3-2014

An Active Resource Orchestration Framework for PAN-scale Sensor-rich Environments

Youngki LEE
Singapore Management University, YOUNGKILEE@smu.edu.sg

Chulhong Min
KAIST

Younghyun Ju
KAIST

Seungwoo Kang
KAIST

Yunseok Rhee
Hankoon University of Foreign Language

See next page for additional authors

Citation

LEE, Youngki; Min, Chulhong; Ju, Younghyun; Kang, Seungwoo; Rhee, Yunseok; and SONG, Junehwa. An Active Resource Orchestration Framework for PAN-scale Sensor-rich Environments. (2014). *IEEE Transactions on Mobile Computing*. 13, (3), 596-610.
Available at: https://ink.library.smu.edu.sg/sis_research/2065

An Active Resource Orchestration Framework for PAN-scale Sensor-rich Environments

Youngki Lee, Chulhong Min, Younghyun Ju, Seungwoo Kang, Yunseok Rhee, Junehwa Song

Abstract—In this paper, we present Orchestrator, an active resource orchestration framework for a *PAN-scale sensor-rich mobile computing platform*. Incorporating diverse sensing devices connected to a mobile phone, the platform will serve as a common base to accommodate personal context-aware applications. A major challenge for the platform is to simultaneously support concurrent applications requiring continuous and complex context processing, with highly scarce and dynamic resources. To address the challenge, we build Orchestrator, which actively coordinates applications' resource uses over the distributed mobile and sensor devices. As a key approach, it adopts an *active resource use orchestration*, which prepares multiple alternative plans for application requests and selectively applies them according to resource availability and demands at runtime. Through the selection, it resolves resource contention among applications and helps them efficiently share resources. With such system-level supports, applications become capable of providing long-running services under dynamic circumstances with scarce resources. Also, the platform can host a number of applications stably, exploiting its full resource capacity. We build a Orchestrator prototype on off-the-shelf mobile devices and sensor motes and show its effectiveness in terms of application supportability and resource use efficiency.

Index Terms— Context monitoring, Active resource orchestration, PAN-scale sensor-rich environments.

1 Introduction

A smart mobile device expands its role as a gateway for personal pervasive services. It will form a *PAN-scale sensor rich environment* with diverse wearable or space-embedded sensors, e.g., e-watch, sensing garments, and textile electrodes in bed sheets [18], [23]. As a common platform, a mobile device will accommodate various *personal context-aware* applications, e.g., dietary monitoring, life assistant [28], elderly support [29]. The applications monitor user contexts continuously [16], and provide highly proactive and situational services. The context monitoring often requires multi-step complex processing across the mobile and sensor devices (e.g., for a 'running' context, acceleration sensing on multiple body-worn sensors, FFT-based feature extraction, and classifying the features through a decision tree [17]).

This new environment raises an important challenge; the platform should run a number of concurrent applications with highly scarce and dynamic resources. Greedy resource use by an application would significantly aggravate contentions among multiple applications and deepen skewed uses of specific devices. This can lead to substantial reduction of overall system capacity. Specifically, we first note that many tiny sensor devices have strictly constrained resources. For example, a MicaZ mote has 8MHz CPU and 4KB RAM; it is even incapable of running a light FFT library, *kiss_fft* [25], often used to extract frequency-domain features. More challenging, availability of sensor devices changes dynamically due to their wearable forms and mobility of users. For example, a user may take off a sensor-equipped watch, or enter a sensor-embedded office. Also, changes in applications and their requests continuously affect resource availability of sensor devices.

- *Y. Lee, C. Min, Y. Ju, S. Kang, and J. Song are with the Department of Computer Science, KAIST, 335 Gwahangno, Daejeon 305-701 Republic of Korea. (Telephone: +82-42-350-3546, e-mail: { youngki, chulhong, yhju, swkang, junesong} @nclab.kaist.ac.kr).*
- *Y. Rhee is with the School of Electronics and Information Engineering, HUFS, 89 Wangsan-ri Mohyeon Yongin-si Gyeonggi-do 449-791 Republic of Korea. (Telephone: +82-31-330-4259, e-mail: rheeys@hufs.ac.kr).*

It is almost impossible for individual applications to address these challenges and ensure applications' steady running. Without system-level supports, an application has an extremely limited view on the resource uses of other applications, and hardly negotiates with them for coordinated resource use. Moreover, individual applications hardly adapt to the joins and leaves of heterogeneous sensors and the starts and stops of other applications. For example, consider an application monitoring a user activity using an accelerometer. It fails to run when the very accelerometer is unavailable, e.g., occupied by other concurrent applications or no longer reachable. Even succeeding, the application may redundantly compute the same tasks, wasting limited computational resources.

In this paper, we propose Orchestrator, a novel active resource orchestration framework. Actively interplaying multiple context-aware applications and scarce, dynamic resources within a PAN, Orchestrator hosts concurrent applications stably, exploiting its full resource capacity. More specifically, it helps applications share in resources and processing with a holistic view on the applications and resources. Also, it resolves resource contention between applications. Moreover, it provides continuous context monitoring services, adapting to dynamic sensor membership and their resource availability. With such system supports, applications can provide mobile users with seamless and long-running services by delegating complex resource management details to the system.

1.1 Active Resource Use Orchestration Approach

To enable effective resource orchestration, we take an *active resource use orchestration* approach. A key of this approach is to decouple actual resource selection and binding from applications' logical resource demands. Once applications turn in high-level context specifications to the system, the system actively finds the best combination of resources to process the contexts on-the-fly under the current status of resources and applications.

Digital Object Indentifier 10.1109/TMC.2013.68 1536-1233/13/$31.00 © 2013 IEEE

This is substantially different from an existing approach, *passive resource use management*, adopted in many mobile and sensor systems [5], [6], [9]. Such systems are mostly designed based on application-driven decision in resource selection and binding. Applications explicitly specify the types and amounts of required resources, for instance, as a resource ticket [9], [5]. A system then takes a passive action, simply allocating the requested resources if available. If not, applications themselves reduce their resource use in different ways, e.g., by trading off data fidelity or deactivating certain functionalities, and re-request reduced amount of resources. These approaches, however, impose huge burden to application developers in our environment; it is hardly possible to predict complex resource dynamics and prepare alternative logics matching to individual cases.

To address the problem, Orchestrator takes an active orchestration approach and realizes it as follows (see Fig. 1). First, it prepares alternative resource use plans to monitor a high-level context. Each plan utilizes different combination of sensor devices and their resources, providing opportunities to flexibly adjust applications' resource use. Second, at runtime, Orchestrator selects and executes the best combination of plans for concurrent requests, holistically considering diverse system inputs; 1) the resource demands of applications and 2) resource availability of devices, and 3) system-wide policies. The plans are selected in a way to resolve contentions among concurrent applications and maximize sharing to save resources. Orchestrator flexibly changes executing plans to adapt to dynamic system events such as sensor join/leave. Such holistic coordination and flexible adaptation enable to support multiple context-aware applications as long and balanced as possible.

To generate alternative resource use plans, Orchestrator exploits the diversity of semantic translation. A context can be derived from different sensing modalities, feature sets, and classification methods. For instance, a 'running' context is monitored with diverse methods, e.g., utilizing DC and energy features from acceleration data [17] or statistical features from GPS location data (See Section 4.2 for more details). Alternative plans utilize different combination of devices and their resources. They provide high flexibility in resource coordination compared to the methods that simply under-utilize the designated devices by trading off fidelity or controlling execution period [7], [8], [15].

There have been prior systems to facilitate the resource use adaptation, for instance, Level [7] and Eon [8] for a sensor device and Odyssey [3], [4] for a mobile device. In these works in common, applications register alternative code blocks to the system. At runtime, the systems selectively apply one of them, to best adapt application behavior over changing resource availability. The use of alternatives is a common approach for adaptation, but the proposed active orchestration approach shows uniqueness in terms of preparing and utilizing alternative plans. A key difference is that plans in Orchestrator are mapped to different combination of sensor devices (*inter-device plans*) while the alternatives of other systems are tied to a specific device (*intra-device plans*), e.g., changing execution periods or data fidelity only. Such relaxed association with devices is especially effective in a sensor-rich PAN, where sensors join or leave

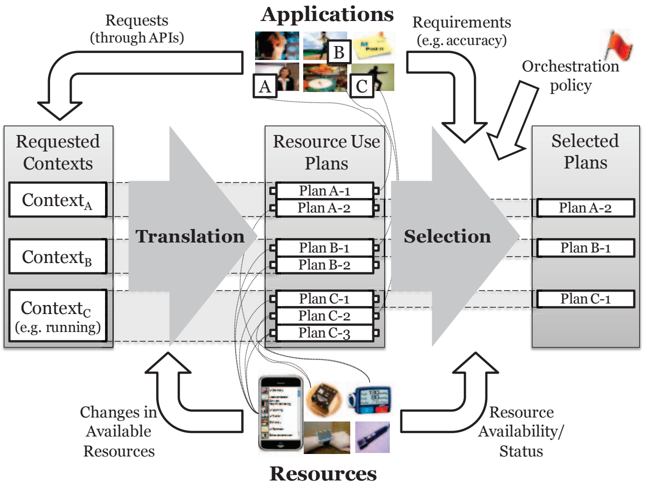


Fig. 1. Active resource use orchestration.

the platform dynamically and their capacity is easily overloaded due to its scarce resource capacity. Second, Orchestrator utilizes its plans to handle resource conflicts on sensor devices among concurrent applications. Most prior systems utilize the alternatives to quickly adapt to changing resource situations; e.g., reducing video quality upon bandwidth reduction [3], or turning off functionality when battery drains sooner than expected [7].

The contribution of this paper is summarized as follows. First, it proposes a new resource coordination system for distributed personal sensor devices in a PAN. The system newly handles resource scarcity and dynamics problem while monitoring multiple contexts for concurrent applications. Second, we propose an active resource orchestration approach; it relaxes the association between applications and devices, and thus enables flexible coordination and adaptation. Third, to realize the approach, we provide a novel planning mechanism including the two-phase translation, plan selection and adaptation. Importantly, it provides the system primitives to acquire resource demands of alternative plans and resource availability of sensor devices. Finally, we implement Orchestrator prototype over off-the-shelf mobile and sensor devices and extensively show its coordination and adaptation capabilities.

In the rest of this paper, we first discuss related work in Section 2. Section 3 motivates our work. Section 4 describes the architecture and techniques of Orchestrator. In Section 5, we present the implementation and show evaluation in Section 6. Finally, we conclude the paper in Section 7.

2 Related Work and Background

2.1 Context Monitoring Systems

As diverse context-aware applications emerge [1], [2], [24], a common underlying platform is increasingly required to coordinate resource use of applications. As early efforts, Titan has been proposed to enable context recognition in dynamic BAN environments [12], [13]. Titan dynamically reconfigures sensor nodes to adapt the execution of tasks for activity recognition. However, it considers only a single application to run at a time, and does not address complicated issues arising with concurrent applications.

SeeMon [16] is our early attempt to build a context monitoring platform. Orchestrator significantly extends SeeMon with new essential functionalities. First, Orchestrator supports generic sensor devices that have multiple sensing modules and processing capability while SeeMon considers sensors as mere data sources. Orchestrator thus incorporates distributed architecture to actively leverage generic sensor devices. With the architecture, diverse tasks constituting a processing pipeline can be flexibly offloaded to sensors, e.g., to reduce communication costs. With this design, a new important challenge arises; concurrent applications may easily conflict for the use of resource-scarce sensor devices. In SeeMon, sensors only determine whether to sense and send data. Orchestrator resolves such conflicts through inter-device planning, which is clearly distinguished from SeeMon that improves energy efficiency by simply turning off sensors. Second, Orchestrator newly deals with resource dynamics problem when sensors join and leave. Such sensor dynamics would be common in daily lives due to user mobility and wearable form factors of sensors. SeeMon, however, hardly supports such dynamics; it uses a single method for a context, mainly decision-tree based one, and optimizes resource use within the method.

There have also been research efforts to improve energy efficiency for context monitoring [31], [32], [33]. They reduce energy consumption while maintaining a reasonable accuracy by changing a set of used sensors [31], data sampling rates [45], or offloaded tasks [32]. Unlike Orchestrator, most works focus on reducing the energy usage only for a single context on a single device; they do not address the complicated resource problems occurring when diverse contexts should be monitored over multiple distributed devices. Recently, a technique is proposed to select the best set of sensors and their parameters considering multiple concurrent contexts [33]. While it shares the high level idea with Orchestrator, it focuses on problem modeling and algorithm design whereas Orchestrator focuses on system design issues including plan generation and selection, adaptation, and resource profiling.

2.2 Resource Management Systems

Significant research efforts have been made to effectively manage the use of limited resources of mobile and sensor devices [3], [4], [5], [6], [7], [8], [9]. The potential approaches to design resource management systems can be classified into two different ones, i.e., an *application-driven* and *system-driven management*. While the former tries to give full control over the resource use to the applications, the latter actively involves in the resource use of applications and controls it in a system level.

2.2.1 Systems with Application-Driven Approach

Several systems such as Pixie [9], ECOSystem [5], and Chameleon [6] have taken the application-driven approach. They expose APIs for resource allocation to applications. Applications determine the types and amounts of resources required to execute program codes, and explicitly request the resources through the APIs. For example, Pixie provides resource tickets, e.g., <*Energy, 700mJ, 10sec*>, and Chameleon provides systems calls such as *set-speed()* to control CPU speed directly. These systems play a passive role to bind and allocate the use of the requested resources. In general, the application-driven approach provides applications with high flexibility to control their resource use while imposing much burden to applications.

This approach, however, has limitations to be applied for PAN-scale sensor-rich environment. The complexity in context processing incurs high burden on programming, compromising potential flexibility allowed to developers. Also, considerable efforts are required to identify and specify resource demands for intended context processing. Moreover, it is difficult that the developers should implement different adaptation and coordination strategies suitable to various resource situations.

2.2.2 Systems with System-Driven Approach

We consider that a system-driven approach [7], [8], [5] is more suitable as the solution of our target environment. This approach hides applications from the details of complex resource management. As such, they can focus on application-specific logics such as UI.

However, many existing sensor systems with this approach are still application-aided, e.g., Eon [8], Levels [7]. Applications need to provide multiple code blocks, each of which corresponds to high level resource states; then the systems help applications adapt its energy use for changing battery status. Also, these systems are still limited in dealing with contention among concurrent applications or dynamic sensor availability; they focus on a single application under fixed sensor membership.

Orchestrator shares a high-level design with Odyssey [3] in that both systems selectively use alternative logics for adaptation and conflicts resolution. However, Orchestrator has several unique features. First, it targets emerging context-aware applications, while Odyssey targets conventional applications such as a web browser and a video player. To handle highly scarce sensor resources and dynamics, Orchestrator creates alternatives in a way to leverage different combinations of sensor devices. In Odyssey, on the other hand, applications mostly control data fidelity altering the resource use within a device. Second, in Odyssey, alternatives are mainly used for agile adaptation over sudden fluctuations in resource availability such as sudden bandwidth drop. Instead, Orchestrator applies alternatives to coordinate sensor resource use of competing applications, in addition to the adaptation to dynamic sensor availability. Although Odyssey considers concurrent applications, its coordination is a lot simpler only considering proportional, static distribution of bandwidth on a single mobile device.

In the literature of sensor network, coordinating energy use over multiple nodes has been importantly studied. Many works propose techniques to balance the communication loads of sensors for routing [10], [11]. This can be considered as a system-driven approach in that sensor systems apply alternative routing path to balance the use of energy resources. However, it is hard to take such a balancing approach in Orchestrator environment since multiple competing applications utilize heterogeneous

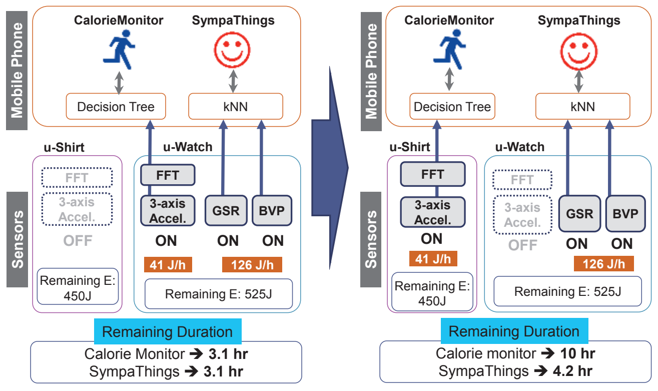


Fig. 2. Motivating cases of Orchestrator (Scene 2)

sensor nodes with different resource use characteristics. It is significantly different from traditional WSNs in which homogeneous sensor nodes work for a single application.

Recently, a high-level service orchestration model [30] has been proposed to provide a service over MANET (Mobile Ad-hoc NETwork). It provides a model to determine mapping between high-level services and low-level resources, considering the dynamic construction of MANET. However, it does not consider models for multiple concurrent applications and their coordination, while focusing more on mobility-awareness and adaptation to support a single application. Also, the main focus of the work is to build a mathematical model while ours try to build real system design and empirical experiments.

3 Motivating Cases

Souneil, a middle-aged man, wears a u-watch that incorporates an accelerometer, a BVP, and a GSR sensor. He uses a *CalorieMonitor* for his weight control every day. It continuously recognizes user activities such as running and walking by sensing acceleration data from the watch, extracting frequency-domain features with FFT and classifying the activities with a decision tree. When running CalorieMonitor only, Orchestrator plans to offload the feature extraction logic to the wrist sensor such that data communication and battery consumption is reduced.

Scene 1: He goes to a fitness room to run on a treadmill. As he gets easily bored with treadmill running, he prefers to play an *exer-game*, *SwanBoat* [26]. SwanBoat leverages his arm and hand gestures as gaming interaction, to make the running more fun and social. For gesture recognition, the u-watch needs to sense acceleration data at a high frequency and send 20 packets/sec. to the mobile device for further processing. While CalorieMonitor is in operation, SwanBoat cannot send the additional data due to conflicts in bandwidth use. The situation is overcome with Orchestrator at work. It identifies that the two applications can share the raw acceleration data although the data is processed afterwards through different inference logics (FFT and decision tree for CalorieMonitor, and heuristic wave form analysis for SwanBoat). Thus, by moving the FFT processing for CalorieMonitor from the u-watch to the mobile device, the u-watch is able to transfer just a single stream of the raw acceleration data, throttling the bandwidth consumption below the availability.

Scene 2: After running and taking a shower, he wears a u-shirt, embedding a 3-axis accelerometer on the waist, and goes to his office, a space where sympathetic interactions are enabled with smart objects (See Fig. 2). With *SympaThings* running on his mobile device, the lamp and the picture frame adapt their color and contents to his affective states which are recognized by processing sensing data from GSR and BVP sensors in the u-watch.

SympaThings consumes about 126 J/hour in the u-watch to sense the data. (The u-watch has already been operating for CalorieMonitor, consuming battery at the rate of 41 J/hour.) Equipped with a small coin battery, the u-watch has remaining energy of 525 J at the moment, and can support the two applications just for 3.1 additional hours. Meanwhile, the calorie expenditure monitor can alternatively operate using another accelerometer in the u-shirt which has more available energy.

As shown in Fig. 2, Orchestrator identifies the new u-shirt sensor and resolves unnecessary battery contention on the u-watch. Identifying that the new sensor has 450 J of available battery, it hands over the acceleration sensing and feature extraction tasks to the sensor. The u-watch sensor, then, can run SympaThings tasks only. This balanced energy use stretches the duration of SympaThings and CalorieMonitor to 4.2 and 10 hours, respectively.

4. Architecture Design

4.1 Architecture Overview

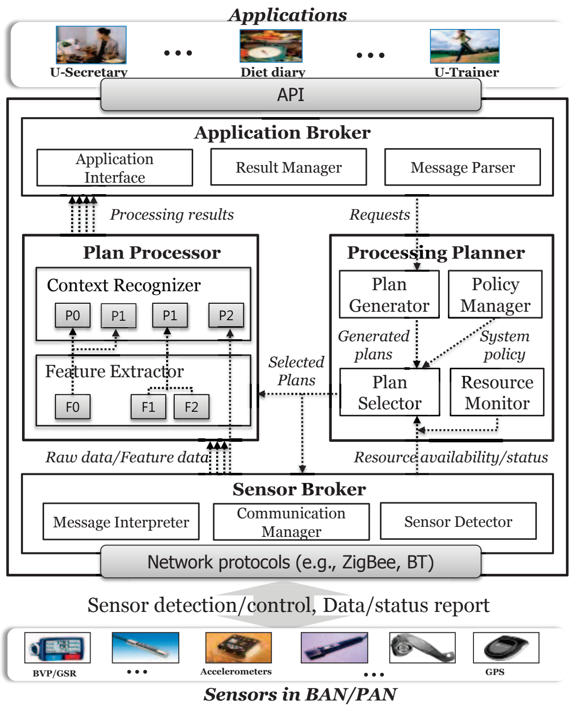
We design the Orchestrator architecture to enable the active orchestration approach. The architecture spans a mobile device and multiple sensor devices (see Fig. 3).

To use the platform, mobile applications register their requests via APIs (see Section 4.2). The *application broker* manages interactions with applications, including request registration/deregistration, delivery of processing result, and notification of processing failure.

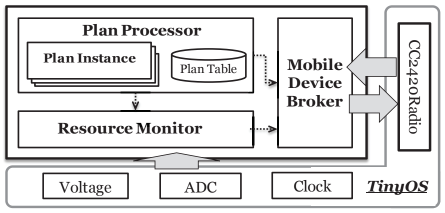
Given the registered requests, the *processing planner* decides how to process the requests with the available devices and resources. It plays a key role as a control center for resource use orchestration and consists of two major sub-components: the *plan generator* and the *plan selector*. The plan generator dynamically updates applicable plans based on available sensors and their capabilities (Section 4.3). Among the generated plans, the plan selector decides a set of plans to execute, which supports maximal requests with available resources and best meets an orchestration policy (Section 4.4). The selection changes adaptively, reflecting dynamic availability of devices and their resources (Section 4.5)

For effective planning, the *resource monitors* keep track of the status of CPU, memory, energy, and bandwidth on sensors and a mobile device (Section 4.6). The status is periodically reported to the plan selector for runtime adaptation. The monitors are designed to minimize monitoring overhead while providing reasonable accuracy.

Once plans are selected, the *plan processors* in the mobile device and sensors cooperatively process the plans. The sensor-side processor performs the early-stage tasks of the context processing pipeline such as sensing and optionally feature extraction tasks. The processor in the



(a) architecture on mobile device



(b) architecture on sensor device

Fig. 3. Orchestrator architecture.

mobile device executes the rest, i.e., feature extraction and context recognition tasks or the latter only, and completing plan execution. Note that we develop the plan processor in the mobile device to dynamically compose and share diverse processing modules for plan execution. In our current implementation, the unit tasks comprising all the usable plans are statically stored in the plan processors of the mobile and sensor devices; at runtime, the plan selector simply triggers the associated sensors to execute the selected tasks. We can further extend the sensor system to dynamically load new processing modules at runtime by adopting over-the-air programming (OTAP).

4.2 Application Programming Interface

Orchestrator provides applications with programming interfaces that abstract complicated low-level resources, while providing rich semantics for a wide range of context monitoring. Applications do not need to specify which sensors to use, what types of data to collect, how often to collect, which feature extraction and classification modules to apply, and where to execute modules. Fig. 4 shows a set of APIs currently supported by Orchestrator.

registerCMQ() is a key API. Using this API, applications easily specify a context of interest as a form of query statement, called Context Monitoring Query [16]. We add an 'ACCURACY' condition, which specifies the minimal accuracy requirement on the specified context value. For

Context Monitoring APIs	*CMQ_ID* = registerCMQ (*CMQ_statement*, *query_result_handler*, *query_status_handler*)
	deregisterCMQ (*CMQ_ID*)
	updateCMQ (*CMQ_ID*, *CMQ_statement*)

Cf. query_result_handler (CMQ_ID, query_result)
query_status_handler (CMQ_ID, query_status)

Fig. 4. Orchestrator APIs.

example, assume that an application wants to know if a user is running with more than 90% of accuracy. Then, the developer specifies the query as below.

registerCMQ("CONTEXT Activity == running,
ACCURACY 90%, DURATION 7 days",
callback_for_result, callback_for_status).

Once the query is registered, Orchestrator notifies the application of query results whenever the condition starts or ends to be satisfied by calling the *callback_for_result* function. Via the *callback_for_status* function, Orchestrator notifies query status, e.g., the query becomes no longer activated or the currently achievable accuracy is 92%. *updateCMQ()* allows applications to change a registered query if necessary upon the update of query status.

4.3 Plan Generation

As a first step to resource orchestration, the plan generator prepares alternative plans for resource uses. A *plan* is the basic abstraction that represents the resource use to handle a request over distributed devices. It is also associated with the expected accuracy of context recognition. A key idea to obtain alternative plans is to exploit the diversity of context recognition methods. First, a context can be recognized by a variety of processing methods. For example, a 'running' activity can be inferred with frequency-domain features of acceleration data and a decision tree [17] as well as with time-domain statistical features and Naïve Bayes [18]. Second, the same context can be recognized by different sensing modalities. For instance, affective states of individuals can be recognized by biomedical sensors such as BVP, GSR, and ECG [21]. It can be also done by using a microphone with voice-related features such as pitch and formant [22]. Lastly, a context can be monitored with different combinations of sensor devices, e.g., the different position of accelerometers for activity recognition [14], [15], [18].

We develop a *two-phase translation method* to prepare a set of usable plans at runtime. The method first loads multiple *logical plans* (LPlans) for each context, prepared by system developers in advance. An LPlan represents a set of sensing and processing modules to derive the corresponding context. At runtime, the logical plans for the context of interests are translated into *physical plans* (PPlans); each LPlan then associated with available physical resources. The concept of translating context-level query is also proposed in SeeMon [16], but used differently. SeeMon translates a context into a fixed plan, i.e., feature-level range query upon that shared processing is enabled. Orchestrator, however, translates a context into multiple alternative plans with which Orchestrator flexibly substitues the resource use to monitor the context.

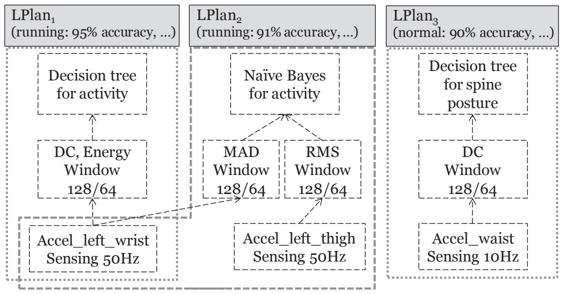

LPlan1 (running: 95% accuracy, ...)
LPlan2 (running: 91% accuracy, ...)
LPlan3 (normal: 90% accuracy, ...)
Decision tree for activity
Naïve Bayes for activity
Decision tree for spine posture
DC, Energy Window 128/64
MAD Window 128/64
RMS Window 128/64
DC Window 128/64
Accel_left_wrist Sensing 50Hz
Accel_left_thigh Sensing 50Hz
Accel_waist Sensing 10Hz

Fig. 5. Example LPlans.

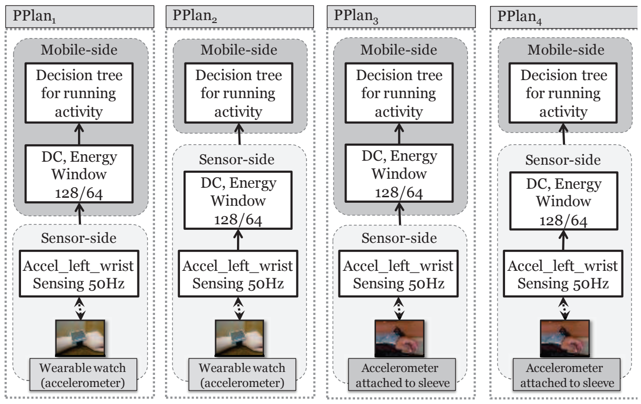

PPlan1
PPlan2
PPlan3
PPlan4
Mobile-side
Sensor-side
Decision tree for running activity
DC, Energy Window 128/64
Accel_left_wrist Sensing 50Hz
Wearable watch (accelerometer)
Accelerometer attached to sleeve

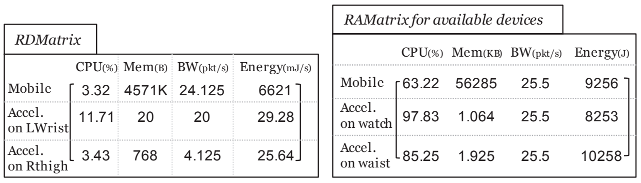
Fig. 6. Example PPlans for LPlan1.

Fig. 5 shows three example LPlans in graph representation: two for 'running context' and one for 'spine posture' context. For the 'running', *LPlan*$_1$ utilizes acceleration data from a left wrist, extracts two frequency-domain features, DC and energy, and runs a decision tree classifier. On the other hand, *LPlan*$_2$ utilizes two time-domain statistical features, RMS and MAD, and a Naïve Bayes classifier. To incorporate diverse LPlans, Orchestrator provides plan developers with a simple description language as well as a variety of processing modules commonly used for context monitoring.

The plan generator translates such LPlans into PPlans by associating available sensor devices at runtime. To increase diversity of PPlans, it maximally leverages *sensor mappings* and *distribution mappings*. The former utilizes the multiple sensors for an LPlan that are eligible to serve the sensing and processing tasks in the LPlan. For the sensor mapping, Orchestrator facilitates plan developers to specify the requirements on sensors, e.g., sensor data type and sampling rate, sensor position, and processing capability. For example, in Fig. 5, the sensor of *LPlan*$_1$ is described as 'accelerometer, left wrist, 50Hz'. At runtime, Orchestrator identifies the sensors satisfying the requirements. The latter exploits the distributions of processing modules into sensors and a mobile device. Fig. 6 shows four example PPlans for an LPlan$_1$ when the user has two available sensors, a watch-embedded accelerometer and a sleeve-attached one on a left wrist.

4.4 Plan Selection

The core of the effective orchestration is to properly select PPlans to execute. Through the plan selection, Orchestrator supports application requests maximally even with highly limited resources. Also, it meets a system-wide policy and the accuracy requirements of applications.



RDMatrix

	CPU(%)	Mem(B)	BW(pkt/s)	Energy(mJ/s)
Mobile	3.32	4571K	24.125	6621
Accel. on LWrist	11.71	20	20	29.28
Accel. on Rthigh	3.43	768	4.125	25.64

RAMatrix for available devices

	CPU(%)	Mem(KB)	BW(pkt/s)	Energy(J)
Mobile	63.22	56285	25.5	9256
Accel. on watch	97.83	1.064	25.5	8253
Accel. on waist	85.25	1.925	25.5	10258

Fig. 7. Examples of RDMatrix and RAMatrix.

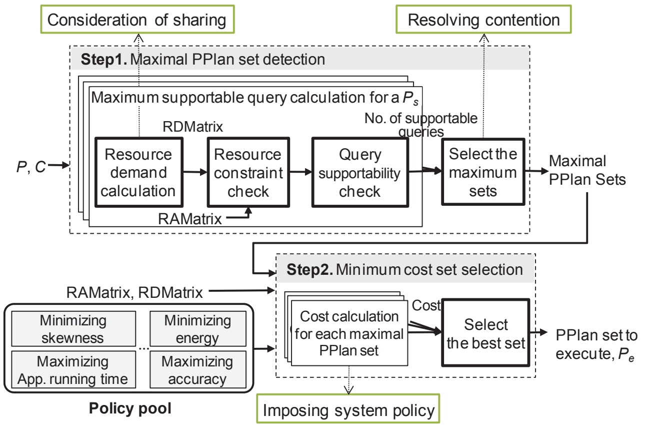

Consideration of sharing
Resolving contention
Step1. Maximal PPlan set detection
Maximum supportable query calculation for a P_s
RDMatrix
No. of supportable queries
P, C
Resource demand calculation
Resource constraint check
Query supportability check
Select the maximum sets
Maximal PPlan Sets
RAMatrix
Step2. Minimum cost set selection
RAMatrix, RDMatrix
Minimizing skewness
Minimizing energy
Maximizing App. running time
Maximizing accuracy
Policy pool
Cost calculation for each maximal PPlan set
Cost
Select the best set
PPlan set to execute, P_e
Imposing system policy

Fig. 8. Plan selection process.

For clear description, we define the selection problem as follows. Given $C=\{c_i \mid c_i$ is a context to monitor$\}$ and $P=\{p_{i,j} \mid p_{i,j}$ is a j^{th} PPlan for a context $c_i\}$, the plan selector determines P_e, a subset of P to execute. Among all possible subsets, P_e should support the maximal number of queries under given resource constraints while the cost of P_e, Cost(P_e) is minimized. Here, the cost function, Cost(), describes the system policy that should be satisfied to achieve desired system operations, e.g. minimizing energy consumption or maximizing recognition accuracy.

For the selection, Orchestrator provides common system primitives that abstract the resource demand of applications and the real-time resource availability. First, a function, *GetRDMatrix*(P_s), provides the resource demand to execute a set of PPlans, P_s, in the form of a matrix, i.e., *RDMatrix*. Second, *GetRAMatrix*() returns the list of available devices and their resource status such as CPU, memory and energy as a matrix, i.e., *RAMatrix*. Fig. 7 shows examples of *RDMatrix* and *RAMatrix*.

Fig. 8 illustrates the plan selection process. It consists of two major processing steps, i.e., (1) detection of maximal PPlan sets and (2) selection of the minimum cost set.

Step1. The plan selector detects the maximal PPlan sets, each of which supports maximum number of queries with available sensors. Fig. 9 shows the pseudo code. To obtain the sets, the selector first computes the number of supportable queries for every possible PPlan set, $P_s \in 2^P$. The computation involves three sub-steps, resource demand calculation, resource constraint check, and query supportability check.

First, *GetRDMatrix*(P_s) calculates the resource demand of P_s by aggregating the resource demand of each PPlan which belongs to P_s. If more than two PPlans in P_s execute the same processing modules in the same device, the resource demand for the module is taken into account only once. Second, with the *RDMatrix*, resource constraints are

```
Step 1. Maximal PPlan set detection

Input: $C = \{c_1, c_2, \ldots, c_n\}$, $P = \{p_{1,1}, \ldots, p_{n,m}\}$
Output: $\{P_M\}$, a set of maximal PPlan sets

1.  $\{P_M\} \leftarrow ø$, $\{P_c\} \leftarrow ø$, $maxQueries = 0$
2.  for $\forall Ps_i$, where $Ps_i \in 2^P$  // $2^P$ is the power set of $P$
3.      // Calculate resource demand
4.      $RDMatrix = GetRDMatrix(Ps_i)$
5.      // Check resource constraints with RDMatrix and RAMatrix
6.      if resource demand < available resources
7.          add $Ps_i$ to $\{P_c\}$
8.          // Check query supportability
9.          $nSupportableQueries = NumSupportableQueries(Ps_i)$
10.         $maxQueries = \mathrm{Max}(maxQueries, nSupportableQueries)$
11. for $\forall Ps_i$, where $Ps_i \in \{P_c\}$
12.     if $maxQueries == NumSupportableQueries(Ps_i)$
13.         add $Ps_i$ into $\{P_M\}$
14. return $\{P_M\}$
```

Fig. 9. Pseudo code for the maximal PPlan set detection.

```
Step 2. Minimum cost set selection

Input: $\{P_M\}$, a set of maximal PPlan sets
Output: $P_e$, a set of PPlans that minimizes the cost function

1. $P_e \leftarrow ø$
2. $cost \leftarrow \infty$
3. for $\forall Ps_i$, where $Ps_i \in \{P_M\}$
4.     if $cost > \mathbf{Cost}(Ps_i)$
5.         $cost \leftarrow \mathbf{Cost}(Ps_i)$
6.         $P_e \leftarrow Ps_i$
7. return $P_e$
```

Fig. 10. Pseudo code for the minimum cost set selection.

checked considering the resource availability exposed by *GetRAMatrix*(). This filters out the PPlan sets that violate resource constraints. The constraints are satisfied only if every device in *RDMatrix* exists in *RAMatrix* and every element in the *RDMatrix* is smaller than the corresponding one in the *RAMatrix*. Since the supportability of each PPlan is highly dependent on that of other PPlans, we check the constraint of a PPlan set as a whole, not as an individual PPlan. Given the plan sets that passed the previous constraint check, the last step selects the ones that support the maximum number of queries. Here, the accuracy conditions for the queries are also checked.

Step 2. In the second step, the plan selector determines the minimum cost set among candidate maximal PPlan sets. It is likely that there often exist multiple maximal PPlan sets since Orchestrator prepares diverse PPlans for contexts and utilizes various combinations of them. Fig. 10 shows the pseudo code. This step calculates the cost corresponding to each maximal PPlan set and selects the one with the minimum cost. The cost is calculated with the *RDMatrix* of the plan set as well as *RAMatrix*. Diverse cost functions can be employed from the policy pool.

4.4.1 Resource Use Policies

Orchestrator supports diverse resource use policies, according to the operation goals of the system. Due to energy limitation of devices, policies are often specified with respect to energy use on devices or running time of applications. Orchestrator adopts several representative cost functions for effective energy utilization, e.g., minimizing the total energy consumption, maximizing the sum of query running time, minimizing the skewness of remaining battery of devices. For high-quality services, it deploys a policy to maximize context recognition accuracy.

```
Function: Cost($Ps$)
Input: $Ps$, a candidate maximal plan set
Output: cost of $Ps$

1. $\mathbf{RDMatrix} \leftarrow \mathbf{GetRDMatrix}(Ps)$
2. $totalEvaluationTime \leftarrow 0$
3. for $\forall p_i$, where $p_i \in Ps$,
    $planEvaluationTime \leftarrow \infty$
    for $\forall d_j$ where $d_j$ is a device to execute $p_i$
        $planEvaluationTime \leftarrow \mathrm{Min}(planEvluationTime,$
            $(\mathbf{RAMatrix}(d_j, ENERGY) / \mathbf{RDMatrix}(d_j, ENERGY)))$
    $totalEvaluationTime \leftarrow totalEvaluationTime +$
                                   $planEvalutaionTime$
4. Return $(1 / totalEvaluationTime)$
```

Fig. 11. A cost function to maximize the query running time

The policies are specified in the form of cost functions. The functions are easily specified by utilizing the system primitives, *GetRDMatrix()* and *GetRAMatrix()*. Consider the policy to maximize the sum of query running time as in Fig. 11. In the function, the RDMatrix is firstly retrieved for the given P_s to figure out the energy demand. For each PPlan, $p_i \in P_s$, the expected running times of queries are calculated with the RDMatrix and the remaining energy of devices from RAMatrix. The running times of p_i are determined by the device which supports p_i for the least time among all required devices to execute p_i. Finally, the function calculates the total sum of the running times.

4.5 Plan Adaptation

Continuous changes in resource availability and application requests affect the operation of Orchestrator. For example, an application may request new contexts, and a wearable watch or u-shirt may join Orchestrator. Due to such changes, the selected PPlans at a time do not guarantee the optimal behavior at another time. To continuously adjust to the new resource demands and availability, the plans are adaptively re-selected at runtime. Through the plan adaptation, Orchestrator keeps supporting application requests seamlessly, resolves newly occurring resource contentions, and continues to best meet the system policy. Note that the adaptation in PAN-scale sensor-rich environment is hardly addressed in previous context monitoring systems [1], [16]; they mainly work under the assumption that the sensor devices are always available.

For effective adaptation, it is important to determine when to reselect the plans. It is clear that new plans should be applied when there are changes in the set of contexts to monitor, *C*, and the set of available PPlans, *P*. This is because *C* and *P* are the major inputs of the plan selection, and the changes in *C* and *P* may disqualify the previously selected PPlan set. Also, changes in resource status could trigger the plan selection. For example, the energy drain of devices periodically needs to trigger the adaptation since it could change the costs of PPlan sets.

Five types of events trigger the adaptation process. When an event occurs, the plan selection is performed with new inputs, C_{new} and P_{new}, changed by the event. P_{new} is obtained considering all LPlans, which enables flexible use of LPlans during the adaptation. Table 1 summarizes

TABLE 1 Adaptation events

Event	Inputs (C_{new}, P_{new}) for plan reselection
Registration of a query (regarding a context c_n)	$C_{new} = C_{old} \cup \{c_n\}$ $P_{new} = P_{old} \cup \{p_{n,j} \mid p_{n,j}$ is a j^{th} PPlan for $c_n\}$
Deregistration of a query (regarding a context, c_d)	$C_{new} = C_{old} - \{c_d\}$ $P_{new} = P_{old} - \{p_{d,j} \mid p_{d,j}$ is a j^{th} PPlan for $c_d\}$
Join of a sensor (s_n)	$P_{new} = P_{old} \cup \{p_{i,j} \mid p_{i,j}$ is a new j^{th} PPlan for a context, c_i, enabled by the new sensor, $s_n\}$
Leave of a sensor (s_d)	$P_{new} = P_{old} - \{p_{i,j} \mid p_{i,j}$ is j^{th} PPlan for a context, c_i, that utilizes the leaved sensor, $s_d\}$
Resource status changes, e.g., energy drain of devices	No changes in C and P

the event types and corresponding changes in the inputs.

4.5.1. Incremental Adaptation

For efficient adaptation, Orchestrator adopts an incremental adaptation method. When there are a number of contexts to monitor and corresponding PPlans, it might be costly to reselect the new plan set by considering whole available PPlans upon every event and redeploy new processing modules. To address the issue, we develop an effective heuristic solution. When the heuristic solution is used, Orchestrator periodically performs global selection to avoid errors that might be accumulated due to repeated incremental selection.

The heuristics finds the subset of contexts and corresponding PPlans that are directly affected by the triggering events. Then, it locally applies the selection process only for the subset upon each event. First, upon a query registration, Orchestrator performs the selection only with the requested context and corresponding PPlans. If the context has been already monitored, the executed PPlan is shared. Second, upon the query deregistration, Orchestrator simply stops executing the corresponding PPlans; it does not perform plan selection again. Third, upon the sensor join, Orchestrator first generates new PPlans utilizing the sensor. If some of queries are newly enabled by the new PPlans, Orchestrator executes the new PPlans. Also, if the new PPlans are more cost-effective than the currently executed PPlans, the new PPlans replace the current one. Finally, when a sensor leaves, some of the running PPlans may be disabled. In this case, Orchestrator finds new PPlans for the affected queries and replaces the disabled plans with new ones.

4.6 Resource Monitoring and Demand Profiling

Resource availability and demand information is essential for Orchestrator to select the best plans. To obtain up-to-date and accurate information, Orchestrator develops *Resource status monitor* to obtain diverse resource status and *Resource demand profiler* to estimate resource consumption for each PPlan. Using these, Orchestrator maintains resource information on RAMatrix and RDMatrix. Moreover, it adopts a sensor detection protocol to identify up-to-date sensor availability. The detection process is initiated by the heartbeat messages periodically generated by sensors. The mobile device listens to the messages and detects new and dead/out-of-scope sensors.

4.6.1 Resource Status Monitoring

We design and implement resource monitors in sensors and mobile devices for energy, CPU, memory, and bandwidth. For mobile devices, Orchestrator simply utilizes the information provided by resource monitoring tools of operating systems. For sensors, we develop our own light-weight monitors. We describe the sensor-side resource monitor below in detail. Currently, we target the MicaZ motes using ZigBee protocols and Tiny OS.

Feature Type	Window /Slide	Feature Processing	Device Type	CPU (%)	Memory (bytes)	Energy (µJ/s)	Bandwidth (pkts/s)
Frequency domain features (4 features)	64/32	On Sensor	Sensor	4.47	576	11.984	3.125
			MD	0.28	160K	27305	
		On MD	Sensor	11.13	20	17.019	20
			MD	2.00	184K	158923	
	128/64	On Sensor	Sensor	3.62	1152	10.786	1.5625
			MD	0.13	160K	4829	
		On MD	Sensor	11.13	20	17.019	20
			MD	1.94	242K	126528	
Statistical features (7 features)	64/32	On Sensor	Sensor	2.17	384	13.265	6.25
			MD	0.17	213K	11047	
		On MD	Sensor	11.13	20	17.019	20
			MD	2.12	221K	98031	
	128/64	On Sensor	Sensor	2.85	768	13.376	3.125
			MD	0.24	217K	16107	
		On MD	Sensor	11.13	20	17.019	20
			MD	1.12	225K	105110	

Fig. 12. Example of resource demand profiles

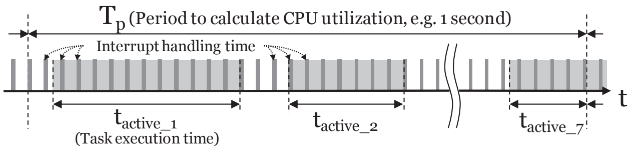


Fig. 13. Example of CPU utilization.

Energy Monitor: For energy monitoring of sensors, Orchestrator adopts a voltage-based method [7]. This is practical since many of widely-used sensors such as MicaZ provide real-time voltage readings. It estimates remaining energy from voltage readings based on pre-built voltage-energy translation maps. However, we find out that the method can cause estimation errors since the voltage reading provided by a sensor could be different from the real voltage up to 5%. The 5% error in voltage readings can cause 27.8% error in estimated energy since the voltage-energy conversion function is non-linear. We compensate the errors based on an additional map between the voltage readings and the real voltages. By applying the map, the energy monitor achieves a high level of accuracy, i.e., the errors under 0.85%.

CPU Monitor: In Orchestrator, the CPU cycle of a sensor device is occupied by two major operations: 1) executing assigned tasks and 2) handling timer interrupts for sensing, storing, transmission (see Fig. 13). Among them, the CPU monitor only considers the CPU cycle for task execution since the interrupt handling cost is relatively small, i.e., 4.5% in our measurements. More specifically, the monitor measures CPU utilization as ($\sum t_{active\_i} / T_p$), where t_{active_i} is the execution time of a $task_i$ and T_p is a period to calculate CPU utilization; in our implementation, T_p is set to 1 second. We measure t_{active_i} by recording timestamps with a system call, *system.getTime32*().

Memory Monitor: The available memory size, M_{av}, is obtained as $M_{max} - \sum_i M_{used}(task_i)$, where M_{max} is the maximum available memory and $M_{used}(task_i)$ is memory used for a $task_i$. $\sum_i M_{used}(task_i)$ is computed as $\sum_i (M_F(task_i)) + Max(M_T(task_i))$, where $M_F(task_i)$ and $M_T(task_i)$ denote the size of *Fixed* and *Temporary Space* for a $task_i$, respectively. *Fixed Space* is continuously occupied by a task to store sensor readings and some internal states until the task is deregistered. *Temporary Space* is allocated and used temporarily only when the task is scheduled to run.

Bandwidth Monitor: Wireless network bandwidth is a shared resource for all sensors and a mobile device. The available bandwidth for all devices is measured and managed in the mobile device. An available bandwidth, B_{av} is measured as $B_{max} - B_{used}$, where B_{max} is the maximum available bandwidth and B_{used} is the bandwidth being currently utilized. According to our experiments, B_{max} is about 40 kbps for ZigBee (802.15.4).

4.6.2 Resource Demand Profiling

As described in Section 4.4., Orchestrator calculates RDMatrix, a matrix representation of resource demand with respect to a set of PPlans. Calculating the RDMatrix is twofold: 1) profiling resource demands of processing modules used for PPlans and 2) computing the total resource demand based on the profiles.

First, Orchestrator collects the resource demand profiles for diverse context processing tasks in pre-runtime. The offline profiling works well in our environment where the energy consumption of sensors is stable over time, although other methods can be also applied as discussed in [37]. Tasks for context monitoring usually perform periodic operations such as sensing and transmitting data at the fixed time interval. Accordingly, the energy consumption to execute such tasks is unlikely to fluctuate over time. PowerTutor [37] proposes online monitoring of power consumption on smartphones; it continuously monitors the hardware status on representative components such as CPU and network interface and estimates the result by aggregating the power consumption of each component. However, it does not provide the measurement on sensor devices, and also imposes high overheads to trace hardware status. Fig. 12 shows a part of profiles for several tasks. For profiling, we used a clone of MicaZ, and a SONY Ultra Mobile PC with 1.33 GHz CPU. We scaled down the CPU frequency to 600MHz and the energy consumption is profiled with a multi-meter.

From the profiles, Orchestrator calculates the total resource demand of PPlans at run time. For the purpose, it first identifies processing modules needed to execute the PPlans. Then, it computes the total demand of PPlans by adding up the resource demands of all the modules to the baseline resource demand. In our experiments, this method provides reasonable accuracy; it achieves 98% accuracy on the estimation of resource demand for the sensors we used. Resource demand estimation is a well-studied problem in the literature [4], [7], [6], [8]. It is beyond the scope of this paper to fully compare to other approaches.

4.7 Limitation and Discussion

We discuss potential limitations of our planning methods.

Burden for LPlan preparation. A potential burden to realize our planning method is to explore and implement usable LPlans for diverse contexts. It could be costly for individual developers to prepare such a variety of LPlans separately; however, the cost is bearable to adopt in a common platform like Orchestrator. Such efforts are required by system developers only once a priori and most application developers can simply use developed plans afterwards. Moreover, frequently used context types are not highly diverse and applicable inference algorithms are also limited to a few; thus, with reasonable efforts, system developers can identify and build multiple processing plans for commonly interested contexts.

We currently use heuristic methods to acquire diverse plans, and show several representative examples. In the near future, however, we believe that a context processing ontology can be built and diverse plans can be systematically managed for various context types. Such processing ontology may be more easily built extending existing context models [34]. Through the ontology, multi-party developers can easily share new processing plans. Also, necessary modules can be downloaded from clouds on demands and loaded to the platform dynamically.

Planning optimization. Our planning method can be further improved by adopting diverse optimization techniques. A first potential optimization is to enable sharing while processing concurrent requests. Sharing can be performed at different levels; when multiple queries monitor the same context, a plan is executed only once and the inferred contexts are de-multiplexed to all relevant applications. When plans for different queries share a common operator or a part of plans, the overlapping part can be executed once and the intermediate results can be shared.

To enable effective sharing, we first need to organize the context processor to flexibly share intermediate or final results in any level. A possible way to develop such sharable context processor is to leverage our recent work, SymPhoney [35]; it organizes context processing pipelines as dataflow graphs of unit operators and executes the operators in an event-driven manner. With such event-driven execution engine, sharing of operators or partial sub-plans can be implemented easily. Another important consideration to enable sharing is that the cost reduction by sharing should be taken into account during the plan selection such that sharable plans can be selected preferably. For proper cost evaluation, we may adopt a new technique to estimate resource demands in an operator-level; the system should be able to estimate resource consumption of diverse combinations of operators that can be shared among multiple queries.

Another interesting optimization technique can be devised to support composite queries such as conjunction or disjunction of contexts, as in SeeMon [16] or Deamon [36]. For example, if a query asks for user activities when he is at a company, the system does not need to monitor the user activity when he is outside of company. Leveraging composite query structure as in the example, Orchestrator can deactivate monitoring of some contexts that do not influence the result generation. Similar idea has been proposed in SeeMon to turn off unnecessary sensors, and we can generalize the idea for the resource use planning in Orchestrator. It is also possible to apply other common intra-device techniques such as duty cycling or sensor data sampling; we could potentially adopt the technique proposed in SymPhoney to intelligently adjust execution periods of highly complex processing pipelines.

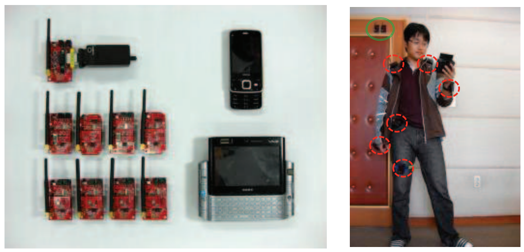


Fig. 14. Hardware setup.

Sensor	Sensor location (sensor ID)	Sampling rate	Feature	Feature generation rate	Context type (# of possible values)	Context value examples
Four 2-axis accelerometer	Right(3) wrist, Right thigh(4), Left wrist(5), Waist (6)	50Hz x 8	DC, Energy, RMS, MAD, Percentile	0.78Hz x 8	Activity (4)	Run, Sit, Walk, Stand
Two light Sensors	Body(2), Space(102)	0.78Hz x 2	Illumination	0.78Hz x 2	Light (7)	Dark, Bright
Two temperature Sensors	Body(1), Space(101)	0.78Hz x 2	Temperature	0.78Hz x 2	Temperature(8)	Cool, Hot
Two humidity sensors	Body(1), Space(101)	0.78Hz x 2	Humidity	0.78Hz x 2	Humidity (6)	Dry, Humid

Fig. 15. Sensor, feature, context profile used in the prototype.

Quality of services. In terms of applications, the quality of services (QoS), e.g., recognition accuracy, provided by Orchestrator might vary from time to time. We believe that the slight QoS difference caused by different plans does not cause severe problem for many non-critical daily applications; current Android sensing APIs do not also guarantee fine-granule QoS for GPS and accelerometer sensing services. For the hard QoS requirements, Orchestrator may conservatively initiate plan adaptation or check QoS condition further during plan selection process.

5 Implementation

We implemented the Orchestrator architecture as a prototype system. First, we implemented the architecture of a mobile device in two platforms: (1) standard C/C++ over Linux, (2) Open C/C++ over S60 SDK and Symbian OS. Their total lines are about 13,000. Second, the sensor architecture is implemented in NesC on top of TinyOS 1.1.11. The total lines are about 2,300.

We deployed the prototype system on various types of mobile devices and sensors. Fig. 14 shows a snapshot of currently used hardware. First, we mainly deployed the prototype on two mobile devices, (1) a SONY UMPC, with Intel U1500 1.33 GHz CPU and 1GB RAM, and (2) a smartphone, Nokia N96 with Dual ARM9 264MHz processor and 128MB RAM. Second, we incorporate various sensors widely adopted for context-aware applications. As presented in Fig. 15, we use eight USS-2400 sensor nodes (MicaZ clones), i.e., four 2-axis accelerometers, two light, and two temperature/humidity sensors. They are equipped with Atmega 128L MCU, CC2420 RF transceiver supporting ZigBee, and TinyOS. For communication between the mobile device and sensors, we attach one base sensor node to the mobile device. The node receives sensor data from other sensor nodes and forwards the data to a mobile device. Also, it transmits control messages to the sensor nodes on behalf of the mobile device.

Currently, the plan processor in the mobile-side architecture includes eight feature extraction modules (see Fig. 15). We used *kiss_fft* [25], a FFT library, to derive frequency-domain features. It also provides a recognition module implementing a decision tree algorithm. To generate multiple plans to monitor activity, we combine diverse feature and sensor sets. As feature sets, we use frequency-domain features (e.g., DC, Energy) and time-domain statistical features (e.g., RMS, MAD). For sensor sets, we use all combinations of sensors on the left/right wrist, right thigh, and waist. We trained the activity contexts via annotation-based learning [17]. The learning was done with C4.5 decision tree by Weka, a Java-based open source machine learning tool [19]. We implemented feature extractors on sensor nodes to offload feature extraction tasks. We used a highly optimized *avr-fft* library written in an assembly language for FFT computation on sensors.

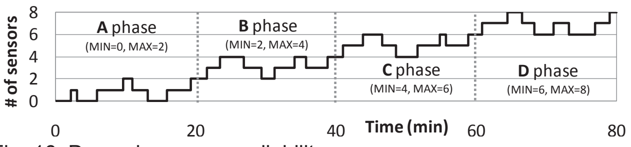


Fig. 16. Dynamic sensor availability.

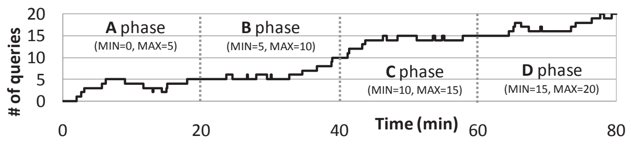


Fig. 17. Dynamic query workload.

TABLE 2. Query model

Parameter	Default value
# of queries	20
# of context types per query	1
Distribution of context type	Activity (50%), Other Contexts (50%)
Distribution of context value	Uniform distribution
Distribution of accuracy requirement (only for activity)	Uniform distribution (min and max from training data)

6 Evaluation

6.1 Experimental Setup

We demonstrate the effectiveness of Orchestrator under dynamic changes in sensor devices and requests. For the experiments, we used the devices described in Section 5.

Sensor availability: For experiments, we first vary the number of available sensors as shown in Fig. 16. For the total 80 minute period, we randomly add or remove a sensor every 2.5 minute among 8 sensors. To examine the effect of number of available sensors, we divide the total time into four 20 minute phases such that each phase predefines different MIN/MAX sensor numbers. Note that sensor composition is diverse even with the same number of sensors.

Query workloads: We also generate a dynamic query workload as shown in Fig. 17. We add or delete a query every 1.5 minute. Also, we set the MIN/MAX numbers of queries for each phase. Table 2 summarizes the parameters and default values used for the query generation.

Baseline: As a baseline to evaluate Orchestrator, we develop the system lacking of resource coordination capability. The system supports a query with a single fixed plan that provides the best recognition accuracy. We denote it as Conventional Context Recognizer (CCR), since conventional context-aware systems mostly adopt a sin-

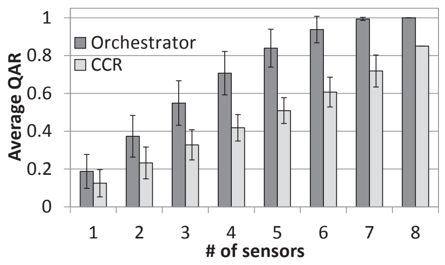


Fig. 18. Effect of sensor availability on QAR.

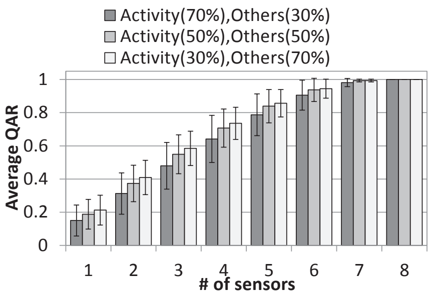


Fig. 19. Effect of request types (Orchestrator).

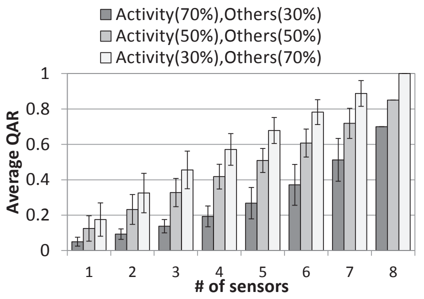


Fig. 20. Effect of request types (CCR).

gle recognition method to handle a context [20]. In CCR, all employed sensors are always on and send raw data. A mobile device extracts features from the sensor data and runs recognition modules. Note that CCR does not utilize newly joined sensors and nor deals with sensor leaves. For Orchestrator, we use the policy that minimizes the total energy consumption of available sensors by default.

Metrics: We measure the effectiveness of Orchestrator in terms of application supportability and resource utilization. First, for the application supportability, we measure *query activation* and *quality of context*. To quantify the former, we use the number of activated queries (NAQ) and the query activation ratio (QAR). We regard that a query is activated if a PPlan exists for the query and is executed with available resources. QAR is formally defined as follows.

$$QAR = \frac{\sum T_A(q_i)}{\sum T_R(q_i)}, \quad q_i \text{ is an } i^{th} \text{ query in a registered query set}, Q$$
$$T_A(q_i): \text{total activation time of } q_i$$
$$T_R(q_i): \text{total registration time of } q_i$$

To quantify the quality of context, we measure the instant accuracy and the overall accuracy. The instant accuracy denotes the average accuracy of activated queries at a moment. The overall accuracy (OA) is the time-averaged instant accuracies for the whole period of experiment.

$$instant\ accuracy = \frac{\sum Acc_C(q_i)}{NAQ},$$
$$q_i \text{ is an } i^{th} \text{ query in an activated query set, } Q$$
$$Acc_C(q_i)\text{: currently achieved accuracy of } q_i$$
$$,$$
$$OA = \frac{\sum \sum_{p_{i,j}} Acc(p_{i,j}) \times T_A(p_{i,j})}{\sum T_A(q_i)}, \ p_{i,j} \text{ is an } j^{th} \text{ plan of } q_i$$
$$Acc(p_{i,j})\text{: accuracy of } p_{i,j}$$
$$T_A(p_{i,j})\text{: activation time of } p_{i,j}$$

As the metrics for resource utilization, we use the number of activated sensors (NAS) and the energy consumption (EC) of sensors in Joule (J). We regard that a *sensor is activated* if the sensor executes certain tasks comprising any selected PPlans. In CCR, all available sensors are considered to be activated since it does not control the sensors.

6.2 Query Supportability

We first evaluate the query supportability under the variation of sensor availability. Fig. 18 shows the average QAR with a standard deviation range as the number of sensors increases. For a number of sensors in the X axis, we measure QAR of every possible combination of sensors and plot average QAR values. For example, a result for two sensors is obtained by averaging QARs for all combinations of two sensors out of 8, i.e., (s1, s2), (s1, s3), …, (s7, s8). As shown in the figure, Orchestrator shows higher QAR than CCR for all ranges of sensor availability. This is because Orchestrator actively identifies executable PPlans even within limited available sensors and selects the best set of plans that resolves the resource conflicts among concurrent requests. On the other hand, CCR utilizes a fixed plan for each query such that it hardly resolves resource conflicts or adapts to dynamic sensor availability. In particular, the QAR gap between Orchestrator and CCR is large when sensor availability is limited. When sensor resources are plenty, every query can be well supported without careful coordination. However, as sensor availability becomes limited, the advantage of resource orchestration gets more significant.

We also evaluate QAR variation under different distributions of context types in the query workload. We measure the average QAR for three different distributions, each of which has different portion of 'activity' contexts, i.e., from 30% to 70%. Note that an activity context is more likely to be supported by diverse combinations of sensors on different body positions while other contexts such as temperature have fewer plans to exploit. Fig. 19 and Fig. 20 show the results for Orchestrator and CCR, respectively. As shown in Fig. 20, CCR shows a large variation of QAR over the distributions. To recognize the activity with the best accuracy, CCR adopts a fixed plan that uses three accelerometers on different body positions. It cannot support activity queries if any one out of the three becomes unavailable. As a result, the QAR with more activity queries considerably decreases in CCR. In contrast, the variation of Orchestrator is much smaller than that of CCR, as shown in Fig. 19. Orchestrator shows the high and stable QAR regardless of the context distribution. It prepares alternative plans for the activity context, and substitutes them when certain sensors are not available and corresponding plans cannot be used.

6.3 Orchestration under Dynamic Environments

6.3.1 Dynamic Sensor Availability

In this section, we evaluate Orchestrator under dynamic sensor availability. The number of queries is fixed at 20.

Fig. 21 (a) shows QAR and EC per phase; more sensors are available as the phase changes from A to D. Under phase A and B where available sensors are scarce, Orchestrator activates more queries than CCR, while consuming almost the same amount of energy. The result shows that Orchestrator well supports multiple applications with limited sensor availability. In contrast, under phase C and D where available sensors are plentiful, Orchestrator con-

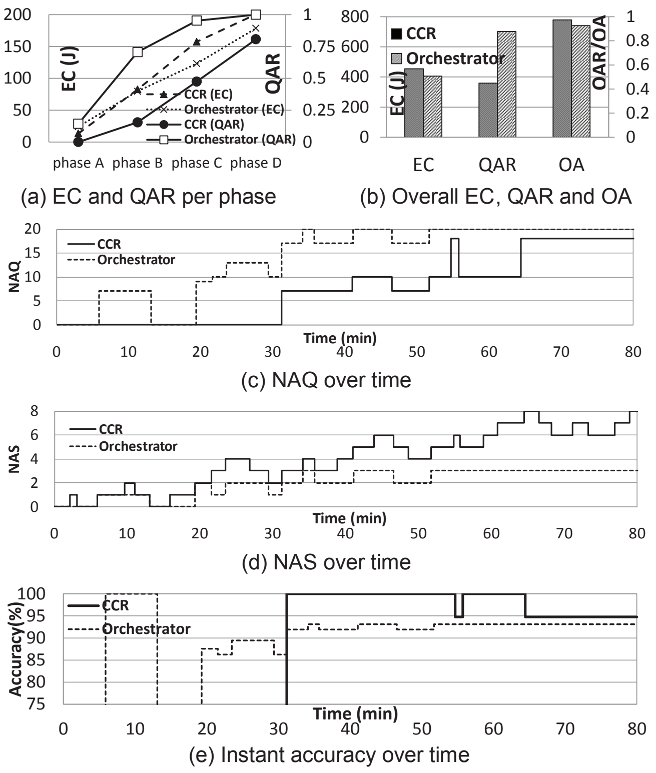


Fig. 21. Orchestration under dynamic sensor availability.

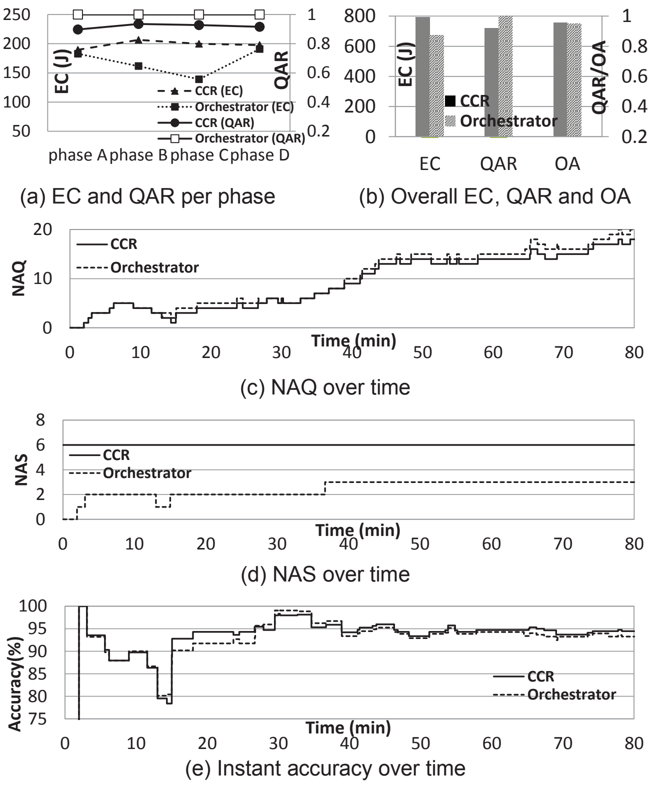


Fig. 22. Orchestration under dynamic query availability.

centrates on energy optimization since the QAR of Orchestrator is already saturated to the maximum, i.e., 1.

Fig. 21 (b) shows EC, QAR, and OA for the total experiment time. To sum up, Orchestrator achieves the two times more QAR with 10.7% reduced energy consumption comparing to CCR. However, CCR shows 4.6% increased OA than Orchestrator. Fig. 21 (c), (d) and (e) show NAQ, NAS and the instant accuracy over time, respectively. Most important, NAQ of Orchestrator is always higher than that of CCR, resulting in higher QAR. This is because Orchestrator utilizes diverse PPlans such that it flexibly supports requests with diverse combinations of sensors. For example, a '*temperature*' query is activated on Orchestrator if any of temperature sensors is available, whereas it is so on CCR only if the designated temperature sensor is available. Also, NAS of Orchestrator is lower than that of CCR. Orchestrator selects PPlans that minimize the number of activated sensors applying the energy optimization policy. In addition, Orchestrator selects PPlans that run feature extraction tasks in sensors rather than the ones that send raw data from sensors. Such sensor-side feature extraction significantly reduces the communication cost, which in turn reduces energy consumption. Interestingly, the accuracy of CCR is higher than that of Orchestrator. By its default policy to minimize the total energy consumption, Orchestrator selects energy-efficient PPlans rather than more accurate ones.

6.3.2 Dynamic Query Workload

We examine the effectiveness of Orchestrator under the dynamic query workload. The number of sensors is fixed at 6 excluding the space-embedded ones, S101 and S102.

Fig. 22 (a) shows EC and QAR per phase. Since the sensor devices are abundant in all phases, the QAR of Orchestrator reaches to the maximum while that of CCR is almost close to the maximum as well. Meanwhile, the ECs of Orchestrator and CCR are kept high, i.e., about 200J. We look into EC per sensor, and discover that a *sensor 4* consumes more energy than other sensors. In our experimental setting, a sensor 3, 4, 5, and 6 as described in Fig. 18 are accelerometers for activity recognition. Based on the energy minimization policy, Orchestrator uses sensor 4 rather than using all sensors together since the accuracy requirements of most activity queries are satisfied only with sensor 4; sensor 4 is placed on the right thigh that is known as the most suitable position for recognizing activities such as running, walking, and standing [17].

Fig. 22 (b) shows EC, QAR, and OA for the total experiment time. Orchestrator achieves better EC (15% reduction) along with slightly better QAR (8.5% improvement) than CCR, whereas CCR shows marginally higher OA (1% increase). Also, Fig. 22 (c), (d), and (e) show NAQ, NAS, and the instant accuracy over time, respectively. The NAQ of CCR is slightly lower than that of Orchestrator in most of time. Although the number of sensors is sufficient to activate all registered queries, some queries are deactivated in CCR. This is because it utilizes only a single recognition method for '*activity*' queries. Although CCR adopts the method to provide the best accuracy, it does not guarantee the best accuracy for every activity instances: sitting, standing, walking, and running. Thus, it could not meet the accuracy requirements of some queries. More important, the NAS of Orchestrator is always lower than that of CCR. All available sensors are activated in CCR, whereas Orchestrator selectively utilizes only some of the sensors to reduce battery consumption. Instead, CCR mostly shows higher accuracy than Orchestrator.

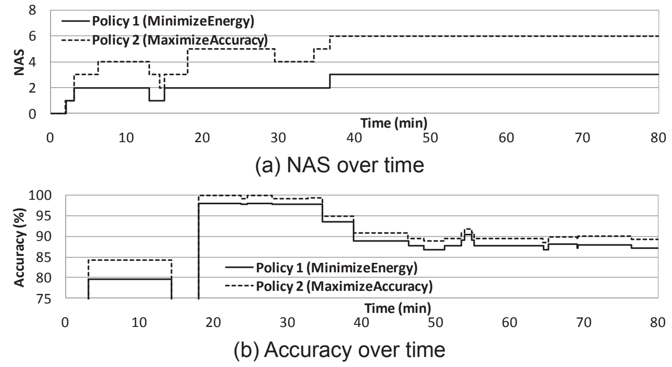


Fig. 23. Effects of resource orchestration policies.

6.3.3 Effects of resource orchestration policies

We further examine if Orchestrator properly applies diverse orchestration policies. We consider two policies: (1) minimizing the total energy consumption of available sensors and (2) maximizing the average accuracy of registered queries. For the experiments, we use the dynamic query workload and fix the number of sensors to 6. Then, we measure the NAS and the instant accuracy over time with the two policies. We only consider the queries containing activity contexts for accuracy measurement.

Fig. 23 (a) and (b) show NAS and the instant accuracy, respectively. In Fig. 23 (a), the NAS of policy 1 is much lower than that of policy 2 during the whole experiment time. Accordingly, the energy consumption is lower with policy 1, which shows the desired operation of Orchestrator. On the other hand, the higher accuracy is guaranteed when policy 2 is used, i.e., 88.8% overall accuracy with policy 1 and 92% with policy 2. In conclusion, Orchestrator well supports diverse orchestration polices that can be easily specified using its system primitives.

6.3.4 Resource Orchestration Costs

We look into the orchestration cost in terms of (1) resource overhead (communication, memory, and energy), (2) reconfiguration time taken for the plan adaptation.

Resource overheads. we first investigate the number of messages exchanged during the total experiment time. Fig. 24 shows the number of messages per message type. In Orchestrator, the control and heartbeat messages are exchanged for coordination, incurring slight communication overheads, e.g., 1.5 kbps under the dynamic query workloads. Orchestrator, however, significantly reduces the number of data messages at the cost of those messages. Compared to CCR, it decreases the number of data messages up to 10 times by transmitting feature values instead of raw data and also selectively utilizing sensors.

Second, we measure the memory usage at runtime. The average memory usage in the mobile device is 57.9KB, which is negligible considering its memory capacity. Each sensor consumes 276B of memory to maintain core data structures for task execution. It additionally consumes 4B to 512B of memory depending on sensor type for data buffering. Note that those values vary depending on executed tasks and runtime parameters such as window size. Considering the memory capacity of MicaZ, i.e., 4KB, Orchestrator can still offload multiple tasks onto sensors.

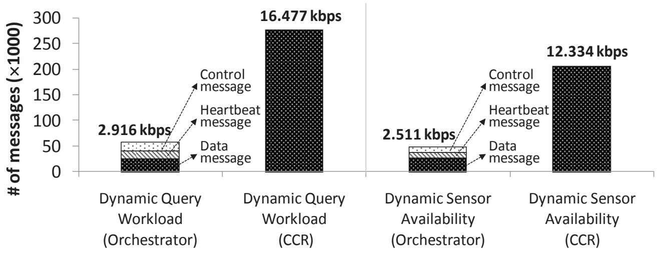


Fig. 24. Communication costs.

TABLE 3. Energy Consumption

Task	Energy consumption (mJ/s)
Idle	15.47
Resource status reading	15.68
Resource status reading /transmission(0.5Hz)	16.42
Heartbeat+ACK exchange(0.5Hz)	17.51
All primitive operations	18.02

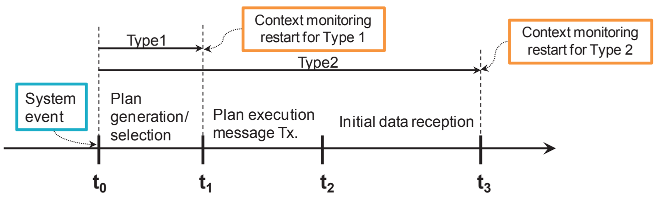

Context monitoring restart for Type 1
Context monitoring restart for Type 2
Type1
Type2
System event
Plan generation/ selection
Plan execution message Tx.
Initial data reception
t_0
t_1
t_2
t_3

Fig. 25. Reconfiguration states and types.

TABLE 4. Reconfiguration time

Setting \ Metrics	Average (ms)	Type1 / Type2	(t_1-t_0) / (t_2-t_1) / (t_3-t_2) (ms)
Dynamic query workload	258	71% / 29%	9 / 697 / 1600
Dynamic sensor availability	2922	45% / 55%	1329 / 1039 / 2381

Last, we measure the energy consumption of the sensor devices using a multi-meter. Orchestrator continuously performs following operations on a sensor device: resource status monitoring, resource status transmission, and heartbeat message transmission. As shown in TABLE 3, the periodic execution of primitive operations is not much attributed to energy overhead (only 2.5mJ/s increase). All operations utilize CPU of the sensor under 1%.

Reconfiguration time. we measure the reconfiguration time for the adaptation under the dynamic query workloads and the dynamic sensor workloads, respectively. As shown in Fig. 25, the reconfiguration time includes the times for re-planning, executing new PPlans, and receiving initial data upon the occurrence of the system events, e.g., sensor join/leave events. A small reconfiguration time is preferable for seamless support for applications.

Table 3 shows the results on reconfiguration time. The average times are 258 and 2,922 milliseconds for dynamic query workloads and sensor workloads, respectively. They are reasonably short in that many daily applications are not sensitive to several-seconds delays and many personal contexts such as activities do not change quickly. Specifically, we classified reconfigurations into two types. The type 1 performs the plan generation and selection only, without plan execution and initial data reception; this case occurs when already running PPlans are reusable for a new query or no better PPlans are identified even with a new sensor. In type 2, all reconfiguration steps are performed; this case occurs when a new sensor provides better PPlans for existing queries or a new query should be processed with a currently inactivated PPlan. Under the dynamic query workloads, the type 1 occurs more often than the type 2 since PPlans can be shared by many queries; thus the reconfiguration time is very short.

7 Conclusion

In this paper, we described Orchestrator, a novel resource orchestration framework to support mobile context monitoring in a PAN-scale sensor-rich mobile platform. Orchestrator enables the platform to host multiple applications stably, exploiting its full resource capacity in a holistic manner. Thus, applications can provide users with seamless, long-running high-quality service under dynamic circumstances with limited resources. We present the design and implementation of Orchestrator running on off-the-shelf mobile devices and sensor motes, and also show its effectiveness in various system environments.

References

[1] E. Miluzzo, et al., “Sensing Meets Mobile Social Networks: The Design Implementation and Evaluation of the CenceMe Application,” Proc. SenSys, 2007.

[2] S. B. Eisenman, et al., “The BikeNet Mobile Sensing System for Cyclist Experience Mapping,” Proc. SenSys, 2007.

[3] B. D. Noble, et al., “Agile Application-Aware Adaptation for Mobility,” Proc. SOSP, 1997.

[4] J. Flinn and M. Satyanarayanan, “Energy-aware Adaptation for Mobile Applications,” Proc. SOSP, 1999.

[5] H. Zeng, X. Fan, and et al., “ECOSystem: Managing Energy as a First Class Operating System Resource,” Proc. ASPLOS, 2002.

[6] X. Liu et al, “Chameleon: Application-Level Power Management,” IEEE Trans. on Mobile Computing, vol. 7, no. 8, Aug. 2008.

[7] A. Lachenmann, P. J. Marrón, D. Minder, and K. Rothermel, “Meeting Lifetime Goals with Energy Levels,” Proc. SenSys, 2007.

[8] J. Sorber, et al., “Eon: A Language and Runtime System for Perpetual Systems,” Proc. SenSys, 2007.

[9] K. Lorincz, B. Chen, J. Waterman, G. W. Allen, and M. Welsh, “Resource Aware Programming in the Pixie OS,” Proc. SenSys, 2007.

[10] K. Seada, et al., “Energy-Efficient Forwarding Strategies for Geographic Routing in Lossy Wireless Sensor Networks,” SenSys, 2004.

[11] W. Heinzelman, A. Chandrakasan, and H. Balakrishnan, “Energy Efficient Communication Protocol for Wireless Microsensor Networks,” Proc. HICSS, 2000.

[12] C. Lombriser, D. Roggen, et al., “Titan: A Tiny Task Network for Dynamically Reconfigurable Heterogeneous Sensor Networks,” Proc. 15. Fachtagung Kommunikation in Verteilten Systemen (KiVS), 2007.

[13] C. Lombriser, et al., “Modeling Service-Oriented Context Processing in Dynamic Body Area Networks,” IEEE JSAC, Jan. 2009.

[14] P. Zappi, et al., “Activity Recognition from On-Body Sensors: Accuracy-Power Trade-Off by Dynamic Sensor Selection,” EWSN, 2008.

[15] K. Murao, T. Terada, Y. Takegawa, and S. Nishio, “A Context-Aware System that Changes Sensor Combinations Considering Energy Consumption,” Proc. Pervasive, 2008.

[16] S. Kang, et al., “SeeMon: Scalable and Energy-efficient Context Monitoring Framework for Sensor-rich Mobile Environments,” Proc. MobiSys, 2008.

[17] L. Bao and S.S. Intille, “Activity recognition from user-annotated acceleration data,” Proc. Pervasive, 2004.

[18] U. Maurer, et al., “Activity Recognition and Monitoring Using Multiple Sensors on Different Body Positions,” Proc. BSN, 2006.

[19] Weka 3: Data Mining Software in Java. http://www.cs.waikato.ac.nz/~ml/weka/

[20] P. Korpipää, et al., “Managing Context Information in Mobile Devices,” IEEE Pervasive Computing, 2003.

[21] H. Andreas, et al. “Emotion Recognition Using Bio-sensors: First Step towards an Automatic Systems,” LNCS 3068, 2004.

[22] V. Kostov and S. Fukuda, “Emotion in User Interface, Voice Interaction System,” Proc. IEEE International Conference on Systems, Man, and Cybernetics, 2000.

[23] M.T. Jones, T.L. Martin, B. Sawyer, “An Architecture for Electronic Textiles,” Proc. BodyNets, 2008.

[24] H. Lu, et al., “SoundSense: scalable sound sensing for people-centric applications on mobile phones,” Proc. MobiSys, 2009.

[25] Kiss FFT. http://kissfft.sourceforge.net/

[26] M. Ahn, et al., “Swan Boat: Pervasive Social Game to Enhance Treadmill Running,” Proc. ACM Multimedia (Demo), 2009.

[27] S. Kang, et al., “Orchestrator: An Active Resource Orchestration Framework for Mobile Context Monitoring in Sensor-rich Mobile Environments,” Proc. PerCom, 2010.

[28] T. Sohn et al., “Place-Its: A Study of Location-Based Reminders on Mobile Phones,” Proc. Conf. Ubiquitous Computing (UbiComp), 2005.

[29] Q. Li et al., “Accurate, Fast Fall Detection Using Gyroscopes and Accelerometer-Derived Posture Information,” Proc. BSN, 2009.

[30] U. Yildiz et al., “On Service Orchestration in Mobile Computing Environments”, Proc. SCC, 2008.

[31] Y. Wang et al., “A Framework of Energy Efficient Mobile Sensing for Automatic User State Recognition”, Proc. MobiSys, 2009.

[32] K. K. Rachuri et al., “SociableSense: exploring the trade-offs of adaptive sampling and computation offloading for social sensing”, Proc. MobiCom, 2011.

[33] N. Roy et al., “Dynamic selection of sensors based on multiple concurrent demands”, Proc. PerCom, 2011.

[34] T. Gu, et al., “A service-oriented middleware for building context-aware services”, Journal of Network and Computer Applications, Vol. 28, Issue 1, Mar. 2004.

[35] Y. Ju, et al., “SymPhoney: A Coordinated Sensing Flow Execution Engine for Concurrent Mobile Sensing Applications”, Proc. SenSys, 2012.

[36] M. Shin et al., “DEAMON: Energy-efficient sensor monitoring”, Proc. SECON., 2009

[37] PowerTutor. http://powertutor.org.

Youngki Lee is currently working toward the PhD degree in KAIST. His research interests include mobile and sensor systems, systems for city-scale services, and large-scale distributed systems and networking.

Chulhong Min is a PhD student at KAIST. His research interests include mobile and pervasive computing systems, ubiquitous services, mobile and sensor systems, and social and culture computing.

Younghyun Ju is a Ph.D. student at KAIST. His research interests include mobile and ubiquitous computing, system support for context-awareness and large-scale distributed systems.

Seungwoo Kang received the PhD degree in computer science department at KAIST. His research interests include mobile and ubiquitous computing.

Yunseok Rhee is a professor, at School of Electronics and Information Engineering, Hankuk University of Foreign Studies, Korea. His research interests include distributed computing, embedded systems.

Junehwa Song received the PhD degree in computer science from the University of Maryland at College Park. He is a professor in the Computer Science Department at KAIST. His research interests include mobile and ubiquitous systems, Internet technologies, and multimedia systems.