer than 10 min. All numbers hereafter reported were collected on a Nokia 6630 phone and a Nokia 9500 communicator (only when WiFi was used). Initially, we measured the cost of different operating modes when the GSM radio was turned off. When BT is turned off, back-light is switched on, and display is switched on, the average power consumption is about 76.20 mW. If the back-light is turned off, the consumption decreases to 14.35 mW. A consumption of 5.75 mW is achieved if also the display is turned off. Turning on BT in page and inquiry scan state increases the power consumption to 8.47 mW. Turning on Contory as well leads to a power consumption of 10.11 mW. We ran all experiments (except UMTS-based tests) with the GSM radio off, back-light off, and display off.

Table 2 reports energy consumption results for all three context provisioning mechanisms. On the provider side, the energy consumption for providing context items is relatively contained. On the requester side, we distinguish three cases.

For BT-based mechanisms, the cost of processing context queries is mostly due to the device discovery phase which lasts approximately 13 sec. Once the BT device is discovered, being periodically notified with context data is fast and the energy cost is definitely low. Results for intSensor were gathered by connecting the BT-GPS device to the phone. While the discovery cost is the same for BT-based intSensor and adHocNetwork, the cost for maintaining a periodic exchange of data is higher for intSensor. This is due to the larger size of the exchanged data (GPS-NMEA data are 340 bytes big) and the packet segmentation BT applies.

For WiFi-based provisioning, energy costs are much higher than in the BT cases. We encountered several problems in running these experiments. Each time a WiFi connection was established on the communicator inserted in the circuit, the communicator switched off after less than 30 sec. New smart phones are low-voltage devices operating from a single Lithium-Ion cell. During the startup phase, the high in-rush current causes the phone's voltage supply to drop due to the multimeter's internal resistance; hence, this drop triggers the internal power management protection circuit to turn off the phone. However, based on the logs we gathered,

Table 2. Energy consumption of different context provisioning mechanisms

Context provisioning method: operation	Energy consumption per cxtItem (Joule) Avg [90% Conf Interval]
adHocNetwork, BT-based: provideCxtItem	0.133 [0.002]
adHocNetwork, BT-based: getCxtItem (one-hop and on-demand query, including discovery)	5.270 [0.010]
adHocNetwork, BT-based: getCxtItem (one hop and periodic query, without discovery)	0.099 [0.007]
intSensor, BT-based: getCxtItem (periodic query, without discovery)	0.422 [0.084]
adHocNetwork, WiFi-based: getCxtItem (one hop and periodic query)	> 0.906 [a]
adHocNetwork, WiFi-based: getCxtItem (two hops and periodic query)	> 1.693 [a]
extInfra, UMTS-based: getCxtItem (on-demand query)	14.076 [0.496]

[a] includes the cost of having back-light switched on

having WiFi connected at full signal (with back light on) drains a constant current of 300 mA, which leads to an average power consumption of 1190 mW. This also means that having WiFi connected is more than 100 times more energy-consuming than having BT in inquiry mode.

In the tests for extInfra provisioning, turning on the GSM radio produces an additional power consumption; this comes in peaks of 450-481 mW and every 50-60 sec. Fig. 4 shows the power consumption for a test in which 5 queries were sent to the infrastructure over UMTS, every 3 min. The maximum power consumption, which corresponds to when the connection is opened and the request for the item is sent, is 1000 mW. Such a high energy cost is mostly due to the cost of opening the UMTS connection. Sending and retrieving larger groups of items in the same time slot largely reduces the energy consumption per item.

To demonstrate how Contory is able to recover from sensor failures by dynamically switching from one context provisioning mechanism to another, we simulated a GPS failure. As Fig. 5 shows, initially the phone is retrieving location data from a GPS device connected through BT. After 155 sec, we caused a GPS failure by manually switching off the GPS device. As a reaction, Contory switches from sensor-based provisioning to ad hoc provisioning and starts collecting location data from a neighboring device. Later on, the GPS device becomes available again. Once the GPS device is discovered, Contory switches back to sensor-based provisioning. The cost in terms of power consumption of the switches is due mostly to the BT device discovery: this varies from 163 mW up to 292 mW.

Experiments Summary These experimental results confirmed the practical feasibility of our approach. The combined use of different context provisioning strategies can bring several benefits. First, it allows to cope with failures of sensing devices by dynamically replacing one context strategy with another. Second, as each con-

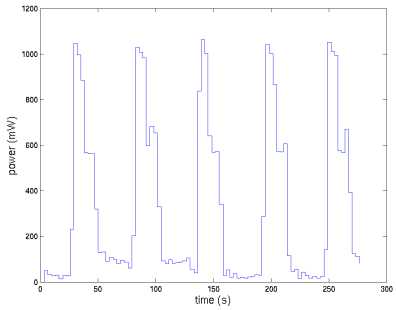

Fig. 4. Power consumption for extInfra provisioning

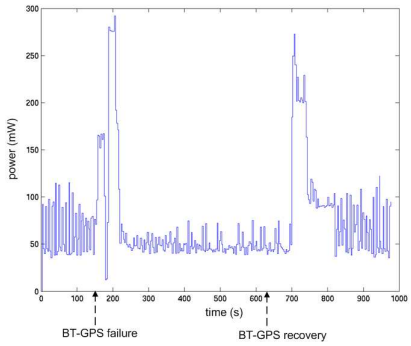

Fig. 5. Contory behaviour in the presence of BT-GPS failure

text provisioning strategy guarantees different performance at different costs, the possibility of flexibly switching from one mechanism to another permits optimizing the utilization of computing and communication resources at run time.

6.2 Sailing Application Prototype

Using the Contory API, we re-implemented the DYNAMOS sailing application [10] and add more context-based services. The use of Contory permitted to decouple the application implementation from underlying communication modules (e.g., the BT JSR-82, the Fuego Core event-based framework, the SM platform), from the repository system, and from sensor technology. The implementation of common services such as connecting BT sensors or communicating with the remote repository was accomplished by simply instantiating context query objects in few lines of code. Moreover, Contory offered support to: *(i)* extend the application's context monitoring range by collecting region-specific observations through ad hoc networks and making those data available to remote clients through the infrastructure support; *(ii)* share context information about multiple entities and across multiple devices;*(iii)* combine information from multiple context sources to enhance context estimation. In the following, we show two services that have been integrated into the previous DYNAMOS application and make use of these features.

WeatherWatcher : it allows users to retrieve weather information in a certain geographical region (e.g., the user wants to know the weather in the proximity of a guest harbor to visit). Weather information consists of temperature, wind, speed, humidity, atmospheric pressure, etc. In a sailing scenario, weather conditions represent an important element for selecting the sailing route, but as this type of information can change very quickly, the information owned by boats currently sailing in such a region is often more reliable than the one provided by official weather stations. Once the user has issued a weather request, if the target region is not dense enough or too far away to support multi-hop ad hoc network provisioning, the query is sent to the remote infrastructure. The

Fig. 6. WeatherWatcher screenshots

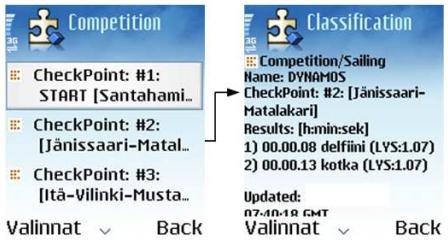

Fig. 7. RegattaClassifier screenshots

infrastructure checks if any WeatherWatcher of users currently sailing in that region has recently provided weather information and returns this information to the requester. Fig. 6 shows the screen interface for this application.

RegattaClassifier : during a regatta competition, this service constantly provides an updated classification of the current winner of the regatta. Virtual checkpoints can be arranged along the route that the boats will take during the competition. Each time a boat reaches a checkpoint, the RegattaClassifier running on the phone's participant (see Fig. 7) communicates to the infrastructure location and speed of the boat (collected using GPS sensors). The infrastructure processes this information and provides each participant with an updated classification and additional statistics of the competition.

7 Related Work

As discussed in Section 2, most research projects investigating context support on mobile devices use sensor-based or infrastructure-based approaches. Approaches exploiting the communication support offered by ad hoc networks have rarely been employed for collecting dynamically changing context data. Our middleware differentiates from these approaches by making use of multiple provisioning mechanisms and by integrating a distributed approach deployed in ad hoc networks.

Our distributed context provisioning mechanism resembles work done to access data stored in sensor networks (e.g., Cougar [17], TinyDB [18]). 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 and data produced by nodes are known at the deployment time, while in MANETs properties and context data differ over time as nodes of different types move across the physical space. Declarative queries are also one of the preferred ways of accessing sensor data [19], [20]. We specialized our query language to offer support for expressing both type and quality of requested context items, and to support long running queries.

Few research projects have focused on implementing practical context support on mobile phones. The ContextPhone [21] is an open-source prototyping platform built on the Series 60 phone platform. It can be used to sense, process, store, and transfer context data. The blackboard-based framework of Korpipää [22] implements a ContextManager which provides a publish-subscribe mechanism and a

database for context data on mobile devices. However, in both works the context sensing relies on information locally available on the device or through BT sensors, whereas in Contory, we provide flexible access to various types of internal and external sensors. According to the classification of Section 2, these works implement an internal sensor-based provisioning mechanism.

8 Conclusions

This paper presented Contory, a middleware specifically deployed to enable easy development of context-aware applications on mobile phones. Our approach provides high flexibility in supporting context provisioning by integrating several context strategies, namely internal sensors-based, infrastructure-based, and distributed provisioning in ad hoc networks. Additionally, Contory offers a unified SQL-like interface for specifying context queries. Using Contory allows context-aware applications to collect context information from different sources without the need to uniquely and continuously rely on their own sensors or on the presence of an external context infrastructure. We demonstrated the feasibility of our approach by deploying Contory in an experimental testbed of smart phones and quantifying its cost in terms of energy consumption. We also used Contory to implement a prototype application for a sailing scenario. Future directions in the development of Contory will focus on providing more efficient and reliable context provisioning in mobile ad hoc networks.

Acknowledgments

This work was partly supported by the DYNAMOS project. The author thanks Cristian Borcea for helpful comments on earlier drafts of this paper, Michael Przybilski for helping in setting up the experimental testbed, and Nishkam Ravi for providing Portable Smart Messages. She also would like to thank the anonymous reviewers who helped improve the paper.

References

1. Schmidt, A., Laerhoven, K.V.: How to Build Smart Appliances? IEEE Personal Communications, Special Issue on Pervasive Computing **8** (2001) 66–71
2. Dey, A.K., Salber, D., Abowd, G.: A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. Human-Computer Interaction **16** (2001) 97–166
3. Hong, I., Landay, J.A.: An Infrastrucutre Approach to Context-aware Computing. Human-Computer Interaction **16** (2001) 287–303
4. Riva, O., di Flora, C.: Contory: A Smart Phone Middleware Supporting Multiple Context Provisioning Strategies. 2nd International Workshop on Services and Infrastructure for the Ubiquitous and Mobile Internet (SIUMI'06) (2006)
5. Schmidt, A., Adoo, K.A., Takaluoma, A., Tuomela, U., Laerhoven, K.V., de Velde, W.V.: Advanced Interaction in Context. In: Proceedings of the First Symposium on Handheld and Ubiquitous Computing (HUC'99), Karlsruhe, Germany (1999) 89–101

6. Yau, S., Karim, F.: A context-sensitive middleware for dynamic integration of mobile devices with network infrastructures. Journal Parallel Distributed Computing **64** (February 2004) 301–317
7. Hong, J., Landay, J.: An Architecture for Privacy-Sensitive Ubiquitous Computing. In: Proceedings of The Second International Conference on Mobile Systems, Applications, and Services (Mobisys'04), Boston, MA (2004) 177–189
8. Bardram, J.E.: 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'05). (2005)
9. Hohl, F., Mehrmann, L., Hamdan, A.: A Context System for a Mobile Service Platform. In: Proceedings of the International Conference on Architecture of Computing Systems(ARCS'02), London, UK, Springer-Verlag (2002) 21–33
10. Riva, O., Toivonen, S.: A Model of Hybrid Service Provisioning Implemented on Smart Phones. In: The 3rd IEEE International Conference on Pervasive Services (ICPS'06), IEEE Computer Society (2006) 47–56
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
12. Crespo, A., Buyukkokten, O., Garcia-Molina, H.: Query Merging: Improving Query Subscription Processing in a Multicast Environment. IEEE Trans. Knowl. Data Eng. **15** (2003) 174–191
13. Borcea, C., Iyer, D., Kang, P., Saxena, A., Iftode, L.: Cooperative Computing for Distributed Embedded Systems. In: Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS 2002), Vienna, Austria (2002) 227–236
14. Ravi, N., Borcea, C., Kang, P., Iftode, L.: Portable Smart Messages for Ubiquitous Java-Enabled Devices. In: The 1st Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous '04). (2004) 412–421
15. Tarkoma, S., Kangasharju, J., Lindholm, T., Raatikainen, K.: Fuego: Experiences with Mobile Data Communication and Synchronization. In: 17th Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC). (2006)
16. Paradiso, J.A., Starner, T.: Energy Scavenging for Mobile and Wireless Electronics. IEEE Pervasive Computing **4** (2005) 18–27
17. Yao, Y., Gehrke, J.: Query Processing in Sensor Networks. In: Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR 2003), Asilomar, CA (2003) 233–244
18. Madden, S., Franklin, M.J., Hellerstein, J.M., Hong, W.: 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), San Diego, California, ACM Press (2003) 491–502
19. Bonnet, P., Gehrke, J., Seshadri, P.: Towards sensor database systems. In: Proceedings of the Second International Conference on Mobile Data Management (MDM '01), London, UK (2001) 3–14
20. Chen, A., Muntz, R.R., Yuen, S., Locher, I., Park, S.I., Srivastava, M.B.: A Support Infrastructure for the Smart Kindergarten. IEEE Pervasive Computing **1** (2002) 49–57
21. Raento, M., Oulasvirta, A., Petit, R., Toivonen, H.: ContextPhone: a prototyping platform for context-aware mobile applications. IEEE Pervasive Computing **4** (2005)
22. Korpipää, P.: Blackboard-based software framework and tool for mobile device context awareness. PhD Thesis. VTT Publications: 579, VTT Electronics, Espoo (2005) http://www.vtt.fi/inf/pdf/publications/2005/P579.pdf.