Development Framework for Mobile Social Applications

Alexandre de Spindler, Michael Grossniklaus, and Moira C. Norrie

Institute for Information Systems, ETH Zurich
CH-8092 Zurich, Switzerland
{despindler,grossniklaus,norrie}@inf.ethz.ch

Abstract. Developments in mobile phone technologies have opened the way for a new generation of mobile social applications that allow users to interact and share information. However, current programming platforms for mobile phones provide limited support for information management and sharing, requiring developers to deal with low-level issues of data persistence, data exchange and vicinity sensing. We present a framework designed to support the requirements of mobile social applications based on a notion of P2P data collections and a flexible event model that controls how and when data is exchanged. We show how the framework can be used by describing the development of a mobile application for collaborative filtering based on opportunistic information sharing.

Keywords: Mobile Social Applications, Development Framework, Adaptive Middleware.

1 Introduction

The increased computational power and storage capacity of mobile phones now makes them capable of hosting a wide range of multimedia services and applications. In addition, the integration of sensing devices such as GPS and connectivity such as Bluetooth and WiFi has made it easier to support location-based services and new forms of information sharing.

As a result of these technical innovations, service providers and application developers are keen to exploit a new potential market for mobile social applications that allow users to interact and share data via their mobile phones. However, programming platforms for mobile phones currently provide little support for flexible forms of information management and sharing. In a rapidly emerging and highly competitive market, this presents companies with a major challenge in terms of the effort required to prototype and validate potential applications.

To address this problem, we have designed an application development framework to support the requirements of mobile social applications. The framework ensures that developers can work at the level of the application domain model, without having to deal with the low-level mechanisms provided in current

P. van Eck, J. Gordijn, and R. Wieringa (Eds.): CAiSE 2009, LNCS 5565, pp. 275–289, 2009.

platforms for dealing with peer-to-peer (P2P) information sharing, data persistence and location sensing. Instead, applications can be designed around a novel concept of P2P collections of persistent data coupled with a flexible event model that can determine how and when data is exchanged and processed.

In this paper, we present the requirements of mobile social applications along with the limitations of existing platforms with respect to these requirements. We then provide the details of our framework and demonstrate its use by describing how we developed an application for collaborative filtering based on P2P information sharing in mobile environments.

Section 2 discusses the limitations of existing platforms for mobile phones with respect to the goal of supporting the development of mobile social applications. In Sect. 3, we then examine the requirements of mobile social applications in detail and describe how our framework supports these requirements. Details of P2P collections and the event model are given in Sect. 4 and Sect. 5, respectively. In Sect. 6, we describe how the collaborative filtering application was implemented using the framework. Concluding remarks are given in Sect. 7.

2 Background

Mobile phones are no longer simply regarded as communication devices, but rather as computing devices capable of, not only hosting a range of applications, but also communicating with each other. This has led to a great deal of interest in mobile social applications which can take advantage of these capabilities to allow users to interact and share information in innovative ways. Applications have been proposed that exploit ad-hoc network connections between phones via Bluetooth or WiFi to support user awareness of social contexts [1,2] or to automatically exchange data between users in shared social contexts [3,4]. In particular, physical copresence has been used as a basis for forming a weakly connected community of users with similar tastes and interests [5,6].

A variety of development toolkits for mobile phones are available. These range from vendor-specific solutions such as iPhone SDK[1], Windows Mobile Edition[2], Symbian[3] and Google Android[4] to the platform independent Java WTK (Wireless Toolkit). These provide integrated emulation environments along with support for the development of user interfaces. They also provide access to typical phone features such as personal information management (PIM) data, the camera, Bluetooth, Internet access and GPS. However, the development of mobile social applications using these toolkits still requires considerable effort since they provide no high-level primitives to support vicinity sensing, location awareness, information sharing and data persistence. As a result, developers have to

[1] http://developer.apple.com/iphone
[2] http://www.microsoft.com/windowsmobile
[3] http://www.symbian.com
[4] http://code.google.com/android

implement components to handle requirements related to these issues for each application and each target platform.

For example, Java WTK uses a simple key-value store for data persistence which means that developers have to define and implement the mapping between Java application objects and key-value pairs for each application. This contrasts with development platforms for PCs such as db4o[5] that support Java object persistence. Support for information sharing is also limited in these platforms and data sharing must be implemented based on sockets able to send and receive binary data. The developer must therefore implement the facilities to serialise and deserialise data, to open and listen to sockets and stream data. Short-range connectivity such as Bluetooth or WiFi can be used to react to peers appearing in the physical vicinity. Using Java WTK, the developer has to implement two listeners, one registered for the discovery of a device and another which is notified about services discovered on a particular device. For each scan of the environment, both listeners must be registered and the developer must also implement the coupling of peer discovery with data sharing.

Frameworks have been developed specifically for P2P connectivity including Mobile Web Services [7] and JXTA [8], but these tend to focus on lower-level forms of data exchange rather than information sharing. For example, JXTA provides the notion of a peer group as a central concept of their metamodel. A group mainly provides facilities for peer discovery, service publishing and message routing. Application development consists of specifying message formats and how they are processed in terms of request and response handling similar to that of service-oriented architectures. This results in a blending of the application logic typically embedded in an object-oriented data model and the collaboration logic specified based on a request-response scheme. Efforts to provide higher level abstractions of P2P networks have either focussed on the allocation and retrieval of identifiers to resources in fixed networks without considering any notion of handling [9] or they offer only a few limited collaboration primitives and lack support for vicinity awareness [10,11].

Within the database research community, a number of P2P database systems, overlay networks and middlewares have been developed including Pastry [12], Piazza [13], PeerDB [14], Hyperion [15], P-Grid [16] and GridVine [17]. However, research efforts have tended to focus on issues of object identity, schema matching and query reformulation, distributed retrieval, indexing and synchronisation as well as transaction management. To date, there has been little effort on supporting developers of mobile applications that utilise P2P connectivity to share information opportunistically with other users in the vicinity.

Based on our own experiences of developing mobile social applications using existing platforms, we realised that there was a need for an application framework that offers functionality for P2P information sharing as high-level primitives. In the next section, we examine the requirements of such a framework in detail before presenting an overview of the framework that we have developed.

[5] http://www.db4o.com

3 Framework

A distinguishing feature of mobile social applications is the notion of collaboration. Each peer follows a set of application-specific rules which determine its behaviour within the collaborative environment. This behaviour includes the local creation, storage and processing of data as well as interacting with other peers by sending, receiving and forwarding data. Such behaviour may be triggered automatically or explicitly by the user. Each peer offers the services of the application to the user independently of the other peers, but the effectiveness of these services depends on the combined effects of local peer behaviour.

To examine the requirements of mobile social applications and illustrate how our framework supports these requirements, we will consider the example of a recommender system. Due to space limitations, a more comprehensive examination cannot be presented here. In previous work [18], we have shown how collaborative filtering (CF) algorithms can be adapted to mobile settings using physical copresence in social contexts as a basis for measuring user similarity. Figure 1 will be used to illustrate how such an application works. Assume users rate items such as music, films or places to go and this data is stored as a collection C of triples (u, i, r) where u is a user, i an item and r a rating. Essentially, we can view the collaborative filtering process as some function f that is applied to C to return the result recommendation R as a list of items. The details of the function f are not relevant to this discussion, but what is important is that each peer has an instance of C and will locally compute $f(C)$ when the recommender service is called. We refer to such application-specific services as the *application logic* of the system, and they may be executed either automatically or upon an explicit request by the user.

An application may have multiple data collections defined by a schema shared by all peers, say $\{C_1, C_2, ...C_n\}$ and a set of participating peers $\{P_1, P_2, ...P_m\}$.

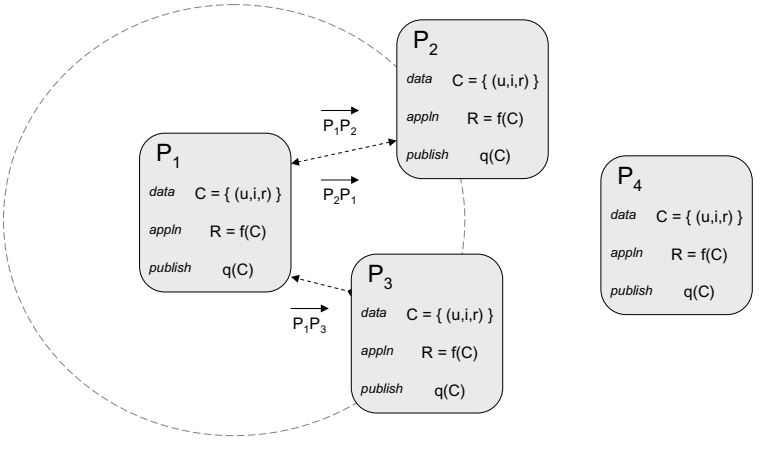

Fig. 1. Collaborative Filtering as a Mobile Social Application

Each peer will have its own instance of each of the application collections and we use $C_i|P_j$ to denote the instance of the collection C_i stored on peer P_j. Note that we prefer to refer to $C_i|P_j$ as an instance of collection C_i rather than as a part of some global collection C_i since the application services running on P_j will operate only on the locally stored collection of the appropriate name, independently of the collections stored on other peers. In the case of the collaborative filtering application illustrated in Fig. 1, there are four peers each of which has a single data collection C containing rating triples of the form (u, i, r) and an application to compute the CF result denoted by $R = f(C)$.

Computing user similarity in centralised CF algorithms can be computationally expensive. In mobile settings, a much simpler approach can be used which takes advantage of the fact that local data comes only from the owner of the device, or from users with similar tastes and interests. The underlying assumption is that users who are close enough to exchange data through ad-hoc connections between mobile devices share social contexts and hence are likely to have similar tastes and interests. Detailed studies related to this assumption have been carried out in a number of projects, see for example [5,6,18], and it is beyond the scope of this paper to discuss this aspect in detail. Our interest here is the fact that mobile social applications often involve some form of opportunistic sharing of information based on ad-hoc connectivity between mobile devices, or possibly mobile and stationary devices. It is therefore important that a development framework for mobile social applications supports a notion of vicinity awareness.

At a given time t, the vicinity of a peer P_i is the set of peers to which P_i is connected, and we denote this by $V_t(P_i) = \{P_1, ..., P_k\}$. In Fig. 1, we use a dashed circle to denote the connectivity range of P_1 at time t and, hence, $V_t(P_1) = \{P_2, P_3\}$. If $P_j \in V_t(P_i)$ then it is possible for peers P_i and P_j to exchange data. The *collaboration logic* of an application will specify *if*, *when* and *what* data is shared. We will discuss the details of how the *if* and *when* can be specified later in the paper when we present the details of our framework. The *what* is specified by associating a query expression q_i with each application collection C_i. We use $P_j \vec{P}_k$ to denote an exchange of data from P_j to P_k. This means that if P_j and P_k both have instances of collections $\{C_1, C_2, ...C_n\}$ then

$$P_j \vec{P}_k \colon \forall C_i \in \{C_1, C_2, ...C_n\}, C_i|P_k := q_i(C_i|P_j) \cup C_i|P_k$$

Figure 1 shows a case where P_1 and P_2 exchange data bilaterally, meaning that each peer sends rating tuples to the other peer and adds the data to its local C collection. The query expression q acts as a filter on the data to be published. In the case of collaborative filtering, only the data pertaining to the actual user of the device, and hence the user currently in the same social context, will be sent to the other peer. In the case of the connection between P_1 and P_3, P_3 sends data to P_1 but P_3 does not receive data from P_1. This is indicated in the figure by the fact that the connection between P_1 and P_3 has an arrow in only one direction. It could be the case that P_3 had previous encounters with P_1 and found their data unreliable and hence placed them on some sort of black list to indicate that they did not want to receive data from them in the case of future

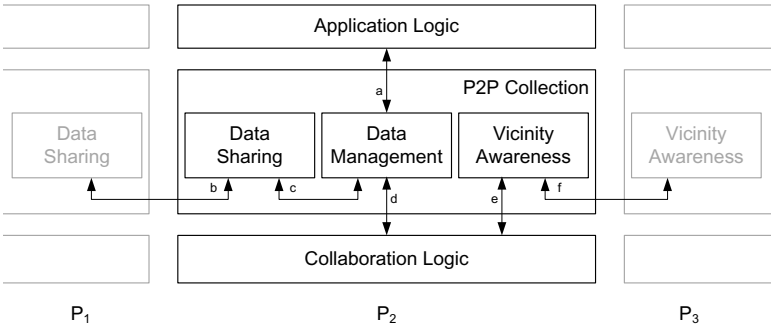

Fig. 2. Framework Overview

encounters. Note that, in practice, it might be that P_1 would publish the data, but P_3 would simply choose not to receive it. Details of this will be given in later sections.

Generally, it should be possible for mobile social applications to have flexibility in determining how and when peers exchange data. For example, there are many ways in which applications might want to control the exchange of data for reasons of privacy. Within our framework, we provide a flexible event processing model that allows applications to determine how and when they share data.

Having looked at the general operation and key requirements of mobile social applications, we see that there are three main functionalities that a framework needs to support. First, it needs to provide basic services for the management and querying of data collections. Second, it should offer developers high-level abstractions to enable data from those collections to be shared via ad-hoc connections to peers. Third, it needs to provide vicinity awareness. At the same time, it is important to separate the concerns of application logic and collaboration logic to ensure maximum flexibility in meeting the requirements of a broad spectrum of users, devices and applications. Figure 2 presents an overview of our framework. The concept of a P2P collection is the central component which encapsulates persistent data storage, data sharing and the ability to sense peers entering and leaving a peer's physical vicinity. As an interface to application logic and user interaction, the framework offers standard data management facilities such as the creation, retrieval, manipulation and deletion of data (a). These facilities are offered in terms of a database management system which includes transaction management support. Furthermore, it offers a second interface allowing collaboration logic to be specified and executed in terms of events and their handling (d,e). By keeping these interfaces independent, we are able to achieve the required separation of concerns. The actual scanning of the physical environment (f) and data sharing (b,c) is encapsulated by the framework.

In the next section, we will present the concept of P2P collections in detail before going on to describe how the collaboration logic can be specified by means of the event processing system.

4 P2P Collections

Programming languages such as Java and C++ have standard libraries that offer various types of collections in terms of interface definitions that declare operations to insert, retrieve and remove data along with concrete implementations that provide the corresponding functionality. Following this paradigm, the central component of our framework—the peer collection—is an alternative collection implementation that provides additional functionality to address the requirements of mobile social applications.

Most programming systems define collections in terms of a collection behaviour and a member type. For example, Java offers collection implementations for sets, lists and maps that, through the use of generics, can be bound to a member type that restricts the possible members of the collection. Our definition of a peer collection follows this approach but extends it to cope with more specific requirements. Generally, a peer collection is characterised by its name n, its member type t and its behaviour b. As we will see, the use of a name to identify the collection is motivated by the requirements of data sharing in a peer-to-peer environment that makes it necessary to identify collections across peers. Our framework introduces additional collection behaviours to support data management. The behaviour $b \in \{\text{set, bag, sequence, ranking}\}$, where {set, bag} are unordered, {sequence, ranking} are ordered and {set, ranking} have no duplicates while {bag, sequence} do.

Similar to common programming environments, methods to add, retrieve and remove data to/from a collection provide basic data management. Peer collections can optionally be marked as persistent with the effect that not only the collection, but also the members are automatically made persistent in a transparent way. In addition, our framework has support for events that get triggered whenever elements are added to, or removed from, peer collections. In Sect. 5, we will discuss how this mechanism can be leveraged to support the decoupling of the collaboration logic.

Our framework also features a low-level query facility that surpasses the data retrieval mechanisms offered by current collection implementations. A query is specified by building a query tree where the inner nodes represent query operations and leaf nodes contain query arguments. Once a query tree has been constructed, its root node is passed to the query evaluator component of the framework which processes the query and returns the result. While a complete presentation of our query facility is outside the scope of this paper, Tab. 1 gives an overview of the most important nodes including those we refer to in this paper. A node may have child nodes and attributes. For example, a selection node has a collection from which members are to be selected as a child and an attribute containing the selection predicates. In order to simplify the task of creating frequently used queries, a query tree builder is provided with the framework. Given the required parameters, it automatically builds the query tree and returns its root node.

Peer collections also address the requirements of data sharing. This additional functionality is provided through a set of methods that can be used to make

Table 1. Example query tree nodes, their children and attributes

Node	#Child Nodes	Attributes
Selection	1	predicates
Intersect	2	–
Union	2	–
Map	1	function
Attribute Access	1	attribute
Collection	–	collection

a collection available for sharing, connect it to other peers and exchange its members. In order for two peers P_1 and P_2 to share data, both peers have to make the collection to be shared available. When the two peers enter in each other's vicinity, available collections can be connected if they have the same name n and member type t. Once two peer collections are connected, all or some of the collection members from each peer are sent to the other peer. A query expression attached to the collection determines which members are sent.

Based on these basic sharing capabilities, our framework also provides a flexible mechanism to control what data is exchanged. This can be done in two ways. A selection query can be bound to a collection to filter data sent to peers. These filter queries are also expressed and evaluated based on the framework's query facilities presented above. In addition, white and black lists can be used to control with which peers data is exchanged. Thus, a collection that has been made available is associated with a positive and negative neighbourhood of peers. The positive neighbourhood contains those peers with which members are shared if they appear in vicinity, while other peers in vicinity will be ignored. If the positive neighbourhood is empty, members will be shared with any peer appearing in vicinity. The negative neighbourhood optionally contains those peers that should not be considered for data sharing even if they appear in vicinity. Similar to the positive neighbourhood, if that collection is empty, no restrictions are assumed to exist. These two neighbourhoods therefore enable a user to define constraints over the social network within which data is shared.

Our framework offers support for vicinity awareness which is used to react upon the appearance or disappearance of peers in the physical vicinity. As well as triggering events to connect collections and share data as described above, an application may react on such events directly. We will present the event mechanism offered by the framework in more detail in the next section.

Based on the Java programming language in conjunction with Java WTK platform, we will now describe how the framework can be implemented. Other programming languages such as C++ or Objective C as well as other platforms such as Symbian, iPhone SDK or Google Android can be supported analogously.

The implementation of the framework consists of two parts. First, there is the application programming interface (API) visible to the developer of a mobile social application together with the implementation of functionality that is common to all platforms. Then, there is the service provider interface (SPI) which needs to be implemented to support the peer collection framework on a

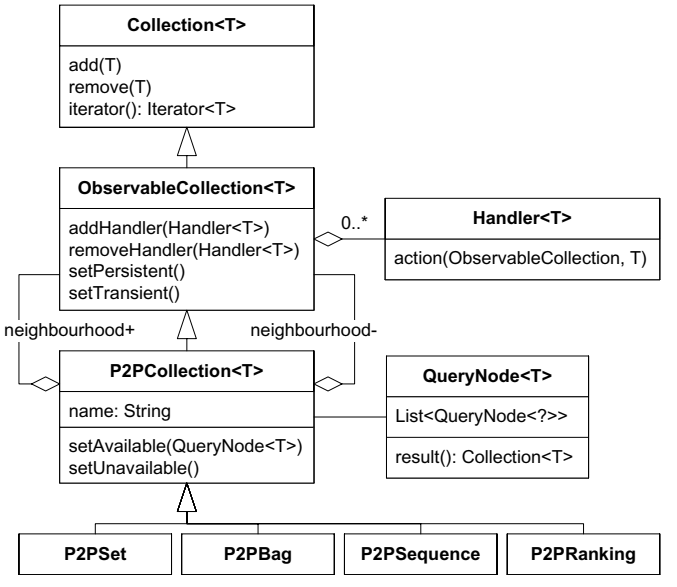

Fig. 3. API of the peer collection framework

given mobile phone and, thus, represents the platform-specific implementation. While other frameworks such as Java WTK address a similar problem, they usually do not cover the entire range of existing devices. In the remainder of this section, we will describe both parts of the framework in turn. Figure 3 gives an overview of the peer collection framework API that is based on the concepts described above. At the top of the figure, a simplified version of the existing Java collection interface is shown, highlighting the methods for adding, retrieving and deleting collection members. Our framework extends the Java collection interface and introduces an interface `ObservableCollection<T>`. Observable collections support the registration of handlers that are invoked whenever an event is triggered through the addition or removal of a collection element. A peer collection is represented by interface `P2PCollection<T>` that defines a collection name as well as methods to make the peer collection available or unavailable. The four different collection behaviours are provided through dedicated implementations of the peer collection interface. Finally, the query facility of the framework is supported by `QueryNode<T>` which serves as the common interface of the various query nodes discussed earlier.

The peer collection framework SPI defines the interfaces for three platform-dependent components and is shown in Fig. 4. One component offers persistent data storage, another the connection technology and a third the scanning of the physical vicinity for other peers. Note that, in contrast to existing platforms, these components have to be implemented once per platform rather than once per application. All persistence mechanisms make use of a single class offering database facilities such as storing, retrieving and deleting objects. This class is

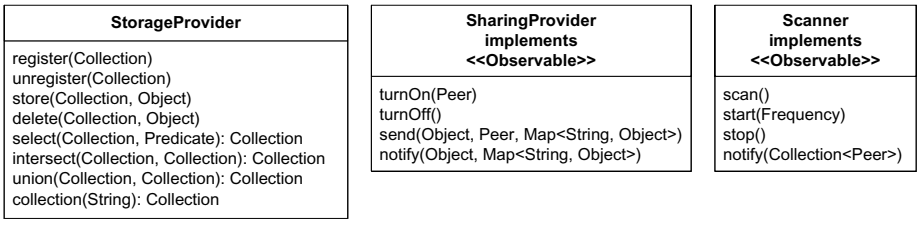

Fig. 4. SPI of the peer collection framework

defined in terms of interface `StorageProvider` and the framework makes use of it based on this interface only which allows it to be adapted to any underlying persistent storage technology such as a record store in the case of Java WTK.

To send and receive data, our framework makes use of a component that is dependent on the connection technology. The implementation of this component is abstracted through interface `SharingProvider`. Developers may turn on and off its availability within the collaborative environment. When turned on, the technology-specific information for reaching the local peer must be provided. In the case of Java sockets, this information includes a host identifier and a port number. Once the peer is turned on, collections may be made available. The peer can be turned off at any time, in which case, no more collections are available and the local peer is no longer available to other peers. This component is defined by a generic interface and can thus be implemented for different connection technologies such as Java sockets, WiFi and Bluetooth.

Finally, connection technologies such as Bluetooth and WiFi are used to scan the physical environment of a peer and discover other peers nearby. As with the other two components, the scanning is implemented by a platform-dependent component which is defined by and used through the interface `Scanner`. This interface declares the means to perform a single scan as well as starting a periodic scan with a frequency that can be specified.

5 Event Processing

Within our framework, it is the appearance of a peer in the vicinity of another that drives the sharing process. Thus, data sharing is linked with an event system composed of events for which handlers can be registered to be notified. While events and handlers can be specified by an application developer, predefined system events exist. Table 2 shows these events along with the arguments to which they are attached and the parameters passed to the registered handlers. To register a handler for an event, the developer needs to implement interface `Handler<T>` shown in Fig. 3 and specify an action method to be executed.

As part of the framework, a system collection `Vicinity` is provided. This collection is maintained by the framework and its members represent those peers that are currently in the physical vicinity. Whenever a new peer is detected, a new object is created and added to the `Vicinity` collection. When a peer moves away,

Table 2. Events, their arguments and parameters passed to the handlers

Event	Argument	Parameters
New Object	–	Object
Object Changed	Object	Object, Attribute
Member Added	Collection	Collection, Member
Member Received	Collection	Collection, Member, Source Peer
Member Removed	Collection	Collection, Member

the respective member is removed from the collection. Note that this collection is not set to be persistent. Since our event model generally supports the triggering of actions when objects are added to collections, vicinity awareness is realised based on addition events associated with the `Vicinity` collection that will be triggered when a new peer enters the physical vicinity. If a collection is made available, a predefined handler for sending collection members is automatically registered with this addition event.

As explained previously, when a collection is made available, the root node of a selection query is passed along. The query is handed over to the handler. The handler action consists of executing the query and sending the result.

6 Collaborative Filtering

To show how our framework is used, we present the implementation of the recommender system introduced in Sect. 3 as a use case. A detailed description of the system has been presented in [18]. A fundamental ability of recommender systems is to infer a rating for a requesting user about a target item unknown to the user. Based on this query, all items known to the system can be sorted according to the inferred rating for a requesting user. In order to recommend an item, the best ranked item(s) can be presented to the user.

Ratings are tuples that contain references to a user and item and the rating value. Consequently, a new tuple is created whenever a user makes a rating. In order to process the fundamental query, a filtering algorithm such as user-based collaborative filtering processes the collection of tuples as follows.

1. Compute the similarity of the requesting user to all other users.
2. Select n most similar users.
3. Aggregate the rating values of the users selected in step 2 for the target item.

The resulting aggregation is the rating value inferred for the requesting user about the target item. However, as was mentioned in Sect. 3, the first two steps can be omitted in a mobile setting where users exchange their own ratings whenever they are in each other's vicinity. Therefore, the main components of this recommender application can be summarised as follows. The application model consists of user and item entities and a relationship representing rating tuples. The application logic performs the rating inference by retrieving all rating tuples

stored locally which contain the target item and aggregating their rating values. The collaboration consists of sending rating tuples made by the local user whenever that user encounters other users in the vicinity while consuming items. Note that peers do not share tuples as soon as they are in each other's vicinity but wait for a configurable amount of time before starting the data transmission.

For illustration, we now describe how this application is implemented using the Java WTK platform. In a first step, the application model is mapped to the Java object model. The concepts of a user and an item are described by classes `User` and `Item`, respectively. These classes declare at least one identifier attribute allowing their instances to be recognised as equal when they are shared among peers. Additionally, attributes such as names and descriptions can be added to provide the users with meaningful information. Finally, we define the class `RatingTuple` to represent ratings as shown below.

```
public class RatingTuple {
   User user;
   Item item;
   float rating;
}
```

To access and share ratings, we create a peer collection named `RatingTuples` with set behaviour and `RatingTuple` as its member type. The following code shows the creation of this collection and how it is set to be persistent.

```
P2PSet<RatingTuple> ratingTuples =new P2PSet<RatingTuple>("RatingTuples");
ratingTuples.setPersistent();
```

Having modelled the application in Java, the developer uses the data management facilities provided by peer collections to implement the application logic. In our simple example, the application logic consists of two main components. First, it needs to give the user the possibility of generating ratings and storing them persistently. Second, it needs to be able to infer ratings about items unknown to the user. The following example shows how to store a user rating by creating a new member of the `RatingTuples` collection. Note that the amount of code is equivalent to that required for existing Java collections.

```
RatingTuple tuple = new RatingTuple(localUser, item, rating);
ratingTuples.add(tuple);
```

To infer ratings, all members of `RatingTuples` containing the target item must be selected. To do so, the query in Fig. 5 is used. It consists of a selection operation where the attribute comparison predicate constrains the item attribute to point to the target item. An attribute access node performs a projection to obtain the rating values of all tuples returned by the selection node.

Using the query tree builder, the code required to construct this query is given below. At runtime, once this query has been executed, the application logic can simply aggregate the rating values returned by the projection node.

```
QueryNode<RatingTuple> collection = Queries.collection("RatingTuples");
QueryNode<RatingTuple> selection =
   Queries.select(collection, "item", targetItem);
QueryNode<Float> projection = Queries.project(selection, "rating");
```

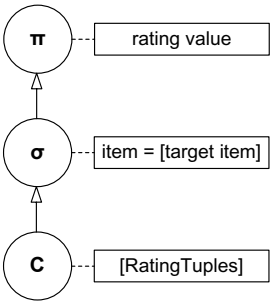

Fig. 5. A query selecting rating tuples containing the target item

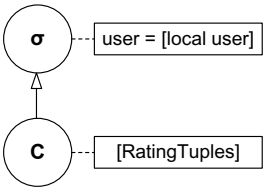

Fig. 6. A query selecting rating tuples containing the local user

To implement the opportunistic sharing of rating tuples, the query to be executed when a peer appears in the vicinity must be specified. Since we only want to send those rating tuples containing the local user, the selection query shown in Fig. 6 is built using the statements given below.

```
QuerNode<RatingTuple> selection =
   Queries.select(collection, "user", localUser);
```

This selection query is given as an argument when the `RatingTuples` collection is made available.

```
ratingTuples.setAvailable(selection);
```

We now compare the effort required to implement this application with that required if using Java WTK. Figure 7 compares the components needed and the amount of interaction required to implement data management, vicinity awareness and data sharing using Java WTK, on the left hand side, and our framework, on the right hand side. To implement the application logic using Java WTK, a Java collection is used to maintain all rating tuples. Consequently, when all tuples with a particular item must be selected, all tuples would have to be accessed in order to select those having the required attribute value. The program code implementing this behaviour would be part of the application logic whereas, using our framework, it is hidden away from the developer by the query facility. The transparent persistence mechanism is a great improvement compared to Java WTK where application objects have to be serialised manually and stored using key-value records. In order to store objects of a particular type based on key-value pairs, an application developer has to program a database-like component and put a lot of effort into overcoming the impedance mismatch between objects and key-value pairs. If objects of different types must be stored, the required effort increases even further and, allowing stored objects to reference each other, would make this even more challenging. As opposed to the simple vicinity awareness mechanism provided by our framework, the developer of a Java WTK application needs to implement the scanning of the environment based on low-level connection technologies such as Bluetooth or WiFi. Moreover,

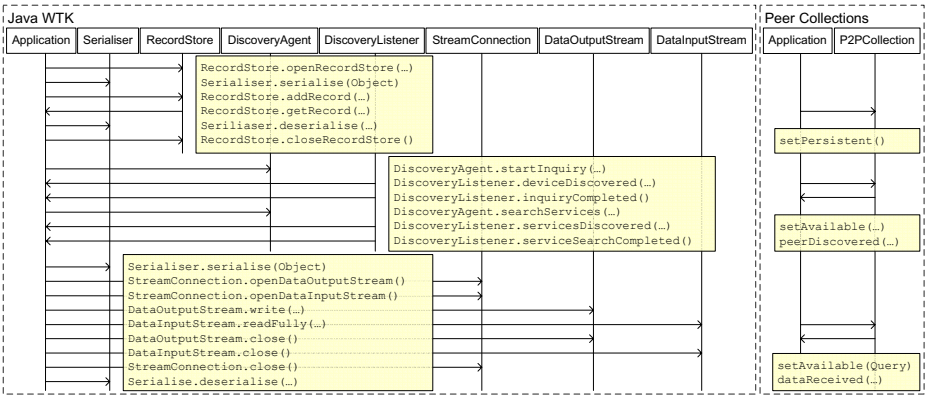

Fig. 7. Comparison of using Java WTK (left) and our framework (right)

using our framework, the application developer does not have to bother with low-level socket-based connectivity and data transmission in terms of serialisation and deserialisation. The fact that the developer can work at the level of the application model by deciding how data collections should be shared presents a significant contribution to the development of mobile social applications.

7 Conclusions

We have motivated a set of novel requirements introduced by the emerging class of mobile social applications. Due to the limitations and heterogeneity of mobile phone development platforms, we have proposed to address these requirements with a framework based on a notion of peer collections. A peer collection encapsulates data management, data sharing and vicinity awareness, all of which are recurring issues in the development of mobile social applications. Further, through the provision of both declarative queries and events associated with peer collections, our framework decouples application and collaboration logic. The merits of our approach have been shown by comparing the implementation of a collaborative filtering application based on our framework with one based on an existing mobile phone platform.

We are currently experimenting with extending our framework to accommodate further mobile social application requirements. Vicinity awareness is currently provided in terms of a real-time representation. One extension is to keep track of peers previously encountered which enables applications to take into account frequencies of encounters and to recognise social contexts. We are also considering support for access control by providing the possibility of associating multiple selection queries with a P2P collection to represent user groups or social contexts.

References

1. Eagle, N., Pentland, A.S.: Reality mining: sensing complex social systems. Personal Ubiquitous Comput. 10(4) (2006)
2. Nicolai, T., Yoneki, E., Behrens, N., Kenn, H.: Exploring social context with the wireless rope. In: OTM 2006 Workshops (2006)
3. Borcea, C., Gupta, A., Kalra, A., Jones, Q., Iftode, L.: The mobisoc middleware for mobile social computing: challenges, design, and early experiences. In: Proc. 1st Intl. Conf. on MOBILe Wireless MiddleWARE, Operating Systems, and Applications (2007)
4. Eagle, N., Pentland, A.: Social serendipity: mobilizing social software. Pervasive Computing, IEEE 4(2) (2005)
5. Counts, S., Geraci, J.: Incorporating Physical Co-presence at Events into Digital Social Networking. In: Proc. CHI 2005 (2005)
6. Lawrence, J., Payne, T.R., Roure, D.D.: Co-presence Communities: Using Pervasive Computing to Support Weak Social Networks. In: Proc. Intl. Workshop on Distributed and Mobile Collaboration (2006)
7. Srirama, S.N., Jarke, M., Prinz, W.: Mobile web services mediation framework. In: Proc. 2nd Workshop on Middleware for Service Oriented Computing (2007)
8. Traversat, B., Arora, A., Abdelaziz, M., Duigou, M., Haywood, C., Hugly, J.C., Pouyoul, E., Yeager, B.: Project JXTA 2.0 Super-Peer Virtual Network. Technical report, Sun Microsystems, Inc. (2003)
9. Aberer, K., Alima, L.O., Ghodsi, A., Girdzijauskas, S., Haridi, S., Hauswirth, M.: The Essence of P2P: A Reference Architecture for Overlay Networks. In: Proc. 5th IEEE Intl. Conf. on Peer-to-Peer Computing (2005)
10. Wang, A.I., Bjornsgard, T., Saxlund, K.: Peer2Me - Rapid Application Framework for Mobile Peer-to-Peer Applications. In: Intl. Symp. on Collaborative Technologies and Systems (2007)
11. Kortuem, G., Schneider, J., Preuitt, D., Thompson, T.G., Fickas, S., Segall, Z.: When Peer-to-Peer comes Face-to-Face: Collaborative Peer-to-Peer Computing in Mobile Ad hoc Networks. In: Proc. Intl. Conf. on Peer-to-Peer Computing (2001)
12. Rowstron, A.I.T., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001)
13. Tatarinov, I., Ives, Z., Madhavan, J., Halevy, A., Suciu, D., Dalvi, N., Dong, X.L., Kadiyska, Y., Miklau, G., Mork, P.: The piazza peer data management project. SIGMOD Rec. 32(3) (2003)
14. Ooi, B.C., Tan, K.L., Zhou, A., Goh, C.H., Li, Y., Liau, C.Y., Ling, B., Ng, W.S., Shu, Y., Wang, X., Zhang, M.: Peerdb: peering into personal databases. In: Proc. ACM SIGMOD Intl. Conf. on Management of Data (2003)
15. Rodríguez-Gianolli, P., Kementsietsidis, A., Garzetti, M., Kiringa, I., Jiang, L., Masud, M., Miller, R.J., Mylopoulos, J.: Data sharing in the hyperion peer database system. In: Proc. 31st VLDB Conf. (2005)
16. Aberer, K., Datta, A., Hauswirth, M., Schmidt, R.: Indexing data-oriented overlay networks. In: Proc. 31st VLDB Conf. (2005)
17. Cudré-Mauroux, P., Agarwal, S., Budura, A., Haghani, P., Aberer, K.: Self-organizing schema mappings in the gridvine peer data management system. In: Proc. 33rd VLDB Conf. (2007)
18. de Spindler, A., Norrie, M.C., Grossniklaus, M.: Recommendation based on Opportunistic Information Sharing between Tourists. Information Technology & Tourism (to appear)