Systems Support for Pervasive Query Processing

Wenwei Xue Qiong Luo Lionel M. Ni
Department of Computer Science
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong
{wwxue, luo, ni}@cs.ust.hk

Abstract

*Database queries, in particular, event-driven continuous queries, are useful for many pervasive computing applications, such as video surveillance. In order to enable these applications, we have developed a pervasive query processing framework called **Aorta**. Unlike traditional database systems, a pervasive query processor requires systems support for managing a large number of networked, heterogeneous devices. In this paper, we present the communication, synchronization, and scheduling mechanisms in Aorta. Even though these techniques have their roots in distributed and parallel systems, we show how these techniques are customized and applied for pervasive query processing. In essence, communication between heterogeneous devices enables network data independence, synchronization on devices protects action atomicity, and scheduling works for adaptive, cost-based multi-query optimization. We have conducted empirical studies on our prototype as well as simulation studies to evaluate the system performance.*

1. Introduction

Pervasive (ubiquitous) computing is an exciting vision in which various kinds of networked computing devices "live" with people in the real world [19]. Monitoring physical-world phenomena (*events*) of interest, pervasive computing applications integrate data acquisition, communication and operations (*actions*) on small devices that are embedded or mobile in the world. For instance, in a building monitoring scenario, a surveillance application automatically operates remotely-controllable cameras to take photos based on the variation in the readings of acceleration sensors. In the meanwhile, it sends the photos to the cell phone of the human manager who may be currently off-duty. Due to the heterogeneous and resource-constrained nature of the devices involved, to date pervasive computing applications are usually difficult to develop and optimize, and their functionality and usefulness are limited.

Recent work in the database and networking areas, including TinyDB [12], Cougar [4][5][21], Direct Diffusion [7][10] and SINA [16], has illustrated the effectiveness of using declarative database queries in querying and tasking a network of sensors. Inspired by these systems, we propose to use database queries for the development and optimization of pervasive computing applications. Specifically, we developed a pervasive query processing framework called *Aorta* to enable these applications. Using this framework, an application can issue SQL-style queries embedded with complex actions on devices (e.g., take a photo towards a location on a camera). These actions embedded in the queries will be executed automatically in response to physical-world events (e.g., object movement).

In this paper, we focus on the systems support issues in Aorta. Specifically, we have designed and implemented a number of mechanisms that are customized for pervasive query processing: (1) a uniform data communication layer that enables device *network data independence* [9], (2) simple locking and probing mechanisms for device synchronization, and (3) two heuristic scheduling algorithms for adaptive, cost-based multi-query optimization.

The remainder of this paper is organized as follows. We give an overview of our query processing framework in Section 2. We present the design of the uniform data communication layer in Section 3. In Section 4, we describe the device synchronization mechanisms we have implemented in Aorta. In Section 5, we propose heuristic algorithms for the scheduling of multiple concurrent action requests. In Section 6, we evaluate the performance of the Aorta system prototype using both empirical studies and simulation studies. We discuss related work in Section 7 and conclude in Section 8.

Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICSCS'05)
1063-6927/05 $20.00 © 2005 **IEEE**

2. Aorta query processing

To give the background of the systems support, in this section we briefly introduce the Aorta pervasive query processing framework. We omit many details about query processing and present only necessary contents to make this paper self-contained. The readers can refer to a previous paper of ours [20] for more thorough description of the action-oriented query optimization and execution techniques in Aorta.

2.1. Architecture of Aorta

The architecture of the Aorta system consists of three major layers.

The top layer of Aorta is a declarative interface that allows pervasive computing applications to specify actions on devices through database queries. This interface alleviates the problem of programmers having to handle various programming APIs for different types or models of devices. We extend the SQL language to allow specifications of user-defined actions and provide a library of system built-in actions for accessing and operating devices.

The bottom layer of Aorta is a uniform data communication layer that manages a number of networked, heterogeneous devices. This layer ensures that the Aorta system, not the individual applications, is responsible for monitoring and tuning the current network infrastructure and the physical status of the devices. Applications only need to see an abstract view of the devices. This abstract view enables application programmers to pay their full attention to the application semantics of their tasks rather than worrying about the lower-level implementation issues, such as data transmission loss, action failure and resource consumption on the devices.

Finally, there is an *action-oriented* query processing engine for queries embedded with actions in the middle. Being the core of our framework, this engine is responsible for generating, optimizing and executing query plans. It interacts with the communication layer to adapt to current device status and network workload. It also provides mechanisms to enable device synchronization and action workload scheduling among multiple queries.

2.2. Declarative application interface

Many pervasive computing applications have an *event-driven* and *action-oriented* processing nature: when the application detects an event, a pre-defined action on some type of devices is triggered. Based on this observation, we implemented a declarative interface in Aorta for applications to specify actions and queries with actions embedded. We call this kind of queries *action-embedded queries*.

Actions are Aorta system built-in or user-defined functions that operate devices. For a user-defined action, the user must pre-compile the code block of the action into a dynamically linked library, and use the CREATE ACTION command provided by the declarative interface to register it along with an action profile to Aorta. An *action profile* is an XML text file that describes the high-level semantics of the action and will be used in query optimization. As an example, the following command registers a user-defined action sendphoto(), which sends a photo (image file) to a phone with MMS support.

```
CREATE ACTION sendphoto(String phone_no,
                        String photo_pathname)
AS      "lib/users/sendphoto.dll"
PROFILE "profiles/users/sendphoto.xml"
```

After an action is registered to Aorta, applications can specify and register queries with the action embedded using the CERATE AQ command. Figure 1 shows an example action-embedded query in Aorta.

```
CREATE AQ snapshot AS
        SELECT  photo(c.ip, s.loc, "photos/admin")
        FROM    sensor s, camera c
        WHERE   s.accel_x > 500
        AND     coverage(c.id, s.loc)
```

Figure 1. The example snapshot query

As illustrated in this example, the specification of an action-embedded query appears to be an event-driven continuous query with a name. In this query, the system-provided action *photo(camera_ip, location, directory)* operates the camera with IP address *camera_ip* to move its head to the direction pointing to *location* and to take a medium-size photo; then the action stores the photo to *directory*. The constraint *s.accel_x > 500* in the query condition specifies the events of interest (e.g., someone pushes the door and causes a movement of the door together with the sensor attached on it). The set of candidate cameras that are suitable for executing the photo() action whenever an event is detected is specified using the system-provided Boolean function *coverage (camera_id, location)*. This Boolean function returns TRUE if the camera with ID *camera_id* has a view range that covers *location*.

2.3. Cost-based query optimization

As shown in the example in Figure 1, the snapshot query states that whenever a sensor detects a

movement at a certain speed, all cameras are examined to see whether they can take a photo towards the sensor's location (e.g., whether they are candidates for executing the photo() action upon this event). However, even if multiple candidate devices are available to take the action, it is sufficient to let some, instead of all, devices to take the action. This application semantics in pervasive computing brings opportunities to query optimization. In Aorta, the query optimizer will select the current "best" camera in terms of cost among all candidates to execute the action.

The cost-based device selection optimization is necessary because the cost of an action execution on different candidate devices may be different. Furthermore, if a device is selected for too much workload, it may slow down or malfunction. For generality, we define the cost of an action on a device as the execution time of the action on the device.

Consider the cost of the system built-in action photo() in Figure 1. On the AXIS 2130(R) PTZ network cameras [3] we used, the time spent on moving the head of a camera to a target position depends on the current position of the head (the pan, tilt, zoom values). As a result, when a photo() action is executed on a camera, the starting head position of the camera affects the execution time (cost) of the action on it. Moreover, the execution of a photo() action moves the head of the camera to a new position, which in turn affects the cost of the subsequent photo() action to be executed on it.

In this example we see that, the cost of an action execution on a device may depend on the *current physical status* of the device. Furthermore, an action execution may change the current physical status of the device and in turn the cost of subsequent action executions on the device. In the photo example, the current physical status of a camera that is related to the photo() action is the camera head position. For other actions on other types of devices, the device physical status concerned may be different, e.g., the depth of a sensor in a multi-hop network affects the cost of connecting the sensor for some operation on it.

To enable cost-based device selection optimization for action execution, we put actions as first-class citizens (query operators) inside query execution plans. An action operator contains the name, the input parameters, and the pointer to the method to be executed. Furthermore, we make concurrent queries that have the same embedded action (the name of the function is the same, but the input parameter values may be different) share a single action operator in their query plans. We add the query ID to the input tuples of a query so that the operator knows which tuples are for which query. Such action operator sharing saves system resources and facilitates group optimization of actions.

We have proposed a cost model for the optimizer to estimate the cost of an action execution on a candidate device. The core component of the cost model is the action profile, which specifies the composition of an action in terms of the sequential and/or parallel execution of a number of atomic operations. The costs of atomic operations are obtained from empirical measurements. The cost of an action is then estimated based on the action profile and the estimated costs of the atomic operations on the type of devices. Our results from a number of experiments have validated that our cost model is reasonably accurate. The cost model is also used by the two heuristic scheduling algorithms that we propose in Section 5.

In addition to the query optimization and execution issues, a number of systems issues must be addressed to support pervasive query processing in our framework. Specifically, pervasive query processing requires data communication across networks of heterogeneous devices, synchronization on the devices, and scheduling action workload among the devices. The design and implementation of these systems support mechanisms is the focus of this paper.

In the following sections, we present in detail our design considerations and approaches for the data communication, device synchronization and action workload scheduling mechanisms in Aorta.

3. Uniform data communication layer

Pervasive computing applications usually involve heterogeneous devices that have completely different hardware architectures and software APIs. Moreover, even for the same type of devices, the capabilities and interfaces of different models may vary widely. Examples of these devices include smart sensors, network cameras, cell phones, smart cards, and so on.

We have designed and implemented a uniform data communication layer in Aorta. The goal of the communication layer is to handle heterogeneous networking protocols and to provide a dynamic, logical view of networked devices for applications. This layer mainly consists of three components: (1) the profiles of devices, (2) the scan operators for devices, and (3) a set of basic communication methods for each type of devices. We describe these components in order. For brevity, in the remainder of the paper we say "a type of devices" in short for "a type or model of devices".

3.1. Device profiles

The communication layer maintains the profiles of devices. In Aorta, we use device profiles to describe the physical characteristics of devices. These profiles are generated and registered to the system and are updated dynamically by the system administrator.

A typical example of profiles that are common to all types of devices involved in Aorta is the *device catalog*. A device catalog is an XML text file that keeps the names of the attributes supported by the type of devices (e.g., temperature and light for sensors), the pointers to the system built-in methods for acquiring the values of the attributes, and the information about the semantics and properties of the attributes.

Furthermore, for each type of devices, there is also an atomic_operation_cost.xml file included in its profiles. This file lists all atomic operations on the type of devices and their corresponding estimated costs. We define an *atomic operation* as the smallest unit of operation that a type of devices can perform. Examples of atomic operations on devices include "take a photo of a specified size (small, medium or large)" on cameras, "receive an SMS/MMS message" on phones, and "beep/blink once" on sensors.

The estimated cost of an atomic operation is measured by our homegrown programs using some cost metric; the cost metric we currently use is the time required to finish the operation. Our tests show that an atomic operation has almost the same cost on devices of the same type. These estimated costs will be used by the query engine in query optimization.

3.2. Scan operators for devices

The communication layer abstracts each type of devices into a virtual relational table. It then provides special "scan operators" as simple interfaces for the query engine to acquire device data tuples from these virtual tables. Each tuple of a virtual device table (e.g., the sensor table) is from a specific device of the corresponding type; it is generated on-the-fly when requested by the query engine. Such virtual table abstraction has been widely adopted by previous work [12][21] in sensor databases and proved to be effective for encapsulating device data. With such abstraction, the query engine can regard the underlying device network as a distributed database consisting of various horizontally partitioned device tables.

The attributes of a virtual device table can be either real-time data such as sensor readings and camera video feeds, or static data such as sensor locations (we assume the location of a sensor is fixed), camera IP addresses and phone numbers. We call the former *sensory attributes* and the latter *non-sensory attributes*. We categorize the attributes that describe device status (e.g., sensor or phone battery voltages, camera zoom level) into sensory attributes, since their values are acquired by "sensing" the current physical status of the devices. Consequently, the implementation of a scan operator on different attributes varies by the categories of the attributes. Specifically, sensory data must be acquired dynamically whereas non-sensory data may be stored statically.

3.3. Basic communication methods

In order to enable data acquisition and transmission across heterogeneous devices, the communication layer implements a common interface that defines a set of basic communication methods such as connect(), close(), send() and receive(). These methods wrap around the heterogeneous networking protocols of the various types of devices in Aorta. Each type of devices inherits this interface in its own communication module, with a specific implementation based on the software programming APIs supported by the type of devices.

These basic communication methods are the building blocks of many query operators in the query engine, for example, the scan operators and action operators. The uniform interface provided by these methods shields the engine from the heterogeneity and complexity of the underlying device network.

The design of the device profiles, scan operators and basic communication methods all follow a generic structure. Our consideration is to make the communication layer easily extensible for new types of devices in the future.

4. Device synchronization

Device synchronization is another important issue in pervasive computing systems design. The major goal of device synchronization is to ensure the correct application semantics of individual actions executed on unreliable physical devices. Our current Aorta system prototype mainly includes two device synchronization mechanisms: (1) a locking mechanism, and (2) a probing mechanism.

The locking mechanism is used to avoid concurrent execution of multiple actions on a single device. When a device has been selected to execute an action, the optimizer will lock it until it finishes executing the action (i.e., the call to the code block of the action method returns). Subsequent actions on this device cannot start before the device is unlocked.

To illustrate the necessity of such locking mechanism, let us consider queries embedded with the photo() action again. Suppose a camera has been

selected by the optimizer to execute a photo() action for a query among all candidates. When the camera is executing the action, it is selected to execute a photo() action for another query. As a result, the head of the camera will be redirected to the second target position before it finishes taking the first photo. Such interference between concurrent photo() actions on a device leads to either unclear (blurred) photos or photos taken at wrong positions. Furthermore, it is possible that when a camera is busy with the first action, it will fail to execute the second action or has a very long delay for it. We observed all of these problems in practice and our locking mechanism eliminated these problems.

The probing mechanism is for the optimizer to examine each candidate before deciding whether it should be included in the device selection optimization. A probe on a candidate device includes the transmission of several messages between the optimizer and the device.

The major role of the probing mechanism is to check the current availability of a candidate device. This availability checking is necessary because physical devices in pervasive computing are intrinsically unreliable. They may join, move around, or leave the network dynamically in a way unpredictable to the system. As examples, the current generation sensors usually communicate via a wireless radio channel of a high packet loss rate [6]; a camera may suffer from network connection delay and produce blurred photos occasionally; and a phone may become unreachable when its owner moves into an area that is out of the coverage of the service provider.

Consequently, the optimizer must establish connection to a candidate device and make sure it is currently available before proceeding to estimate its cost for executing an action. A system-provided TIMEOUT value is set for each type of devices to break the probe on unresponsive devices. These malfunctioning devices will be automatically excluded in the device selection optimization.

Additionally, by probing a candidate device the optimizer can gather information about the current physical status of the device. This information is useful and even necessary in the device selection optimization. What kind of device physical status is concerned and how it is considered in the optimization is specified in the action profile.

5. Action workload scheduling

In this section, we discuss the problem of action workload scheduling in Aorta. As action-embedded queries are continuous queries with a long-running nature, it is expected that there will always be a large number of queries running concurrently in the system. In this scenario, multiple action requests from different queries may appear in the optimizer at the same time or within a short time interval. Consequently, a scheduling mechanism is necessary to distribute multiple action requests to the devices that are currently available. We define an *action request* as the request from a query for the execution of an action with instantiated input parameter values for the action.

We first formulate the action workload scheduling problem in Aorta in Section 5.1. We then present two heuristic algorithms to solve the problem in Section 5.2.

5.1. Problem formulation

Given multiple action requests that appear in a shared action operator, the ***Action Workload Scheduling Problem*** we consider in Aorta is formulated in Figure 2. The object function of the problem is to minimize the maximum completion time (or called *makespan*) of the set R of action requests. The completion time is defined as the interval between the time when these requests appear in the shared action operator and the time when all of them have been serviced. We chose this object function because our goal is to balance the action workload on all available devices and improve device utilization.

Problem: Action Workload Scheduling

Input: A set R of n action requests ($r_1, r_2, \ldots, r_n$).
A set D of m devices ($d_1, d_2, \ldots, d_m$) that is available for servicing some requests in R.
Each $r_i \in R$ ($1 \le i \le n$) has a candidate device set $D_i \subseteq D$.
Each pair of (r_i, d_j) ($d_j \in D_i$) corresponds to a weight that is equal to the cost of servicing request r_i (i.e., executing the action) on device d_j.

Output: A schedule of R on D (each $r_i \in R$ is assigned to and serviced by a device $d \in D_i$) with the makespan of R minimized.

Figure 2. The action workload scheduling problem in Aorta

Our action workload scheduling problem can be reduced to the classic makespan minimization problem in scheduling theory on unrelated parallel machines with sequence-dependent job setup time and machine eligibility restrictions [13]. The classic problem is known to be NP-hard. Although there are a large number of algorithms proposed in the operational research area for various kinds of scheduling problems, to the best of our knowledge there is little previous work that considers a problem that is equivalent to our workload scheduling problem. The Simulated

Annealing (SA) algorithm proposed by Anagnostopoulos et al. [2] is the only one we know in the literature that has considered all restrictions in the algorithm design (unrelated machines, the sequence-dependent job setup time, and the machine eligibility restrictions for jobs.

In comparison with the classic problem, our problem has the following two special features that are unique in pervasive computing:

1) **Physical status change on devices.** As described in Section 2.3, after executing an action, the current physical status of a device may change, which will in turn change the cost of the subsequent action executed on the device. This concept is similar to the sequence-dependent job setup time in the classic problem. However, in our problem the case is sequence-dependent action execution time instead.

2) **Real-time processing requirement in pervasive computing.** As events in pervasive computing are often transient (e.g., object movement), the action upon an event should be executed in a real-time fashion in order to respond to the event promptly. Consequently, the computational cost of our scheduling algorithm must be small even if the given input size is large. Many existing algorithms in scheduling theory have a considerable computational cost, so they are inapplicable in our scenario.

As a result, we decide to design and implement our own heuristic algorithms to solve the problem.

5.2. Heuristic scheduling algorithms

Existing algorithms for the classic parallel machine scheduling problems can be approximately divided into two categories: (1) algorithms with concurrent job assignment and processing, and (2) algorithms with sequential job assignment and processing. We call scheduling algorithms in these two categories CAP and SAP algorithms, respectively.

For the CAP scheduling algorithms, the assignment and processing of the jobs are performed in parallel. After a job is assigned to a machine, it is immediately executed on the machine. In comparison, for the SAP algorithms, these two procedures are performed in sequence. A job assigned to a machine is queued on the machine first, and may be reassigned to another machine later before the assignment procedure finishes. The processing procedure will not start until the assignment procedure finishes.

As typical examples, the SA algorithm [2] is a SAP algorithm and the well-known List Scheduling (LS) greedy algorithm in scheduling theory [13] is a CAP algorithm. Whenever a machine becomes idle, the LS algorithm schedules any eligible job that has not yet been scheduled on the machine.

Based on this SAP/CAP classification of the scheduling algorithms, we have designed and implemented two heuristic algorithms to solve the action workload scheduling problem in Aorta. They are shown in Figure 3. Algorithm 1 is SAP and Algorithm 2 is CAP. We proposed these two different algorithms to compare and evaluate their performance under different workloads.

Algorithm 1 consists of two subcomponents, which we call LERFA (Least Eligible Request First Assignment) and SRFE (Shortest Request First Execution). Both of them are greedy algorithms.

LERFA performs the assignment with the heuristics being the number of candidate devices of each request. It starts with the request that has the least number of candidate devices (i.e., the least eligible request), and assigns the request to the candidate device that will have the minimum total estimated workload if this request is serviced on the device. It then goes on to assign the next least eligible request until it finishes the assignment of all requests to devices. If two requests have the same number of candidate devices, LERFA assigns them in a random order.

SRFE prioritizes and services the requests that have been assigned to a single device. It always selects the request with the least estimated cost (i.e., the shortest request) for the device to service. As the execution of an action may change the physical status of a device, the physical status of the device is updated before selecting each new request for the device to service.

Algorithm 2 is also a greedy algorithm with the heuristic being servicing the shortest request first. Consequently, we call the algorithm SRFAE (Shortest Request First Assignment and Execution). In each round of the assignment and execution, the algorithm always selects an unserviced request with the least estimated cost on some device, assigns it to and services it immediately on the device. If the device is currently busy with a previously assigned request, the newly assigned request is queued in the device's request input queue. The requests queued in the input queue of a device are serviced in the FIFO order.

After an action request is assigned to a device and serviced or queued on the device, SRFAE updates the assigned workload of the device. The estimated costs of other unserviced requests that can be serviced by the device are also recalculated to reflect the workload increase on the device. Such cost recalculation is based on the new physical status of the device after servicing the newly assigned request. The purpose is to provide accurate information for the next round of assignment and execution.

Algorithm 1.1: Least Eligible Request First Assignment

1. **for** each device d_j ($1 \le j \le m$) in D **do**
2. initialize its assigned workload $W_j = 0$;
3. $i = 1$;
4. **while** there are unassigned requests **do**
5. **for** each request r that has i candidate devices **do**
6. **for** each candidate device d_k of r **do**
7. C_{rk} = the estimated cost for servicing r on d_k;
8. $E_k = W_k + C_{rk}$;
9. select the device d_l that has the least E value among the i candidate devices of r and assign r to d_l;
10. C_{rl} = the estimated cost for servicing r on d_l;
11. W_l += C_{rl};
12. i ++;

Algorithm 1.2: Shortest Request First Execution (on a single device *d*)

1. lock d;
2. **while** there are unserviced requests **do**
3. update the current physical status of d;
4. **for** each remaining request r **do**
5. C_r = the estimated cost for servicing r at this moment;
6. select the request with the least C value and service it;
7. unlock d;

Algorithm 2: Shortest Request First Assignment and Execution

1. **for** each request r_i ($1 \le i \le n$) in R **do**
2. **for** each device d_j in D_i **do**
3. insert (r_i, d_j) as a node in a balanced binary search tree T, the key of the node is the weight of this request-device pair;
4. **for** each device d_j ($1 \le j \le m$) in D **do**
5. initialize its assigned workload $W_j = 0$;
6. lock d_j;
7. **while** T is not empty **do**
8. extract the node a from T that has the least key value;
9. Find (r_i, d_j) that a corresponds to and assign request r_i to device d_j;
10. **if** d_j is currently free **then**
11. service r_i on d_j;
12. **else**
13. queue r_i in the input queue of d_j;
14. w = the key value of a;
15. delete a from T and mark r_i as serviced;
16. **for** each unserviced request r_l in R **do**
17. **if** $d_j \in D_l$ **then**
18. C_{lj} = the estimated cost for servicing r_l on d_j after servicing r_i;
19. a_l = the node that corresponds to (r_l, d_j) in T;
20. update the key value of a_l to be $C_{lj} + w$;
21. **for** each device d_j ($1 \le j \le m$) in D **do**
22. unlock d_j;

Figure 3. Heuristic algorithms for action workload scheduling

Note that both our algorithms and the existing SA and LS algorithms result in a schedule that is nearly optimal but not guaranteed to be optimal. In general, scheduling problems can be formulated into a 0/1 Mixed Integer Program (MIP) and be solved optimally by solving the MIP [2]. However, as our problem is NP-hard, the optimal MIP is too computationally expensive to be feasible in our scenario even if the given input size is small. This expensiveness has been verified by the results of Anagnostopoulos et al. [2]: with an input size n = 4 and m = 8, the optimal MIP required nearly one and a half hour running time on a machine with a 1GHz CPU.

6. Experiments

In order to evaluate the performance of our Aorta system prototype, we have developed an action-enabled monitoring application on it for the pervasive lab in our department. The pervasive lab is established to accommodate cross-area research activities on pervasive computing. It has desktops with removable hard disks, and various types of devices such as sensors, cameras, and phones.

Without loss of generality, we mainly present the results of our empirical and simulation studies for queries embedded with the photo() action.

6.1. Experimental setup

The experiments involved one Pentium R(M) 1.5GHz notebook with 512MB memory, two AXIS 2130(R) PTZ network cameras [3], and ten Berkeley MICA2 motes with MTS310CA sensor boards [6]. The two cameras were mounted on the ceiling of the pervasive lab. The ten motes were put at ten different places of interest in the lab. The location of each mote was in the view range of at least one camera. To ensure that photos towards the same location taken by the two cameras had almost the same visual quality (i.e., view size), we configured each camera to automatically tune its zoom level based on the distance between itself and the target location.

Our Aorta system prototype with the pervasive lab monitoring application was implemented in Java (JDK 1.4.1). A package that contains the source code and configuration files of the system is available at the Aorta project web site http://www.cs.ust.hk/aorta.

6.2. Effects of device synchronization

We first show the effects of the device synchronization mechanism we implemented in Aorta using results from empirical studies. We generated 10

queries embedded with the photo() action and registered them with the system in a batch. In this workload, a photo of Mote i's location was required to be taken by the i-th query every minute ($1 \le i \le 10$).

We found that without device synchronization, in each minute the photo() action requests of the 10 queries interfered with one another (i.e., mixed their operations on the two cameras) in an unpredictable fashion. More than half of the action requests failed (i.e., connection to the camera timed out), resulted in blurred photos, or took photos at wrong positions.

In contrast, with our device synchronization mechanism implemented, the percentage of these action failures reduced to nearly 10%. The large decrease in the action failure rate is because with device synchronization, the system will not assign a new request to a camera that is busy serving another request or is currently unavailable. The non-zero action failure rate even with device synchronization was due to the heavy workload caused by the ten queries continuously operating on the two cameras. Due to the hardware limitations of the physical devices, zero action failure on them seems to be extremely rare in real-world applications.

6.3. Evaluation of the scheduling algorithms

In this section, we present the performance evaluation of five scheduling algorithms. The LERFA + SRFE and SRFAE algorithms are proposed by us. The LS and SA algorithms are from previous work in parallel machine scheduling. Finally, the RANDOM algorithm was included as the baseline for comparison. It randomly assigns action requests to available devices for execution. All scheduling algorithms under study assigned requests to devices as well as executed the actions on them. The cost metric in our study was the service time (i.e., the expected action execution time).

In order to enable controllable performance studies, we developed a homegrown camera simulator to simulate the AXIS 2130(R) PTZ network cameras in the pervasive lab. We tuned the camera simulator through extensive tests on the real cameras, so that a photo() action (and other actions on camera) executed on a simulated camera had similar effects (e.g., time for head movement) to that on a real camera. All experiments presented later in this section were conducted using this camera simulator.

In the first set of experiments, we fixed the number of cameras to be 10. We used three synthetic action workloads, each of which consisted of 10, 20 and 30 action requests, respectively. In each workload, the cost (service time) of an action request was randomly selected from the interval [0.36, 5.36], which is the range of the execution time (in seconds) of a photo() action on an AXIS 2130(R) camera. For each request, all of the 10 cameras were candidates for its execution.

Figure 4 shows the makespans achieved by the five scheduling algorithms. Each point in the figure is the average of results from ten independent runs. Note that the makespan values shown in the figure included both the computational cost of the scheduling algorithm (the scheduling time), and the time spent on servicing the requests on the cameras (the service time).

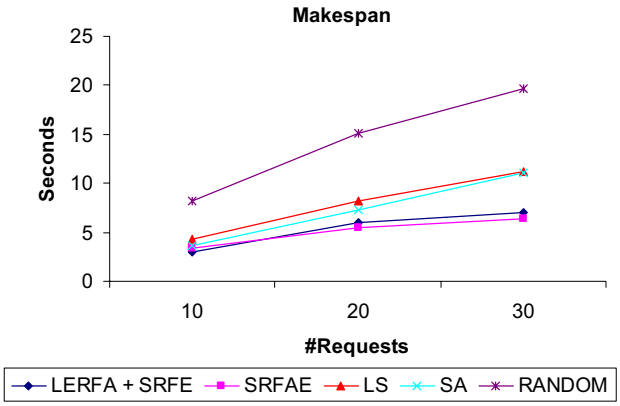

Figure 4. Performance of various scheduling algorithms with a uniform workload

In the figure we see that, for all five algorithms, the makespan increased when the total number of requests increased. The performance of the RANDOM algorithm was much worse than the other four. The two algorithms we proposed had similar performance and both of them outperformed the existing LS and SA algorithms by about 20%-40%. For instance, in the workload consisting of 20 requests, the makespans achieved by LERFA + SRFE, SRFE, LS and SA were 5.73, 5.18, 8.21 and 7.29 seconds, respectively.

Figure 4 also illustrates that with a uniform workload, the two algorithms we proposed scaled better than the two existing algorithms. Both LS and SA resulted in a nearly linear increase on the makespan when the number of requests increased linearly. In comparison, the two algorithms we proposed yielded sub-linear increase. Such scalability is desirable in pervasive computing, where real-time actions upon a large number of events are often required. The performance improvement of our proposed algorithms over existing algorithms validated the effectiveness of the heuristics we adopted.

Figure 5 shows the time breakdown of the five algorithms with the workload consisting of 20 requests. In the figure we see that the scheduling times of the four algorithms other than SA were negligible in comparison with the service time. As we described in Section 5.1, negligible scheduling time is a requirement of scheduling algorithms in pervasive computing.

Among all five algorithms, the SA algorithm achieved the least service time (4.81 seconds), which happens to be the optimal schedule in this special case. However, the scheduling time of SA was much longer than those of the other four, which greatly affected the overall performance of the algorithm. In comparison, our proposed algorithms both achieved nearly optimal schedules (the differences to the optimal schedule is less than 1 second) with a negligible scheduling time.

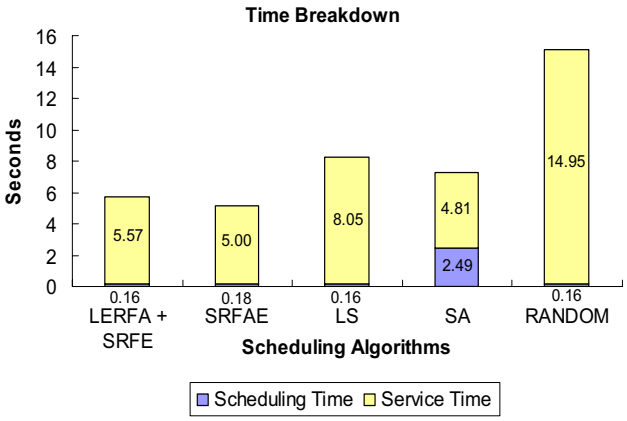

Figure 5. Time breakdown of the five scheduling algorithms

We have also run a number of other experiments with the number of cameras and requests varied but still kept all cameras as candidates for each request in the workload. The results show that with a uniformly distributed workload, the performance of the four scheduling algorithms (except for RANDOM) was only affected by the average number of requests scheduled on each device (i.e., #requests / #devices).

Next, we fixed the number of cameras and requests involved in the scheduling to be 10 and 20, respectively. We changed the action workload distribution on the 10 cameras from a uniform distribution to a skewed one. Specifically, in a skewed workload, half of the 20 requests each had 10 cameras as its candidate devices; for the other half, each could only be serviced on a subset of the 10 cameras. The size of the subset determined the skewness of the request distribution. We define *skewness* to be the size of the subset divided by the total number of cameras.

Figure 6 shows the makespans achieved by the five algorithms when the workload skewness varied from 0.2 to 0.4. In the figure we see that, the SA algorithm performed the worst with skewed workloads. This was because the algorithm had a very long scheduling time, which completely dominated the service time of the scheduling. For the other four algorithms, the makespans decreased when the skewness increased, due to the increasing opportunity of distributing the skewed workload to more candidate devices. Our proposed algorithms still had the best performance among all five.

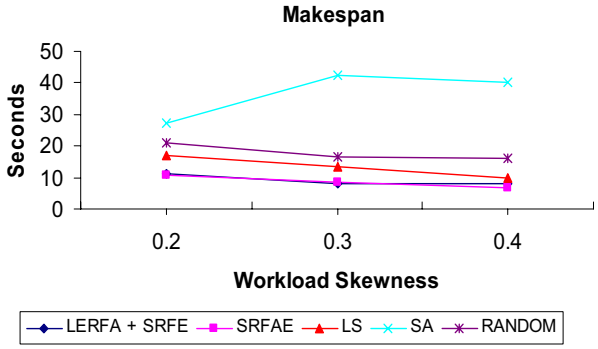

Figure 6. Performance of various scheduling algorithms with a skewed workload

7. Related work

Recent work in pervasive computing has focused on networks of homogeneous devices, e.g., RFID (Radio Frequency Identification) tags [14] and cell phones [17]. In comparison, Aorta manages a number of heterogeneous devices and enables the data acquisition and transmission across these devices through a uniform data communication layer.

Flinn et al. [8] proposed the design of a remote execution system in pervasive computing named Spectra. The goal of Spectra is to improve application performance and result quality while reducing the energy consumption on mobile devices. The best application execution plan selection in Spectra shares a similar spirit with the device selection optimization in Aorta. However, the authors of Spectra did not consider the issue of device heterogeneity as we did in Aorta. The clients (devices) in their implementation were PC-grade machines in mobile computing as opposed to our cameras, phones, and sensors.

There have been recent publications on event-driven processing over distributed sensor networks. Abdelzaher et al. [1] developed an environmental programming paradigm called EnvioTrack that tracks physical-world events and invokes pre-defined computation for them. In Direct Diffusion [7][10], a distributed communication paradigm was proposed for disseminating data to sensor nodes that match the interest of the data. The data is named attribute-value pairs that encapsulate the semantics of the events detected by nodes in the network. In comparison, in Aorta we encapsulate the event-driven processing nature of pervasive computing applications in the specification of declarative database queries. Furthermore, we focused on invoking various types of actions on heterogeneous devices upon events.

Finally, previous work on load balancing in distributed computing systems [11][15][18] studied the scheduling of a number of tasks on an interconnected network of computers. In comparison, in our action workload scheduling problem there is no connection or communication among the devices for executing the workload. We have further considered the physical status changes on devices and the real-time processing requirement in pervasive computing when designing our scheduling algorithms.

8. Conclusions

In this paper, we focus on presenting the systems support issues in our Aorta pervasive query processing framework. We have designed and implemented a uniform data communication layer that enables network data independence over a number of heterogeneous devices. We have implemented locking and probing mechanisms in Aorta for device synchronization. We have also proposed two heuristic algorithms to solve the action workload scheduling problem. Our results from both empirical studies and simulation studies demonstrate that these systems mechanisms helped Aorta to balance device workload and to achieve good performance.

Future work includes extending the uniform data communication layer to support new types of devices, studying more sophisticated device synchronization mechanisms, and investigating scheduling techniques for a large number of heterogeneous devices. The Aorta source code and publications are available at the project web page http://www.cs.ust.hk/aorta.

9. Acknowledgement

This work was supported by grants AoE/E-01/99, HKUST6158/03E, HKUST6263/04E and HKUST6264/04E, all from the Hong Kong RGC (Research Grants Council).

10. References

[1] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. EnviroTrack: Towards an Environmental Computing Paradigm for Distributed Sensor Networks. ICDCS 2004.

[2] Georgios C. Anagnstopoulos and Ghaith Rabadi. A Simulated Annealing Algorithm for the Unrelated Parallel Machine Scheduling Problem. In Proceedings of the 8th International Symposium on Manufacturing with Applications, 2002.

[3] Axis Communications. http://www.axis.com

[4] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Querying the Physical World. IEEE Personal Communications, Vol. 7, No. 5, October 2000.

[5] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards Sensor Database Systems. MDM 2001.

[6] Crossbow Corp. http://www.xbow.com

[7] Deborah Estrin, John Heidemann, Ramesh Govindan, and Satish Kumar. Next Century Challenges: Scalable Coordination in Sensor Networks. MobiCom 1999.

[8] Jason Flinn, So Young Park, and M. Satyanarayanan. Balancing Performance, Energy, and Quality in Pervasive Computing. ICDCS 2002.

[9] Joseph M. Hellerstein. Toward Network Data Independence. SIGMOD Record, Vol. 32, No. 3, September 2003.

[10] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. MobiCom 2000.

[11] Virginia Mary Lo. Heuristic Algorithm for Task Assignment in Distributed Systems. IEEE Transaction on Computers, Vol. 37, No. 11, November 1988.

[12] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. The Design of an Acquisitional Query Processor for Sensor Networks. SIGMOD 2003.

[13] Michael Pinedo. Scheduling Theory, Algorithms, and Systems. 2nd Edition, Prentice Hall, 2002.

[14] Kay Römer, Thomas Schoch, Friedemann Mattern, and Thomas Dübendorfer. Smart Identification Frameworks for Ubiquitous Computing Applications. PERCOM 2003.

[15] Keith W. Ross and David D. Yao. Optimal Load Balancing and Scheduling in a Distributed Computer System. Journal of the Association for Computing Machinery, Vol. 38, No. 3, July 1991.

[16] Chien-Chung Shen, Chavalit Srisathapornphat, and Chaiporn Jaikaeo. Sensor Information Networking Architecture and Applications. IEEE Personal Communications, Vol. 8, No. 4, August 2001.

[17] Frank Stajano and Alan Jones. The Thinnest Of Clients: Controlling It All Via Cellphone. ACM SIGMOBILE Mobile Computing and Communication Review, Vol. 2, No. 4, October 1998.

[18] Asser N. Tantawi and Don Towsley. Optimal Static Load Balancing in Distributed Computer Systems. Journal of the Association for Computing Machinery, Vol. 32, No. 2, April 1985.

[19] Mark Weiser. The Computer for the 21st Century. Scientific American, Vol. 265, No. 3, September 1991.

[20] Wenwei Xue and Qiong Luo. Action-Oriented Query Processing for Pervasive Computing. CIDR 2005.

[21] Yong Yao and Johannes Gehrke. Query Processing for Sensor Networks. CIDR 2003.