FutureWare: Designing a Middleware for Anticipatory Mobile Computing

Abhinav Mehrotra, Veljko Pejovic, and Mirco Musolesi

Abstract—Ubiquitous computing is moving from context-awareness to context-prediction. In order to build truly anticipatory systems developers have to deal with many challenges, from multimodal sensing to modeling context from sensed data, and, when necessary, coordinating multiple predictive models across devices. Novel expressive programming interfaces and paradigms are needed for this new class of mobile and ubiquitous applications.
In this paper we present FutureWare, a middleware for seamless development of mobile applications that rely on context prediction. FutureWare exposes an expressive API to lift the burden of mobile sensing, individual and group behavior modeling, and future context querying, from an application developer. We implement FutureWare as an Android library, and through a scenario-based testing and a demo app we show that it represents an efficient way of supporting anticipatory applications, reducing the necessary coding effort by two orders of magnitude.

Index Terms—Anticipatory computing, mobile middleware, mobile sensing, prediction.

1 Introduction

Equipped with accelerometers, GPS, light, humidity, and orientation sensors, boasting high-power computation resources, and also being with their owners at all times, today's smartphones represent an ideal platform for context-aware computing. A range of different aspects of the context, such as users' physical activity [1], co-location with friends [2], social activity [3], even emotions [4], can be extracted thanks to smartphone's multimodal sensors and machine learning [5]. Contextual information, in turn, can augment mobile applications ranging from healthcare [6], [7], safety [8], environment monitoring [9], transport [10], [11] to human-computer interactions [12], [13], [14].

Most of the work in the area of context-aware computing focuses on inferring current context, yet highly personal everyday use of the smartphone allows us to go beyond and identify the inherent patterns of human behavior from the sensed data. In a nutshell, machine learning models of human behavior can be built, and, on top of them, *predictions* of future context can be made. Compared to merely inferring context from currently sensed data, context predictions are more difficult to make. Certain modalities, however, tend to be more suitable for prediction – mobility is one such example. People follow circadian rhythms, commute to work, relax during weekends and so on. Consequently, behavioral patterns can be mined and human mobility can be predicted with a high level of accuracy [15], [16]. In addition to mobility, predictive models of other contextual aspects such as mobile users' network connectivity [17], [18], communication patterns [19], application usage [20], [21] and even for health state [6] have been demonstrated.

Context prediction has the potential to revolutionize mobile applications. This is evident through a recent wave of commercial applications such as MindMeld [22], GoogleNow [23], Yahoo Aviate [24], Microsoft Cortana [25], Alexa [26] and Siri [27]. These applications harness data obtained from different sources such as mobile sensors, emails, Web navigation history, and calendar entries, to provide information relevant for immediate-future context of a user. These predictive applications can, for example, pop-up a Wikipedia article which is about to be relevant for a point raised in a discussion, check a user in to a pending flight, or provide the duration of a commute before being explicitly asked by the user. Yet, we believe that these are just forerunners to truly anticipatory applications that will act autonomously on the basis of past, present and predicted future context, and modify the future to satisfy users' requirements [28]. Such applications will be able to suggest alternative routes to a driver who is about to experience congestion, will provide well-being coaching to prevent behavior-related health issues, and will organize our busy days to minimize the predicted stress levels [29].

However, realizing anticipatory applications relying on predicted context is challenging due to the intricacy of implementing prediction logic on a mobile device. Such logic, necessary for any context-aware application, includes sensor sampling, managing multiple streams of sensed data, labeling sensor data, and building machine learning models that can be queried in order to infer high-level context of a user. Yet, in the context prediction case, the prediction logic needs encompass machine learning models that describe *the evolution* of the sensed context, as well as further probing to ensure that the models are capturing possible irregularities in user behavior. Moreover, predictive applications need to overcome specific challenges of the mobile platform, for example, they need to maintain the prediction

- *A. Mehrotra is with the Department of Geography, University College London, UK.*
  E-mail: a.mehrotra@ucl.ac.uk
- *V. Pejovic is with the Faculty of Computing and Information Science, University of Ljubljana, Slovenia.*
  E-mail: veljko.pejovic@fri.uni-lj.si
- *M. Musolesi is with the Department of Geography, University College London, UK.*
  E-mail: m.musolesi@ucl.ac.uk

models and the information about the prediction request across the device and the OS shutdowns. Finally, advanced anticipatory applications might be interested in predictions of a context modality (i.e., context types, such as location, movement and network connectivity, to name a few), only when the future state of some other modality satisfies a certain condition (e.g., know the application that the user will launch, the next time her Internet connectivity is down), something that conventional mobile sensing applications do not have to deal with. Therefore, compared to the standard sense-and-classify apps, anticipatory context-aware applications can be significantly more challenging to implement, as the developer needs to master all of the above aspects. A general purpose platform with an easy-to-use API that hides the intricacy of implementing prediction logic is poised to unlock the true potential of context prediction, speeding up the process of building and deploying predictive mobile applications.

To address these challenges we present FutureWare – a middleware that lifts the burden of context prediction from application developers. FutureWare comes with a publish-subscribe interface that offers asynchronous subscriptions to a one-time or a stream of predicted future contexts. The middleware exposes an expressive and flexible API to abstract the means of mobile sensing, individual and group behavior modeling, and future context querying, and lets the developers concentrate on high level functionalities of mobile applications that rely on context prediction. FutureWare is designed with the restrictions of mobile devices in mind, thus the middleware has a minimal data storage and memory footprint, minimizes energy consumption during sensing and modeling, keeps the data transmission burden low, and ensures privacy of mobile users.

FutureWare is composed of two parts: a mobile middleware residing on smartphones, and a centralized server component necessary for group-based predictions. We implement the mobile middleware as an Android library and the server component as an executable Java archive. Integration with existing mobile sensing tools lets FutureWare sense a range of different modalities, including, but not limited to, location, sound, light, and physical movement.

To evaluate the middleware, we first provide several examples that demonstrate the expressivity of the programming interface of FutureWare in mobile and Internet of Things (IoT) settings. Second, we analyze the features of a variety of existing applications that could be built through FutureWare. In our analysis we show that the FutureWare API has the potential of providing support for all the fundamental predictive functionalities of the apps taken into consideration. Finally, we discuss a real world case study, the design and implementation of a predictive application, namely an intelligent lock screen that predicts applications that a user is about to request, and enables quick launching of these applications through direct placing of selected application icons on the lock screen. We gathered and analyzed usage logs from the lock screen applications run by nine users over ten days. The results show that all the prediction techniques outperform the random selection-based application icon placement strategy. We implemented the lock screen application both with and without using the FutureWare middleware in order to evaluate the programming effort needed to develop such an application. The code analysis of these applications shows that by using FutureWare, the same predictive functionality can be implemented with a drastically reduced coding effort.

Specific contributions that FutureWare brings to the field of predictive mobile computing can be summarized as follows:

1) **Expressive and extensible API for context prediction.** FutureWare provides an expressive API that lets a developer configure the parameters of context prediction: types of sensors that will be sampled, how often should the sampling happen, machine learning tools upon which prediction models will be based, and many other configuration parameters. Furthermore, FutureWare API asynchronously accepts the input provided by the overlying application about the correctness of the latest prediction (i.e., the user's feedback), and this information is used in the next model training iteration. Finally, the extensible API of FutureWare enables easy integration of our middleware with third-party machine learning models.
2) **Filtered context prediction querying.** Querying FutureWare for the upcoming state of the context can be done through topic-based and content-based subscriptions to the middleware. In a topic-based subscription the subscriber (i.e., the overlying application) can specify the modalities (e.g. user's mobility, physical activity, etc.) of interest for prediction, whereas, a content-based subscription allows the subscriber to specify the modalities of interest as well as the conditions under which the subscriber is to be notified about the predicted context (e.g., a user's predicted location when the user is running).
3) **Multi-user context predictions.** FutureWare supports group-based predictions. A single user-based predictions involve only the individuals for whom the prediction is made and can be computed on each user's mobile. By contrast, in the case of group-based predictions, individual predictions for the users belonging to a group are collected and aggregated on the server in order to make the final prediction. Here, individual predictions for all users can either be made locally and the prediction results are transmitted to the server. Alternatively, the server maintains models of these users to make their predictions whenever required. We note, however, that building predictive models to learn collective behavior of users is orthogonal to the scope of FutureWare's group-prediction mechanism. In other words, FutureWare does not exploit the data of multiple users to train a single group-prediction models for making predictions about the behavior of these users. Instead, FutureWare's group-prediction mechanism exploits the predictions obtained by means of individual-based models of a group of users separately and merge them in order to make predictions about generic (common) behavioral patterns of the users.

To clarify the scope of our work, we emphasize that the focus of FutureWare is the provision of highly expressive programming abstractions for making context predictions. The main contribution is not on the implementation of the

forecasting algorithms themselves or a predictive application, but on the design, implementation and evaluation of the expressiveness of the framework to support seamless development of next generation anticipatory mobile and Internet of Things applications.

2 Related Work

Several middleware systems were proposed in order to relieve the developers from the burden of interacting with and managing low-level sensors, delivering energy-efficient sensing on battery constrained mobile phones, and ensuring the privacy compliance. Energy Efficient Mobile Sensing System (EEMSS) [30], Jigsaw [31], AnonySense [32], Pogo [33], PRISM [34], Aware [35], and SenSocial [3] are examples of such middleware. All of the above projects address the challenges of mobile sensing for the recognition of current context. These projects are orthogonal to FutureWare; our middleware focuses on supporting context prediction by using the knowledge of the past contextual information.

A few existing works have proposed middleware frameworks for mining user behavior by using longitudinal context data [36], [37], [38]. The Acquisitional Context Engine (ACE) [36] is an example of such a system that maintains a knowledge base of the contextual events on the server and mines the co-occurrence patterns among these context events to sense the users context in an energy-efficient manner. Another example is the MobileMiner [37] that proposed the idea of on-device mining of mobile user's frequent co-occurrence patterns to predict which context events frequently occur together. Similarly, PrefMiner [38] exploits the contextual information and notification data in order to discover the patterns of users' interaction with mobile notifications. These frameworks are built for specific predicting problem. FutureWare, on the other hand, has no such limitations and it is designed to provide an API for enabling seamless development of mobile apps that rely on context predictions. Moreover, unlike previous approaches that are based on some specific machine learning algorithms, FutureWare enables the integration of third-party machine learning algorithms that allows the developers to use any prediction techniques while building an application overlying on the FutureWare middleware. To the best of our knowledge, FutureWare is the first middleware that abstracts the intricacy of implementing sensing and predictive features in a mobile app. Furthermore, to hide the complexity of managing the sensing and prediction streams, FutureWare uses a publish-subscribe interface in order to offer asynchronous communication with the middleware.

A number of commercial applications have been built by using anticipatory mobile computing techniques [39]. Applications such as MindMeld [22], GoogleNow [23], Yahoo Aviate [24] and Microsoft Cortana [25] exploit users' personal data in order to forecast the context and provide context-aware services beforehand. All of the above applications infer the cues of current and future events from various sources, such as microphone, calendar entries, emails, search queries and many others, in order to provide relevant information to the user, however they do not provide a generic API for predicting future events and context. To address this, FutureWare enables further exploitation of the collected data to predict the future context of the user. Moreover, FutureWare can be used not only as a source of personal predictions, but also to make group predictions.

The above listed efforts in providing mobile sensing and machine learning support to application developers, as well as a flurry of commercial applications relying on context prediction, point to the gap that our work aims to fill: FutureWare provides support throughout the whole process, from sampling and storing the sensed data, over machine learning modeling, to managing and utilizing predictions for anticipatory mobile computing.

3 FutureWare At A Glance

The FutureWare middleware is distributed over two components, one residing on mobiles to support individual-based predictions and the other on a centralized server to make group-based predictions. Figure 1 presents the high-level architecture and the flow of information in our system. Once an application subscribes to FutureWare, a *Predictor* (i.e., an abstraction that facilitates construction and querying of the context evolution models) is initialized with the configuration specified by the overlying application. The Predictor is also revised as new sensor data comes in, to ensure a high accuracy of predictions. If a subscription requires predictions pertaining to more than one client, a group prediction query is sent to the server that in turn communicates with the relevant clients by sending a remote prediction query. These clients return the predicted data that is aggregated by the server to forecast the context pertaining to the group. Once the prediction[1] (individual or group-based) is made, the predicted data is handed over to the filter that checks if the optional developer-defined conditions are satisfied, before the prediction result is sent to the overlying application. The filtering conditions are specified while making a subscription, and ensure that only relevant predictions are forwarded to the application.

To illustrate the potential of FutureWare we consider two possible scenarios in which application developers can benefit from the abstractions provided by our middleware.
Scenario 1. Alice uses Internet-based mobile applications while she commutes to work by train, yet faces patchy connectivity on the route. To deal with this poor network connectivity, she runs PreFetcher that predicts when she will leave the connectivity range and which applications she is likely to use thereafter in order to prefetch the content of these applications before the predicted time of disconnection. PreFetcher periodically samples Alice's context, and learns her mobility and network connectivity patterns; then it predicts if she is going to leave the connectivity range in the next forecast period or not. If PreFetcher predicts that she will be out of coverage, it infers the applications Alice is going to use and prefetches all the related application data before she leaves the network coverage area.
Scenario 2. Alice was not able to attend Bob's birthday party because she was not in town. She wants to give Bob a present next time they meet. Therefore, she sets a predictive

1. We use the term prediction here for indicating the computation of probabilities for future events, but as a limit case, these might refer to current events, as in traditional publish-subscribe systems.

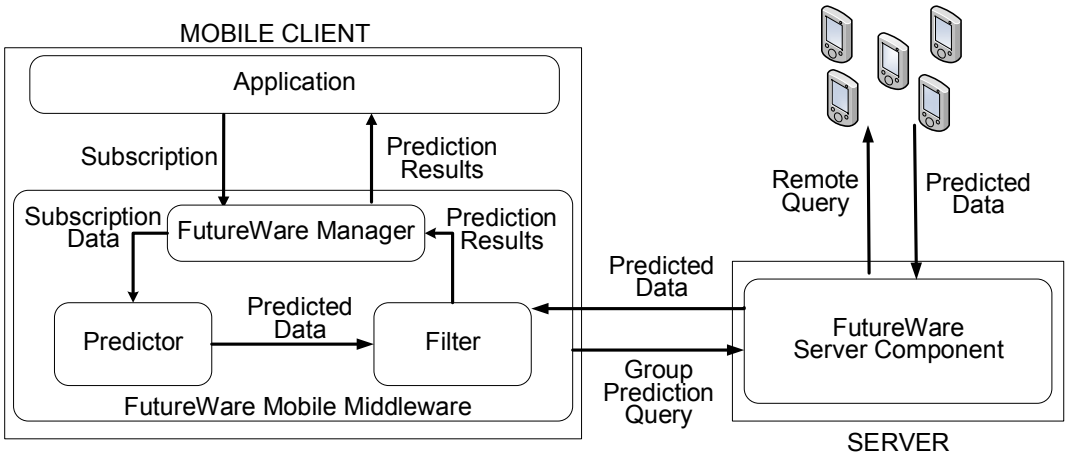

Fig. 1. **FutureWare architecture overview.** FutureWare is distributed over a server and participating mobiles. The mobile middleware exploits the user's contextual information to make predictions. The server-side of the middleware remotely aggregates predictions concerning the context of multiple users to derive group forecasting. The overlying application can subscribe to either individual or group-related future context information.

alarm for reminding her to bring the present before meeting Bob. The application subscribes to FutureWare in order to get notified an hour in advance of the next predicted encounter. FutureWare predicts the time for next encounter based on the mobility patterns of both Alice and Bob, and triggers the application an hour before the predicted time. On receiving this trigger, the application triggers a notification on Alice's mobile that reminds her to take a present for Bob, if the meeting is about to happen in the next hour. In case Alice snoozes the alarm, the application requests that FutureWare makes another prediction for the encounter. Crucially, this mechanism works only if Bob agrees to use the same encounter predictor application.

The implementation of these scenarios with the help of FutureWare API is discussed in Section 7.

FutureWare middleware enables seamless development of the anticipatory functionalities for their apps by hiding the complexity of constructing and maintaining predictive models as well as by facilitating them with a set of APIs to build sophisticated logics for their anticipatory apps. On the other hand, since the interfacing with other services (such online social networks) could be subject to the type of application and depend on the developers choice, FutureWare does not include any API for implementation of such features.

4 Design of Key Abstractions

The middleware design principles focus mainly on the following aspects: (i) abstracting the complexity of the implemented features; (ii) providing flexibility for extending the platform; (iii) enabling the customization of predictions; (iv) exposing intuitive and easy to use APIs.

4.1 Predictor

This is the key abstraction of the middleware that is provided for constructing and querying behavioral models. Behavioral models are built from sensed context data [40], and querying a model provides predictions of a user's future context/behavior. Developers can access instances of the the `Predictor` class according to a publish-subscribe paradigm through the `FutureWareManager` class. Such abstractions let middleware internally manage multiple context prediction subscriptions. Therefore, at the time of binding with the middleware, an overlying application can define the requirements of the context prediction it needs through a set of subscriptions by means of the `FutureWareManager` class. The middleware will then automatically instantiate a new predictor, or bind an existing one, for each subscription as needed. More specifically, in case the app has created a new subscription that requires a predictive model with identical context modalities as another existing subscription, the middleware will bind this new subscription to the existing predictor rather than creating a duplicate one. However, since the middleware is implemented as a library (discussed in Section 6), the predictors are not shared across applications. This also ensures that the middleware complies with the privacy of users' data based on the permission they have given to specific apps. We discuss the details of the subscription mechanism in the next section.

To construct a behavioral model, the predictor uses the historical data (collected by the middleware or provided by the overlying app) as specified in the subscription of that predictor. It is worth noting that the overlying app must already have the permission from the users to collect the data that is required for making the prediction. In order to construct a model, the `generateDataModel` function (of `Predictor` class) is invoked by passing the historical data (that is collected and stored in the phone's memory by that middleware) along with the labels of each data point.

Types of Predictions

In general, anticipatory apps make two types of predictions: (i) the time when the user will be in a given state (i.e., context states); (ii) the state at a given time in future. In order to perform the former, the instance of the Predictor class invokes `predictionRequest` function by passing

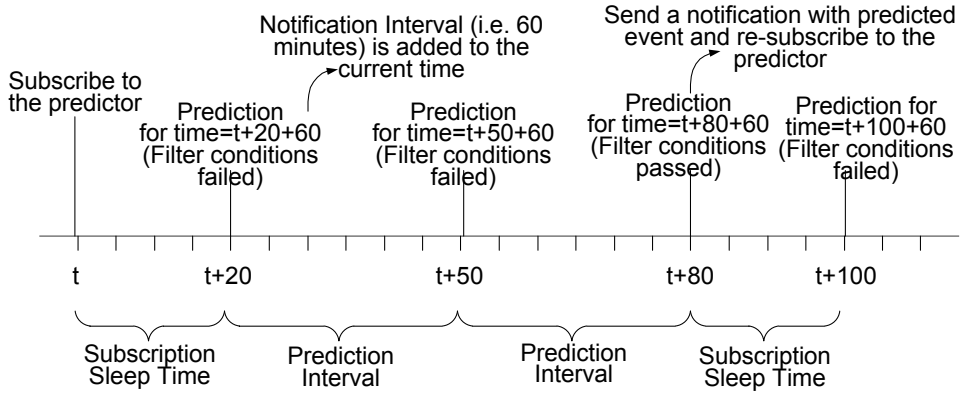

Fig. 2. Timeline for a prediction configuration object. Here, subscription sleep time is 20 minutes, notification interval is 60 minutes, and prediction interval is 30 minutes. Subscription sleep time is the delay before the predictor is subscribed to start making predictions, notification interval is the future time for which the context is to be predicted, and prediction interval is the time gap between two consecutive predictions.

the states that are to be predicted along with the current context state as arguments. Whereas, in the case of latter, the `predictionRequest` function is invoked by providing the time horizon after which user's context has to be predicted along with the current context state. In order to facilitate such predictions, FutureWare comes with a set of built-in machine learning tools, such as frequency-based and Markov Chain algorithms, which are used to compute the probability of future occurrence of a class in a given time interval. The flexible API of FutureWare enables easy integration of our middleware with third-party machine learning tools that suit the application requirements. This can be done simply by providing a class path of an external machine learning algorithm through the *Prediction Configuration Object* (discussed later in this section) while making a subscription. The external prediction algorithm must implement the `PredictorInterface` interface, override all the methods of the interface which are internally used by the middleware for making predictions, and have a nullary constructor with "public" access modifier so that the middleware can use reflection to instantiate it.

Training and Storing Prediction Models

The Predictor resides both on the mobile middleware and the server. The mobile side Predictor is in charge of individual-based predictions and the server side Predictor supports group-based predictions. Once an application subscribes to FutureWare, by default the client-side Predictor builds the behavioral models of the user and stores them locally in the phone's memory. However, the developer can also choose to build the models on the server in order to avoid heavy computations on the client. This can be done by simply enabling the transmission of data to the server by invoking `enableDataTransmissionToServer` function from `PrivacyManager` class. In such a case, the mobile middleware sends the context information collected since the last model training to the server and requests model retraining. Once the freshly trained model is ready, the server informs the mobile middleware, which then downloads and stores it locally. Hence, FutureWare offers flexibility for processing model training either locally on the client or on the server. FutureWare manages context sensing and labeling, so that behavioral models can be retrained after each context life-cycle time period (i.e., the developer-defined expiry time of the model).

Since the mobile app could be killed by the OS due to resource limitations or by the users themselves, it is important to periodically store the trained prediction models so that the middleware does not have train them from scratch once the app is relaunched. To do so, in FutureWare the models are stored (in the external memory) after every iteration of model training, which allows the middleware always keep the freshly trained model. FutureWare also allows the developer to integrate third-party machine learning algorithm for building the prediction models (discussed in Section 4.2.4). To support the functionality of storing trained models for third-party ML algorithms, FutureWare exposes *storeModel* and *loadModel* that can be implemented to store and load the model in an arbitrary file format.

4.2 Publish-Subscribe Paradigm

The FutureWare Manager is the core component of the mobile middleware and represents a point of entry for the overlying application. We design FutureWare Manager as a publish-subscribe interface to offer asynchronous communication with the middleware. This enables the overlying application to specify its interest, in order to be notified of any predicted event that matches the application's registered interest. The FutureWare Manager exposes the `subscribeOnce` and `subscribe` methods to subscribe for a single event or a stream of future events that are predicted by the middleware. In order to access these methods, an application has to create an object of the `FutureWareManager` class, which is designed as a singleton class to ensure that there is only one instance of the middleware running at a time. To make a subscription of either type (i.e., for a single event or a stream of future events), an application should provide the following arguments: (i) modalities of interest, (ii) listener, (iii) filter, and (iv) prediction configuration object. Details of these arguments are discussed below in this section.

FutureWare allows the overlying application to make multiple subscriptions. On receiving a new subscription,

the FutureWare Manager passes the subscription object to the Event Manager that maintains a list of all subscriptions along with their registered prediction configurations. The Event Manager periodically initiates sensing, classifies the sensed data to obtain high level contextual information and stores this contextual information that is used for re-training models. All classes involved in the process of making a subscription implement standard `Serializable` interface. This allows the middleware to store objects of these classes as string and later deserialize the objects in case the application restarts after getting killed by the OS.

4.2.1 Modalities of interest

Modalities of interest refers to a list of context modalities that the middleware needs to predict for a specific subscription. All supported context modalities (such as location, physical activity, calling behavior and so on) are implemented as separate classes that extend `PredictorData` class. Moreover, for each of these classes, their predefined objects are exposed as static variables of the `PredictorData` type. This allows a single API call for finding all the supported context modalities. For instance, if a subscription requires a prediction of a user's location, it can simply create an empty list, add a single element (i.e., `PredictorData.LOCATION`) to it, and pass it to a newly-created subscription.

4.2.2 Listener

Listener interface contains a method signature for `onNewEvent` through which the middleware notifies the overlying application about the prediction results that match the registered interests. In other words, `FutureWareListener` enables the middleware to support asynchronous communication for a subscription as results can be passed to the subscriber once they are ready. A listener can be any developer made component that implements the `FutureWareListener` interface. A listener can be used to receive notifications for multiple subscriptions that can be classified according to the subscription IDs. Since the mobile OS could kill an application, the middleware could lose all references to these listeners. Therefore, FutureWare stores the canonical names of the registered listeners (i.e., any class that implements `FutureWareListener`) that can be obtained by invoking `getClass` and `getCanonicalName` methods as following: `listener.getClass().getCanonicalName()`.
Once the results are ready, the middleware uses the stored canonical name (let's say `canonical_name`) of the registered listener class to instantiate an object that class through the following Java method call `Class.forName(canonical_name)`. Finally, this listener class object is used to trigger `onNewEvent` for delivering new results.

4.2.3 Filter

The key feature of the middleware is its ability to support both *sensor-based* and *event-based* subscriptions for context prediction. A sensor-based subscription enables an application to specify the modalities of interest. For example, subscription to get periodically notified about the user's location in the subsequent hour. Whereas, an event-based subscription allows an application to define the modalities of interest as well as the filtering conditions under which the application should be notified. Unlike the sensor-based subscriptions for which the middleware publishes all the predicted events for the registered modality, the event-based subscriptions only receive the predicted events that match the registered modality and filter conditions. For example, we can have a subscription that ensures the application is notified an hour in advance of the user's visit to a restaurant.

In order to create an event-based subscription, a filter must be provided at the time of subscription initialization to specify the conditions under which the application should be notified. Whereas, a null value can be passed in the case of a sensor-based subscription as there are no conditions to checked. A filter consists of a set of conditions which can be based on: (i) time interval of a day, (ii) the probability of the prediction, or (iii) a certain value of the context of a user or a group of users. A condition can be written as a Java conditional statement and each condition consists of a variable, a comparison operator, and a target value of the variable. Since there are three types of conditions supported by the FutureWare, therefore, a variable of a condition can be a the time, prediction probability or modality type. These supported values of variables can be accessed through `Variable` class. Similarly, the supported operators and target values relevant to the used variable could be accessed from `Operators` and `Values` classes respectively.

Let us assume that a developer wants to be notified only when the user's predicted location is "home" and the probability of this prediction is greater than fifty percent. Such a filter (f) can be written in the following way: $f = Variable.Sensor.Location + Operators.EqualTo + Values.Location.Home + Operators.And + Variable.PredictionProbability + Operators.Greater + Values.Probability(0.5)$. Alternatively, the filter f can also be defined as: $f =$ "$Location == Home \;\&\&\; Probability > 0.5$". In this example there are two conditions 1) predicted location is home, and 2) the probability of prediction is greater than 50%. Thus, the predicted data for the relevant subscription is sent to the application only if the user's predicted location is `Home` with a probability of 50% minimum.

FutureWare stores all filters in the internal memory as serializable strings. This is done in order to re-instantiate filters for active subscriptions in case the app restarted itself after it is killed by the OS. This way the middleware helps in avoiding the permanent allocation of volatile memory consumed for these filters.

4.2.4 Prediction configuration object

The goal of FutureWare is to streamline the development and at the same time support a wide range of possible anticipatory mobile applications. Expressiveness and easy-to-use primitives are key design requirements of our middleware, particularly, having in mind that a simple prediction of a single context modality (e.g. location), may be insufficient for many applications (that would require joint location and physical activity prediction, for example). Moreover, applications might also require different machine learning

approaches and predictions with varying prediction horizons (i.e. how far in future should the middleware look). In order to increase the flexibility of the platform, FutureWare provides a *Prediction Configuration Object* (implemented by the `PredictionConfig` class), that offers the possibility of configuring the prediction and notification (i.e., information about the newly predicted event) process according to developer's requirements. In addition, the *Prediction Configuration Object* can also be used to manage the trade-off between prediction accuracy and resource consumption: frequent sensing and model querying requires significant battery and CPU resources [41]. *Prediction Configuration Object*, as is, offers a large number of potential configurations that an anticipatory application might need for its predictive functionalities. However, it is flexible enough to enable developers to construct their own prediction architectures.

The *Prediction Configuration Object* is defined through a set of key value pairs where each key refers to a specific setting of the predictor. In order to increase the usability of this API, the middleware expose it in the same way as other standard configuration APIs of Android. In the FutureWare middleware, this abstraction is implemented as $Configuration$ class and the configuration keys are implemented as static variables of this class. The following are the five configuration keys that can be used to customize a prediction task:

(i) **Notification Interval (Prediction Horizon)**: the time interval that defines how far in the future should the model look to make a prediction. It is added to the current time when a prediction is made to obtain the future time for which a prediction is to be made. More specifically, *Notification Interval* can be defined by setting the `NotificationInterval` key to any integer value (i.e., time period in minutes).

(ii) **Prediction Interval**: the time gap between two consecutive predictions (given that the former prediction failed to satisfy the filter conditions). To define this interval, developers can assign any integer value (i.e., time period in minutes) to the `PredictionInterval` key.

(iii) **Predictor Module Selector**: the selection for a built-in prediction algorithm or the absolute path of a third-party machine learning algorithm. Developers can pick a built-in algorithm for building their prediction model through the `PredictorCollection` class, which provides codes for all supported algorithms. One of these values can then be assigned to the `PredictorNameOrPath` key and added to the *Prediction Configuration Object* . When a third-party machine learning algorithm is used, its absolute path can be linked to the `PredictorNameOrPath` key.

(iv) **Subscription Lease Time**: this defines the expiry time for a subscription, which enables developers to create dynamic subscriptions with the predefined lease. This expiry period can be configured by setting the `SubscriptionLeasePeriod` key to any integer value (i.e., time period in minutes).

(v) **Subscription Sleep Time**: the time delay before making the first prediction for a subscription (Subscription Sleep Time is also valid for the re-subscriptions made after every successful prediction). In order to define this interval, developers can assign any integer value (i.e., time period in minutes) to the `PredictionPeriod` key. The motivation for introducing this configuration key is to allow the app to collect enough data for training the model at the first instance. A classic problem in developing predictive mobile apps is the initial training of the predictors: when an app is installed by a user, his/her behavioral data is not usually available to train the prediction model. This key is used to set the interval between the start time of an app and the time of the first training of a prediction model.

Let us consider the prediction requirements summarized in the time line showed in Figure 2. In this specific case the *Prediction Configuration Object* has to be set with the following values: (i) Notification Interval = 60 minutes; (ii) Prediction Interval = 30 minutes; (iii) Predictor Module Selector = `MarkovChain`; (iv) Subscription Lease Time = `SubscriptionTypeContinuous`; (v) Subscription Sleep Time = 20 minutes.

In this example, a subscription (made at time t) requests the predictor to make predictions, for example, about a user's future location, continuously by using the Markov-Chain-based prediction algorithm. As the subscription sleep time is defined as 20 minutes, the middleware waits for that amount of time before subscribing to the predictor for making predictions. At time t+20 the predictor makes a prediction for time t+20+60 (i.e., current time + notification interval). If the prediction results satisfy a given filter, for example, `location == home`, they are passed to the application listener by clearing the filter conditions and the process resets, otherwise the next prediction is made after a time interval provided by prediction interval. Therefore, the second and third predictions are made with an interval of 30 minutes (i.e., defined by prediction interval). However, the third prediction satisfies the filtering conditions and is passed to the application, and thus the subscription cycle resets and the middleware sleeps for the given subscription sleep time (i.e., 20 minutes in this example) before subscribing again to the predictor.

Finally, in to order to provide an efficient context prediction platform, FutureWare uses a single machine learning-based model for multiple subscriptions (belonging to the same application) that have a similar configuration when it comes to context prediction. For this, FutureWare matches the configuration of every new subscription with the configurations of existing subscriptions in order to identify matching subscriptions. For example, two different subscriptions request a user's future location 20 minutes in advance of the location change. In such a case only a single model is constructed by the middleware to support both subscriptions.

5 Interaction between FutureWare Components for Making Predictions

In Figure 3 we present the location of the components within the architecture of FutureWare and types of data that are exchanged among these components. The goal of these components is to efficiently support the abstractions we explained earlier. The middleware exposes a concise

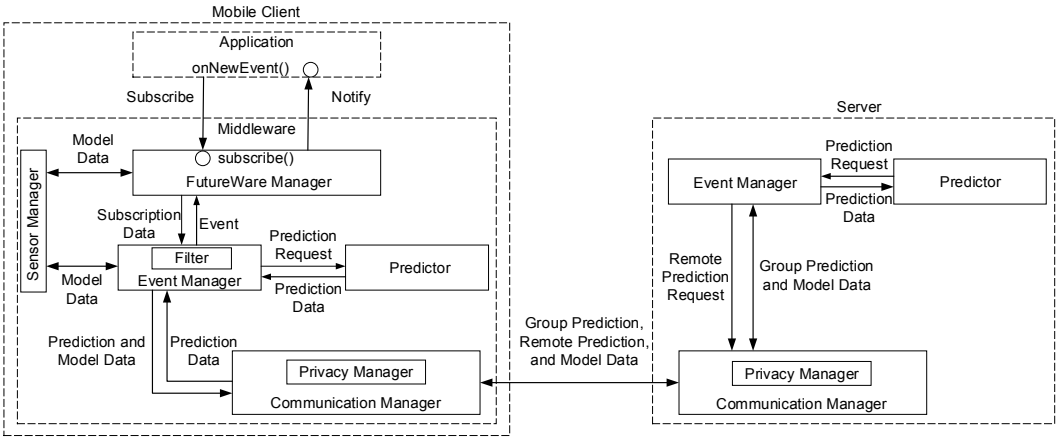

Fig. 3. FutureWare components of mobile client and server-side middleware. The server can control multiple clients; we show a single client for clarity.

Algorithm 1 Pseudo-code for implementing the prediction task presented in Scenario 1 (discussed in Section 3) by using the FutureWare abstractions.

$PC1 \leftarrow$ Prediction Configuration Object with Notification Interval and Prediction Interval as t_{event} and t_{gap} respectively
$filter \leftarrow$ Filter as "$WiFi == Unavailable$"
$sid1 \leftarrow$ subscribe for predicting WiFi with Filter and Prediction Configuration Object as $filter$ and $PC1$ respectively
function ONNEWEVENT($data$)
  if $data.subscriptionId == sid1$ **then**
    $PC2 \leftarrow$ Prediction Configuration Object with Notification Interval as t_{event}
    $sid2 \leftarrow$ subscribe for predicting app usage with Prediction Configuration Object as $PC2$
  end if
  if $data.subscriptionId == sid2$ **then**
    $results \leftarrow$ data.result
  end if
end function

set of API calls that are sufficient for controlling these components.

5.1 Making Individual-based Predictions

In FutureWare context predictions for all the subscriptions are managed by the Event Manager. Based on the prediction horizon (provided as `PredictionInterval` through the `PredictionConfig` class) a timer is set to remind the Event Manager to make context predictions for the relevant subscription. On receiving a trigger to make prediction, the Event Manager fetches the relevant behavioral models from a phone's database and instantiates the Predictor with this model. Once the Predictor is instantiated, the Event Manager instructs the Sensor Manager to get current context data and sends a prediction request to the Predictor for predicting the requested context modality (specified as modalities of interest while making the subscription). Finally, when all conditions provided via subscription filter are satisfied, the Predictor passes the predicted results to the Event Manager, which then asks the FutureWare manager to trigger the results to the subscription's listener.

5.2 Making Group-based Predictions

In order to make group predictions, the Event Manager (client side) transmits a group prediction request to the server component via the Communication Manager. This request contains the user identification codes of a group of users, the modality for which the prediction is to be made, and the future time at which the context is to be predicted or the context modality value that is to be predicted.

On receiving a group prediction request, the Event Manager (server side) obtains the individual-based prediction results for the requested set of users. In order to so, the Event Manager (server side) uses the behavioral models of the users who opt to builds share their contextual information and build models on server. For the users who opt not to share the raw context data with the server client, the server component sends a remote prediction request to the mobile client of the corresponding users and obtains the prediction results of the requested query. On receiving the prediction results of all users specified in the group-prediction query, the Event Manager (server side) merges them to make predictions about generic (common) behavioral patterns of these users. Finally, the predicted result is transmitted back to the mobile client of the user who generated the request.

FutureWare does not exploit the data of multiple users to train a single group-prediction model for making predictions about the behavior of these users. Instead, FutureWare's group-prediction mechanism exploits the predictions obtained by means of individual-based models of a group of users separately and merges them in order

to make predictions about generic (common) behavioral patterns of the users.

Ensuring Privacy Compliance
Since group-based predictions deal with sharing individual's predicted context information with others, FutureWare exposes API calls for the developers to define the application's privacy policies. These can be dynamically defined by the developer or exposed as settings to give users the control over their data sharing policies. These policies can be defined by specifying the sensor modalities and friends' ids through the `allowPredictorAccess` and `givePredictionModelReadAccessToFriends` functions of the `PrivacyManager` class, and then call `commitChanges` function to add these policies to the Privacy Manager (on both mobile and server). For instance, a user can define her privacy policies such that a set of Facebook friends can be permitted to use predicted location and accelerometer data, whereas, another set of users connected on Foursquare can be allowed to access only the predicted location data.

Once a group prediction request is received, the Privacy Manager (server side) screens this request to confirm that it complies with the privacy policies of all the relevant users. In other words, it checks if all the users included in the group prediction have granted permission to share their prediction data of the requested sensor with the requester. To support dynamic handling of privacy policies, every time a user changes their privacy policies, the Privacy Manager (server side) is notified to update them. This ensures that the group prediction could not be made if any user in the group who was initially sharing the prediction results data with the requester has later decided to revoked the permission to share their data, or the vice versa.

Finally, on successfully passing the privacy screening, a group prediction request is transferred to the Event Manager (server side), that makes the group-prediction by obtaining and merging prediction results of individuals through the models of users available on the server or by sending remote prediction requests (as discussed earlier in this section). Note that the requests that fail the privacy screening are notified about the failure for privacy check (without sharing the users for which it failed) and are stored in the *Group Prediction Stack*. This allows the mobile client to either cancel the group-prediction subscription or leave it until the privacy checks are cleared. In the case of latter, the group-prediction requests in the *Group Prediction Stack* are screened again when user's privacy policies are modified.

To better understand the process of ensuring privacy compliance, let us consider an example by assuming that Alice has made a request to predict her next encounter with Bob. In order to make this prediction, the middleware requires that Bob has already allowed to share his location prediction data with Alice. If this was not permitted, the middleware would not allow Alice's mobile client to make the requested prediction. Similarly, if Bob has initially allowed to share his predicted location with Alice but later on he decides to revoke this location sharing policy, Alice would not be allowed to make any more predictions about her next encounter with Bob.

6 Implementation

The FutureWare mobile middleware is implemented as an Android Java library and the server component consists of an executable Java library. All the manager components – FutureWare, Event, Privacy Policy and Communication Managers – are implemented as singleton Java classes to secure the uniqueness of the global state. As far as the implementation of the mobile middleware is concerned, we follow the best practice of Android programming and ensure that heavy processing tasks are performed on their individual background threads. FutureWare is released as an open source project[2].

Sensing and Classification. The middleware builds a knowledge base about the user's contextual information by streaming sensor data through the Sensor Manager. To implement the Sensor Manager, the FutureWare includes its own sensing library capable of sampling data from numerous sensors including: location, physical activity, environment sound, network connectivity, Bluetooth environment, application usage, phone call, SMS logs, notifications and screen events. The library is designed for adaptive sensing mechanism that samples sensor data only when there is significant change in their values. Such a mechanism reduces the consumption of energy and, thus, the app has low impact on battery life. The library has been used and tested in multiple apps developed for different studies [38], [42], [43].

The FutureWare Manager exposes the API calls to define the sampling cycle rate (i.e., time window after which a sensor is sampled) of a sensor in a key-value object. The developers can adjust the sensing rate to manage the trade-off between accuracy and energy consumption, as high sampling rate consumes more battery but reduces the chance to miss important context information and vice versa. The current version of FutureWare also provides a k-Nearest Neighbors classifier to group the raw sensed data into higher level context classes (with additional classifiers planned for future releases). Note that this classifier is used to label the raw sensor data, but not for making predictions. Additionally, the middleware relies on the Google's Activity Recognition library to obtain higher level information about the user's physical activity [1].

Mobile operating system vendors have started centralizing control over the mobile's resources. This not only saves phone's resources by prohibiting individual applications from overusing the phone's resources, but also coordinates the needs of multiple applications – the same reading can be served to multiple applications, if the data "freshness" requirements are met. Therefore, instead of device-wide mobile sensing, FutureWare supports per-application sensing because this allows for much cleaner privacy and permission management. We also underline the fact that predictors might not be shared since different models are trained for each user.

Data Storage. The mobile middleware stores the user's temporal contextual information (context data at different

2. https://github.com/AbhinavMehrotra/FutureWareMiddleware

times) and the behavioral models, encapsulated as JSON-formatted strings, locally in the SQLite database. To maintain the updated behavioral models, the mobile middleware replaces the old models with the newly generated models after each context life-cycle period. The information about the user's privacy policies and the list of subscriptions are stored in an external storage as the serialized strings of their object. These strings are de-serialized and used by the FutureWare Manager to re-instantiate the state of the mobile middleware when the overlying application restarts. This ensures that the persistence of the models is preserved even if the application is abruptly terminated by the Android OS.

The user's temporal contextual information and privacy policies are transmitted to the server if the user allows sharing. The server component uses a MongoDB database [44] to store this information encapsulated as JSON-formatted strings. To maintain the updated information, the mobile middleware transmits an updated privacy policy to the server every time it is modified on the client. On the other hand, users' temporal contextual information is transmitted periodically after a defined context life-cycle time period. The context life-cycle can be set through the API calls exposed by the Sensor Manager. All the group prediction requests that fail to comply with privacy policies are stored in the database as serialized objects and once the privacy policies are modified, these requests are de-serialized and go through a privacy check again.

Server Client Communication. The possibility of making a group prediction in FutureWare is enabled by a persistent communication link between a mobile client and a centralized server component. FutureWare implements a Mosquitto broker [45] on the server side and each client runs a MQTT client service in the background. To transmit a group prediction request, the mobile middleware creates a prediction query and publishes it through the MQTT service. The published requests are received by the server component via the Mosquitto broker. FutureWare uses MQTT over HTTP protocols due to the fact that MQTT is based on the push paradigm, thus, unlike HTTP-based solutions, does not require continuous polling from the mobile side, resulting in a lower battery consumption. For example, in the case of group-based prediction request, the mobile client does not continuously poll the server to check if the prediction results are available, instead the server transmits the results once they are computed.

7 Evaluation

In this section we evaluate the performance of the FutureWare middleware by focusing on the aspects related to memory efficiency, energy management, scalability and API expressivity.

7.1 Source Code and Memory Footprint

We evaluate FutureWare on a LG Nexus 5 phone with 2 GB of RAM, and a quad-core 2.3 GHz Krait 400 CPU, running a clean slate Android 5 (Lollipop) operating system. We use third-party measurement tools, namely Count Lines of Code (CLOC) [46], and Android Dalvik Debug Monitor Server (DDMS) [47] to quantify the middleware performance. We use CLOC to obtain code count statistics. The Android-based mobile middleware comprises of 85 Java classes containing 7052 lines of code, while the server component is made of 28 Java classes and 1734 lines of code.

TABLE 1
Memory footprint for sample FutureWare applications.

Application	Heap-size allowed (MB)	Heap-size allocated (MB)	Objects
Stub	16.847	16.243	46872
SingleMod	17.453	16.585	51951
MultiMod	17.496	16.792	52175

TABLE 2
Memory required to store the sensor data obtained per sensing cycle.

Sensor	Memory consumed (Bytes)
GPS	58
Physical Activity	37
Microphone	48
Bluetooth	40
WiFi	40
App Usage	53
Call	52
SMS	52

In the Android OS, paused applications are retained in the phone's memory so that the OS can maintain the state of these application instances. However, as discussed in the documentation of Android's activity lifecycle [48], the OS could kill the application process (which destroys not only the activity but everything else running in the process, including the sensing of contextual information) to release the memory allocated to it when another higher priority application requests additional memory that is not available at the moment. The likelihood of the OS killing a certain process depends on the state of the process at that time, which in turn, depends on the state of the activity running in the process. Applications with activities in the background state are most likely to be killed, whereas, applications with foreground activities are least likely to be killed by the OS[3]. Consequently, if an application uses a large amount of memory, it may cause other background applications to be killed. Thus, it is essential for an application to occupy as little memory space as possible.

We evaluate the memory footprint of three test applications: a first simple stub application (*Stub*) that invokes FutureWare without making any subscription, a second one (*SingleMod*) that subscribes to FutureWare for a continuous prediction of a single modality – location – and a third one (*MultiMod*) that subscribes to FutureWare for predictions of multiple modalities – WiFi connectivity, communication patterns (calls and SMSs), applications currently in use, current activity (from the Google Activity Recognition API) and location.

The memory footprint obtained via the Android DDMS tool is shown in Table 1. Compared to *SingleMod*, the fully

3. The documentation of activity state and ejection from memory [49] provides detailed information on the relationship between state and vulnerability to ejection.

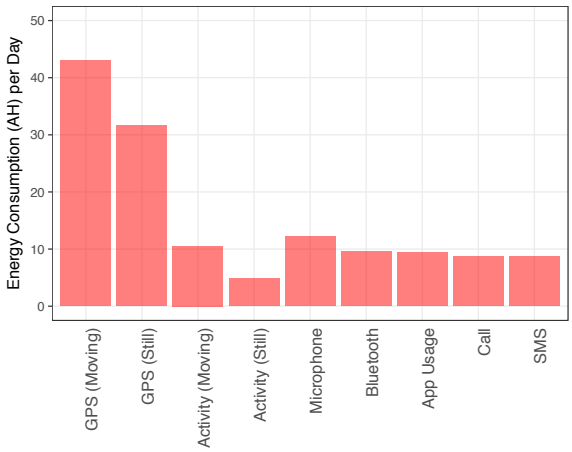

Fig. 4. Average battery charge consumed per day by FutureWare for different sensors.

TABLE 3
Battery charge consumed for transmitting sensor data (obtained per sensing cycle) to the server.

Sensor	Energy consumed (mAh)
GPS	2.5
Physical Activity	1.5
Microphone	2.0
Bluetooth	1.7
WiFi	1.7
App Usage	2.1
Call	2.1
SMS	2.1

loaded *MultiMod* uses only 207 KB of extra memory to provide a much broader set of context predictions.

Additionally, we also evaluate the memory requirements for storing the data obtained through each sensor. Since the sampling rate can be set by the developers depending on the requirements of their application, we compute the memory required to store the sensor data obtained per sensing cycle. As shown in Table 2, FutureWare requires 40-58 Bytes of memory to store the sensor data. This demonstrates that the maximum storage required (when all sensors are sampled at 1 min intervals) for daily sensor data ranges from 57.6 to 83.5 KB.

7.2 Energy Management

One of the key challenges for mobile sensing-based applications is to optimize energy consumption. Continuous sensing could have an adverse impact on the phone's battery life. Moreover, the energy consumption by an app depends not only on the type of sensor, but also on the sampling duty cycle of these sensors (i.e., interval at which sensor values are sampled). In FutureWare, we use an adaptive sampling technique in which the duty cycle is not constant but it relies on the previously sampled values. Details of the implementation of this sampling technique are discussed in [50]. In particular, we use this is used for location sensing as continuous sensing of GPS may lead to a twenty-fold reduction in the battery lifetime [51] compared to no GPS sampling. Similarly, for physical activity sensing we rely on the Google's Activity Recognition API that uses the same technique internally [1].

We now evaluate the energy requirements of FutureWare for primary key task of mobile sensing. In order to perform this evaluation, we develop an application with a background service instance that samples sensors and write the data to a file. We investigate energy requirements of each of the sensing modalities supported by FutureWare: GPS location, physical activity, microphone, Bluetooth, app usage, call and SMS logs. We performed 1 hour experiments in which sensing is separately performed every 60 seconds for each sensor. Since activity and location sensing is performed with an adaptive sensing technique, we evaluate the energy consumption of these sensors in two scenarios: (i) when the user is continuously moving; and (ii) when the user is not moving at all. We chose these scenarios in order to get the maximum (by the former) and the minimum (by the latter) energy consumption values. The energy profiling was carried out using PowerTutor [52].

Figure 4 shows the energy consumed by FutureWare for different sensors. As expected, different sensor modalities are characterized by remarkably different energy costs. The GPS sensor consumes the most (i.e., 42 mAh) and the minimum energy is consumed by the activity sensor (i.e., 4 mAh). This indicates that the energy cost of the FutureWare middleware ranges from 0.17% to 1.82% considering an average phone that comes with 2300 mAh battery.

Often, however, GPS readings are shared at operating system level across multiple applications and, therefore, the energy consumption associated to this sensor is actually lower in practice. At the same time, we observe there is negligible difference in the energy consumption for sampling app usage, call and SMS logs. This is due to the fact that there are no sensors for sampling these values, instead their values are sampled from the logs of the operating system with a negligible energy use overhead.

Furthermore, we also evaluate the energy consumed for transmitting sensor data to the server, which is needed for the case of group-based predictions. As discussed above, we compute the energy consumed for transmitting sensor data obtained per sensing cycle because the sampling rate can vary depending on the application requirements. As shown in Table 3, FutureWare consumes 1.7-2.5 mAh for transmission of data obtained per sensing cycle. This demonstrates that over a day 2.5-3.6 Ah of battery charge is consumed to transmit sensor data to the server. In other words, FutureWare consumes 0.07-0.1% of battery of an average phone that comes with 2300 mAh battery.

7.3 System Scalability

In this section we discuss the scalability of our middleware with respect to the number of active prediction subscriptions. Increasing a number of subscriptions initiated by an overlying application may influence the CPU load as it would require additional computations. We measure the impact on CPU load by increasing the number of subscriptions from 1 to 10.

As shown in Figure 5, the CPU load grows linearly as the number of subscription increases. More specifically,

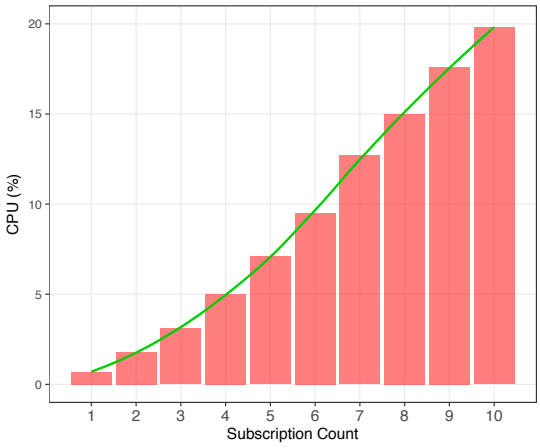

Fig. 5. CPU load for the prediction task with increasing number of subscription.

```
PredictionConfig pc1 = new PredictionConfig();
pc1.put(NotificationInterval, t_event);
pc1.put(PredictionInterval, t_gap);
PredictionConfig pc2 = new PredictionConfig();
pc2.put(NotificationInterval, t_event);
ModalitiesOfInterest  wifi =
     ModalitiesOfInterest.WiFi;
ModalitiesOfInterest  app =
     ModalitiesOfInterest.App;
Filter filter = new Filter("WiFi ==
     Unavailable");
int sid1 = subscribe(wifi, filter, pc1, this);
onNewEvent(PredictionData data) {
  if(data.getSubscriptionId() == sid1)
    int sid2 = subscribeOnce(app, null, pc2,
     this);
  else {
    for(PredictionResult pr :
     data.getResults()) {
      String app = pr.getState();
      /* Prefetch data for this app */
    }
  }
}
```

Fig. 6. A code snippet for making one-time and periodic predictions using the primitives provided by FutureWare.

to perform a prediction task, we observe a CPU load of around 1% for a single subscription and around 20% for 10 subscriptions. Here, the CPU load is computed only for the prediction task, which includes loading the model and predicting the output, without considering the impact of sensing. The measurements of CPU load was also carried out using PowerTutor.

The results demonstrate that a simple anticipatory mobile app that requires two or three predictive models add a significantly low amount of additional CPU load and, thus, it should not hinder the overall users' mobile experience. However, in the case of a sophisticated anticipatory mobile app that is based on multiple predictive functionalities, as the number of subscriptions increases, the CPU requirements of FutureWare also increases. In such cases, FutureWare's *Prediction Configuration Object* can be used to configure the values of the prediction interval and sleep time (discussed in Section 4) to lower down the CPU load.

7.4 Conceptual Apps Demonstrating API Usability

In this subsection, in order to evaluate expressiveness of the FutureWare API, we perform scenario-based usability testing frequently used by the API designers [53], [54]. We present a series of novel anticipatory applications based on different predictive scenarios and their implementation by using FutureWare API. Since the scope of this work is to evaluate expressiveness of the FutureWare API, we present the code for implementing the required predictive functionalities. The evaluation of the usability of these applications is indeed an interesting aspect to be investigated, but it is outside the scope of this article.

7.4.1 Making One-time and Periodic Predictions

We first demonstrate how an application developer can use FutureWare for implementing applications that rely on both one-time and periodic predictions. Such a prediction is useful in Scenario 1 discussed in Section 3. Here, two modalities (i.e., network connectivity failure and application usage) are predicted. However, to prefetch applications before network disconnection, we need to perform network connectivity prediction continuously and predict the usage of applications in the near future only when a network disconnection is predicted. Figure 6 presents a code snippet that shows how it is possible to implement the key prediction mechanisms by relying on FutureWare. `PredictionConfig` is used to specify the prediction settings. Then, in order to define the time horizon for disconnection prediction, we set `NotificationInterval` equal to `t_event`. The time gap between two consecutive predictions is defined by `PredictionInterval`. In the example, it is set to `t_gap`. Time `t_gap` and `t_event` can be set according to the required trade-off between resource (CPU and battery) utilization and prediction accuracy. We then create a subscription `sid1` to periodically predict WiFi disconnections. Finally, once we predict that the WiFi will not be available in the next `t_event`, we make another one-off prediction `sid2` to retrieve the applications used in the next hour.

7.4.2 Making Group-based Predictions

In this scenario we show how an application developer can use FutureWare to implement applications that rely on group-based prediction of the future occurrence time of a given event. Such a prediction scenario is useful for implementing the application described in Scenario 2 discussed above. In this case the event to be predicted Alice's next encounter with Bob.

Figure 7 presents a code snippet that shows how we can implement the required predictive functionalities with FutureWare. We use `PredictionConfig` to specify the required prediction settings. As we need to predict the location of Alice and Bob in the next hour, we set the time horizon (i.e., `NotificationInterval`) to 60. Then, in order to specify the time gap between two consecutive predictions we set `PredictionInterval` as `t_gap`. We create two filters: (i) `f1`: to make predictions for Bob's future

```
PredictionConfig pc = new PredictionConfig();
pc.put(NotificationInterval, 60);
pc.put(PredictionInterval, t_gap);
ModalitiesOfInterest  moi =
     ModalitiesOfInterest.Location;
Filter f1 = new Filter("User == " +
     friend_osn_id + " Time > " +
     Time.getCurrentTime());
Filter f2 = new Filter("Time > " +
     Time.getCurrentTime());
int sid1 = subscribe(moi, f1, pc, this);
int sid2 = subscribe(moi, f2, pc, this);
PredictionResult pr1 = null;
PredictionResult pr2 = null;
onNewEvent(PredictionData data) {
  if(data.getSubscriptionId() == sid1)
    pr1 = data. getMostProbableResult();
  else
    pr2 = data. getMostProbableResult();
  if(pr1 != null && pr2 != null &&
     pr1.getState() == pr2.getState() &&
     pr1.getTime().getHour() ==
     pr2.getTime().getHour())
  /* Trigger an alarm at
     pr1.getTime().getHour()-1 hours for next
     encounter. */
}
```

Fig. 7. A code snippet for the prediction of the next encounter with a friend, implemented using FutureWare. Filter f1 ensures the location of the person with an OSN identifier as "$friend\_osn\_id$" is predicted, while filter f2 is used to predict the location of the user running the app (i.e., owner of the device). Later, the stream of predicted locations of these two users are compared until a match (i.e., their encounter) is found.

locations (assuming his identifier is $friend\_osn\_id$); (ii) `f2`: to make predictions for Alice's future locations.

We create two subscriptions with `f1` and `f2` filters but the same `ModalitiesOfInterest` (as `Location`). On receiving the prediction data for `sid1` and `sid2`, we store the prediction result in `pr1` and `pr2` respectively. Additionally, each time a prediction data is received, we check if the predicted locations and times (in hours) of both the users are the same. Finally, when the above conditions are satisfied, an alarm can be triggered to notify Alice that she is likely to meet Bob at `pr1.getTime().getHour()`.

The code snippet presented in Figure 7 runs on the mobile (client) side. On detecting a subscription for group-based prediction, the middleware itself transmits it to the server-side middleware that starts performing the required prediction task.

7.4.3 Predicting First And Last Occurrences of an Event

In this scenario we consider the case of predicting the time of the first and last occurrences of an event starting within a given time interval. Let us consider a potential Internet of Things application, namely a smart coffee machine linked with the corresponding smartphone app that learns a user's coffee making pattern to predictively start the coffee machine in advance of the user's first instance of making coffee in a day and switches the coffee machine OFF (if it is left switched ON) after the last instance of making coffee in a day. Figure 8 presents the code snippet for implementing a mobile service for such an intelligent coffee machine by using the abstractions offered by FutureWare. We use two

```
PredictionConfig pc = new PredictionConfig();
pc.put(NotificationInterval, t_notify);
ModalitiesOfInterest  moi =
     ModalitiesOfInterest.Activity;
int t_start = 0;
Filter filter = new Filter("Time > " + t_start);
/* Predict when to switch the coffee machine ON
     */
while(true) {
  int s_first = subscribeOnce(moi, filter, pc,
     new FutureWareListener() {
    onNewEvent(PredictionData data) {
      PredictionResult pr = data.getResult();
      String activity = pr.getState();
      int prob = pr.getProbability();
      if(activity.equals("MakingCoffee") &&
     prob > prob_min)
        /* User will make first coffee of the
     day at (t_start - t_notify) hour */
      else
        t_start++;
    }
  } );
}

/* Predict when to switch the coffee machine
     off */
t_start = 0;
while(true) {
  subscribeOnce(moi, filter, pc, new
     FutureWareListener() {
    int s_last = onNewEvent(PredictionData
     data)  {
      PredictionResult pr = data.getResult();
      String activity = pr.getState();
      int prob = pr.getProbability();
      if(activity.equals("MakingCoffee") &&
     prob > prob_min)
        /* User will make last coffee of the
     day at (t_start + t_notify) hour */
      else
        t_start == 0 ? 23 : t_start--;
    }
  } );
}
```

Fig. 8. A code snippet for implementing a Internet of Things service that predictively notifies the smart coffee machine about the user's first and last instance of making coffee.

subscriptions 1) `s_first` to predict the first instance of making coffee; 2) `s_last` to predict the last instance of making coffee.

In order to evaluate `s_first` we predict the probability that the user will make coffee (i.e., `Activity == MakingCoffee`) in a loop for each subsequent hour starting from midnight and once the predicted probability is greater than a given probability value (say `prob_min`), we exit the loop and switch the coffee machine ON in advance of the predicted time for fist instance of making coffee. In case of `s_last` we predict the same but in a loop for each preceding hour starting from midnight and once the predicted probability is greater than `prob_min`, we exit the loop and switch the coffee machine OFF in case it is left switched ON after the predicted time for last instance of making coffee. In order to train the model we assume that the coffee machine detects if the user is indeed making coffee and transmits this information to the mobile application. We define

a `PredictionConfig` with `NotificationInterval` as `t_notify` (i.e., the time in advance for notifying about the predicted event) and an appropriate `Filter` capturing the notification requirements. The value of `t_start` depends on the subscription (i.e., `s_first` and `s_last`) and the iteration count.

7.5 Analyzing API Usability through Real-world Apps

In this subsection we evaluate how FutureWare can be used to develop some popular existing applications to show its potential practical uses. In order to perform this evaluation, we selected eight apps from the following two domains: (i) Personal Assistant, and (ii) Health and Personalization. There are three personal assistant apps (GoogleNow, Cortana and MindMeld), and five health and personalization apps (Yahoo Aviate, Z Launcher, EverythingMe, Neura Platform and PrefMiner). As shown in Table 4, our analysis demonstrates that FutureWare is capable of providing support for all fundamental predictive functionalities of the examined apps. These functionalities are: (i) predicting next task based on contextual information, (ii) predictive suggestions to assist in the current task, (iii) predicting next app, (iv) predictive reminders, and (v) predicting irrelevant notifications. These functionalities, as discussed in the previous section, could be achieved through a proper setup of the *Prediction Configuration Object* while making subscription for one-time or continuous predictions.

FutureWare can also be used together with other libraries to improve the usability of apps by extending features such as speech recognition and gesture recognition. Example of such libraries include Android Speech Recognition [58] and Android Gesture Recognition [59] APIs. These features along with other non-supported features (such as information search and transparency of prediction model) are orthogonal to the scope of FutureWare and can be implemented at the application level.

8 Implementation Support for Developers

The implementation of the prediction logic is not a trivial task for developers who do not have expertise in building anticipatory mobile applications. In this section we discuss how FutureWare provides support for implementing such applications, by examining some key aspects of the development and testing process.

8.1 Data Collection and User's Permissions

The main component of an anticipatory mobile app is the data that is obtained through mobile sensors and interaction with the users. FutureWare removes the burden of implementing the sensor sampling process and the management of the sensor data from the developer. An overlying app does not have to subscribe to sensors as the underlying subscriptions to the sensors are automatically done by the middleware. At the same time, the middleware offers primitives to tailor the sampling rate of sensors in order to manage the trade-off between accuracy and energy consumption (discussed in Section 6). However, the middleware assumes that the app built on top of it has the necessary permissions to query the required sensors. In other words, the middleware does not explicitly ask users for permissions. In case the middleware identifies a missing permission for a modality requested in a subscription, it throws an error when the app tries to create that subscription.

8.2 Selecting the Right Machine Learning Algorithm

We envision FutureWare as a framework whose utility will grow with the evolution of context prediction methods. Therefore, we deliberately enable easy integration of our middleware with third-party machine learning models. The middleware offers additional flexibility by enabling the reuse of the third-party ML components by different subscriptions. Overall, the middleware enables the developers to select the ML algorithm based on the predictive task they try to implement. For example, it could be a classification or regression task and accordingly there are variety of algorithms (such as linear regression [60], support vector machines [61] and random forest [62]) to model the task.

Developers can integrate any third-party ML component, such as those provided by Weka for Android [63], by creating a Java class that implements the `PredictorInterface` interface offered by the middleware. The `PredictorInterface` exposes a method signature named `generateModel` that can be implemented to train (and return) prediction models. Another important method signature is `predictionRequest` that can be used to make predictions. The arguments of this method are the current values of the context modalities that are required to make the prediction and a reference to the model itself. The method returns the prediction results wrapped in the `PredictionResult` class. To summarize, these functions allow the app to integrate third-party ML components by enabling the two fundamental steps of training of a model and making predictions using it.

8.3 Reasoning about Prediction Requirements

Context prediction is not usually a part of an average developer's repertoire, thus it might be challenging for them to reason about it while designing an application. To facilitate the process we suggest an approach that builds upon the standard context inference pipeline (Figure 4 in [39]). Namely, one should start from the high-level context aspect that they would like to predict, and follow down to the sensing modalities and their features that had been shown to be relevant for inferring that specific context.

We refer the reader to [39] where the authors have presented a breakdown of commonly used features and machine learning approaches that can then be used for a particular domain (Table 2 and Table 3 in [39]).

8.4 Quick Implementation and Validation Procedure

FutureWare enables rapid prototyping of anticipatory mobile applications, which in turn enables quick sanity checking of envisioned applications. Anticipatory mobile applications are by their nature complex and their testing needs to cover both the performance of low-level sensing, feature extraction, and machine learning methods' implementation. FutureWare comes with built-in sensing and machine learning implementation, however, it also allows developers

TABLE 4
Review of FutureWare API support for existing applications.

Domain	Application	Supported Features	Non-supported Features
Personal Assistant	GoogleNow [23]	Predict next task based on time and location	Speech recognition
	Cortana [25]	Predict next task based on time and location	Information search
	MindMeld [22]	Predictive suggestions to perform current task	Information search
Health and Personalization	Yahoo Aviate [24]	Predictive home-screen based on time and location	–
	Z Launcher [55]	Predictive home-screen based on the context	Gesture recognition
	EverythingMe [56]	Predictive app-bar based on location	–
	Neura platform [57]	Predictive reminders for medication and staying active	–
	PrefMiner [38]	Predictive suggestions for blocking irrelevant notifications	Transparency of model

to incorporate their own sensing and machine learning mechanisms that can be linked to the middleware through `FileManager` and `PredictionConfiguration` APIs respectively. We suggest that anticipatory mobile application developers independently test sensing and machine learning components using standard procedures (e.g. using unit testing).

Integration and validation testing are the next steps, and here the abstraction *Predictor* (discussed in Section 4) not only enables quick implementation of anticipatory mobile applications, but also reduces the time needed for testing these applications. To test an application, developers could provide previously collected data and link them with the middleware through the `FileManager` API. The data is then used to train the model once a new subscription is made. Furthermore, developers can wait until the new context is captured and investigate their models' performance on it, or they can also provide mock test data (through the `FileManager` API) to speed-up the testing process. In this way, developers do not have to rely on a separate platform for validating the performance of their models and can also quickly test their prediction mechanism.

8.5 Bootstrapping the Prediction Models

A classic problem in developing predictive mobile applications is the initial *bootstrapping* and training of the predictors: when an app is installed by a user, his/her behavioral data is not usually available to train the prediction model. The training of the model is only possible after collecting a certain amount of data, which becomes available gradually over time. For example, if the app aims to build a mobility prediction model it might fail to make any accurate predictions until it has collected a sufficient amount of location data. This amount depends on the type of underlying classifier.

In order to address this issue, previous studies have shown that group-based models could be constructed to make predictions at the initial stage and, then, the performance of the predictors can be gradually improved by successive re-training over time as more data is collected [64]. Separate models could be trained for each group of users who possess identical behavioral or personal characteristics. During the deployment, these models are then assigned to new users based on their behavioral or personal characteristics. These characteristics could be chosen by the developers based on the app domain. For example, the developer of a mobility prediction app can use profession and age group as the two factors to cluster users. It is worth noting that group-based models could not be used for predicting very specific activities, instead they should be adopted to make predictions about generic behavioral patterns of users.

FutureWare allows for pre-training the model by exploiting historical data of other users. For example, developers could collect such behavioral data with test users and validate the predictive power of the models through an offline analysis during the app development phase.

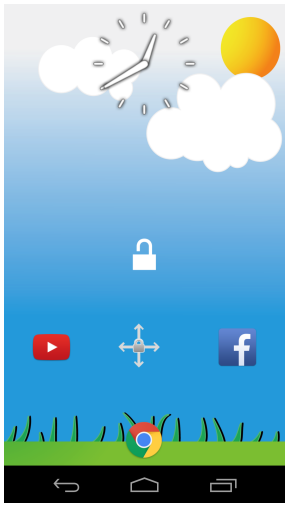

Fig. 9. QuickLaunch, a lock screen application. The lock icon (in the centre) can be slided towards an application icon in order to launch that application. Also, the phone can be unlocked (without launching any application) by sliding the lock icon towards the unlock icon (i.e., towards top in this case).

9 Case Study: Predictive Lock Screen App

9.1 Overview

To show a potential practical use of FutureWare in developing a predictive mobile application, we now present the design of a simple illustrative application named QuickLaunch – an intelligent lock screen to launch applications without unlocking the phone. Given a growing number of application installed on the smartphones, managing applications on a home screen is cumbersome. QuickLaunch reduces the time a user spends in navigating through a home screen to find an application icon. QuickLaunch is a replacement for Android's default lock screen that learns the user's application usage behavior and provides easy and fast access to the desired application at the right time. We

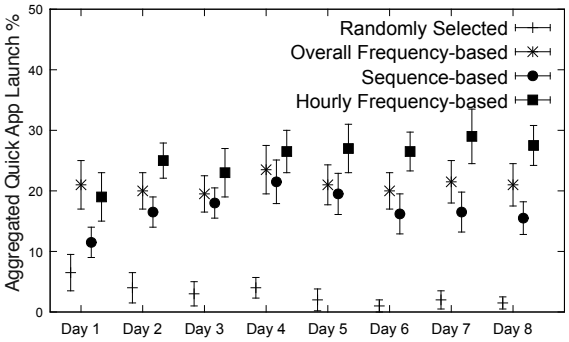

Fig. 10. Quick Launch application results. We show aggregated quick app launch percentages along with the confidence intervals of nine users for each day.

present this application as an illustrative case study. The goal of this implementation is not to develop and evaluate a particularly novel solution for predictive lock screens but in showing the expressiveness of FutureWare.

QuickLaunch builds a behavioral model of the user's application usage to predict what applications will be used by the user in near future and places the relevant icons on the lock screen for quick access (see Figure 9). For example, on predicting that the user will use Chrome, Google Maps and Skype in the next hour, QuickLaunch displays the relevant icons on the lock screen so that user can launch these applications by swiping the lock icon towards an application icon, without unlocking and searching for the apps. However, if the required application icon is not present on the lock screen, the user can simply swipe the lock icon towards the unlock icon to go straight to the home screen. The position of the unlock icon changes on each launch of the lock screen so that the users do not establish the habit of swiping the lock icon in a single direction without looking at the available application icons.

9.2 Programming Effort Comparison

We demonstrate the benefits of using FutureWare by evaluating FutureWare's potential to reduce the programming effort for the development of mobile applications that rely on context prediction. We compute the lines of code (LOC) needed for the implementation of QuickLaunch both with and without using the FutureWare middleware. For a fair measure of programming effort between the two versions of the application, we use the same ML tools in both cases, and do not include these tools in the LOC computation. In total, FutureWare halves the LOC from 6085 to 2720, for implementing the QuickLaunch application. However, in the case of programming effort comparison for implementing only the predictive functionalities used in the QuickLaunch application, FutureWare reduces the LOC eighty three folds from 3406 to 41 LOC.

9.3 Prediction Results

QuickLaunch relies on FutureWare to predict application usage. It subscribes to the middleware to make predictions periodically at the time interval of 60 minutes for predicting the application usage in the next 60 minutes. QuickLaunch exploits the following FutureWare components: (i) hourly frequency-based predictor: predicts the most used applications in the given hour of a day, for example, the most used applications during 22.00 to 23.00 hours; (ii) overall frequency-based predictor: predicts the most used applications from the entire data set; (iii) sequence-based predictor: uses the last application to predict the next applications based on the sequential history of applications. However, for the evaluation purpose we also consider (iv) randomly selected applications. Every time the user unlocks the mobile device, QuickLaunch arranges the application icons based on a strategy that is sequentially chosen from the above four strategies: hourly frequency-based, overall frequency-based, sequence-based and randomly selected.

For completeness we now present results related to the performance of the prediction mechanisms to show a possible real-world use of the middleware[4]. We collected the application usage and the lock screen logs of nine users for ten days, including the number of times the lock screen was seen by the users and the usage of the applications launched by them. The first two days were considered as training period for the predictive mechanisms at the basis of the application. In Figure 10 we present the aggregated quick app launch percentages for all strategies on each day. The results show that all three prediction techniques outperform the randomly selected strategy. Comparing the three prediction strategies, the hourly frequency-based predictions were better than the overall frequency-based and sequential-based predictions.

10 Discussion and Conclusion

In this paper we have presented the design, implementation and evaluation of FutureWare, a middleware that abstracts the complexity of unified context sensing and prediction from the application developer. FutureWare is not limited to a single aspect of the context: it supports location, physical activity, and communication patterns prediction, to name a few. The novelty of FutureWare resides in the set of powerful abstractions that are provided to the users to build novel applications for anticipatory mobile computing.

The main goal of FutureWare is to enable a novel set of anticipatory applications that bring intelligent decisions based on the past, present, and the predicted future context. Anticipatory applications can revolutionize the way we use mobile devices, and in this paper we have presented a few novel conceptual anticipatory application scenarios that can be written in a few tens of lines of code by using FutureWare. By means of a demo application, we have shown that FutureWare can drastically reduce the development effort and time. The aim of this work was not the design of novel prediction algorithms or a predictive application, but of an expressive programming framework for applications that rely on prediction algorithms.

4. At the same time, the focus of our work is not on the implementation of predictive mechanisms themselves, but on the design of a framework for reducing the complexity of implementing anticipatory mobile applications. More complex and accurate algorithms can be integrated in the middleware. The API of FutureWare has been designed to allow an easy integration of third-party predictive components.

We believe that the field of anticipatory mobile computing is still in its nascency. In particular, we see room for improvement when it comes to prediction accuracy that can be enhanced by designing more sophisticated behavioral models. Furthermore, in anticipatory applications predictions are tightly bound to application's needs. Due to these reasons, we leave an option for the developers to define learners that suit their requirements and integrate them with the middleware. FutureWare provides a framework for the design, implementation and evaluation of more refined prediction models and we hope it will represent a useful tool for developers and researchers working in this field. We plan to release the middleware as open source tool for the community and ask developers to provide us feedback on the usability of the middleware. This would help us accurately evaluate the effectiveness of the API offered by the middleware and will also enable us to further improve it in terms of both usability and versatility.

11 Acknowledgements

This work was supported through the EPSRC grants EP/L018829/2 and EP/P016278/1 at UCL and by The Alan Turing Institute under the EPSRC grant EP/N510129/1.

Veljko Pejovic acknowledges the financial support from the Slovenian Research Agency (research core funding No. P2-0098).

References

[1] "Google's Activity Recognition Application," http://developer.android.com/training/location/activity-recognition.html.
[2] D. Quercia and L. Capra, "Friendsensing: recommending friends using mobile phones," in *RecSys'09*, New York City, NY, USA, October 2009.
[3] A. Mehrotra, V. Pejovic, and M. Musolesi, "SenSocial: A Middleware for Integrating Online Social Networks and Mobile Sensing Data Streams," in *Middleware'14*, Bordeaux, France, December 2014.
[4] K. K. Rachuri, M. Musolesi, C. Mascolo, J. Rentfrow, C. Longworth, and A. Aucinas, "EmotionSense: A Mobile Phones based Adaptive Platform for Experimental Social Psychology Research," in *UbiComp'10*, Copenhagen, Denmark, September 2010.
[5] N. D. Lane and P. Georgiev, "Can deep learning revolutionize mobile sensing?" in *HotMobile'15*, Santa Fe, New Mexico, February 2015.
[6] A. Mehrotra, R. Hendley, and M. Musolesi, "Towards Multi-modal Anticipatory Monitoring of Depressive States through the Analysis of Human-Smartphone," in *Adjunct UbiComp'16*, September 2016.
[7] A. Mehrotra and M. Musolesi, "Designing effective movement digital biomarkers for unobtrusive emotional state mobile monitoring," in *MobiSys'17 Adjunct (Workshop on Digital Biomarkers)*, 2017.
[8] T. Wang, G. Cardone, A. Corradi, L. Torresani, and A. T. Campbell, "WalkSafe: A Pedestrian Safety App for Mobile Phone Users Who Walk and Talk While Crossing Roads," in *HotMobile'12*, San Diego, CA, USA, February 2012.
[9] M. Mun, S. Reddy, K. Shilton, N. Yau, J. Burke, D. Estrin, M. Hansen, E. Howard, R. West, and P. Boda, "PEIR, the Personal Environmental Impact Report, as a Platform for Participatory Sensing Systems Research," in *MobiSys'09*, Krakow, Poland, June 2009.
[10] "Mobile Millennium Project," http://traffic.berkeley.edu.
[11] A. Thiagarajan, L. Ravindranath, K. LaCurts, S. Madden, H. Balakrishnan, S. Toledo, and J. Eriksson, "VTrack: Accurate, Energy-Aware Road Traffic Delay Estimation Using Mobile Phones," in *SenSys'09*, Berkeley, CA, USA, November 2009.
[12] J. Ruiz, Y. Li, and E. Lank, "User-defined motion gestures for mobile interaction," in *CHI'11*, Vancouver, Canada, April 2011.
[13] A. Mehrotra, M. Musolesi, R. Hendley, and V. Pejovic, "Designing Content-driven Intelligent Notification Mechanisms for Mobile Applications," in *UbiComp'15*, September 2015.
[14] A. Mehrotra, S. Muller, G. Harari, S. Gosling, C. Mascolo, M. Musolesi, and P. J. Rentfrow, "Understanding the role of places and activities on mobile phone interaction and usage patterns," *IMWUT*, vol. 1, no. 3, 2017.
[15] S. Scellato, M. Musolesi, C. Mascolo, V. Latora, and C. A., "Nextplace: A spatio-temporal prediction framework for pervasive systems," in *Pervasive'11*, San Francisco, CA, USA, June 2011.
[16] A. Noulas, S. Scellato, N. Lathia, and C. Mascolo, "Mining User Mobility Features for Next Place Prediction in Location-based Services," in *ICDM'12*, Brussels, Belgium, December 2012.
[17] A. J. Nicholson and B. D. Noble, "Breadcrumbs: forecasting mobile connectivity," in *MobiCom'08*, San Francisco, CA, USA, September 2008.
[18] J. Manweiler, N. Santhapuri, R. R. Choudhury, and S. Nelakuditi, "Predicting length of stay at WiFi hotspots," in *INFOCOM'13*, Turin, Italy, April 2013.
[19] M. Pielot, "Large-scale evaluation of call-availability prediction," in *UbiComp'14*, Seattle, WA, USA, September 2014.
[20] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu, "Fast app launching for mobile devices using predictive user context," in *MobiSys'12*, Lake District, UK, June 2012.
[21] C. Shin, J.-H. Hong, and A. K. Dey, "Understanding and prediction of mobile application usage for smart phones," in *UbiComp'12*, Pittsburgh, PA, USA, September 2012.
[22] "Mindmeld," http://www.expectlabs.com/mindmeld/.
[23] "Google Now," http://www.google.com/landing/now/.
[24] "Yahoo Aviate," http://aviate.yahoo.com/.
[25] "Cortana," www.windowsphone.com/en-us/features-8-1/.
[26] "Alexa," https://www.alexa.com.
[27] "Siri," https://www.apple.com/uk/ios/siri/.
[28] R. Rosen, *Anticipatory Systems*. Pergamon, 1985.
[29] V. Pejovic and M. Musolesi, "Anticipatory mobile computing for behaviour change interventions," in *UbiComp'14 Adjunct*, Seattle, WA, USA, September 2014.
[30] Y. Wang, J. Lin, M. Annavaram, Q. A. Jacobson, J. Hong, and B. Krishnamachari, "A Framework of Energy Efficient Mobile Sensing for Automatic User State Recognition," in *MobiSys'09*, Krakow, Poland, June 2009.
[31] H. Lu, J. Yang, Z. Liu, N. Lane, T. Choudhury, and A. Campbell, "The Jigsaw Continuous Sensing Engine for Mobile Phone Applications," in *SenSys'10*, Zurich, Switzerland, November 2010.
[32] C. Cornelius, A. Kapadia, D. Kotz, D. Peebles, M. Shin, and N. Triandopoulos, "Anonysense: Privacy-Aware People-Centric Sensing," in *MobiSys'08*, Breckenridge, CO, June 2008.
[33] N. Brouwers and K. Langendoen, "Pogo, a Middleware for Mobile Phone Sensing," in *Middleware'12*, Montreal, Canada, December 2012.
[34] T. Das, P. Mohan, V. N. Padmanabhan, R. Ramjee, and A. Sharma, "PRISM: Platform for Remote Sensing Using Smartphones," in *MobiSys'10*, San Francisco, CA, USA, June 2010.
[35] D. Ferreira, V. Kostakos, and A. K. Dey, "Aware: mobile context instrumentation framework," *Frontiers in ICT*, vol. 2, no. 6, pp. 1–9, 2015.
[36] S. Nath, "ACE: Exploring Correlation for Energy-Efficient and Continuous Context Sensing," in *MobiSys'12*, Lake District, UK, June 2012.
[37] V. Srinivasan, S. Moghaddam, A. Mukherji, K. K. Rachuri, C. Xu, and E. M. Tapia, "Mobileminer: Mining your frequent patterns on your phone," in *UbiComp'14*, Seattle, WA, USA, September 2014.
[38] A. Mehrotra, R. Hendley, and M. Musolesi, "PrefMiner: Mining User's Preferences for Intelligent Mobile Notification Management," in *UbiComp'16*, September 2016.
[39] V. Pejovic and M. Musolesi, "Anticipatory mobile computing: A survey of the state of the art and research challenges," *ACM Computing Surveys*, vol. 47, no. 3, pp. 1–47, 2015.
[40] A. Campbell and T. Choudhury, "From smart to cognitive phones," *Pervasive Computing*, 2012.
[41] K. K. Rachuri, C. Mascolo, M. Musolesi, and P. J. Rentfrow, "SociableSense: Exploring the Trade-offs of Adaptive Sampling and Computation Offloading for Social Sensing," in *MobiCom'11*, Las Vegas, NV, USA, September 2011.
[42] A. Mehrotra, V. Pejovic, J. Vermeulen, R. Hendley, and M. Musolesi, "My Phone and Me: Understanding People's Receptivity to

Mobile Notifications," in *CHI'16*. ACM, April 2016, pp. 1021–1032.
[43] A. Mehrotra, F. Tsapeli, R. Hendley, and M. Musolesi, "Mytraces: Investigating correlation and causation between users' emotional states and mobile phone interaction," *IMWUT*, vol. 1, no. 3, 2017.
[44] *MongoDB*, 2013, http://www.mongodb.org.
[45] "Mosquitto MQTT Broker," http://mosquitto.org/.
[46] "CLOC – Count Lines of Code," http://cloc.sourceforge.net.
[47] "Android DDMS," http://developer.android.com/tools/debugging/ddms.html.
[48] "Android Activity Lifecycle," https://developer.android.com/guide/components/activities.html.
[49] "Android Activity State and Ejection from Memory," https://developer.android.com/guide/components/activities/activity-lifecycle.html#asem.
[50] L. Canzian and M. Musolesi, "Trajectories of depression: unobtrusive monitoring of depressive states by means of smartphone mobility traces analysis," in *UbiComp'15*, September 2015.
[51] F. Ben Abdesslem, A. Phillips, and T. Henderson, "Less is more: energy-efficient mobile sensing with senseless," in *SIGCOMM'09 Adjunct*, Barcelona, Spain, August 2009.
[52] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones," in *CODES/ISSS'10*, Scottsdale, AZ, USA, October 2010.
[53] S. Clarke, "Measuring API usability," *Doctor Dobbs Journal*, vol. 29, no. 5, pp. S1–S5, 2004.
[54] J. M. Daughtry, U. Farooq, J. Stylos, and B. A. Myers, "API Usability: CHI'2009 Special Interest Group Meeting," in *CHI'09 Extended Abstracts*, Boston, USA, April 2009.
[55] "Z Launcher," https://www.zlauncher.com.
[56] "EverythingMe," http://emlauncher.com.
[57] "Neura platform," https://www.theneura.com.
[58] "Android speech recognition," https://developer.android.com/reference/android/speech/package-summary.html.
[59] "Android gestures recognition," https://developer.android.com/training/gestures/index.html.
[60] J. Neter, M. H. Kutner, C. J. Nachtsheim, and W. Wasserman, *Applied linear statistical models*. Irwin Chicago, 1996.
[61] N. Cristianini and J. Shawe-Taylor, *An introduction to support vector machines and other kernel-based learning methods*. Cambridge University Press, 2000.
[62] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.
[63] "WEKA," https://www.cs.waikato.ac.nz/ml/weka.
[64] N. D. Lane, Y. Xu, H. Lu, S. Hu, T. Choudhury, A. T. Campbell, and F. Zhao, "Enabling Large-scale Human Activity Inference on Smartphones using Community Similarity Networks (CSN)," in *UbiComp'11*, Beijing, China, September 2011.

Abhinav Mehrotra Abhinav Mehrotra is a Research Associate at University College London. His main areas of interest include human behavior modeling and digital health. More specifically, his current research focuses on understanding and predicting human behavior by using contextual information obtained via embedded sensors for the development of intelligent systems. He also works on projects in the digital health area, in particular on the development of techniques for predicting mental health symptoms through the analysis of mobile sensing data. He obtained his PhD in computer science from the University of Birmingham, UK, where he worked on intelligent mobile notification systems.

Veljko Pejovic Veljko Pejovic is an assistant professor of computer science at the University of Ljubljana, Slovenia. His broader interests include mobile computing, HCI, resource-efficient computing, and the interaction of technology and society. He obtained his PhD in computer science from the University of California, Santa Barbara, USA, where he worked on mobile and wireless technologies for bringing connectivity to remote rural regions. Before joining the University of Ljubljana, he worked as a research fellow at the University of Birmingham, UK, investigating and developing mobile-based behavioral change interventions.

Mirco Musolesi Mirco Musolesi is a Reader in Data Science at the Department of Geography at University College London and a Turing Fellow at the Alan Turing Institute. At UCL he leads the Intelligent Social Systems Lab. He received a PhD in Computer Science from University College London and a Master in Electronic Engineering from the University of Bologna. After postdoctoral work at Dartmouth College and Cambridge, he held academic posts at St Andrews and Birmingham. The research focus of his lab is on sensing, modeling, understanding and predicting human behavior in space and time, at different scales, using the 'digital traces' we generate daily in our online and offline lives. He is interested in developing mathematical and computational models as well as implementing real-world systems based on them. This work has applications in a variety of domains, such as ubiquitous systems design, healthcare and security&privacy.