Flocks: Enabling Dynamic Group Interactions in Mobile Social Networking Applications

Elisa Gonzalez Boix
egonzale@vub.ac.be

Andoni Lombide Carreton
alombide@vub.ac.be

Christophe Scholliers
cfscholl@vub.ac.be

Tom Van Cutsem
tvcutsem@vub.ac.be

Wolfgang De Meuter
wdmeuter@vub.ac.be

Theo D'Hondt
tjdhondt@vub.ac.be

Software Languages Lab
Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussel, Belgium

ABSTRACT

Mobile social networking applications enable end-users to interact *on the move*. Current applications model user groups as simple lists which have to be manually *enumerated*. This representation is both unsuitable and inefficient for group interactions: due to the openness and the mobility to which these applications are exposed, the contents of such lists are likely to change frequently. Updating the lists manually while interacting with users quickly becomes impractical. In this paper, we introduce an alternative representation for user groups named *flocks*. A flock represents a loosely-defined user group in terms of an *intensional* description. The flock content is implicitly updated when changes occur, e.g. the users's location. Flocks have group interaction provisions based on asynchronous message passing. Benchmarks indicate that flocks can be implemented efficiently by exploiting structure in their definitions. We present the flock abstraction and its implementation as the basis of a new distributed framework called *Urbiflock*.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems Distributed Applications; D.3.3 [**Software Engineering**]: Language Constructs and Features—*frameworks, data types and structures*

Keywords

mobile computing, social networks, group management, event-driven programming abstractions

1. INTRODUCTION

In the last few years social networking applications such as Facebook, MySpace and Twitter, have gained tremendous popularity. These applications moved the focus of simple data retrieving towards various social interactions between their users. Some of the traditional social networking applications have a mobile counterpart, but use a centralised server-client architecture and do not exploit so called mobile ad hoc networks. These networks allow mobile devices to join and leave a spontaneously formed network without relying on a server. In this work we focus on abstractions that can take full advantage of such networks allowing users to collaborate and engage in ad-hoc interactions with co-located users. Supported by the advances in mobile technology, a new brand of social networking applications is emerging which enable social interactions *on the move* called *mobile social networking applications* [4, 15].

In mobile social networking applications, the concept of a user group plays a central role as it models the user's social networks. Recently, middleware platforms have been developed especially for these kinds of applications [3, 13, 4]. We observe that in these platforms there is a lack of high-level abstractions to *define* mobile user groups and *interact* with them. Typically, user groups are exposed as simple lists of users. The social application provides a number of predefined user groups (e.g. "the world" or "my direct contacts" in MyNet [13]). In addition, users can create and manage groups by explicitly enumerating the users belonging to it. However, changes in the user's situation or his/her surroundings, usually have to be manually reflected by the end-user. Due to the nature of mobile networks, mobile social networking applications call for more dynamic group management. This paper proposes a high level abstraction to manage the users' social networks and interact with their dynamic nature in a straightforward way. To exemplify the kind of dynamic user groups and interactions that we target, consider the following scenario: Alice and Bob are in the cafeteria of the university when they decide it would be nice to play some badminton. Since reserving the badminton field is rather expensive, Bob decides to invite some extra players by taking his mobile phone and sending a message to the couples *currently* in the neighborhood who like to play badminton. Luckily, Carol and Denis who also wanted to play badminton see the invitation. They reply to Bob's message whereafter they meet and start playing a game. After the game, the four of them get the wild idea to organize a badminton competition for next week. Again Bob takes his mobile phone and decides to send an invitation to *all* couples at the university who like badminton.

In order to realize applications enabling interactions with user groups in an ubiquitous environment there are several challenges which need to be addressed:

- *Semantic Specification of User Groups.* Mobile users should be able to use semantical information in order to organize contacts in user groups and combine them to form more complex groups. This semantic information can be *subjective* information from the point of the user (e.g. users physically nearby) or *objective* information (e.g. the fact that another user likes badminton). Groups can be combined into a new user group of e.g. nearby users who like badminton.

- *Dynamic User Group Composition.* In a ubiquitous environment, user groups are exposed to a higher rate of changes in their contents due to device mobility. The contents of user groups change frequently because users can move out of earshot at any moment in time. Thus, the contents of user groups need to be frequently recomputed to reflect changes in the user's environment. The limited computing power of mobile devices requires that recomputing is as efficient as possible.

- *Interacting with Volatile User Groups.* Users should be able to interact with user groups of which the contents frequently change. Interacting with such volatile user groups with a simple list representation is cumbersome as its contents may change during the interaction. In addition, users should be able to interact with the current contents of the group (e.g. currently nearby badminton couples) and also the semantic contents of the group (e.g. everybody who likes badminton).

To address the challenges of defining and interacting with user groups in an mobile environment, we introduce the *flock* abstraction: a malleable, extensible event-driven representation of user groups abstracting the complexities of dealing with user groups of which the contents frequently change. The flock abstraction is implicitly maintained by the underlying system while remaining configureable by end-users. Flocks are encapsulated in a framework called *Urbiflock* designed both to serve as a testbed for flocks and to ease the development of mobile social networking applications. Several applications have been implemented using the framework in which users can broadcast announcements to each other, browse each other's profiles, launch interactive polls, etc. In summary, the main contributions of this work are:

- The introduction of a declarative event-driven abstraction for representing user groups in mobile social networking applications named a *flock*.

- A set of interaction abstractions to communicate with dynamically changing user groups.

- The implementation of Urbiflock, a mobile social networking application framework developed for Android phones which uses our flock abstraction at the heart of its design.

The rest of this paper is organized as follows. After describing the flock abstraction, we present the Urbiflock framework. We then explain the instantiation of the flock abstraction in Urbiflock and the abstractions designed to interact with flocks. We describe flocks from the point of view of an application programmer and how end-users can create flocks and interact with them in Urbiflock. Before concluding the paper, we evaluate the efficiency of the flock abstraction.

2. FLOCKS

In order to deal with the mentioned challenges of user groups in mobile social networking applications, we introduce the concept of a *flock*. A flock is an abstraction that allows end-users to create and maintain user groups in a straightforward way. Once the flock is created, end-users do not need to be concerned about its maintenance as flocks are implicitly updated by the underlying system.

Semantic Specification of User Groups. A flock is defined in terms of a *characteristic function* that determines which users belong to the flock. In order to be able to define semantic-based user groups, there are several basic characteristic functions that the system needs to provide. First, *subjective* functions, such as an `isNearby` function encoding physical proximity and an `isFriend` function encoding the friendship relationship (i.e. if a user is a friend of another one). Secondly, an `isCompliant` function encoding *objective* information associated to a user (e.g. "is male", "has blue eyes", "likes badminton", etc). We give the intensional description of such characteristic functions as follows:

$$\begin{aligned} isNearby(user) &\Leftrightarrow currentlocation - user.location < 20m \\ likesBadminton(user) &\Leftrightarrow isCompliant(badminton \in user.hobbies) \end{aligned}$$

A characteristic function is basically a predicate applied to a user. In the above examples, the `likesBadminton` function applies a filter on the hobbies of the user and the `isNearby` function tests that the current location of the user is within a certain radius (20 meters). They can be used to create the `nearbyFlock` and `badmintonFlock` flocks as follows:

$$\begin{aligned} nearbyFlock &= \forall user \in \omega : isNearby(user) \\ badmintonFlock &= \forall user \in \omega : likesBadminton(user) \end{aligned}$$

In this description, ω represent the known group of flockrs. Rather than considering each flock in isolation, flocks are designed as a composable abstraction: more complex flocks can be composed from existing flocks by means of logical operations such as `and`, `or` or `not`. The `nearbyBadmintonFlock` for example, denoting the group of users which are nearby and like badminton could be defined as follows:

$$nearbyBadmintonFlock = nearbyFlock \cap badmintonFlock$$

Dynamic User Group Composition. Flocks have been designed to represent a dynamic set of users. Once a flock is created, its characteristic function is recomputed whenever there is an event that *could* alter the contents of the flock. There exist three different kinds of events that may alter a flock corresponding to the three different kinds of basic characteristic functions: (1) *discovery event* triggered when users go online or offline, (2) *friend event* triggered when users get added or removed from the user's friends list and (3) *profile event* triggered when users update their profile information, such as hobbies, social preferences, etc. When one of these events get triggered by the system, all flocks depending on such an event get notified and their characteristic function is applied to the user that generated the event. Programmers do not need to deal with such low-level events. Rather, the characteristic function translates them into addition and removals of users in a flock. Applications can then register an event listener to process them. For example, considering the `nearbyFlock` previously defined, if a user moves out of the specified radius, the `isNearby` function is recomputed removing the disconnected user.

Interacting with Volatile User Groups. In order to deal with intermittent disconnections in mobile networks, communication with a flock has been specified in terms of asynchronous message passing. Interaction with flocks is specified by two operators as follows:

nearbyBadmintonFlock `<-` (*"Badminton at 6pm!"*)
nearbyBadmintonFlock `<+` (*"Tournament next week!"*)

Messages sent via `<-` are only transmitted to the users in the flock at the moment of sending the message. Messages sent via `<+` are transmitted first to the current flock contents and from then, the message is transitively propagated from user to user in order to reach the semantic contents of a flock. This happens by propagating the message to the users that were not reached yet that are in the same flock of the users that received the message originally. The underlying system takes care that despite frequent disconnections those users who are targeted by a message will be reached if they connect with another receiver even if they were not connected at the moment the message was originally sent.

3. URBIFLOCK

We have prototyped flocks as the basis of a new distributed object-oriented framework sculpted for the development of such applications called *Urbiflock*. In Urbiflock, users (called *flockrs*) can meet other users and interact with them, for example by sending each other messages. Flockrs have a profile with information about their identity which can be browsed by other flockrs. By means of flocks, users can create and manage their social networks.

Similar to other social networking applications, programmers can build applications and plug them into Urbiflock. The aim of Urbiflock is to assist in the rapid development of such applications. To this end, it provides programmers with the necessary infrastructure to deal with the highly dynamic environment to which mobile social networking applications are exposed. The framework has been written in the AmbientTalk [18] language, an object-oriented distributed language designed for mobile ad hoc networks. AmbientTalk's asynchronous concurrency model and its built-in peer-to-peer service discovery makes the language suitable for writing such kind of mobile applications. Since AmbientTalk is embedded in Java, developers can reuse all existing Java libraries when writing Urbiflock applications, e.g. GUI components, encryption libraries, etc. Although privacy issues are not the focus of this work, the framework provides hooks to encode privacy strategies as explained later.

3.1 Urbiflock Architecture

The architecture of Urbiflock is shown in figure 1. Every mobile device has one instance of the Urbiflock framework running on top of AmbientTalk (which is in turn hosted by a JVM). The framework is divided in three layers: applications, core, and infrastructure. The application layer consists of two types of applications. Core applications are default applications that provide access to Urbiflock's core such as flock and profile editors. User applications are end-user applications that are plugged into the framework. User applications currently available in Urbiflock are: *I rate you (IR8U)* that allows users to ask proximate users to rate them on a certain subject, and *Guanotes* which is thoroughly ex-

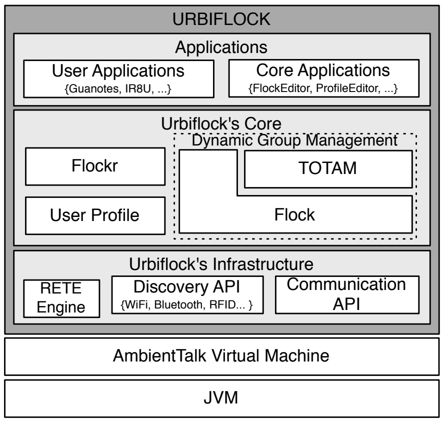


Figure 1: Urbiflock architecture

plained later. End-users need to explicitly add an application before it can be used[1]. Running applications have controlled access to user information via the framework. Applications can have access to the user's flocks and the users who have installed the same application.

Urbiflock's core is composed of the following three facilities used by applications to interact with the framework:

Flockr represents a user in the framework. A flockr has exactly one profile and can create and be registered to multiple flocks. It can also have multiple installed applications. A flockr is also the gateway that other applications can use to get information about the user or about the applications that the user has installed.

User Profile represents the user identity (e.g. name, gender, hobbies, etc.). Profile fields can be used to specify flocks based on semantic information associated with the user. For example, end-users could add a field with their year of graduation which can be in turn used to create a flock of nearby flockrs which graduated the same year. Profiles are highly extensible and besides a number of mandatory fields, end-users can add their own custom fields. From a programmer's perspective, the framework provides several default types of fields (numbers, strings, dates and choices). Moreover, programmers can easily add new data types without having to write too much boiler plate code.

Dynamic Group Management provides developers with the necessary infrastructure to reason and interact with user groups. This module has been designed to take into account the requirements for representing user groups in mobile social networking applications previously explained. It consists of two abstractions: the *flock* abstraction, an intensional event-driven extensible representation of user groups, and the *TOTAM infrastructure*, a tuple space-based framework that enables flock communication. We describe the two abstractions in detail in the next sections.

Finally, the infrastructure layer consists of three low-level abstractions on which the Urbiflock' core is based: a RETE engine [9] which infers the flock contents, and the service discovery and communication APIs to discover and communicate with nearby applications, respectively.

4. FLOCKS IN URBIFLOCK

[1]Due to space limitations, the distribution and deployment of applications is out of the scope of this paper.

As shown in figure 1, user group management is a central concept in the Urbiflock framework. In this section, we explain the integration of the flock abstraction in Urbiflock and the design decisions made. We describe how it can be used by programmers to build a new Urbiflock application, and how it can be used from an end-user's perspective.

4.1 Efficiently Organizing Social Networks

As explained before, flocks are defined in terms of characteristic functions. Such characteristic functions are the key abstraction to support a declarative mechanism to determine which users belongs to a flock. In order to efficiently derive the contents of a flock in the face of frequent updates, flocks are implemented based on a well known caching algorithm named RETE [9]. RETE is a forward-chaining inference engine that caches intermediary results of logical derivations such that changes in the rule set do not trigger an entire rederivation of all possible outcomes.

Flocks are structured as a RETE network where a characteristic function corresponds to a RETE node. A RETE node caches the intermediate results in order to avoid re-evaluating the whole flock when a change in the environment *adds* data to it, for example when a user moves into range. It also allows the efficient re-computation of the flock contents when a change in the environment *removes* data from it, for example when a user moves out of range. The RETE network is kept on a user per user basis resulting in a purely local data structure which is fed by both local and remote events coming from other flockrs. These events are fired by the Urbiflock framework when it detects changes in the flockr' user profile, its friends list or when the service discovery module detects the connection or disconnection of flockrs. These changes produce the profile, friend and discovery events, respectively. Note that while friend and discovery events are locally triggered by the framework, a profile event is also propagated to flockrs in direct communication (via the TOTAM framework as we explain later).

This design allows characteristic functions to be reused in the definition of multiple flocks so that they are evaluated only once when a change occurs. When a more complex flock is composed using logical operators on existing flocks, the existing characteristic functions gets reused and the combination forms the characteristic function of the newly created flock. Apart from being a highly composable and reusable abstraction, it offers a substantial speed-up compared to a non-caching approach as we show in the evaluation section.

4.2 Flocks From a Programmer's Perspective

Urbiflock provides programers with two predefined flocks: `friendsFlock` and `nearbyFlock`. `friendsFlock` represents a list of friends or acquaintances of a user. In Urbiflock, this is the *only* flock that is not automatically derived and thus, requires user intervention to be updated. `nearbyFlock` represents the flockrs that are physically colocated, i.e. the flockrs that are within communication range of the device on which Urbiflock is running. Whenever a user moves in or out of range, the framework updates `nearbyFlock` to reflect the changes in the network topology. Additionally, programmers can create custom flocks. To this end, the framework exposes the flock abstraction as an object with the API given in table 1.

The programmer can create a flock in two different ways. First, a flock can be created by calling the flock constructor and passing it a characteristic function as shown below.

`new(charactFunc)`	Flock constructor.
`and(aFlock)`	Returns a flock composed of two flocks using `and` predicate.
`or(aFlock)`	Returns a flock composed of two flocks using `or` predicate.
`not()`	Returns the negation of the flock.
`getSnapshot()`	Returns a snapshot of the flock.
`addListener(aListener)`	Registers a listener to the flock that must implement two methods: `notifyFlockrAdded(aFlockr)` `notifyFlockrRemoved(aFlockr)`

Table 1: Flock API

```
def maleFlock := flock.new(isMale);
```

We show later in this section how such an `isMale` characteristic function is created. The second way of creating a flock is to compose it out of other flocks using one or more of the logical operations defined on the flock abstraction. The example below shows how the previously defined `maleFlock` can be composed to create the `nearbyMaleFlock` flock containing all nearby flockrs that are male.

```
def nearbyMaleFlock := nearbyFlock.and(maleFlock);
```

The `and` and `or` methods of the flock API have an optional predicate parameter to allow more complex combinations using variables (see section 7).

In order to create flocks, the Urbiflock framework offers two built-in characteristic functions: the `isFriend` function checks if a flockr is in the user's buddy list and the `isNearby` function checks if a flockr is physically close to the user. Additionally, users can build a characteristic function based on profile information. For example, the definition of the `isMale` characteristic function used in the `maleFlock` definition previously introduced is shown below.

```
def isMale := makeIsCompliantCharacteristic(
  { |flockr| flockr.profile.gender == 'Male });
def maleFlock := flock.new(isMale);
```

In this example, the `isMale` function filters flockrs on their gender. The `makeIsCompliantCharacteristic` function is part of the Urbiflock API and allows the programmer to quickly create a characteristic function that matches on properties of a flockr's profile. The matching consists of applying the predicate passed as an argument to a concrete flockr (of which the profile can be accessed). The `makeIsCompliantCharacteristic` function can be used in a similar way to create a flock denoting the group of users that like badminton as follows:

```
def likesBadminton := makeIsCompliantCharacteristic(
{ |flkr| flkr.profile.interests.includes('badminton)});
def badmintonFlock := flock.new(likesBadminton);
```

We will show later how this flock is combined with a flock denoting the couples in the neighborhood to define the user group that Bob targets in our scenario (i.e. all the couples in the neighborhood who like badminton).

Applications can be notified of the addition and removal of flockrs in a flock by registering them as listeners on flocks. For example, the Urbiflock framework offers as a core application a simple viewer which shows the list of the current flockrs in a given flock. The code snippet below shows the part of the `flockViewer` implementation which takes care of registering the application to receive flock updates. The last line of code shows how a `flockViewer` instance is created and registered as a listener to the `nearbyMaleFlock`.

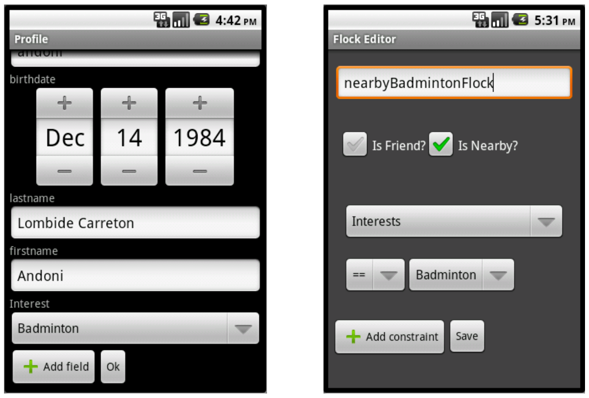
```
def flockViewer := object: {
  // ...
  def notifyFlockrAdded(aFlockr) {
    // Add aFlockr to the list of flockrs
  };
  def notifyFlockrRemoved(aFlockr) {
    // Remove aFlockr from the list of flockrs
  } };
nearbyMaleFlock.addListener(flockViewer.new());
```

In response to the corresponding flock update events, `nearbyMaleFlock` calls the `notifyFlockrAdded` and `notifyFlockrRemoved` notification methods of the listener that in turn updates the user interface with the new contents of the flock.

Finally, the API provides the `getSnapshot` method that returns a list of flockrs. This list corresponds to the contents of the flock at the point in time when the snapshot was taken.

4.3 Flocks From an End-User's Perspective

Urbiflock offers some core applications to allow end-users to easily create, edit, view and compose flocks. Consider again the example of a user who is looking for people nearby that like to play badminton. The criterion that someone is a badminton player is based on information that is not in the standard user profile. Badminton players could agree to extend their user profiles as shown in figure 2.



Profile

birthdate

Dec 14 1984

lastname

Lombide Carreton

firstname

Andoni

Interest

Badminton

Add field Ok

Flock Editor

nearbyBadmintonFlock

Is Friend? Is Nearby? ✓

Interests

== Badminton

Add constraint Save

Figure 2: Matching on custom profile fields

When adding custom fields to the profile, the user can specify the type of the field such as a number, a piece of text, a date, a choice, etc. Once this information is added to the user profile, it can be used in the definition of a flock to match with other flockrs. These flockrs must have a field in their profile with the same name and type of value. In our example the user chooses to match on other flockrs that like badminton based on the `Interests` field, as shown in figure 2. The underlying implementation optimizes the verification of these criteria by translating them into a RETE network as explained before.

5. INTERACTING WITH FLOCKS

To enable mobile social interactivity, users should be able to communicate both with the currently connected flockrs in a flock (e.g. flockrs belonging to a flock which are currently reachable) and the semantically determined contents of the flock (e.g. everybody who belongs to a flock regardless of being currently connected). Due to the dynamics of peer-to-peer mobile networks, a good communication mechanism should make it possible to communicate with a flock taking into account that all flockrs belonging to it may not be connected at the same time while maintaining privacy in their interaction and limiting network traffic. The basic communications API of the framework provides means to address and communicate with individual flockrs by means of remote objects references. However, communicating with a volatile group of flockrs in this fashion does not scale (as it would require to manually manage a group of remote references of which the contents frequently change). Instead, we have designed a dedicated communication abstraction for flocks called *Tuples on the Ambient (TOTAM)* [17].

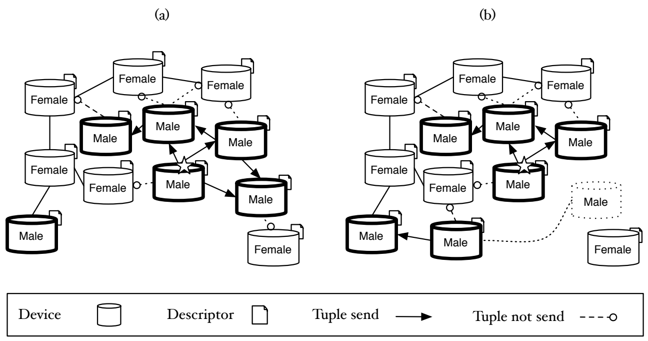


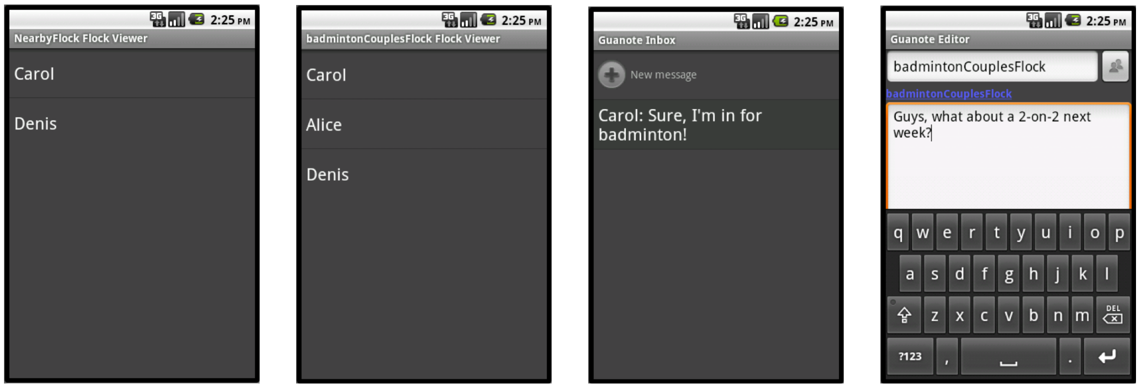
Figure 3: Operational sketch of a scoped tuple

TOTAM is a tuple space-based framework that allows flockrs to communicate with other flockrs belonging to a flock by exchanging tuples. Rather than sending tuples to all connected peers, TOTAM tuples have been augmented with a dynamic scoping mechanism. Every tuple carries the definition of the flock that targets which allows the tuple itself to determine whether a flockr is in its *scope* before it is physically interchanged between two devices.

Figure 3 illustrates the exchange of tuples through several devices hosting the Urbiflock framework. Consider that a user running Urbiflock on the device with a star has the definition of two flocks: the male and female flock. This user sends a message to his male flock. This interaction is implemented using the TOTAM framework by means of a tuple to be propagated to the semantically defined contents of the male flock. Which devices belong to a flockr in the male flock is determined by means of a *descriptor* associated to every device. This descriptor contains semantic information that is used by the tuples at sending time to decide whether a certain location is in their scope. In Urbiflock, the user profile is used as the descriptor.

Figure 3(a) shows that a tuple is injected from the device with a star. This device is connected to four male users and one female user. As the scope of the tuple is limited to male users the tuple will only be sent to the four male users connected. From those four users the tuple is transitively propagated obeying the scope of the tuple until all connected male users are reached without being transmitted to female users. Note that one male user is not transitively connected to the sending device and thus does not receive the tuple. Figure 3(b) shows that a male user moved into the range of the isolated male user and transmits the previously received tuple to the male user as he did not receive the tuple yet. Again, the tuple is not transmitted to the female users.

It is important to note from this operational sketch, TO-



Carol: Sure, I'm in for badminton!

Guys, what about a 2-on-2 next week?

Figure 4: Bob's (a) NearbyFlock, (b) badmintonCouplesFlock, (c) Guanotes inbox and (d) Guanote editor

TAM allows communication with flockrs that are not connected at the same time, e.g. the male user isolated at first receives a tuple without being connected at any time with the originator of the tuple (the star device). In addition, we ensure that only flockrs belonging to the targeted flock are used as *routers*, i.e only potential targeted flockrs carry tuples. Such a tuple propagation strategy has the benefits of enhancing privacy and decreasing the burden on the network traffic. Note that it is not possible to guarantee that all users in the scope of the tuple are reached: a targeted flockr must be at least *once* in communication range with a router or the originator device to receive a tuple. However, other peer-to-peer propagation strategies (e.g. a flooding algorithm) could have been encoded in TOTAM.

5.1 Interacting With Flocks From a Programmer's Perspective

In order to interact with flocks, programmers do not have to manually insert tuples using the TOTAM framework as in traditional tuple space middleware [10]. Rather, the flock abstraction provides programmers with two methods (in addition to the ones discussed in table 1): the `sendToCurrentContents` method sends an asynchronous message to the contents of a flock at the point in time of executing the operation, and the `sendToAll` method sends an asynchronous message to all flockrs belonging to a flock. These methods make use of the TOTAM framework to exchange the necessary tuples to send the corresponding message to the targeted flockrs.

We show how these methods can be used our scenario below. For example, Bob's Urbiflock sends his initial message to invite nearby couples for a badminton match as follows:

```
badmintonCouplesFlock.sendToCurrentContents("Let's play
  some badminton at 6pm");
```

The `sendToCurrentContents` method first determines the contents of the flock at this moment in time (by means of the `getSnapshot` method) and inserts a tuple in the TOTAM framework whose scope is limited to flockrs belonging to the resulting list of flockrs. In the second part of the scenario, Bob would like to reach all couples playing badminton in Urbiflock. This can be encoded as follows:

```
badmintonCouplesFlock.sendToAll("Badminton tournament
  next week!");
```

In response to the `sendToAll` messsage, Urbiflock inserts a tuple in the TOTAM framework of which scope is the `badmintonCouplesFlock`. To determine the flockrs belonging to it, the tuple recomputes the flock definition at every device where it is transmitted. These two methods have an optional parameter specifying a timeout to obtain time-based delivery guarantees on sent messages.

Urbiflock applications can subscribe a listener with the TOTAM framework to receive messages. The snippet below shows the skeleton code of an application that registers itself to display the contents of a message.

```
TOTAM.registerArrivalTupleListener( object: {
  def receiveTuple(tuple) {
    // display the contents of the tuple
  }});
```

5.2 Guanotes: An Application Interacting With Flocks

To demonstrate the use of the TOTAM framework in an Urbiflock application, we have implemented the Guanotes application. Inspired by the Wall plug-in of Facebook where people can post messages on somebody else's wall, the Guanotes application allows end-users to interact with flockrs belonging to a particular flock by means of messages called *guanotes*. A guanote consists of a message and a receiver list denoting the targeted audience. A guanote can be sent to the flockrs belonging to a flock which are currently in their surroundings or to a flock as a whole (i.e. the semantic contents of the flock). In addition, a guanote can also target individual flockrs as is the case in the Wall plugin.

Guanotes keeps track of the connected flockrs which are running the application. Guanotes instances communicate with each other by means of the TOTAM framework. A guanote is implemented as a kind of tuple. Figure 4 shows the Urbiflock GUI on Bob's device during the process of typing a guanote after playing a match with Carol and Denis. In particular, four different screenshots are displayed: (a) the contents of Bob's `NearbyFlock`, (b) the contents of Bob's `badmintonCouplesFlock`, (c) Bob's Guanotes inbox (that contains a guanote previously received from Carol replying to his first invitation), and (d) the editor for a new guanote to invite all his friend's couples to participate in a badminton tournament. Note that there are few flockrs colocated at this time with Bob (people in the `NearbyFlock` shown in Figure 4 (a)) which belong to the `badmintonCouplesFlock` flock (b). As he would like to reach all couples interested in badminton for the tournament, he selects the `badmintonCouplesFlock` in the receiver list in Figure 4 (c). In order to send a guanote to the current flockrs in the `badmintonCouplesFlock`, Bob had to long press on the flock name in the receiver list and select the "Expand flock..." option.

6. IMPLEMENTATION

Urbiflock is available for download with AmbientTalk at `http://tiny.cc/urbiflock`. Our experimental setup consists of phones running Android 2.0 or higher that communicate by means of TCP broadcasting on a wireless ad hoc WiFi network. Urbiflock can be also deployed on phones running J2ME under the connected device configuration.

Urbiflock's discovery module uses AmbientTalk's discovery engine based on multicast messaging via UDP. This means that the `isNearby` characteristic function infers that a flockr is nearby to another flockr if the latter is reachable within the same WiFi network. However, `isNearby` can be easily adapted to work with a different technology (e.g. Bluetooth). Below is the skeleton implementation of `isNearby`.

```
def isNearbyCharac :=
  extend: UnfilteredCharacteristic with: {
    // methods invoked by the discovery module.
   def notifyJoined(flockr){ //insert flockr in RETE};
   def notifyLeft(flockr){ //remove flockr from RETE};
};
discoveryModule.registerListener(isNearbyCharac);
```

The `UnfilteredCharacteristic` is a prototypical characteristic function that encapsulates the behavior of the underlying RETE network. In this example, we extend it (by means of the `extend:with:` construct) with additional behavior to insert or remove flockrs from the RETE node caching the flockrs in the `nearbyFlock` when they move in or out of range. For this purpose, the `isNearby` characteristic function registers itself to the discovery module to be notified when a flockr is in close range. It suffices to register this characteristic function as a listener to another discovery module to alter its behavior.

7. EVALUATION

In this section, we report on benchmarks of our flock prototype implemented in the Urbiflock framework [2]. The aim of our benchmarks is to measure the speed-up acquired by using our flock abstraction compared to a hand-crafted solution which does not cache intermediate derivations. We benchmarked our flock abstraction by means of the flock used in our scenario, which contains all the couples in the neighbourhood who like badminton. The code to construct this flock is shown below.

```
def nearbyCouplesFlock := nearbyFlock.and(
  nearbyFlock,  {|P1,P2|  P1.partnerId == P2.id});
def nearbyBadmintonFlock := nearbyCouplesFlock.and(
  badmintonFlock);
```

We first construct a flock which contains all the people whose partner is nearby. This flock combines the `nearbyFlock` with itself, by specifying that the partner id of nearby person `P1` has to have the same id as the nearby person `P2`. The `nearbyCouplesFlock` is in turn combined with the `badmintonFlock` to construct the `nearbyBadmintonFlock`. In the ad hoc implementation of the `nearbyBadmintonFlock`, a user list is kept to represent the current people in the surroundings. Every time a new user appears in the network the entire contents of

[2]Benchmarks of the network performance of the underlying AmbientTalk language can be found in [7].

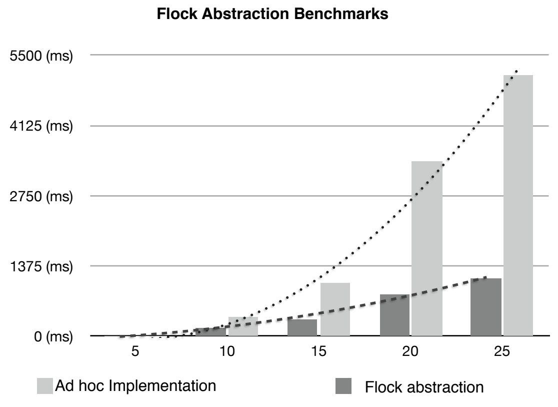


Figure 5: Flock contents derivation times.

the list are inspected in order to determine whether his/her partner is also in the immediate vicinity and if he/she likes badminton. Despite not being a complex flock, the code to be written for the ad hoc implementation was longer and required explicit management of the discovered users. The benchmarks in figure 5 show the processing time of computing the flock contents with a varying number of users in the immediate vicinity. These tests were conducted on a HTC Desire running an official version of Android 2.2. For each simulation variation (1 till 25) the test starts with an empty list of nearby users and stops when the total amount of users is added. For each measurement, we took the average of 10 tests, discarding the extremes. As shown in figure 5 we measured a 4.5x speed-up in the case of 25 people for the `nearbyBadmintonFlock`. This speed-up is due to the caching of the derivation of the flocks which drastically reduces the number of comparisons that need to be performed.

8. RELATED WORK

Several areas of related work to the flock abstraction exist, including mobile social applications, frameworks and communication abstractions for mobile networks.

Mobile User Groups Other possible approaches are contact recommendation algorithms, such as in Cluestr [11], VENETA [19] and [4]. Other approaches use context information defined by geographical location of devices [16], safe distance [6] or device communication range [5]. Friendlee [1] uses social network information (such as the most frequently called people by a user) to rank user contacts. Wang et al. [20] propose abstractions to support and manage *persistent social groups*. In contrast to flocks, members in a social group are chosen by consensus between group members and interactions amongst them last over a period of time with a concrete beginning and end.

Mobile Social Networking Applications Slam [8] is a mobile group-based messaging and media sharing application, but uses a centralized infrastructure and groups of contacts have to be listed extensionally. MyNet [13] is a secure platform which enables end-users to organize their social networks by representing them as lists. Three built-in user groups are provided: "the world", "my direct contacts", "my extended contacts" (i.e. users two social hops away). Similarly, in BT Communities [3], the notion of proximity is defined by Bluetooth connectivity. Urbiflock provides a more abstract way independent of the networking technology to determine which users are in a user group.

Group Interactions Group interaction has also been studied in the context of mobile ad hoc networks. M2MI [14] introduces a data type called a *handle* that is used to denote a dynamic group of remote Java objects of the same interface. M2MI only uses network connectivity and Java interface types of objects to decide which objects are in the group. In contrast to our interaction model, M2MI invocations on objects that are not immediately reachable are lost. Other approaches provide delivery guarantees by exploiting context information such as location and/or distance [16, 6, 5], or by means of efficient content routing algorithms [15, 12]. These alternative group communication mechanism could be encoded in TOTAM as tuple protocols.

9. CONCLUSIONS AND FUTURE WORK

We presented a novel abstraction to represent user groups in mobile social networking applications named *flocks*, designed to deal with the openness and the mobility to which these applications are exposed. Flocks are customizable based on semantic information derived from the user's profile, the user's friend links, and the physical colocation of users. Custom flock criteria can be easily defined and composed using logical operators to form more complex flocks that are *declaratively specified*. Programmers do not need to deal with the low-level events that alter the flock contents. Flock contents are *efficiently recomputed* by a RETE-like inference engine. Additionally, flocks provide an event-based interface to allow applications to be notified of changes in their contents. Finally, we provide an *asynchronous interaction model* to communicate with the currently reachable users in the flock or the semantics contents of the flock.

Flocks have been prototyped in Urbiflock, a framework for mobile social networking applications. As future work we would like to augment Urbiflock with semi-automatic means to infer relationships based on recommendation algorithms [4], or sensing data, e.g. using RFID technology [2].

10. ACKNOWLEDGMENTS

Elisa Gonzalez Boix is funded by the prospective research for Brussels program of IWOIB-IRSIB. Andoni Lombide Carreton and Christophe Scholliers are funded by a doctoral scholarship of IWT-Vlaanderen. Tom Van Cutsem is a postdoctoral fellow of FWO-Vlaanderen.

11. REFERENCES

[1] A. Ankolekar, G. Szabo, Y. Luon, B. A. Huberman, D. Wilkinson, and F. Wu. Friendlee: a mobile application for your social life. In *Human-Computer Interaction with Mobile Devices and Services (MobileHCI'09)*, pages 1–4. ACM, 2009.

[2] A. Barrat, C. Cattuto, V. Colizza, J.-F. Pinton, W. V. den Broeck, and A. Vespignani. High resolution dynamical mapping of social interactions with active rfid. *CoRR*, abs/0811.4170, 2008.

[3] R. Beale. Supporting social interaction with smart phones. *IEEE Pervasive Computing*, 4(2):35–41, 2005.

[4] S. Ben Mokhtar and L. Capra. From pervasive to social computing: Algorithms and deployments. In *ICPS 2009*, pages 169–178. ACM, 2009.

[5] L. Briesemeister and G. Hommel. Localized group membership service for ad hoc networks. In *Inter. Workshop on Ad Hoc Networking (IWAHN)*, pages 94–100, 2002.

[6] G. catalin Roman, Q. Huang, and A. Hazemi. Consistent group membership in ad hoc networks. In *ICSE*, pages 381–388, 2001.

[7] J. Collins and R. Bagrodia. Programming in mobile ad hoc networks. In *4th Annual International Conference on Wireless Internet (WICON '08)*, pages 1–9, 2008.

[8] S. Counts. Group-based mobile messaging in support of the social side of leisure. *Computer Supported Cooperative Work (CSCW)*, 16(1-2):75–97, 2007.

[9] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. In *Artificial Intelligence & Databases*, pages 547–557. Kaufmann Publishers, INC., San Mateo, CA, 1989.

[10] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan 1985.

[11] R. Grob, M. Kuhn, R. Wattenhofer, and M. Wirz. Cluestr: mobile social networking for enhanced group communication. In *Int. Conf. on Supporting Group Work (GROUP '09)*, pages 81–90. ACM, 2009.

[12] E. Jaho and I. Stavrakakis. Joint Interest- and Locality-Aware Content Dissemination in Social Networks. In *Wireless On demand Network Systems and Services*, pages 161–168. IEEE Press, 2009.

[13] D. N. Kalofonos, Z. Antoniou, F. D. Reynolds, M. Van-Kleek, J. Strauss, and P. Wisner. Mynet: A platform for secure p2p personal and social networking services. In *PerCom 2008*, pages 135–146, 2008.

[14] A. Kaminsky and H.-P. Bischof. Many-to-many invocation: a new object oriented paradigm for ad hoc collaborative systems. In *OOPSLA 2002*, pages 72–73. ACM Press, 2002.

[15] A. Mashhadi, S. Ben Mokhtar, and L. Capra. Habit: Leveraging human mobility and social network for efficient content dissemination in manets. In *Inter. Symp. on a World of Wireless, Mobile and Multimedia Networks*. IEEE, 2009.

[16] R. Meier and V. Cahill. Exploiting proximity in event-based middleware for collaborative mobile applications. In *Distributed Applications and Interoperable Systems (DAIS)*, pages 285–296. Springer-Verlag, 2003.

[17] C. Scholliers, E. Gonzalez Boix, and W. De Meuter. Totam: Scoped tuples for the ambient. In *Proc. of the CAMPUS Workshop collocated with DisCoTec'09 federated event*, volume 19, pages 19–34. EASST, 2009.

[18] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, and W. De Meuter. Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In *Inter. Conf. of the Chilean Comp. Science Society*, pages 3–12. IEEE C.S., 2007.

[19] M. Von Arb, M. Bader, M. Kuhn, and R. Wattenhofer. Veneta: Serverless friend-of-friend detection in mobile social networking. In *WiMob 2008*, pages 184–189. IEEE Computer Society, 2008.

[20] B. Wang, J. Bodily, S. K. S. Gupta, and E. K. S. Gupta. Supporting persistent social groups in ubiquitous computing environments using context-aware ephemeral group service. In *PerCom 2004*, pages 287–296. IEEE Computer Society, 2004.