In this paper, we present Orchestrator, an active resource orchestration framework for mobile context monitoring. Emerging pervasive environments will introduce a PAN-scale sensor-rich mobile platform consisting of a mobile device and many wearable and space-embedded sensors. In such environments, it is challenging to enable multiple context-aware applications requiring continuous context monitoring to simultaneously run and share highly scarce and dynamic resources. Orchestrator enables multiple applications to effectively share the resources while exploiting the full capacity of overall system resources and providing high-quality service to users. For effective orchestration, we propose an active resource use orchestration approach that actively finds appropriate resource uses for applications and flexibly utilizes them depending on dynamic system conditions. Orchestrator is built upon a prototype platform that consists of off-the-shelf mobile devices and sensor motes. We present the detailed design, implementation, and evaluation of Orchestrator. The evaluation results show that Orchestrator enables applications in a resource-efficient way.

I. INTRODUCTION

Smart mobile devices have been the pivot for personal services. Many diverse sensors around the mobile device will enable highly proactive services with the help of a lot of personal context-aware applications, e.g., u-Trainer, dietary monitoring, and health monitoring. In such PAN-scale sensor-rich environments, mobile devices will serve as the common computing platform which accommodates diverse wearable sensors or nearby space-embedded sensors, e.g., e-watch, sensing garments, and textile electrodes in bed sheets. Then a mobile device will usually run multiple context-aware applications at a time which mainly focus on continuous monitoring of users’ context with diverse sensors [19]. The context monitoring often requires multi-step and complex processing over multiple devices at the same time, e.g., for a ‘running’ context, sensing of body-worn accelerometers, filtering, FFT-based feature extraction, and classifying the features through a decision tree [20].

The challenge in this new environment is that the platform should simultaneously support a number of applications with highly scarce and dynamic resources. Greedy and injudicious resource use would significantly aggravate contention among multiple applications and accelerate skewed use of some specific devices. This can also lead to reduced overall system capacity. Especially, we note that most of the devices have highly constrained resources, often less than the capacity required for a context monitoring. More challenging, availability of such resources dynamically changes due to their wearable forms and mobility of users. Sensors join or leave the platform frequently as users may take off a sensor-equipped smart watch [21], or enter a new sensor-embedded smart office. Also, resource usage by running applications or environmental factors such as interference continuously affects the resource availability.

It is almost impossible for each application to address the challenges without system-level supports. First, context monitoring entails very complex processing, and often involves burdensome device-level exploration [19][20]. Second, it is more challenging to ensure applications’ steady running under highly scarce and dynamic resource availability. To achieve this, each application should efficiently share the constrained resources with other applications. Without system-level support, however, each individual application has an extremely limited view of the existence or resource uses of other applications, and further cannot negotiate with the concurrent applications for coordinated and efficient resource utilization. Thus, system-level support is compelling for the effective and coordinated use of scarce computing resources among multiple applications.

In this paper, we propose a novel active resource orchestration system, Orchestrator for the PAN-scale sensor-rich mobile computing platform. Actively interplaying between context-aware applications and scarce dynamic resources, Orchestrator enables the mobile platform to host a number of applications stably, exploiting its full resource capacity in a holistic manner. More specifically, it helps applications efficiently share resources and processing as much as possible. It also seamlessly adapts the applications to dynamic resource availability by resolving resource contention between applications or selecting the best processing plan according to the resource availability at that time. In addition, Orchestrator can apply a system-level policy such as energy optimization with an integrated view of the total resource uses and status. With such system supports, applications become capable of providing mobile users with seamless, long-running high-quality service under dynamic circumstances with scarce resources.

To design effective resource orchestration, we take an active resource use orchestration approach that actively finds out the best combination of resources for requested context.
monitoring under the current status of resources and applications. This is substantially different from an existing approach, passive resource use management [3][4][7][10]. A key feature of the active approach is to decouple application’s logical resource demand from physical resource allocation. It enables postponing resource selection and binding until resources’ availability is sufficiently explored. Under this active approach, applications do nothing but turn in high-level context specifications to the system, and comply with the system’s decision on resource allocation. Most of mobile and sensor systems, on the other hand, have taken passive resource use management approaches based on application-driven decision in resource selection and binding. In such approaches, applications should explicitly specify the type and amount of resources as in resource ticket [10] and resource descriptor [3]. A system simply allocates the requested resources if available. If not, applications would change their resource use plans according to predefined code blocks, e.g., by trading off data or functional fidelity. However, these approaches impose huge burdens on the programmer, and their flexibility cannot be fully utilized in practice since it is almost impossible for applications to estimate dynamic resource status and prepare well adapted code blocks for all the cases.

Orchestrator realizes the proposed approach as follows. First, it prepares multiple alternative resource use plans, each of which can process a high-level context from applications. Such alternatives result from the diversity of semantic translation. That is, a context can be derived by utilizing many different sensing modalities as well as feature extraction and classification methods. For instance, when a context quality is required by an application is conditionally tolerable under a particular situation, a ‘running’ context is monitored with diverse methods, e.g., utilizing DC and energy features from acceleration data [20], or statistical features from GPS location data. Second, at runtime, it dynamically adapts a processing plan to reflect resource availability, running applications’ requests, system-level policies in a holistic manner. Such flexibility and adaptation enable Orchestrator to support multiple context-aware applications, extending their running time while balancing their resource usage, in environments with highly limited and dynamic resources.

To the best of our knowledge, Orchestrator is an initial attempt to provide an active resource orchestration system, recognizing the PAN-scale sensor-rich mobile platform as a common underlying computing platform. Recently, many systems have been proposed for effective resource management of mobile devices [3][4][5][6][7] and sensors [8][9][10] comprising PAN. They are mostly designed to manage resources, especially battery in most cases, for applications on a single computing device. Such device-centric resource management, however, can hardly be utilized in our target environment, in which multiple sensors and a mobile device cooperatively serve multiple applications.

This paper is organized as follows. We discuss related work in Section II. Section III and IV describe the architecture and main functions of Orchestrator, respectively. Section V presents the implementation of Orchestrator prototype. We discuss experimental results in Section VI. Finally, we conclude the paper in Section VII.

II. RELATED WORK

Recently, sensor-enabled mobile applications have been emerging in diverse application domains [1][2]. They provide useful mobile services based on the contexts of a user and her surroundings that are captured using diverse sensors such as accelerometers, microphones, and GPS. As many applications emerge to run on a mobile device with a variety of sensors, common underlying frameworks such as Orchestrator will be increasingly necessary to coordinate multiple applications and allow them to efficiently share resources in a systematic way.

There have been broad research efforts to effectively manage the use of limited resources in different environments. Some research proposed systems to manage the resources of a mobile device, to name a few, Odyssey [3][4][5], ECOSystem [6], and Chameleon [7]. Also, there have been ongoing research efforts to efficiently manage and utilize the limited resources of a sensor node such as Levels [8], Eon [9], and Pixie [10]. Some research has also been made to manage resource use in sensor networks. To increase the lifetime of the whole sensor network, algorithms for energy-efficient routing, filtering, and in-network processing have been proposed [11][12][13]. There has been, however, relatively little attention to develop a platform or to address issues for PAN-scale sensor-rich mobile environments. Orchestrator emphasizes a holistic view of resources of multiple devices as well as multiple applications that share the resources. Through the holistic approach, it effectively orchestrates the resource use of the applications.

Several systems such as Pixie [10], Odyssey [3], and Chameleon [7] have taken an application-driven approach to resource management. 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 provided APIs. The systems play a rather simple role to bind and allocate the use of the requested resources.

We consider that a system-driven approach [8][9][6] is more suitable as the solution of the target environment as it gets too complex and dynamic for individual applications to deal with. This approach hides from applications the details of complex resource management. As such, they can focus on specifying their application logics. Instead, systems determine and allocate the required resources at runtime, reflecting resource status and system policies. However, many of the existing systems with this approach are still application-aided, e.g., Eon [9], Levels [8]. Applications provide multiple program code blocks, each mapped to highly abstracted resource levels.

There has also been research to support adaptive context data provision. Titan [14][15] supports context recognition in dynamic BAN environments. However, it allows only a single recognition algorithm to run at a time. Activity recognition techniques proposed in [16][17][18] allow performing recognition processes adaptively to improve
energy efficiency. Their focus was on reducing energy consumption only for activity recognition. None of the mentioned research tackles challenging issues arising when multiple applications run simultaneously and share dynamic and scarce resources over multiple devices.

SeeMon [19] was our early attempt at a context monitoring platform for a PAN-scale sensor-rich mobile environment. While it proposed important techniques, e.g., locality-aware context processing and ESS (essential sensor set)-based energy control, it did not address complicated stream sources and sinks. Reflecting changing conditions, while it proposed important techniques, e.g., failure of components used, they reselect appropriate components to continue the execution of applications. However, they are not directly applicable to our target environments; they were developed and tested mainly in rather powerful desktop environments. Also, since their focus was on developing techniques to make a pervasive application adaptive to dynamic environments, they hardly consider problems due to scarce resources shared by multiple applications such as resolving resource contention and achieving high efficiency.

III. ORCHESTRATOR ARCHITECTURE

We design the architecture of Orchestrator to enable active resource use orchestration. The architecture spans a mobile device and multiple sensors (Fig. 1).

**Plan generation and selection:** The processing planner located in the mobile device plays a key role as a control center for resource use orchestration. It includes two major components: the plan generator and the plan selector. The plan generator dynamically creates a variety of applicable plans based on available sensors and their processing capability at runtime. Among the generated plans, the plan selector decides a set of plans to execute, which supports maximal application requests with available resources and best meets an orchestration policy enforced by the policy manager. The selection changes adaptively, reflecting dynamic resource status of sensors and the mobile device.

**Plan execution:** Orchestrator employs the plan processors in both the mobile device and sensors to execute the selected plans. They cooperatively process the plans, often involving sensing, feature extraction, and context recognition tasks. The plan processor in a sensor performs sensing tasks and optionally executes feature extraction tasks so that processing is offloaded onto the sensor. The plan processor in the mobile device executes the rest of the plan, i.e., feature extraction and context recognition tasks or the latter only, and completing plan execution. Note that we develop the plan processors to dynamically compose and share diverse processing modules for plan execution.

**Resource monitoring:** The resource monitors keep track of available resource status of sensors and a mobile device, respectively. They continuously monitor the status of CPU, memory, energy, and bandwidth. The status is periodically reported to the plan selector for runtime adaptation. The monitors are designed to minimize monitoring overhead while keeping reasonable level of accuracy.

**Communication protocols:** For effective cooperation between the mobile device and sensors, we develop a range of communication protocols. For example, a sensor detection protocol enables the mobile device to automatically detect available sensors. The detected sensors report heartbeat messages and resource status through a status reporting protocol. The mobile device sends diverse control messages based on a control message protocol.

**Application interface:** The application broker in the mobile device interacts with applications through Orchestrator APIs. Fig. 2 shows the APIs. Using registerCMQO(), applications specify contexts of interests as a form of a query statement and register the query along with two callback functions to receive query results and status.
The query specification is extended from the Context Monitoring Query presented in our previous work [19]. For 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).
```

Applications are notified of query results via the specified callback functions. Also, they are provided with query evaluation status, e.g., “not activated” in case that the processing planner cannot find any available processing plan for query evaluation. Note that the APIs allow application developers to specify high-level contexts only without caring about low-level resource status.

IV. ACTIVE RESOURCE USE ORCHESTRATION

A. Processing Plan Generation

The plan generator creates a wide variety of processing plans regarding each context to monitor. The plans utilize different combinations of devices and their resources, and thus, serve as a basis for effective orchestration. A key idea to obtain diverse plans is to exploit the diversity of context recognition methods and sensing modalities. For example, Orchestrator detects a ‘running’ activity with frequency-domain features from acceleration data and a decision tree classifier [20]. Another method is possible with time-domain statistical features and a Naïve Bayes classifier [21]. Also, each method can utilize different combinations of accelerometers located on various body positions [16][17][21]. Moreover, there are alternative ways to derive affective state of individuals with different sensing modalities. It is possible to recognize the state using biomedical sensors such as BVP, GSR, and ECG and features such as heart rate, heart rate variability, and inter-beat interval [24]. The state can be also identified by voice-related features such as pitch [25].

To generate diverse processing plans, we develop a two-phase translation method, including a logical translation and a physical translation. First, the logical translation maps a context into multiple logical processing plans, LPlans. An LPlan represents a set of processing modules to derive the context. It often consists of sensing, feature extraction, and recognition modules, and optionally includes the accuracy of context recognition. Second, the physical translation associates an LPlan with different sensors and computing resources to run the modules of the LPlan, accordingly generating physical processing plans, PPlans. Together with logical translation, this second translation increases the possibility to prepare for a number of resource usage options.

Fig. 3 shows example LPlans. The example, running activity and spine posture are monitored. There are two LPlans for the ‘running’ context. LPlan 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 utilizes acceleration data from a left thigh and a left wrist, extracts two time-domain statistical features, RMS (root mean square) and MAD (mean absolute deviation), and finally recognizes the context through a Naïve Bayes classifier. For the ‘spine posture’ context, there is an LPlan, LPlan.

New LPlans can be easily incorporated in Orchestrator by specifying them with the relevant description and adding corresponding modules. The plan processor is designed to flexibly adopt and execute new modules, which will be required to support the new LPlans.

The plan generator dynamically performs the physical translation for available LPlans. It is necessary since available sensors and their resource status continuously change at runtime. When sensors are newly registered or deregistered, the plan generator updates available PPlans with the sensors. Orchestrator readjusts plan execution with the updated PPlans.

To increase the number of available PPlans, the plan generator maximally exploits the possibility of diverse hardware resource mapping through a sensor mapping and a distribution mapping. The sensor mapping contributes to the diversity by examining different sensors that are eligible to serve adequate sensor data and processing modules for an LPlan. The distribution mapping exploits the different distributions of modules into sensors and a mobile device.

Fig. 4 shows examples of physical plans (PPlans) for LPlan.

Fig. 4 shows examples of logical plans (LPlans).
B. Holistic Resource-aware Plan Selection

The key of the effective orchestration is to properly select PPlans to execute among diverse alternatives. Through the plan selection, Orchestrator resolves contentions and maximizes sharing in resource use of multiple applications, and thus, supports application requests maximally with highly limited computing resources. At the same time, Orchestrator meets a system-wide policy for resource use in system operation, e.g., minimizing the total energy consumption.

For clear description of the plan selection, we define the selection problem as follows. Given \( C = \{ c_i \mid c_i \text{ is a context to monitor} \} \) and \( P = \{ p_{ij} \mid p_{ij} \text{ is a } j^{th} \text{ PPlan for a context } c_i \} \), the plan selector determines \( P_j \) a subset of \( P \) to execute. Among all possible subsets, \( P_j \) should support the most number of queries under given resource constraints while the cost of \( P_j \), \( \text{Cost}(P_j) \) is minimized. Here, the cost function, \( \text{Cost()} \), describes a system policy that should be satisfied to achieve desired system operation.

Fig. 5 shows overall 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.

In the first step, the plan selector detects the maximal PPlan sets, each of which supports maximum number of queries with available resources. To obtain the sets, the plan selector computes the number of supportable queries for every possible PPlan set, \( P_x \in 2^P \), and selects the ones supporting maximum queries. The computation involves three sub-steps as in the Fig. 5. In the first sub-step, the resource demand to execute \( P_x \) is calculated. Basically, it is computed by aggregating the resource demand of each plan. In the calculation, the module sharing among multiple plans are importantly considered; if more than two PPlans in \( P_x \) execute the same processing modules in the same device, the resource demand to execute the module is taken into account only once. In the second sub-step, it is examined if the resource demand of \( P_x \) exceeds the available amount of system resources. Through the process, it filters out the PPlan sets that violate resource constraints. The last sub-step calculates the number of supportable queries for the plan sets that pass the constraint check. Here, the accuracy conditions are also checked for the queries specified with minimum accuracy requirement. Finally, the plan selector selects the ones that support maximum number of queries.

In the second step, the plan selector determines the minimum cost set among the candidate maximal PPlan sets. It is likely that there often exist multiple maximal PPlan sets. This step calculates the cost corresponding to each set and selects the one with minimum cost. Here, diverse cost functions can be employed from the policy pool. Note that the module sharing is also considered in the cost calculation.

Orchestrator provides a couple of system primitives that abstract the resource demand of applications and the real-time resource availability of the system. First, a function, \( \text{GetRDMatrix}(P) \), provides the resource demand to execute a set of PPlans, \( P_x \), in the form of a matrix. Second, \( \text{GetRAMatrix}() \) returns the list of available devices and status of their resources such as CPU, memory and energy as a matrix. Fig. 6 shows examples of \( \text{RDMatrix} \) and \( \text{RAMatrix} \). Note that the system primitives facilitate the specification of cost functions for various policies. For example, the policy to minimize total energy consumption can be easily specified as shown in Fig. 7.

C. Dynamic Plan Adaptation

Orchestrator adapts to changes in resource availability and application requests at runtime by continuously altering \( P_x \), a set of PPlans to execute. The adaptation enables Orchestrator to support application requests seamlessly, to resolve newly occurring resource contentions, and to best meet the system policy continuously.

Orchestrator performs the plan selection process again when there are changes in a set of contexts to monitor, \( C \), or in the set of corresponding PPlans, \( P \). This is because the changes in \( C \) or \( P \) could disqualify the previously selected PPlan set from the best choice for execution. Also, changes in resource status could trigger the selection process regardless of changes in \( C \) or \( P \). For example, the energy drain of devices periodically triggers the process when...
Orchestrator applies a policy to minimize skewness among remaining energy of devices.

Five types of events trigger the selection process of Orchestrator. When an event occurs, Orchestrator performs the plan selection described in Section IV.B with new inputs, \( C_{\text{new}} \) and \( P_{\text{new}} \) changed by the event. Table I summarizes the event types and corresponding changes in the inputs of the plan selection. First, upon a registration of a query regarding a new context, \( C_{\text{new}} \) and \( P_{\text{new}} \) are reselected. Deregistration of a context is handled similarly as shown in Table I. Second, the join or leave of a sensor triggers the selection process as it changes the available PPlan set, \( P \). Some queries might be newly supported with the new PPlans, and other queries may not be supported anymore with old PPlans. Upon a join of a new sensor, the new available PPlans are inserted in \( P \) and the plan selection is executed. On the other hand, upon a leave of a sensor, the PPlans enabled by the sensor are removed from \( P \). Finally, in case of events by resource status changes, Orchestrator just reselects the PPlans with previous \( P \) and \( C \) once again.

For efficient adaptation, Orchestrator also adopts an incremental plan selection method. When there are a number of contexts to monitor and corresponding PPlans, it might be costly to reselect the whole \( P \) upon every event and redeploy new processing modules. To address the issue, the plan selector employs an incremental selection method as an effective heuristic solution. When the heuristic is used, Orchestrator periodically performs global selection to avoid errors that might accumulate due to repeated incremental selection.

The heuristic finds subset of contexts, \( C_{\text{sub}} \subseteq C \), and corresponding PPlans, \( P_{\text{sub}} \subseteq P \) that are directly affected by the triggering events. Then, it locally applies the selection process only for the subset. The selection is performed as follows upon each event. First, upon a query registration, Orchestrator performs the selection with only the requested context and corresponding PPlans. If the context has been already monitored, the executed PPlan is shared. Upon the query deregistration, Orchestrator simply stops executing the processing modules of the corresponding PPlans unless the modules are shared to support other queries; it does not perform plan selection again. Third, upon the sensor join, Orchestrator first generates new PPlans utilizing the sensor. In case that the new PPlans enable some of the queries that cannot be processed with the previously generated plans, Orchestrator executes the new PPlans. Also, if the new

<table>
<thead>
<tr>
<th>Event</th>
<th>Inputs ( (C_{\text{new}}, P_{\text{new}}) ) for plan reselection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registration of a query (regarding a context ( c_{\text{new}} ))</td>
<td>( C_{\text{new}} = {C_{\text{old}} \cup {c_{\text{new}}}) | ( P_{\text{new}} = P_{\text{old}} \cup {P_{\text{new}}}) is a ( j )-th PPlan for ( c_{\text{new}} )</td>
</tr>
<tr>
<td>Deregistration of a query (regarding a context, ( c_{\text{old}} ))</td>
<td>( C_{\text{new}} = C_{\text{old}} ) | ( P_{\text{new}} = P_{\text{old}} ) is a ( j )-th PPlan for ( c_{\text{old}} )</td>
</tr>
<tr>
<td>Join of a sensor ( (s_{\text{new}}) )</td>
<td>( P_{\text{new}} = P_{\text{old}} \cup {P_{\text{new}}}) is a ( j )-th PPlan for a context, ( c_{\text{new}} ) enabled by the new sensor, ( s_{\text{new}} )</td>
</tr>
<tr>
<td>Leave of a sensor ( (s_{\text{old}}) )</td>
<td>( P_{\text{new}} = P_{\text{old}} ) | ( P_{\text{new}} ) is a ( j )-th PPlan for a context, ( c_{\text{old}} ) that utilizes the lefted sensor, ( s_{\text{old}} )</td>
</tr>
<tr>
<td>Resource status changes, e.g., energy drain of devices</td>
<td>No changes in ( C ) and ( P )</td>
</tr>
</tbody>
</table>

PPlans are more cost-effective than the currently executed PPlans, the new PPlans replace the current ones. 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 currently executing plans with new ones.

D. Resource Detection, Monitoring and Demand Profiling

Orchestrator continuously updates RAMatrix and builds RDMatrix for resources orchestration. As a base to update RAMatrix, Orchestrator identifies sensors that become newly available or unavailable. Then, it updates RAMatrix by monitoring resource availability of the detected sensors and a mobile device in real-time. RDMatrix is generated through the off-line profiling. The following briefly describes the main ideas and methods implemented in Orchestrator.

**Dynamic Sensor Detection Protocol:** The protocol enables Orchestrator to dynamically detect available sensors in real-time. It is designed with a sensor-initiated approach, which achieves a high level of energy efficiency for energy-constrained sensors. The sensors need to turn on their radio transceivers only for a short period of time in order to send a heartbeat message and wait for corresponding ACK messages. We omit the protocol details.

**Resource Status Monitoring:** Resource monitors continuously track the availability of energy, CPU, memory, and bandwidth. For a mobile device, Orchestrator simply utilizes the resource information provided by resource monitoring tools in operating systems. For sensors, we develop our own monitors targeting MicaZ Motes.

For energy monitoring of sensors, Orchestrator adopts a voltage-based method [8]. It estimates remaining energy from voltage readings based on pre-built voltage-energy translation maps.

The CPU cycle is mostly occupied by two major operations: executing assigned tasks and handling timer interrupts for sensing, storing, and transmission (See Fig. 8). Among them, the CPU monitor only considers the CPU cycle for task execution. More specifically, the monitor measures CPU utilization as \( \frac{\sum t_{\text{active}}}{T_p} \). To measure \( t_{\text{active}} \), we record timestamps right before and after a task execution.

The available memory size, \( M_{\text{av}} \), is obtained as \( M_{\text{max}} - \sum M_{\text{used}}(\text{task}_i) \), where \( M_{\text{max}} \) is the maximum available memory and \( M_{\text{used}}(\text{task}_i) \) is memory used for a task. \( \sum M_{\text{used}}(\text{task}_i) \) is computed as \( \sum (M_{\text{CPU}}(\text{task}_i) + \sum M_{\text{mem}}(\text{task}_i)) \), where \( M_{\text{CPU}}(\text{task}_i) \) and \( M_{\text{mem}}(\text{task}_i) \) denote the size of Fixed and Temporary Space for a task, 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

![Figure 8: Example of CPU utilization.](image-url)
allocated and used temporarily only when the task is scheduled to run.

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.

Resource Demand Profiling: We collect resource demand profiles of processing modules used for PPlans before runtime. The total demand is calculated by adding up the demands of all relevant modules to the baseline resource demand.

V. IMPLEMENTATION

We have 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 1.1.1. The total lines are about 2,300.

We deployed the prototype system on various types of mobile devices and sensors. Fig. 9 shows a snapshot of currently used hardware. First, we have deployed the prototype on two different mobile devices, (1) Ultra Mobile PC (UMPC), SONY VAIO UX27LN with Intel® U1500 1.33 GHz CPU and 1GB RAM, and (2) a smart phone, Nokia N96 with Dual ARM9 264MHz processor and 128MB RAM. Second, we have incorporated various wireless sensors that have been widely adopted for context-aware applications (See Fig. 10 for sensor details). We used kiss_fft [26], a Fast Fourier Transform library, to derive frequency-domain features. It also provides a recognition module implementing a decision tree algorithm. To generate diverse plans to monitor activity contexts, 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, PRC, MAD). For sensor sets, we use all combinations of sensors on the left wrist, right wrist, right thigh, and waist. We trained the activity contexts via annotation-based learning [20]. The learning was done with C4.5 decision tree by Weka, a Java-based open source machine learning tool [22]. We implemented feature extractors on sensor nodes to offload feature extraction tasks. We used a highly optimized avr-fft library which is written in an assembly language for FFT computation on sensors.

A. Example Scenario and Operation of Orchestrator

We test and inspect the operations of Orchestrator under real situation. A student in our lab carried the UMPC and 6 sensors running an Orchestrator prototype while entering/leaving smart spaces or wearing/taking off wearable sensors on campus for 2 hours. Fig. 11 shows a short part of the processing log in his Orchestrator, describing query and sensor status according to changes in application requests and sensors. At 2:30 p.m., 6 sensors and 4 queries were registered. During 8 minutes, three events occurred; two sensor changes (E1 and E3) and an application request change (E2). Accordingly, processing plans were adaptively changed. For example, at 2:33 p.m., he entered a smart lab. Then, a processing plan using on-body sensors replaced the plan using space-embedded sensors for energy optimization.

VI. EVALUATION

A. Experimental Setup

We demonstrate the effectiveness of Orchestrator under dynamic changes in resources and application requests. For
the experiments, we used the UMPC and the eight USS-2400 sensor nodes described in Section V.

**Sensor availability:** To make the environment dynamic for experiments, we continuously vary the number of available sensors as shown in Fig. 12. For the total 80min period, we randomly add or delete a sensor every 2.5 min in average among 8 sensors. To examine the effect of number of available sensors, we divide the total time into four 20 min 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. 13. We add or delete a query every 1.5 min in average. Also, we set the MIN/MAX numbers of queries for each phase. Note that the queries are generated by carefully reflecting our previous example scenario. Table II summarizes the parameters and default values used for the query generation.

**Alternatives:** We compare Orchestrator with Conventional Context Recognizer (CCR) which models existing context-aware systems [23] lacking of resource orchestration functionalities. CCR supports a query with a single and fixed recognition method that provides the best recognition accuracy with designated devices. All employed sensors are always turned 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 nor deals with sensor leaves. For Orchestrator, we use the orchestration policy that minimizes the total energy consumption of available sensors as a default policy.

**Metrics:** We measure the effectiveness of Orchestrator in terms of application support and resource utilization. First, we measure the level of application support through *query activation*. We regard that a query is activated if a processing plan exists for the query and is executed with available resources. To quantify the query activation, we use Number of Activated Queries (NAQ) and Query Activation Ratio (QAR). QAR is formally defined as follows.

\[ QAR = \frac{\sum_{q_i} T_a(q_i)}{\sum_{q_i} T_r(q_i)} \]

where \( q_i \) is an i-th query in a registered query set, \( Q \) a query, \( T_a(q_i) \) total activation time of \( q_i \), and \( T_r(q_i) \) total registration time of \( q_i \).

As the metrics for resource utilization, we use Number of Activated Sensors (NAS) and the Energy Consumption (EC) of sensors in Joule (J). We regard that an available sensor is activated if the sensor executes certain tasks comprising any PPlan. In CCR, all available sensors are considered to be activated since CCR does not control the sensors. In general, higher NAS over time results in higher EC since the number of activated sensors is the dominant factor of energy consumption.

**B. Resource Orchestration under Dynamic Environments**

1) **Dynamic Sensor Availability**

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

Fig. 14 (a) shows QAR and EC per phase where more sensors are available as phase changes from A to D. In the graph, we can observe the key characteristics of Orchestrator. Under phase A and B where available sensors are scarce, Orchestrator activates more queries than CCR, while consuming almost the same amount of energy. It demonstrates that Orchestrator coordinates resource use well to support applications under resource-scarce environments. In contrast, under phase C and D where available sensors are plentiful, Orchestrator concentrates on energy optimization since the QAR of Orchestrator is already saturated to the maximum, 1. It shows that Orchestrator coordinates resource use well to maximize utilization under resource-abundant environments.

Fig. 14 (b) shows QAR and EC for total experiment time. To sum up, Orchestrator achieves better QAR (95% improvement) and less EC (10.7% reduction) than CCR. Fig. 14 (c) and (d) show NAQ and NAS over time, respectively. Most importantly, the NAQ of Orchestrator is always much higher than that of CCR, resulting in higher QAR. This is
because Orchestrator prepares for and utilizes diverse PPlans such that it is highly likely to support application requests with diverse combinations of sensors. Also, the NAS of Orchestrator is lower than that of CCR. Orchestrator selects PPlans in a way to minimize the number of activated sensors applying the energy optimization policy. Specifically, we found that Orchestrator selects PPlans that run feature extraction tasks in sensors rather than those that send only raw data from sensors. Thus, sensor-based feature extraction significantly reduces the overall communication cost, which in turn reduces energy consumption.

2) Dynamic Query Workload

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

Fig. 15 (a) shows EC and QAR per phase where more sensors are available as phase changes from A to D. Since sensor resources are abundant in all phases, the QAR of both Orchestrator and CCR are saturated to the maximum, 1. Meanwhile, the ECs of Orchestrator and CCR are kept high, i.e., about 200J. We look into EC per sensor, and discover that sensor 4 consumes more energy than other sensors. In our experimental setting, sensor 3, 4, 5, and 6 are accelerometers for activity recognition. Based on energy minimizing policy, Orchestrator tried to utilize sensor 4 rather than 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 which is known as the most suitable position for recognizing activities such as running, walking, and standing [20].

Fig. 15 (b) shows EC and QAR for total experiment time. Orchestrator achieves better EC (15% reduction) along with slightly better QAR (8.5% improvement) than CCR. Also, Fig. 15 (c) and (d) show NAQ and NAS over time in detail. 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. Thus, it does not meet the accuracy requirements of some queries. More important, the NAS of Orchestrator is always lower than that of CCR. This is because all available sensors are activated in CCR whereas Orchestrator selectively utilizes the sensors. It results in lower energy consumption, and better resource utilization eventually.

3) Orchestration Cost

We examine orchestration cost under the dynamic sensor and query workloads in terms of communication and memory cost.

First, we measure the number of messages exchanged during the total experiment time. To investigate the communication cost, we classify the messages into three types: control messages, heartbeat messages including sensor status information, and data reporting messages. Fig. 16 shows the number of messages for each message type. In Orchestrator, the control and heartbeat messages are additionally exchanged for active resource use orchestration, thereby incurring a slight communication cost, e.g., about 1.5 kbps for the dynamic query case. However, Orchestrator significantly reduces the number of data messages at the cost of those messages. Compared to CCR, it is decreased up to 10 times. Note that the number of control and heartbeat messages is controllable by configuring the heartbeat period.

Also, we measure the memory size consumed at runtime, which is averaged over time. The average memory sizes in the mobile device are 59.6KB under the dynamic sensor availability and 56.2KB under the dynamic query workload, respectively. The processing planner, a core component for orchestration in the mobile device, consumes 31.5KB and 30.6KB, respectively. Considering the memory capacity of the mobile device, they are negligible. In addition, each sensor consumes 276B of memory to maintain data structures storing executed tasks. The average sizes used in sensors vary from 278B to 7698 depending on sensor type; the plan processor uses 512B in accelerometers and 4B in other sensors to buffer sensor data. Note that those values vary depending on executed tasks and runtime parameters such as window size. Considering the memory capacity of MicaZ, 4KB, and the memory taken by TinyOS, 329B, Orchestrator can offload multiple tasks such as feature extraction onto sensors.

C. Effects of Resource Orchestration Policies

We determine if Orchestrator can properly apply multiple orchestration policies. We consider two sample policies: (1) minimizing the total energy consumption of available sensors and (2) maximizing the average accuracy of
registered queries. For the experiment, we use the dynamic query workload and fix the number of sensors to 6. Then, we measure the NAS and the average accuracy of activated queries over time for the two policies, respectively. We only consider the queries with activity contexts for accuracy measurement.

From the experiment, we could see Orchestrator supports the two policies well. We could observe that the NAS of the first policy is much lower than that of the second during the whole experiment time. Accordingly, the energy consumption is lower with the first policy, which shows the desired operation of Orchestrator. On the other hand, higher accuracy is achieved when the second policy is used, i.e., about 4% increases in average.

VII. 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.

ACKNOWLEDGMENT

We thank our shepherd, David J. Yates, and the anonymous reviewers for their valuable comments to improve the quality of this paper. This work was partially supported by Defense Acquisition Program Administration and Agency for Defense Development, Korea under the contract.

REFERENCES