Erschienen in: On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops : OTM Confederated International Workshops and Posters, AWeSOMe, CAMS, OTM Academy Doctoral Consortium, MONET, OnToContent, ORM, PerSys, PPN, RDDS, SSWS, and SWWS 2007, Vilamoura, Portugal, November 25-30, 2007, Proceedings, Part I / Robert Meersmann ... - Heidelberg [u.a.] : Springer, 2007. - S. 437-446. - (Lecture Notes in Computer Science ; 4805). - ISBN 978-3-540-76887-6

A Collection-Oriented Framework for 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. Personal mobile devices and mobile ad-hoc networks are the technical ingredients to build social networks in mobile environments. We present a framework that leverages social networks in order to obtain a collaborative environment for social applications. The framework provides a flexible means for the specification of collaboration protocols and allows their transparent integration with the application logic. Consequently, the framework supports a clean separation of collaboration and application logic which simplifies the development of social applications.

1 Introduction

Continuous advances in mobile technologies and communication infrastructures provide new ways of connecting people. On one hand connectivity to a fixed network where everybody can reach anybody is being increasingly made available by means of wireless hotspots and 3G networks. On the other hand, mobile devices feature short-ranged connectivity facilities such as Wi-Fi and Bluetooth that allow users to connect to other users in their vicnity and share information opportunistically. An advantage of the latter is that it provides an easy way of connecting people who have something in common based on physical co-presence in social contexts.

Mobile devices able to build short-range connections are therefore an ideal means for building social networks that establish a collaborative environment for running social applications. However, currently there is limited support for developing such applications and developers must implement the application and collaboration logic as well as the connectivity using a regular programming language such as Java with stripped down libraries. Some efforts have been made to support application development by offering peer-to-peer connectivity frameworks such as Mobile Web Services and JXTA [1] but they still require the developer to implement collaboration logic in a service-oriented architecture or in terms of message processing. Further efforts in providing higher level abstractions of peer-to-peer networks have either focussed on the allocation and retrieval of data in fixed networks [2] or they offer only few and limited collaboration primitives [3,4].

Our goal was to provide developers with a high-level framework that would support the forms of information sharing required in social applications. We base the framework on the concept of *peer collections* that allow data to be managed and shared in a flexible way. Handlers bound to a set of predefined events implement the collaboration logic in terms of how the data in these collections are shared and collaboration can be integrated transparently with the application logic. Furthermore, a set of predefined handlers is available that covers frequently used collaboration logics. This makes it possible for a user to design and deploy social applications simply by creating data collections and assigning handlers to events. We have implemented a Java binding for our framework which allows social applications to run on mobile devices featuring Bluetooth connectivity and the Java ME platform.

In Sect. 2 we motivate our framework and discuss the general requirements. Section 3 then presents the concept of peer collections using an example. Details of the framework are given in Sect. 4 while we show how social applications can be implemented based on the framework in Sect. 5. Concluding remarks are given in Sect. 6.

2 Motivation and Requirements

Previous projects have shown how physical co-presence can be used as a basis for forming a weakly connected community of members sharing similarities in terms of taste and interest [5,6,7]. There is, therefore, an increasing interest in social applications which use opportunistic information sharing in mobile environments based on ad-hoc network connections using Bluetooth or Wi-Fi. At present, such applications running in a mobile environment must be developed using a programming platform with little or no support for flexible forms of information sharing and/or distributed computation.

A distinguishing feature of social applications is the notion of collaboration. In the case of a mobile environment with opportunistic information sharing, each peer follows a set of local rules which determine its behaviour within the collaborative environment. This behaviour may include the creation, manipulation or deletion of data as well as interacting with other peers such as sending and receiving data. Local behaviour may be triggered internally or as a result of interaction. The sum of all local behaviour per peer results in the global behaviour determined by a common goal such as the allocation of application-specific data and its relevance preserving dissemination. Note that a peer could be following multiple sets of rules in which case it is collaborating on achieving multiple goals. A social application may include one or multiple goals while a peer may participate in a single or multiple applications.

As an example, consider the case of location-based information management where peers store location-specific data. As the peers move around, the data stored moves with them which affects the difference between where data is stored versus the location it refers to. A possible goal could be to minimise this difference, i.e. keeping the data at the relevant location. In [8], a protocol is proposed where peers exchange location-specific data in a way that data stays at the location of relevance despite all peers moving around freely.

We wanted to develop a framework to support developers of such applications by providing simple, high-level abstractions for opportunistic information sharing with flexible means of specifying the desired collaboration logic necessary to achieve the overall common goals. We achieved this by adopting a basic collection model for the management of data and extending the concept of a collection familiar in object-oriented programming languages with a second interface that allows the collaborative aspects of a social application to be specified through event handlers. These collections are shared, persistent data structures that allow peers to share data by connecting their collection instances and hence we refer to them as *peer collections*.

Since the collaboration logic is defined by a second interface to the normal one for managing and accessing the contents of a collection, the collaboration and application logics are kept orthogonal to each other and collaboration can be specified and integrated transparently with an application.

In the next section, we present our collection model in detail, describing how it allows both application and user data to be shared in mobile environments.

3 Peer Collections

A collection is a widely used concept for data management. In programming platforms such as Java or C++, standard libraries offer this concept in terms of an interface definition declaring operations for inserting, retrieving and removing data. A variety of implementations of this interface allows the developer to choose one according to his needs in terms of the performance for particular operations. However, the use of the interface definition throughout the application code makes it possible to exchange the collection implementation with ease. The central component of our framework, the peer collection, is an implementation of a collection which can therefore easily be integrated with an application.

A peer collection has a name which allows it to be identified and retrieved in absence of references and makes it possible for collection instances residing on different peers to be connected in the manner described later in this section. A peer collection is created by providing a locally unique name and specifying its member type such as string, file or an application-specific data type. Our framework automatically maintains a root collection containing all locally created collections where they can be retrieved by their name. Additionally, a collection of peers available in the vicinity is maintained and serves as a starting point for event handling.

Once a peer collection has been created, items can be added, retrieved and removed using its data management interface. However, our collection features a second interface allowing it to be made available and share its members. Collection instances residing on different peers can be connected if they have the same name and member types, which we refer to as them being *connectable*. When a collection is made available, another collection is created which represents its neighbourhood of peers with which its members may be shared.

The collaboration logic is specified in terms of event handling. The main events are the local creation of a member, the receipt of a remotely created member, the appearance of a new available peer in the vicinity and a new neighbour. For each peer collection, specific handlers for these events may be specified. Default handlers are specified automatically upon creation of the collection and predefined handlers may be assigned or custom observers implemented and assigned by the developer.

As part of our framework, we provide a graphical development tool which offers the means to create, retrieve and remove collections and members as well as connecting them within a set of available peers. This development tool can be used to test social applications and as a basis for developing application-specific user interfaces. We use it here to illustrate the concept of peer collections using a simple chat application. The main component of the tool is a viewer on peer collections allowing members to be created, browsed and removed. When started, a view on the root collection is opened. Consequently, collections can be created which can each be opened in their own view.

In Fig. 1 we show two views on collections residing on two different peers, a peer identified as `craig.ethz.ch` on the left and another one `morag.ethz.ch` on the right. A collection view features two panels, one of them to manage its members and one for user management (as shown in this figure). The user management panel shows the two user collections, the one containing available peers in the vicinity at the bottom and the other one representing the neighbourhood on top.

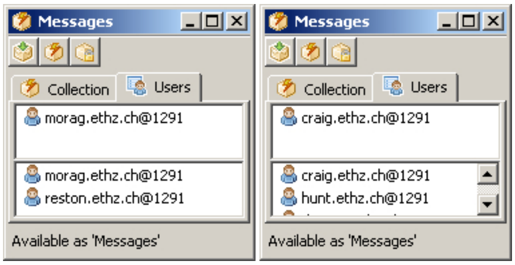
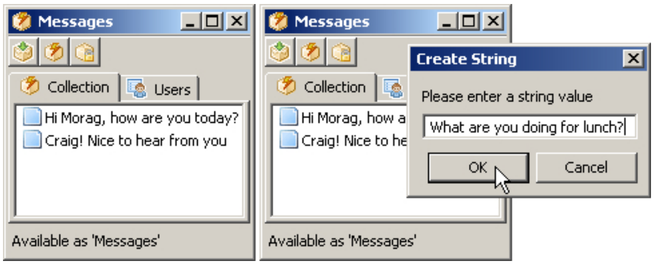


Fig. 1. Collection views on two different peers: `craig.ethz.ch` on the left and `morag.ethz.ch` on the right

Both collection instances are connectable, their names being “Messages” and their member types being strings, hence the peers may share their members. For this purpose, each peer has been put into each other’s collection neighbourhood as shown in the figure.

The creation and transmission of members is depicted in Fig. 2 where both collection views are shown, the one of craig on the left and morag on the right. The members created on both peers have been sent to each other. The first member has been created by craig and the second member was a response from morag. In this figure we see how a chat application can be implemented based



Hi Morag, how are you today?
Craig! Nice to hear from you

Please enter a string value
What are you doing for lunch?

Fig. 2. Collection views on two different peers: creating and sharing members

on a collection with a string member type that preserves the insertion order. The peer on the right is just about to create a new member which will be the next element of their current discussion thread.

In the example chat application, predefined handlers could be used for all possible events. If a peer is newly available, the users are asked whether they want to add it to the current collection's neighbourhood. In the event of a new neighbour, the user is asked whether he wants to share all existing local members with the new neighbour. Upon local creation of a new member, the member is broadcast to all peers in its neighbourhood while a remotely received member is added locally and may be forwarded to all other users in the neighbourhood if the user agrees.

A group chat can be arranged simply by adding more peers into the collection's neighbourhood. In the case of some peers having different neighbours, we effectively support multihop chatting. Since we can create multiple collections, each one representing one chat with its specific neighbours we can also support multiple parallel chats. Each collection represents a single discussion thread and can be managed separately.

In contrast to the collections offered by common programming language libraries, our collections support the notion of super- and sub-collections based on containment relations. Members of a sub-collection are automatically members of all its super-collections while the members of a super-collection are not necessarily members of any or all of its sub-collections. Sub-collections allow the notion of concept specialisation to be represented which simplifies the creation and use of more complex information management applications.

For example, we could extend our chatting application by declaring a `review` type containing a string typed attribute. A review instance would be created by the user in the same manner as a string while chatting, in order to create information possibly relevant to other users such as a short comment on a book, theatre play or restaurant. A collection "Reviews" would contain all review instances while sub-collections "Books", "Plays", "Restaurants" could be created to represent the special categories of reviews. We might also want to associate some reviews with a specific location and time captured when the user entered the review. To represent these reviews, we could create a subtype `spatio_temporal_review` with additional location and time attributes. In addition to placing such reviews in the collections for specific categories such as “Plays” and “Restaurants”, it would also be possible to dynamically create a collection “LocalReviews” containing all spatio-temporal reviews that are considered to be currently highly relevant because the time and location values are close to those of the user. Note that this means that a comment on a particular theatre show would then be accessible either through the “LocalReviews” collection whenever the user is close to or attending that show or through the collection “Plays” independently of the user’s location.

Having this review management domain modelled with collections we can introduce collaboration. The single collection “Reviews” containing all reviews could be made to collaborate in exchanging them with other peers. For example, on the event of a new peer in the vicinity, both users could be asked whether they want to exchange all their reviews. However, in the case of a browser application, users in the vicinity could simply browse each other’s collections in order to exchange reviews of particular interest. Since this application domain is modelled using multiple collections, each collection may share its members according to the same or different rules. In the case of collection hierarchies, the rules can be chosen to be inherited or overwritten.

4 Framework

Having presented the idea of peer collections by means of an example in the previous section, we now describe our framework to support the development of applications based on this concept. Figure 3 shows the main components of the framework. As can be seen from the figure, an application uses one or multiple peer collections as regular data structures. Note that, since our collection is observable, it can be integrated with a model-view-control design by acting as the model. The application may observe the collection and be notified about addition and removal events. Our collection makes use of a persistence subcomponent which allows it to store its members in persistent memory. Another component encapsulates connectivity by maintaining a set of peers in its neighbourhood and a set of available ones. This component hides the mechanisms to send and receive messages, their format definitions and how they are created and processed. It declares a peer type which encapsulates the details required for a particular connectivity technology to connect to a particular peer. Finally, the collaboration interface is offered by a handling component which observes both the collection and connectivity component.

The collaboration interface is built around a set of events that occur in a mobile distributed environment. In our framework we support events for the receipt of remote data, the local creation of data and a newly available peer as well as a newly connected one which we call a new neighbour. When one of these events is triggered, a peer may execute one or multiple actions. The set of primitives that are available to an event handler are: connecting to and disconnecting from another peer, creating and deleting data, accepting and rejecting data received from another peer as well as sending or forwarding data to other peers. In Tab. 1,

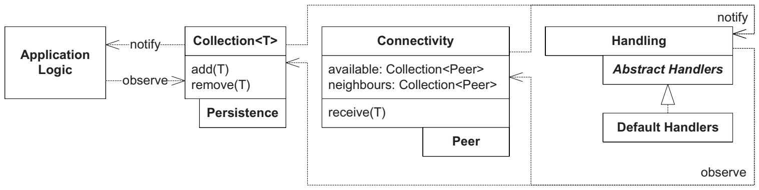


Fig. 3. The framework, its main components and collaboration means

we summarise the events and possible primitives as well as the defined default policy. Note that some primitives are not bound to any particular event. The deletion of data and the disconnection from a neighbour may be executed at any time. For this purpose, a thread can be run that will present each data item and each neighbouring peer to specific handlers which must decide whether to delete or disconnect, respectively. In addition to the predefined primitives, a developer may also choose to execute arbitrary actions upon any of the available events which is not indicated in the table.

Table 1. Events, primitive action and default policies

Events	Primitive Action	Default Policy
Local creation of data	Send to single peer or neighbourhood	Neighbourhood
Reception of remote data	Accept or reject	Accept
Acceptance of remote data	Forward to single peer or neighbourhood	Neighbourhood
New available peer	Add to neighbourhood	No
New neighbour	Share existing members	No
Periodic	Delete data, remove neighbour	None

Developing an application within our framework consists of declaring member types, creating collections and assigning handlers to events. When creating a collection, its member type and a name must be provided which will serve for identification purposes. By default, there is one collection containing all other ones including itself. As a consequence, all collections are also members of a collection and therefore our framework features a meta circular meta model which allows all facilities offered to be applied on collections as well. Furthermore, collections may be retrieved by their names which renders it unnecessary to maintain references throughout an application. An existing collection can be made available in which case a collection representing its neighbourhood is created implicitly. Once it is made available, members created locally can be propagated to all of its neighbours having an available collection with the same name and member type. Conversely, when a local peer is put in the neighbourhood of another peer's collection, members created remotely may be received.

Figure 4 shows the interface of the handling component. As can be seen from the figure, methods corresponding to the events described above are available. Our framework offers three options for specifying local rules. First, default handlers are assigned automatically if no other handler is specified explicitly. Second, a set of commonly used predefined handlers is currently being developed. Third,

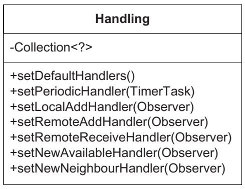

Handling
-Collection<?>
+setDefaultHandlers()
+setPeriodicHandler(TimerTask)
+setLocalAddHandler(Observer)
+setRemoteAddHandler(Observer)
+setRemoteReceiveHandler(Observer)
+setNewAvailableHandler(Observer)
+setNewNeighbourHandler(Observer)

Fig. 4. The handling component for assigning handlers to events

handlers can be implemented and assigned by the application developer. The developer may choose from extending default and predefined handlers in order to adapt them to his needs. Alternatively, they may choose to implement abstract handler definitions which allows them to specify an arbitrary action as well as approving or disapproving the execution of the primitives available.

Our framework makes use of a connection technology-dependent component in order to send and receive messages. While this component is hidden from the developer, a user may turn on and off its availability within the collaborative environment. When turned on, the technology-specific information for reaching this peer must be provided. In the case of Java sockets, this information includes a host name 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, Wi-Fi and Bluetooth.

The connectivity components of all collection instances make use of this single connection component. As a consequence, messages are destined for a particular collection which requires a way of assigning messages received to the target collection. By default, we use the name of the collection under which it is made available. However, a different name can be chosen in which case a local collection may be connected to a remote collection with a different name. For example, in the case of an application using one way member transmission, every peer would have an incoming and outgoing collection. Thus, local outgoing collections should be connected with remote incoming collections. This can be achieved by making all incoming collections available with the name of the outgoing collection.

Note that security and privacy aspects are considered on multiple levels. First, mobile devices featuring short range connectivity technologies provide the user with the ability to toggle their availability at will. Second, since our current prototype is a Java application, it is fully embedded in the Java sandbox model, preventing it from behaving maliciously outside the Java virtual machine. Finally, our framework caters for privacy by following a push paradigm, i.e. data is always sent from the owning peer while requests are not supported.

5 Application Development

In this section we will show how the components of our framework can be used to implement the simple chat application presented in Sect. 3. Figure 5 shows

```
Peer.startPeer(new Peer("reston.ethz.ch", 1291));
Collection<String> collection   = new Collection<String>("Messages");
collection.makeAvailable();

collection.getNeighbourhood().add(new Peer("craig.ethz.ch", 1291));

collection.add("hi Craig, how are you today?");
```

Fig. 5. Programmatic creation and use of chat application

```
Collection<Resource> resources = new Collection<Resource>("Resources");
resources.makeAvailable();
resources.getHandling().setNewNeighbourHandler(
 new NewNeighbourHandler(resources){

  @Override
  protected void handle(Peer neighbour) {
   // send all local user ratings to neighbour
  }
});
resources.getHandling().setRemoteAddHandler(
 new RemoteAddHandler(resources){

  @Override
  protected void handle(Object member, Peer source) {
   // sort all members according to relevance, remove the least relevant
  }

  @Override
  protected boolean forward(Object member, Peer source) {
   // no forwarding
   return false;
  }

  @Override
  protected void send(Object member, Peer source, Peer target) {
   // no forwarding
   return false;
  }
});
```

Fig. 6. Creation of a location-based social application

the source code required to do so. First we start the peer by specifying the connection-technology-specific identification of the local peer, a host name and port number for Java sockets in this example. Then we create a collection with a member type string and make it available. In the second part of the code, we add a peer to the neighbourhood of the new collection to which all members created locally will be sent. In the last part we create a member which is the beginning of the discussion. Note that no handlers are provided in this code since the default behaviour suits the needs of the chat application.

In order to demonstrate the use of events and event handlers in our framework, we also discuss how the location-based application proposed in [8] could be implemented with our framework. To do so, we create a member type called `resource` which aggregates all information to be shared such as the location and time of an observed empty car parking space. Then we create a collection "Resources" which is made available. As shown in Fig. 6, the handling of a new neighbour as well as a remotely added member must be specified since it deviates from the default behaviour.

When a new peer has been added to the neighbourhood, which is made to happen automatically as soon as a new one is available, we share all members. When we receive resources, we sort all existing members together with the remotely added ones according to their relevance which is computed as the distance between the location of the local peer and the location specified in the `resource` object. Finally we delete the least relevant resources.

6 Conclusions and Future Work

We have presented a framework supporting the development of social applications for mobile environments. Information can be shared through shared data structures known as *peer collections* which provide a simple and yet flexible means of organising both application and user data. We have shown how a particular collaboration logic can be implemented by associating event handlers with these collections and have described how the framework is implemented and also how application development is supported.

References

1. 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)
2. 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. Intl. on Peer-to-Peer Computing (2005)
3. 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)
4. 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)
5. 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)
6. Counts, S., Geraci, J.: Incorporating Physical Co-presence at Events into Digital Social Networking. In: Extended Abstracts on Human Factors in Computing Systems (2005)
7. de Spindler, A., Norrie, M.C., Grossniklaus, M., Signer, B.: Spatio-Temporal Proximity as a Basis for Collaborative Filtering in Mobile Environments. In: Proc. Intl. Workshop on Ubiquitous Mobile Information and Collaboration Systems (2006)
8. Xu, B., Ouksel, A., Wolfson, O.: Opportunistic Resource Exchange in Inter-Vehicle Ad-Hoc Networks. In: Proc. Intl. Conf. on Mobile Data Management, Berkeley, USA (2004)