Object-Oriented Middleware for Location-Aware Systems

Riku Järvensivu
Institute of Software Systems
Tampere University of Technology
P.O. Box 553, FIN-33101
Tampere, Finland
galaxy@iki.fi

Risto Pitkänen
Institute of Software Systems
Tampere University of Technology
P.O. Box 553, FIN-33101
Tampere, Finland
rike@cs.tut.fi

Tommi Mikkonen
Institute of Software Systems
Tampere University of Technology
P.O. Box 553, FIN-33101
Tampere, Finland
tjm@cs.tut.fi

ABSTRACT

Location-based systems are often aiming at the separation of the concept of location to a particular system, or, alternatively, treat location as yet another issue that should be fit into a larger prescribed application framework. This may lead to increasingly complex system architectures, when application logic uses location only as a facility. As a solution that gathers the best sides of both approaches, we propose an approach where object-oriented middleware is used for handling location-based issues in applications that run in mobile devices. In this paper, we will introduce distribution of software and hardware components, as well as sketch the internal architecture of the middleware in terms of the most relevant interfaces. Towards the end of the paper, a sample application is also introduced.

Keywords

Location-based systems, location-based services, object-oriented middleware, software architecture

1. INTRODUCTION

Modern wireless technologies combined with positioning techniques enable a vast amount of location-aware applications. Examples of such applications are navigation aids, location-aware information services such as local weather, tourist information and mobile advertising, location-sensitive games, and location-based dating services. Enabling technologies include programmable smart phones and PDAs, mobile telecommunication networks and related technologies (e.g. GSM, GPRS and UMTS), short-range radio links (e.g. WLAN and Bluetooth), and positioning systems (e.g. GPS and cellular positioning).

Location-based applications typically consist of both terminal and server components. Depending on positioning technology, ready-to-use location data may be produced by terminals (as in GPS) or network components (as in computing positions in a network component based on the cellular structure of a mobile network). Furthermore, applications may comprise various server and terminal components which collaborate to execute their functions.

The location-based domain is a heterogeneous one, with many different terminal devices, operating systems, server platforms and positioning technologies. As developers will want to produce applications that run on various combinations of platforms to enable widely adaptable mass-market applications, APIs (e.g. [10, 7]) and products have already begun emerging to respond to the need for a layer of abstraction above specific positioning platforms. Similarly, general terminal and server software portability is aimed at for example by using the Java programming language that is compiled into portable bytecode to be executed by a virtual machine. There is, however, room for one more layer of abstraction above these rather primitive levels.

Location-based applications are inherently distributed systems. Modern distributed systems are normally built on some kind of middleware, making distribution as transparent to application programmers as possible. Currently, the most common distribution middleware industry standards, such as CORBA, Enterprise JavaBeans and Java/RMI are based on the object-oriented paradigm. Our goal is to design a middleware specifically tailored for location-aware mobile applications, catering for commonly arising requirements in both terminal and server domains. We have specified and implemented an experimental prototype of such a middleware, called OLOS (Object-oriented LOcation System).

OLOS builds on Java and RMI, and it integrates the concept of location seamlessly with an object-oriented view of the world. Location becomes an attribute of a class of objects. Furthermore, OLOS provides useful facilities for applications that run in a highly asymmetric environment comprising small terminals equipped with low-bandwidth connectivity, and high-capacity servers with high-bandwidth networking capabilities, which is indicated as one of the fundamental challenges in mobile computing in [11]. OLOS integrates with different positioning platforms and APIs by providing an interface for writing positioning system adapters.

OLOS distinguishes between two kinds of location-aware objects. Objects of the first kind are bound to tracked mobile entities such as user terminals. The location of an object of this kind changes as the tracked physical entity moves, i.e., is connected to a physical object. A location-aware service profile of a terminal user is an example of such an object.

Location-aware objects of the second kind may be freely placed and moved in the geographical space, and the location of such an object is purely a virtual notion. A virtual sign placed near to a place of interest is an example of such an object. In addition, OLOS provides a mechanism for clients both in mobile terminal and server domains to create, discover and use location-aware objects. Regardless of where an object is created, its actual code is executed on a server registered for hosting objects of the class in question.

The structure of the rest of this paper is as follows. Section 2 introduces OLOS architecture by describing its distribution model and key interfaces. A sample application is developed in Section 3. Finally, Section 4 concludes the paper with a summary and discussion on results and future plans.

2. OLOS ARCHITECTURE

2.1 OLOS Distribution Model

In the OLOS distribution model, processing requirements of the client-side, i.e., mobile terminals, are cut to bare minimum. In essence, only a thin home interface implementation runs in a portable device for encapsulating remote method calls to remote services. The typical factory approach is used for remote object creation.

Figure 1 represents a typical deployment scenario with mobile terminals and several servers. In a typical configuration OLOS-object factories and server side implementations are run in an application server, which the terminals can communicate with over Java RMI interface. In addition to these, several other components are needed. A key component in OLOS architecture is *OLOS Registry*, which associates objects to location data and acts as an interface for searching objects and factories. OLOS Registry is used transparently to the application programmer by base classes and automatically generated home and factory classes.

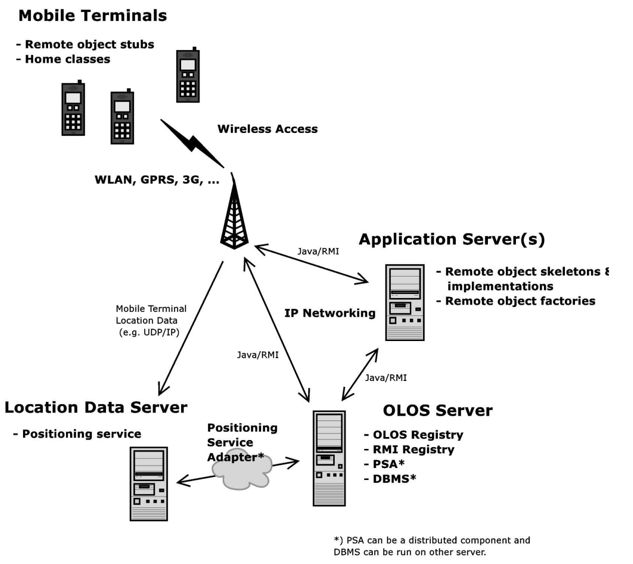


Figure 1: Typical OLOS deployment environment

OLOS Registry has to be connected to a third party positioning service. A positioning service can be any suitable system that can be used to obtain mobile terminal location data, e.g. a WLAN positioning software, a GPS interface, or a 3G mobile phone positioning service. Communication with the positioning service is encapsulated in a component called Positioning Service Adapter (PSA). Positioning service typically is a distributed system itself, and depending on the type of interface it provides, at least parts of the related PSA are often run in the same machine as OLOS Registry.

The current version of OLOS Registry relies heavily on an object-relational database. A simple HSQLDB Java Database Engine [9] has been used to implement the first test version of OLOS Registry. Apart from curiosity, it was chosen to this experiment due to its ability to provide in-memory tables for fast query processing. The DBMS is used via JDBC and can be run in any feasible server including the same physical machine as OLOS Registry (as done in Figure 1).

OLOS Registry must be run in a server accessible for the rest of the components. Java/RMI offers a simple naming service called *RMI Registry*. This naming service has a major limitation: Bound remote object server side implementations have to be run on the same physical machine as the naming service daemon itself. This limitation was implemented mainly because of security reasons in the early JDKs, and it still exists [8, 6]. Nevertheless, RMI Registry is used in this early version of OLOS (a more advanced naming service such as JNDI could be adopted later). In the OLOS deployment architecture, naming service's purpose is to make OLOS Registry easily accessible to the clients. Client-side programs and object factories are supplied with a deployment descriptor which is used to access the registry transparently to application programmers. Deployment descriptors in OLOS are human-readable text files processed and generated with Java Properties-objects.

2.2 Key Interfaces

As already mentioned, OLOS objects are location-aware remote objects of two exclusive subtypes: *physical* and *virtual*. Physical OLOS objects encapsulate a policy whereby the location of an object coincides with the location of the mobile device that created the object. Virtual OLOS object instances represent entities whose locations can be programmatically set, i.e. have no counterparts in the physical world.

From application programmer's point of view, OLOS is a relatively thin extension of Java/RMI to implement object location awareness. OLOS objects are distributed Java/RMI objects with added functionality to gain location awareness. In addition, a home class is generated for each OLOS object class to provide basic object life cycle services and location-based queries.

To create a new OLOS-based application, application programmer derives needed location-aware classes from provided abstract base classes. This is done by first deriving a remote interface from either `PhysicalOLOSObject` or `VirtualOLOSObject` depending on required object type. The created interface declaration is compiled using *OLOS Interface Compiler*, a convenience tool for code generation, which generates home and factory classes and a factory deployment descriptor draft. After a successful interface compilation, a remote implementation of the interface has to be provided deriving `PhysicalOLOSRemoteObject` or `VirtualOLOSRemoteObject` abstract base classes.

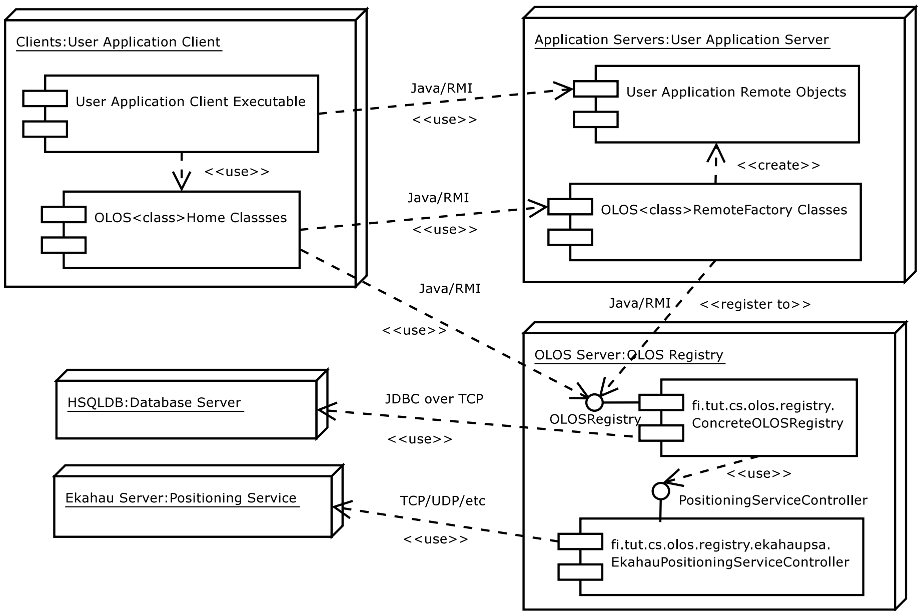

Clients:User Application Client
User Application Client Executable
<<use>>
OLOS<class>Home Classses
Java/RMI
<<use>>
Java/RMI
<<use>>
Java/RMI
<<use>>
Application Servers:User Application Server
User Application Remote Objects
<<create>>
OLOS<class>RemoteFactory Classes
Java/RMI
<<register to>>
OLOS Server:OLOS Registry
OLOSRegistry
fi.tut.cs.olos.registry.
ConcreteOLOSRegistry
<<use>>
PositioningServiceController
fi.tut.cs.olos.registry.ekahaupsa.
EkahauPositioningServiceController
HSQLDB:Database Server
JDBC over TCP
<<use>>
Ekahau Server:Positioning Service
TCP/UDP/etc
<<use>>

Figure 2: OLOS Middleware deployment example

UML deployment diagram in Figure 2 represents a possible deployment scenario of the application. Generated home and factory class names are derived from the interface as defined in the deployment diagram (`<class>`-tag corresponds to original class name).

Getting hands on an OLOS object is done via a particular home class by requesting creation of a new object or by searching for a suitable object. Searching always applies to recent, non-outdated *perceptions* of the location-aware objects unless other time interval is specified. A perception simply consists of coordinates and a time stamp. Possible search criteria include coordinates with search radius as well as fixed location names. Search methods simply return a remote iterator, which can be queried for the next n result objects.

One of the key features of OLOS is positioning service independence. Positioning Service Adapters play a key role in achieving this. In Figure 2, *Ekahau*-related components refer to a third party WLAN positioning system called Ekahau Positioning Engine [5] and a related PSA developed by our development team to test OLOS in a real life environment (described in more detail in section 3.2). OLOS also includes a stand-alone test PSA which can generate random number based circular track location data.

An application programmer can implement a suitable PSA for a deployed positioning service by providing a class which implements the `LocationServiceController` interface (see grey colored <*implementation*> elements in Figure 3). OLOS Registry instantiates the implementation class using current default class loader. OLOS Middleware provides an implementation of `OLOSLocationListener` interface and registers it (possible more than one instance) to the PSA. All perceptions of registered tracked devices are reported to the middleware through this interface.

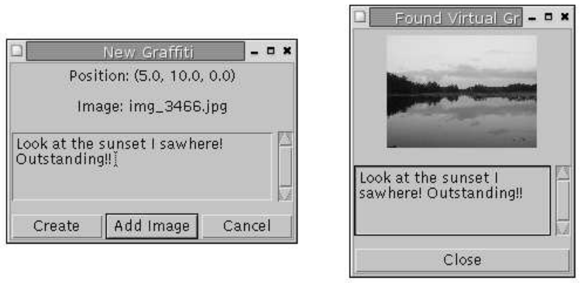
A tracked device is registered to a PSA using positioning service supported key-value pair unambiguously identifying the device to the positioning service. In addition to implementing the `PositioningServiceController` interface (and possibly other needed implementation detail help classes), a PSA might therefore need a client-side component which provides portable device identification data in key-value pairs. This component must provide an implementation of the `TrackedDeviceIdentificationProvider` interface (see Figure 3). A default implementation of the interface returning mobile device's IP address is provided. This is suitable for example for most WLAN positioning services.

3. EXAMPLE APPLICATION: VIRTUAL GRAFFITI

3.1 Functional Specification

To demonstrate and test OLOS, we have defined and implemented a simple example application called Virtual Graffiti. This section describes this application and our test environment for the middleware.

Virtual graffiti are location-bound multimedia messages users can create and leave at their current location. When some other user comes near to the location of a graffiti, she will be shown the multimedia message contained in the graffiti. User interface pictures in Figure 4 demonstrate these application functions.


New Graffiti
Position: (5.0, 10.0, 0.0)
Image: img_3466.jpg
Look at the sunset I sawhere! Outstanding!!

Found Virtual Gr
Look at the sunset I sawhere! Outstanding!!

Figure 4: Dialogs for new and found graffiti

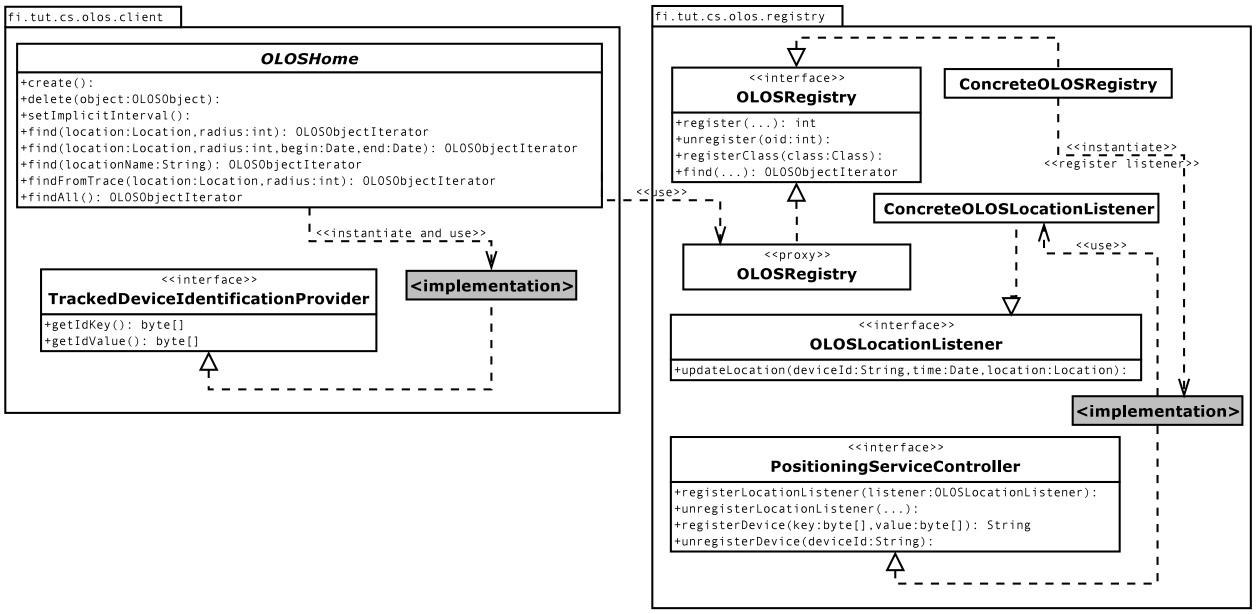

fi.tut.cs.olos.client
OLOSHome
+create():
+delete(object:OLOSObject):
+setImplicitInterval():
+find(location:Location,radius:int): OLOSObjectIterator
+find(location:Location,radius:int,begin:Date,end:Date): OLOSObjectIterator
+find(locationName:String): OLOSObjectIterator
+findFromTrace(location:Location,radius:int): OLOSObjectIterator
+findAll(): OLOSObjectIterator
<<instantiate and use>>
<<interface>>
TrackedDeviceIdentificationProvider
+getIdKey(): byte[]
+getIdValue(): byte[]
<implementation>
<<use>>
fi.tut.cs.olos.registry
<<interface>>
OLOSRegistry
+register(...): int
+unregister(oid:int):
+registerClass(class:Class):
+find(...): OLOSObjectIterator
ConcreteOLOSRegistry
<<instantiate>>
<<register listener>>
ConcreteOLOSLocationListener
<<proxy>>
OLOSRegistry
<<use>>
<<interface>>
OLOSLocationListener
+updateLocation(deviceId:String,time:Date,location:Location):
<implementation>
<<interface>>
PositioningServiceController
+registerLocationListener(listener:OLOSLocationListener):
+unregisterLocationListener(...):
+registerDevice(key:byte[],value:byte[]): String
+unregisterDevice(deviceId:String):

Figure 3: Positioning Service Adapter related classes

Simplified requirements for Virtual Graffiti application are the following:

1. The application must provide a graphical user interface to create and view location bound graffiti.
2. A multimedia message contained in the graffiti consists of text and an optional image. User must be able to select the image file with a standard file select dialog.
3. The application must run in provided test PDAs (detailed specification given in section 3.2). Practically, this means the GUI implementation must be Java AWT-based.

3.2 Test Environment

A WLAN testing network located in the center of city of Tampere is used as a real world test environment for OLOS. Ekahau Positioning Engine [5] has been installed on this test network to get WLAN client terminal location data. The positioning engine is being run in its own dedicated server. It also provides an SDK which offers an event-based local interface to the location data for remote applications. A new event is generated about every 2 seconds containing the latest calculated location estimate of the client terminal [4].

We use Compaq IPAQ 3500-series PDAs with an external PCMCIA WLAN adapter installed in an additional PCMCIA-card cartridge as client devices for testing OLOS in this network. Newer PDAs with an internal WLAN-interface exist in the market, but with the external PCMCIA cartridge we are able to use Ekahau Positioning Engine supported WLAN adapter model. Server side components are run in Sun Microsystems Unix workstations and servers in our software laboratory. Positioning service server is Microsoft Windows based but there is no need to access it any other way but with the SDK provided.

3.3 Implementation

UML class diagram in Figure 5 is a simplified representation of OLOS and Virtual Graffiti application class structure. It includes important classes to understand the functionality of the middleware. Several relations, operations and practically all implementation detail classes are left away from the diagram except for remote object proxies. Grey colored classes are built-in Java classes, while components which application developer has to provide are marked with *user provided* comments and automatically generated classes with *generated* comments. Rest of the classes, i.e. classes without any special markings, belong to the OLOS Middleware.

It is obvious the graffiti itself are objects bound to a particular map location, i.e. to certain coordinates. This indicates functionality provided by OLOS objects of virtual type. We name the class (actually an interface) as VirtualGraffiti and derive it from `VirtualOLOSObject`. In our implementation, the interface needs only two "business logic" methods, i.e. methods which allow us to set and get the multimedia message contained.

We also need a way to assign the graffiti we create to the right position. In other words, we need to get the mobile terminal coordinates from somewhere. Object-oriently thinking, we need an object to represent the user terminal and to implicitly follow its movement and provide an interface to query the position. This indicates it should be physical-type OLOS object and we can derive it from `PhysicalOLOSObject` as shown in Figure 5. This interface has been named as `VGUser`.

The next step is to use the OLOS Interface Compiler. After compiling the interfaces we have several new implementation files: `OLOSVGUserHome` (a class), `OLOSVGUserFactory` (a remote interface), `OLOSVGUserRemoteFactory` (a remote factory implementation), `OLOSVirtualGraffitiHome` (a class), `OLOSVirtualGraffitiFactory` (a remote interface), `OLOSVirtualGraffitiRemoteFactory` (a remote factory implementation) and `.properties`-files for both factories (factory deployment descriptor drafts). These generated components are shown with blue color in the class diagram (Figure 5). We also need to implement the interfaces we

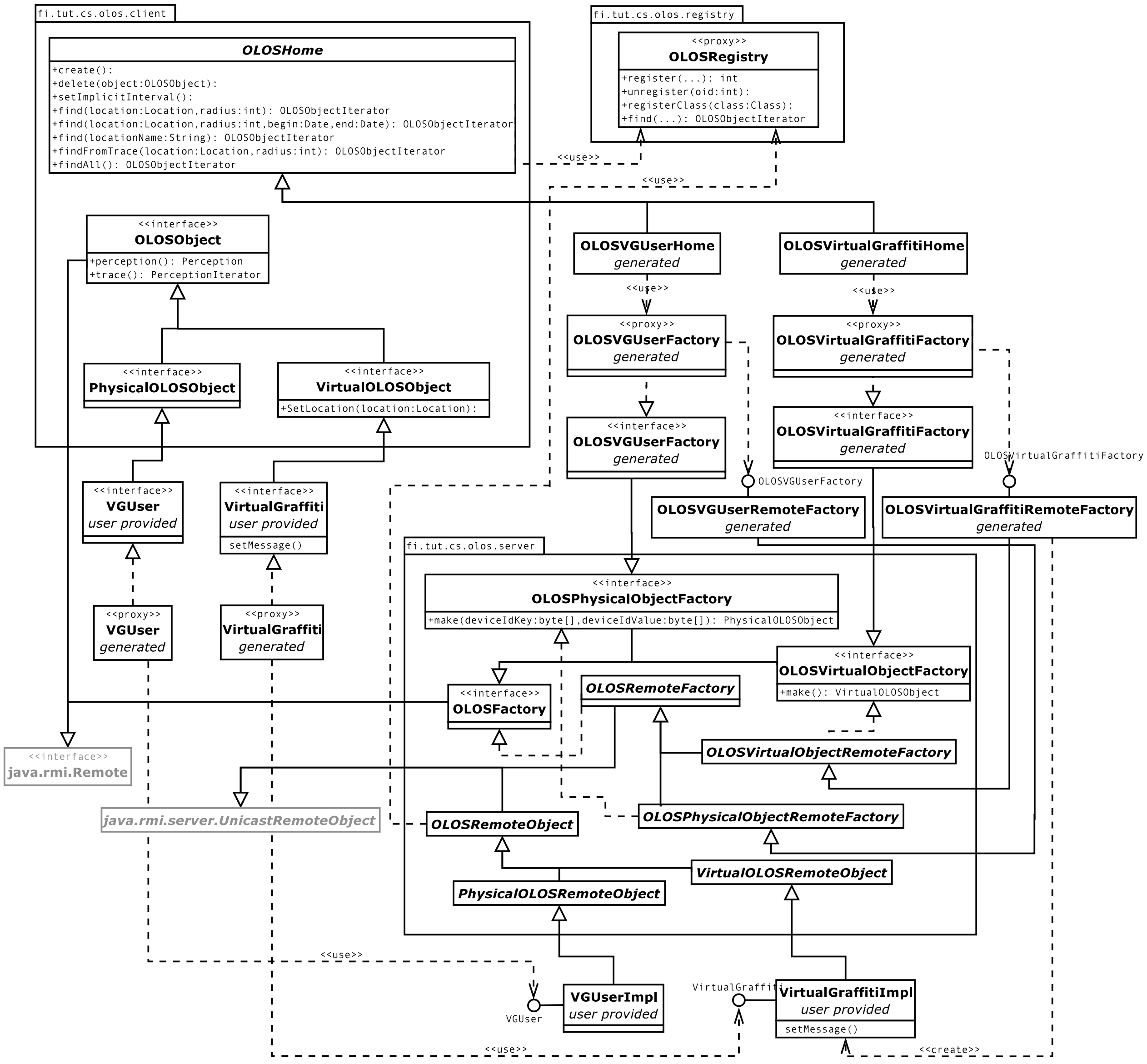

fi.tut.cs.olos.client
OLOSHome
+create():
+delete(object:OLOSObject):
+setImplicitInterval():
+find(location:Location,radius:int): OLOSObjectIterator
+find(location:Location,radius:int,begin:Date,end:Date): OLOSObjectIterator
+find(locationName:String): OLOSObjectIterator
+findFromTrace(location:Location,radius:int): OLOSObjectIterator
+findAll(): OLOSObjectIterator
<<interface>> OLOSObject
+perception(): Perception
+trace(): PerceptionIterator
<<interface>> PhysicalOLOSObject
<<interface>> VirtualOLOSObject
+SetLocation(location:Location):
fi.tut.cs.olos.registry
<<proxy>> OLOSRegistry
+register(...): int
+unregister(oid:int):
+registerClass(class:Class):
+find(...): OLOSObjectIterator
<<use>>
<<use>>
OLOSVGUserHome generated
OLOSVirtualGraffitiHome generated
<<use>>
<<use>>
<<proxy>> OLOSVGUserFactory generated
<<proxy>> OLOSVirtualGraffitiFactory generated
<<interface>> OLOSVGUserFactory generated
<<interface>> OLOSVirtualGraffitiFactory generated
OLOSVirtualGraffitiFactory
OLOSVGUserFactory
OLOSVGUserRemoteFactory generated
OLOSVirtualGraffitiRemoteFactory generated
<<interface>> VGUser user provided
<<interface>> VirtualGraffiti user provided
setMessage()
<<proxy>> VGUser generated
<<proxy>> VirtualGraffiti generated
fi.tut.cs.olos.server
<<interface>> OLOSPhysicalObjectFactory
+make(deviceIdKey:byte[],deviceIdValue:byte[]): PhysicalOLOSObject
<<interface>> OLOSVirtualObjectFactory
+make(): VirtualOLOSObject
<<interface>> OLOSFactory
OLOSRemoteFactory
OLOSVirtualObjectRemoteFactory
<<interface>> java.rmi.Remote
java.rmi.server.UnicastRemoteObject
OLOSRemoteObject
OLOSPhysicalObjectRemoteFactory
VirtualOLOSRemoteObject
PhysicalOLOSRemoteObject
<<use>>
VGUser
VGUserImpl user provided
VirtualGraffiti
VirtualGraffitiImpl user provided
setMessage()
<<use>>
<<create>>

Figure 5: Simplified OLOS class diagram with an example application

created. Implementation classes must be named as `VGUserImpl` and `VirtualGraffitiImpl`. VGUser is trivial to implement and VirtualGraffiti needs implementation of the two business methods specified. In this application, most of the implementation work is needed in the client user application to provide an easy-to-use user interface and to use the home interface to create and find graffiti. Figure 6 contains a UML sequence diagram, which describes the creation of a new VirtualGraffiti instance. Different servers, RMI proxies and remote method invocations between them (calls marked with «rmi») are also marked in the diagram.

When designing OLOS-based applications such as this example, we quickly run into very unsophisticated interface functionality. Neither OLOS object or home class interfaces offer any means for event-based application development. Therefore, the Virtual Graffiti application must poll for nearby graffiti. This major limitation in OLOS is a deliberate simplification in the first test version of the middleware to keep it simple enough and to test the concept in reasonable time taking available resources into account. Event-based interfaces are under elaboration for later versions and are considered as a must-have feature for adaptation of the middleware in real-life application development.

A well-designed event-based architecture would allow Virtual Graffiti application to register an event listener that would be notified when graffiti near the user's location are found. Currently, we have to use a polling scheme described in the UML sequence diagram in Figure 7. Every time we poll for nearby graffiti, we have to first ask VGUser for latest Perception (marked with letter 'p' in the diagram). After receiving the Perception we check if it is recent and get Location it represents (Location object is marked with letter 'l' in the method parameters). When the user terminal location is known, we can commit the actual find-operation with location information and fixed search radius (marked with 'r') as parameters. If no graffiti are found, the search immedi-

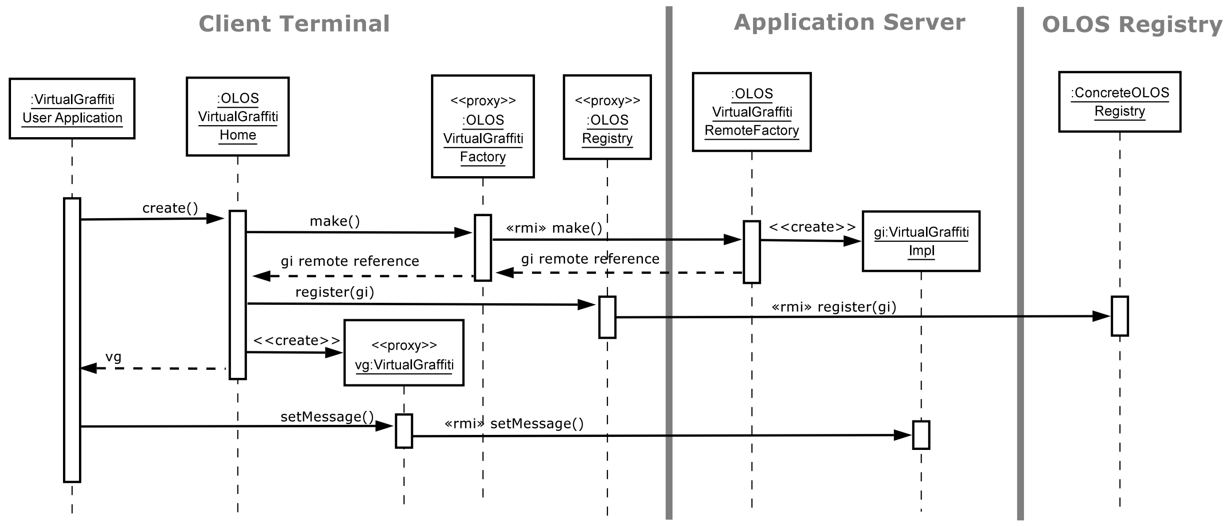


Figure 6: Creating a new VirtualGraffiti-instance

ately returns a null for maximum performance. If results are found, a remote iterator is created to provide a safe interface to the results. Finally, the user application can query the iterator for a `java.util.List` instance of preferred amount of actual results (remote object references).

After implementing the user application (providing a GUI) the application can be deployed. Assuming OLOS Registry, positioning service, DBMS etc. are properly configured and running, deployment simply consists of starting needed object factories. In this early version of the middleware, object factories are started by running `FactoryStarter` class with a factory deployment descriptor as a command line argument. This class instantiates a given factory class and registers it to OLOS Registry. In addition, a user application deployment descriptor needs to be created supplying RMI Registry address and port as well as a `PositioningServiceIdentificationProvider` implementation class name to the generated home classes. After deployment, the application is ready to be run.

4. DISCUSSION

Location awareness is a common feature of many novel applications. However, the treatment of location has typically been an application specific item that is bound to directly benefiting from the concept of location as such (e.g. [3]), or, alternatively, a minor detail of a wider framework where location awareness only forms a small part (e.g. [1]). As an attempt to gain the best of both worlds, we introduced OLOS, a location-based middleware that follows the practices of Java RMI to provide the concept of location to mobile devices.

Currently, we are in the middle of testing the OLOS system alpha release, with the majority of software and installations already in place and running. Moreover, the architecture and functionality has been validated with a test positioning service adapter. In these tests, location changes have been simulated as we had no access to the real location database. While writing this, latest version of Ekahau Positioning Engine has just been successfully deployed and calibrated in the WLAN testing network located in city of Tampere, Finland. We are about to start real world tests of OLOS using the Ekahau positioning engine as a back-end, accompanied by an appropriate positioning service adapter.

The future plans for OLOS are many. To begin with, we wish to incorporate a better strategy for tracking the movement of objects within the middleware instead of the current polling mechanism, as already mentioned. This would liberate the applications from requesting information on the locations of objects that they are interested in, and would let the middleware to send an event upon e.g. a change in the location of an object. In addition, we wish to extend the number of applications that have been implemented on top of the framework. Moreover, there are many aspects related to the installation of software in servers as well as to the deployment of required programs in mobile devices that should be clarified for full-scale installation of the middleware.

Instead of using an object-relational database in the OLOS Registry, we have been planning on using more effective spatial data structures for in-memory caching and persistently storing the location data. Performance of OLOS Registry is a crucial factor considering responsiveness of OLOS. Other optimizations in OLOS architecture, which could lead to significant reduction in the processing work required, include implementation of some sort of *Positioning Quality of Service* (as introduced in [2]) and policies in which the calculation of the latest location of an object could be requested through the OLOS Registry only when needed. OLOS architecture is also able to support multiple object factories for single class of location-aware objects, but currently only one OLOS Registry. A natural way to improve scalability of the system would be to distribute the OLOS Registry, or better yet, support unlimited number of registries without a single entry point.

5. REFERENCES

[1] P. Bellavista, A. Corradi, and C. Stefanelli. Mobile agent middleware for mobile computing. *IEEE Computer*, pages 73–81, March 2001.

[2] P. Biswas, S. Han, and J. Wu. Location Caching in the Mobile Middleware Platform. In *IEEE Third International Conference on Mobile Data Management*, pages 172–173, January 2002.

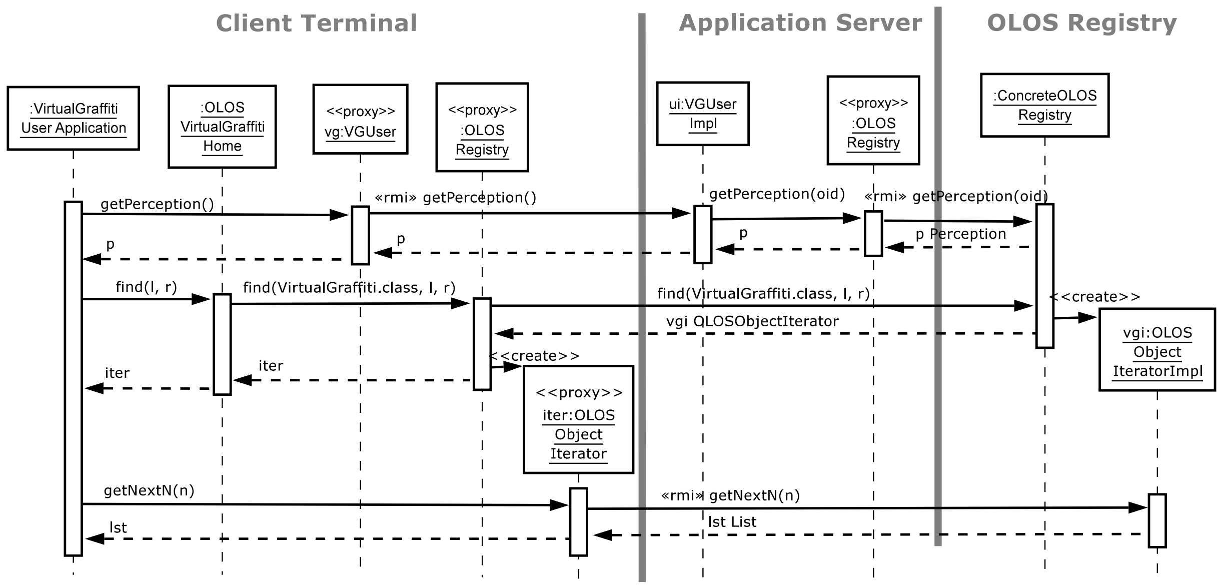


Figure 7: Searching for nearby VirtualGraffiti instances

[3] X. Chen, Y. Chen, and F. Rao. An efficient spatial publish/subscribe system for intelligent location-based services. In *2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, June 2003. `www.eecg.toronto.edu/debs03/papers/chen_etal_debs03.pdf`.

[4] Ekahau, Inc. *Ekahau Positioning Engine 2.0: Developer Guide*, 2002.

[5] Ekahau, Inc. Ekahau Positioning Engine Product Homepage. At `http://www.ekahau.com/products/positioningengine/`, 2003.

[6] W. Emmerich. *Engineering Distributed Objects*. John Wiley & Sons, Ltd., 2000. Pages 109-119 and 215-219.

[7] FiCom Location API Working Group. FiCom Location API 2.0.0 Interface Specification. Edited by: Sampi Lempinen. `http://www.ficom.fi/fi/t_api.html`.

[8] W. Grosso. *Java RMI.* Designing & Building Distributed Applications. O'Reilly & Associates, Inc., first edition, 2002.

[9] T. hsqldb Development Group. HSQL Database Engine Home Page. At `http://hsqldb.sourceforge.net/`, 2003.

[10] Java Community Expert Group. JSR179: Location API for J2ME, 2003. Specification lead: Kimmo Löytänä, Nokia Corporation. `http://www.jcp.org/en/jsr/detail?id=179`.

[11] M. Satyanarayanan. Fundamental challenges in mobile computing. *Technical Report, Carnegie Mellon University, Pittsburgh, PA*, 1996.