Integration and Configuration of Small Wireless Sensors into Ubiquitous Computing Environments

Jong Hee Kang, Gaetano Borriello

Technical Report UW-CSE-03-12-06

Abstract
Ideally, applications in ubiquitous computing environments help users accomplish their tasks in as unobtrusive a way as possible by collecting and processing various information about the users' context. Such information is provided to the applications by a large number of simple wireless sensors embedded throughout the environment or carried by people. In ubiquitous computing environments such as these, two new problems arise. First, the wireless sensors are designed to be inexpensive and low power. Thus, there is minimal computation and communication within the sensors themselves. Much of the computation needs to be performed within the infrastructure where software components running on more capable engines can elaborate the data, make correlations, and discover patterns of interest. As the number of impoverished sensors and accompanying software components increases, it becomes more difficult to configure and manage them. Existing auto-configuration systems, such as UPnP [28], were designed for much more capable devices and are not directly applicable to this environment. Second, the ubiquitous computing applications we are developing want to access sensors based on dynamic attributes, such as the closest user, making it necessary to support rebinding between applications, sensors, and their intermediate software components as the situation changes. Unfortunately, existing discovery systems do not support this fine-grain automatic rebinding. We propose a new approach for the configuration and management of small wireless sensors and software service components that efficiently supports dynamic rebinding to applications. We also demonstrate that our approach is scalable and fills the gaps left by existing auto-configuration and discovery systems.

1 Introduction

Ideally, applications in ubiquitous computing environments help users accomplish their tasks in as unobtrusive a way as possible by collecting and processing various information about the users' context. To collect such information, various types of sensors are deployed in every place in the environment, even in the places where wiring is impossible. To make it easy to deploy sensors in the environment, usually wireless sensors are preferred. However, wireless sensors are powered by battery or harvesting electronic power parasitically from other sources around them [20, 22], and they are designed to have minimal functionality to minimize their power requirements. Thus, they do not support any complex communication protocols or extra computation, and the applications interested in the sensor data rarely communicate directly with the sensors as they might with more intelligent devices. Instead, most of the sensors send data to a nearby base station periodically or when an interesting event occurs through a simple communication protocol, and much of the computation needs to be performed within the infrastructure where software components run on more capable engines. The software components translate, calibrate, fuse data together, and manipulate data in various ways before delivering data to the applications. All these sensor data processing components can be integrated into each application as is typically done today in mostly stove-piped one-off systems. It would be much more desirable if the sensor data processing components could be shared by multiple applications, and the set of these processing components were adaptable at run time as the situation changes. And, the software components need to be upgraded and replaced by new better components while the applications would be expected to be kept running. Thus, it is more efficient to make the sensor data processing components as autonomous services and compose them together dynamically at run time. As the number of impoverished sensors and accompanying software components increases, it becomes more and more difficult to configure and manage them. Even though there exist some systems that perform automatic configuration of devices [26, 29], they are designed for much more capable devices and are not directly applicable to simple wireless sensors and accompanying software service components.

The ubiquitous computing applications we are developing [2, 16] want to access sensors based on dynamic attributes such as where the sensors are currently located, who are around the sensors, and so on. For example, an application may want to keep receiving the temperature data around a person wherever the person is. The person may move from one place to another, and as the personís location changes, the application needs to access different sensors to get the correct data. To make applications easy to write and adaptable, the system should be able to rebind the applications, sensors, and their intermediate software components dynamically based on the dynamic attributes of the sensors. Unfortunately, existing discovery systems do not support this automatic rebinding.

In this paper, we describe a new approach for configuration and management of small wireless sensors and software service components that efficiently support the dynamic rebinding of sensors and software components to applications. We also describe the two applications that we built on top of our new architecture.

2 Related Work

2.1 Ubiquitous Computing Environments

Our work is related to much of the research on intelligent rooms [3, 10, 11, 13, 23] that are equipped with various types of devices including large wall displays, laptops, and handheld devices. The users in the room use these devices to present information to others or exchange information with each other. The applications running in the intelligent rooms use various devices depending on the situation, and the infrastructure maps the needs of each application to the most suitable devices for the current context, and sets up the communication between the application and the devices. This work focuses on how to coordinate the use of output devices, especially large displays, and does not address the issues of collecting context information from sensors.

2.2 Automatic Configuration Systems

Automatic configuration is an important part of approaches such as Jini [26] and UPnP [29]. In these systems, when a device is connected to the local area network, the device registers its existence ñ the services it provides, the properties of those services, and how other entities can access and interact with them ñ with a lookup service. Once the device is registered with the lookup service, it can be used by any clients that want to use the services provided by that device. These systems assume that the devices are intelligent enough to participate in their registration and lookup protocols that can be rather complex. For example, in Jini, the registration and lookup protocols are based on Java RMI [25] that needs TCP/IP protocol support and a Java virtual machine on the device. However, these need complex computation and storage, and are not good for simple wireless sensors that should save battery power so that they can last for long periods of time, possibly years. To use impoverished devices in these systems, a proxy [27] scheme has been proposed. Proxies allow simple devices to communicate with other entities by giving them a more powerful alter ego in the network. We can make simple wireless sensors participate in the automatic configuration system by using proxies for the sensors. However, the sensors do not have any automatic mechanism for connecting themselves with their proxies, and the proxies need to be manually set up so that they can correctly communicate with the sensors.

2.3 Dynamic Service Composition Systems

Among the many existing dynamic service composition systems that compose several simple autonomous services into a complex composite service [4, 5, 8, 14, 18], path-based systems [9, 14] are the most suitable for our purposes because they compose services in pipe-and-filter style that best catches the composition pattern of the intermediate services in ubiquitous computing environments. In Ninja Paths [9], a path ñ a sequence of services from a network service to a client ñ is constructed and the connections between the services are established before the network service starts communicating with the client. Network services in Ninja Paths usually serve streaming data to and from clients, and the examples of the intermediate services include transcoding, translation, and protocol conversion services. The idea of composing services into a path is useful for ubiquitous computing environments. However, it is still based on a connection-oriented communication model that is not suitable for the event-driven communication model used by simple wireless sensors.

2.4 Discovery Systems

In existing discovery systems [1, 6, 26], a device or a software service registers its description with the discovery service. And, the application that wants to use the device locates the device it wants to use through the discovery service. Then, it establishes a connection to the device and communicates with it by sending requests and receiving replies. This model works well when the descriptions of devices do not change frequently. In ubiquitous computing environments, however, the descriptions of devices include the attributes that change as the situation changes. If we use the existing discovery systems in ubiquitous computing environments, each device needs to acquire the changes in its attributes and re-register its new description with the discovery service whenever the situation changes. And, the applications are responsible for locating and connecting to the appropriate devices every time it wants to send a request to the device. The late-binding discovery mechanism [1] does this for the applications. However, it does not fit well for the applications that want to receive a series of data packets from sensors over a relatively long period of time.

3 System Architecture

3.1 Overview

3.1.1 Event-driven communication between sensors and applications

In the existing discovery systems, applications usually communicate with devices through an RPC [24] style request-reply protocol. An application first establishes a connection to the device it is interested in and sends requests and waits for replies. To establish a connection, the application needs to obtain the address of the device through the discovery service. This requires that devices register themselves with the discovery system.

However, applications do not necessarily have to establish explicit connections to sensors to get sensor data. Usually, sensors generate data periodically or when an interesting event occurs, and many applications want to receive data only when the sensors generate it. The publish/subscribe model [19] fits well for this type of interaction. In this model, the data receivers do not have to establish connections to the sender. Instead, they are listening for a particular event type, and the infrastructure delivers the events to the receiver when the events occur. Some applications may want to receive sensor data through request-reply semantics. But, in ubiquitous computing environments where applications are written to react to external events generated in the environment, the publish/subscribe style of event-driven interaction is prevalent [15].

In our architecture, the communication between sensors and applications follows the publish/subscribe model, and the sensors do not register themselves or their proxies with the discovery service. Instead of establishing a connection to sensors, applications just register their interests with the infrastructure. When the sensor data is generated, the infrastructure determines which applications will receive the data and routes the data to them. Thus, the data generated by sensors must be self-describing and include meta-information to be used by the infrastructure to determining the appropriate receivers.

3.1.2 Path-based, per event basis service composition

The raw data generated by sensors is not directly consumed by applications. The raw sensor data is translated, calibrated, fused with other data, and manipulated in various ways before it is actually used by applications. We cannot expect applications to be written for a specific set of sensors. They must be able to evolve over time. Thus several layers of abstraction will be needed. Some of the lower will consist of services that will aggregate, smooth, and calibrate sensor data.

For example, the raw sensor data generated by a temperature sensor is just a series of bytes. It should be first translated into calibrated values so that it represents the actual temperature value that applications really care about. Also, the temperature value from one sensor may be fused with other temperature readings from other sensors in the same room. When the user moves to another room with different types of temperature sensors, the application needs different translation and calibration components. Integrating the sensor data processing components into each application is not efficient for two reasons: the components are shared by multiple applications; the data processing steps for each application may change dynamically as the situation changes. The components should be made as autonomous services and composed dynamically at run time.

The composition pattern of the sensor data processing services follows the pipe-and-filter architectural style [7]. A componentís input data comes from the previous componentís output, and the output of the component goes to the next componentís input, and so on. We borrow this idea from the path-based systems that compose services into a path ñ an ordered sequence of services through which data flows. In path-based systems, a path is constructed and the connections between the services are established before the network service starts sending streaming data to the client. However, in ubiquitous computing environments with small wireless sensors that emit data intermittently, the composition information is not available until the sensor data is generated, and the compositions change frequently. Therefore, the connection-oriented path mechanisms do not catch the characteristics of these environments.

We extend the path-based service composition idea with an event-driven communication model. Services do not use connections to form a path. Instead, services are composed on a per event basis and the composition information is carried within each data event.

3.1.3 Using dynamic attributes in the discovery process

The meta-information attached to the sensor data by sensors includes only static information about the sensor data that can be known at manufacturing time such as the type of the data, sensor ID, and so on. However, the interests registered by applications include not only the static attributes of the sensors, but also the dynamic attributes such as where the sensors are currently located, who are around the sensors, and so on. The infrastructure should bridge the gap between these two and deliver the sensor data to appropriate applications.

We extend the function of existing discovery systems by making the discovery service consult the association service that maintains the dynamic attributes of the sensors and other objects in the environment when they are matching incoming sensor data with applications.

3.2 Putting Them All Together

All the intermediate software services and applications register their descriptions and interests with the infrastructure. They can afford to do this because they are running on a wired infrastructure and do not have power constraints as do the impoverished sensors. The intermediate service description includes the information on the input data, output data, and the operation the service performs on the input data. The application interest includes the information on the data the application is interested in, and the association parameters that specify the dynamic attributes.

When a sensor generates data, it forwards the data along with its meta-information to a nearby gateway. The meta-information attached to the data packet includes the device ID and data-type ID. When the gateway receives a data packet from a sensor, it attaches its gateway ID into the meta-information field and forwards it to the infrastructure. The meta-information that comes with the data has only static attributes of the sensor data. To obtain dynamic attributes of the data, the infrastructure consults the association service. The association service may use the meta-information to figure out the dynamic attribute of the sensor data. Once the infrastructure finds out the dynamic attributes of the data, it figures out which application should receive the data and which intermediate software services should be used to transform the raw sensor data into the appropriate type of data the application is expecting. The result is a path ñ a sequence of intermediate services from the sensor

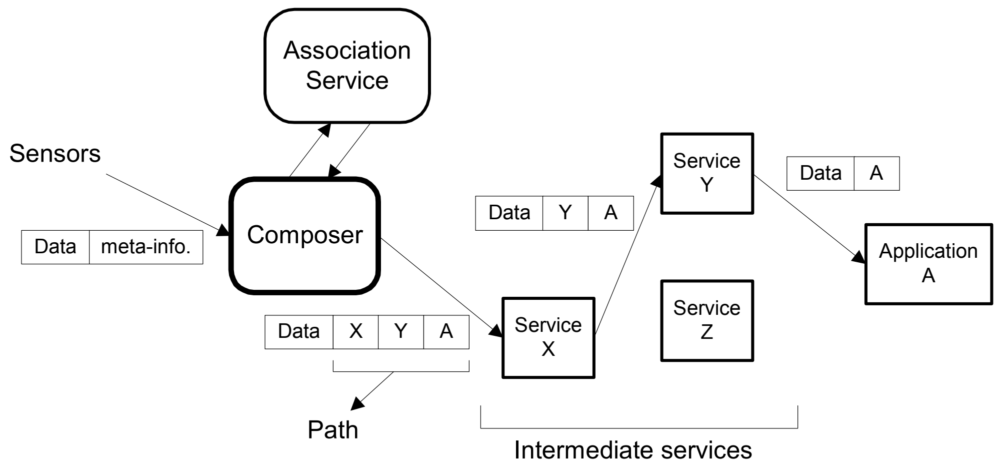


Figure 1. The composer receives data from sensors and generates a path for the data.

to the application. Multiple paths can be produced if more than one application is interested in the same sensor data.

Once a path is produced, the path information is attached to the data packet, and the data packet visits the intermediate services in the order specified in the path information. When an intermediate service receives the data, it processes the data, puts the result into the data packet, and forwards it to the next service in the path information. The last item in the path information is the application. When the data packet reaches the application, the data has been already processed by the intermediate services and transformed into the appropriate forms that the application can directly use.

4 Prototype Implementation

4.1 Sensors

The wireless sensors are implemented using UC Berkeley motes [12] that have been programmed to simply broadcast their data over their radios to a nearby base station either periodically or whenever an interesting event occurs ñ depending on the sensor type. When sending data, each mote includes its ID and the type of the sensor generating the data.

Mote gateways are running on the machines to which mote base stations are connected. The mote base station forwards the radio data packet to the mote gateway through the serial port of the machine. The mote gateway converts the radio data packet into a data message and sends it to the software infrastructure. The mote gateway adds its own ID into the meta-information of the message.

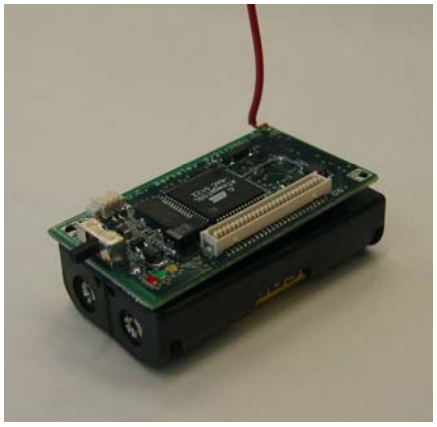


Figure 2. Mote sensor.

4.2 Intermediate Services and Applications

The software infrastructure is implemented on top of Rain [21], an asynchronous event-based service/messaging system developed at Intel Research Seattle. Messages in Rain are XML fragments and the format of the message can be easily changed. The only required fields are sender and recipient tags. All the software entities in the environment are Rain services. And, data, path information, and meta-information are encoded as XML elements and composed into a message. Rain has been implemented in several different languages, and we use the Java implementation for this work.

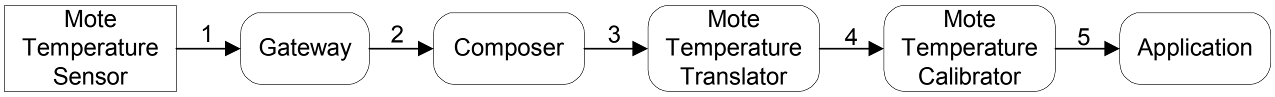
Services and applications register their descriptions and interests with the infrastructure. In our first prototype implementation, we have simplified the contents of the description in a similar way to existing path-based systems [9, 14]. An intermediate service is described by its input data-type and output data-type. The semantics of the serviceís functionality are described by these input and output types. An application interest includes its input data-type and association parameters that are attribute-value pairs. Examples of attributes include location, person, and other physical object. For example, an application may want temperature data in the living room. In this case, the input data type of the application is ëtemperatureí and the association parameter is ëlocation=living roomí.

The lifetime of the descriptions are controlled by a leasing mechanism [8]. Descriptions are updated by periodic advertisement messages from services and applications. Descriptions are deleted from the infrastructure if they cease to be advertised.

When a service receives a data message, it is delivered to the message handler of the service. The message handler processes the data and puts the result into the message. Then, the message containing the result is forwarded to the next service in the path.

4.3 Composer

In the center of the infrastructure is a service named the *composer*. The composer maintains all the descriptions of the intermediate services and interests of the applications in the environment. Its implementation can be centralized or distributed for fault tolerance. Service descriptions are stored as a directed graph. In the graph, nodes represent data-type names, and arcs represent services. The starting node of an arc is the input data-type of the service, and the ending node represents the output data-type of the service. Application interests are stored in a table.



1)

```
7e 3e 7d 1d 27 3b 41 71 ...
```

2)

```
<SENSOR_DATA>
 <TYPE>MoteTempSensorRaw</TYPE>
 <META_INFO>
  <DEVICEID>342</DEVICEID>
  <GATEWAYID>4</GATEWAYID>
 </META_INFO>
 <DATA>
  <VALUE xsi:type="xsd:base64Binary">kQA=</VALUE>
 </DATA>
</SENSOR_DATA>
```

3)

```
<SENSOR_DATA>
 <TYPE>MoteTempSensorRaw</TYPE>
 <META_INFO>
  <DEVICEID>342</DEVICEID>
  <GATEWAYID>4</GATEWAYID>
  <ASSOCIATION type="location">
    living room</ASSOCIATION>
 </META_INFO>
 <DATA>
  <VALUE xsi:type="xsd:base64Binary">kQA=</VALUE>
 </DATA>
 <PATHLIST>
  <SERVICE>
   <serviceid id="....">
     http://10.122.1.60:8080/rain</serviceid>
   <handlerid id="calibration"/>
  </SERVICE>
  <APPLICATION>
   <serviceid id="....">
     http://10.122.1.61:8080/rain</serviceid>
   <handlerid id="fahrenheit"/>
  </APPLICATION>
 </PATHLIST>
</SENSOR_DATA>
```

4)

```
<SENSOR_DATA>
 <TYPE>MoteTempSensor</TYPE>
 <META_INFO>
  <DEVICEID>342</DEVICEID>
  <GATEWAYID>4</GATEWAYID>
  <ASSOCIATION type="location">
    living room</ASSOCIATION>
 </META_INFO>
 <DATA>
  <VALUE xsi:type="Array" arrayType="xsd:int[1]">
   <i xsi:type="xsd:int">145</i>
  </VALUE>
 </DATA>
 <PATHLIST>
  <APPLICATION>
   <serviceid id="....">
     http://10.122.1.61:8080/rain</serviceid>
   <handlerid id="fahrenheit"/>
  </APPLICATION>
 </PATHLIST>
</SENSOR_DATA>
```

5)

```
<SENSOR_DATA>
 <TYPE>TemperatureFahrenheit</TYPE>
 <META_INFO>
  <DEVICEID>342</DEVICEID>
  <GATEWAYID>4</GATEWAYID>
  <ASSOCIATION type="location">
    living room</ASSOCIATION>
 </META_INFO>
 <DATA>
  <VALUE xsi:type="Array" arrayType="xsd:int[1]">
   <i xsi:type="xsd:int">68</i>
  </VALUE>
 </DATA>
 <PATHLIST/>
</SENSOR_DATA>
```

Figure 3. Temperature data of the living room is delivered to the application interested in it. 1) The mote sensor sends a series of bytes to the gateway. 2) The gateway packs the data into a Rain message and forwards it to the composer. 3) The composer gets the association information (location = living room), and finds a path to the application. The path found is the ordered sequence of the mote temperature translator service, mote temperature calibrator service, and the application. The composer forwards the data to the first service of the path attaching information about the remainder of the path to the data. 4) The mote temperature translator service translates the raw data into an integer value and forwards the result to the next service ñ the mote temperature calibrator ñ in the path. 5) The mote temperature calibrator service determines the actual temperature from the calibration data and forwards it to the application.

When a mote gateway receives a data packet from a mote sensor. It puts the data into a data message and forwards it to the composer. Upon receipt of the message, the composer first refers to the *association service*, another infrastructure service that figures out the dynamic attributes from the meta-information of the data. In our prototype, the association service determines the location or proximity based on the gateway ID that receives the sensor data. For example, the association service may return ëlocation=living roomí from the gateway ID. The association service many use different information depending on the association mechanism deployed in the environment. Once the composer gets all the association information for the data, it looks for applications whose association parameters match the dynamic attributes of the incoming data. For each of the matching applications, the composer finds the shortest path between the data-type of the incoming data and the input data-type of the application from the directed graph. If the composer finds a path, it attaches the list of services in the path to the data message and forwards it to the first service in the path.

4.4 Error Handling

The service description information maintained in the composer is kept being refreshed by periodic advertisement messages. However, obsolete descriptions may be present until timeout, and the path may include an unavailable service. When a service cannot forward the message to the next service in the path, the service sends the message back to the composer with the error information giving the composer the opportunity to find a different path for the message and update the service description information.

5 Applications

5.1 PlantCare

The PlantCare system provides autonomous care of plants in home and office environments [16]. It collects data such as current temperature, moisture, and light level through small wireless sensors, decides what actions to take, and sends appropriate commands to a small mobile robot that waters the plants and recharges sensors. The PlantCare system is typical of future ubiquitous computing applications with its many small wireless sensors both on plants and the environment and software services that are both local (sensor calibration) and remote (plant care encyclopedia).

Although, the original PlantCare system was implemented without our infrastructure, we have re-implemented it so that we could better learn from the comparison between the two approaches. The PlantCare system is composed of many services ñ fifteen different types of Rain services, and we will describe only a part of the system that is related to sensor data collection. Each mote sensor is equipped with four sensors for temperature, moisture, light, and current battery level. The mote sensor collects all these readings, packs them together into one data packet, and periodically sends it to a nearby gateway. The mote translator service translates the raw sensor data by unpacking into a much longer XML description that would have cost the sensor extra energy to transmit directly. The mote translator exports four event handlers for each different type of sensor. For example, the temperature data event handler extracts only temperature reading from the raw sensor data and translates it into a value. So, we can think of it as four closely related translator services instantiated as one combined service. There are four calibration services for each of the sensor data types.

The calibration service calibrates sensor data readings using previously collected calibration data. For example, the temperature calibration service converts the temperature reading value 145 (out of the 0-255 range of the A/D converter on the mote) into 68°F. The tables for making this correspondence are built by the robot as it visits sensors and compares their readings to its own higher-quality pre-calibrated

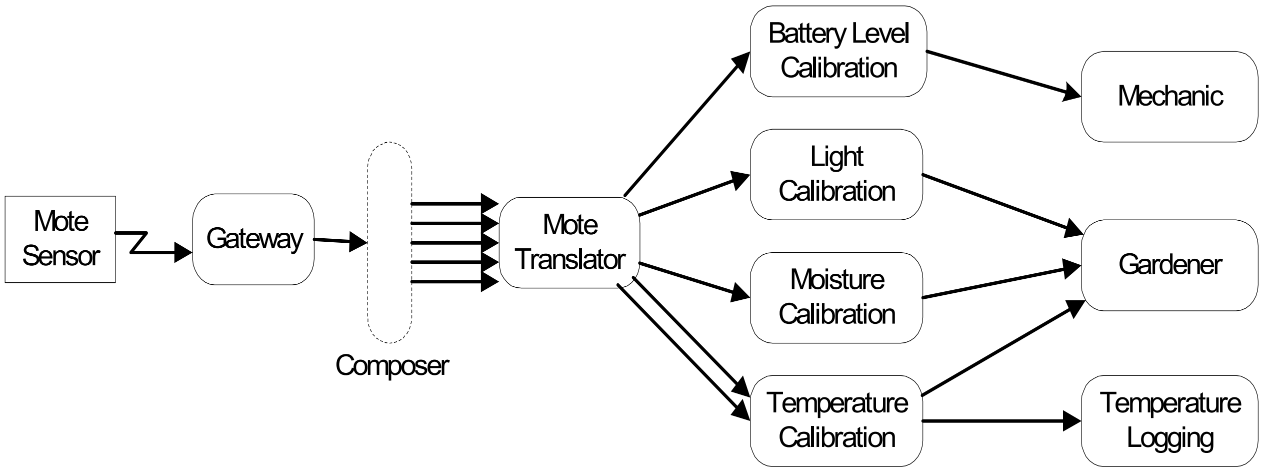


Figure 4. The flow of sensor data in the PlantCare application.

sensor. The robotís readings are also forwarded to the calibration service [17].

The mechanic application monitors the battery level of mote sensors and may cause the robot to visit a plant to recharge its sensors. The gardener application collects environmental data, consults plant care encyclopedia service, and sends appropriate commands to the robot to water the plant. The temperature logging application stores temperature data for future use by other applications. For example, a proper plant positioning application may suggest to the user other spaces where a particular type of plant may find a more appropriate environment in terms of temperature and light levels.

When the gateway receives a data packet from a mote sensor, it forwards the sensor data to the composer. The composer converts the mote ID and the gateway ID into a location name using the association service. Then, the composer looks for applications interested in the sensor data. The mechanic application is interested in the mote battery level, the gardener application is interested in the temperature, light, and moisture data, and the temperature logging application is interested in the temperature data. So, the composer forwards 5 separate messages with different path information. For example, the path information of the message destined to the mechanic is <mote translator service (battery level event handler), battery level calibration service, mechanic application>. Each of the 5 messages travels through the intermediate services specified in the path information and is finally delivered to the application.

Adding a mote sensor into the environment is as simple as just turning on the sensor. Users do not have to stop other services or applications to reconfigure the environment. Once the mote sensor is turned on, the data generated by the sensor is delivered to the appropriate applications by the infrastructure. If the user wants to introduce a new type of sensor into the environment, all she needs to do is to start the translator service and calibration service for the new type of sensor. Applications do not have to be modified or even restarted for the new sensor. We are also investigating how to lookup and start these services automatically on the appropriate local or remote servers. The data generated by the new sensor will be routed through the new translator service and the new calibration service by the infrastructure. Upgrading a software service is also simple. It is done by starting the new upgraded service and stopping the old service. Other services or applications interacting with this service are not affected by it. In connection-oriented systems, replacing existing components often requires restarting interacting components.

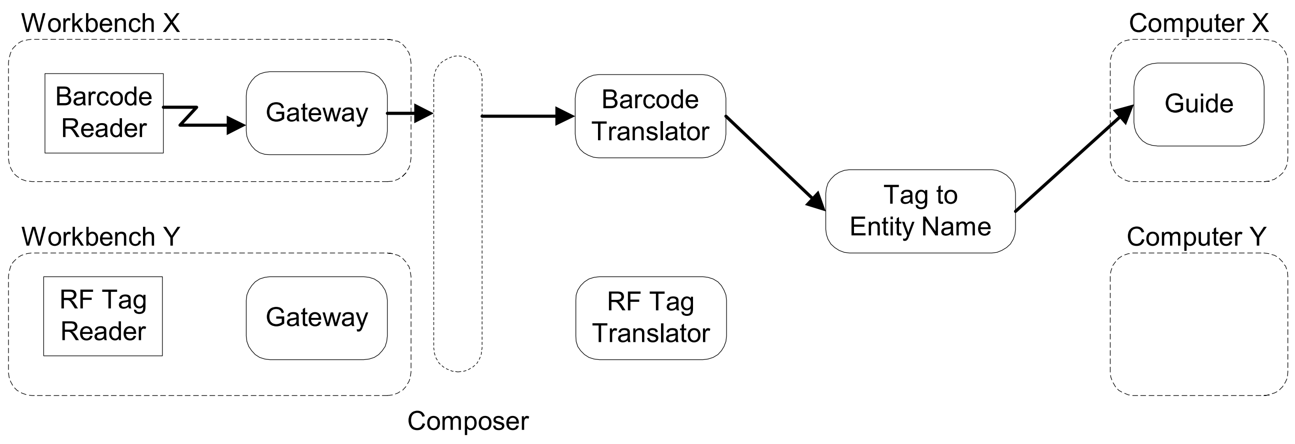
5.2 Labscape

Labscape is an experiment capture system for a cell biology laboratory [2]. In a typical biology laboratory, various experimental instruments are scattered around several workbenches, and the biologist moves around the workbenches during her experiment session. The Labscape application follows the biologist (moves its user interface to the screen nearest them) and helps record the experiment as she moves among workbenches. Each workbench is equipped with a tablet computer and a barcode reader or an RF tag reader. The tablet computer is used for presenting the evolving record of the current experiment to the biologist. A barcode reader and/or RF tag reader is used for scanning the identification tag of each entity involved in the experiment. A body-worn short-range infrared badge is used to determine the userís location.

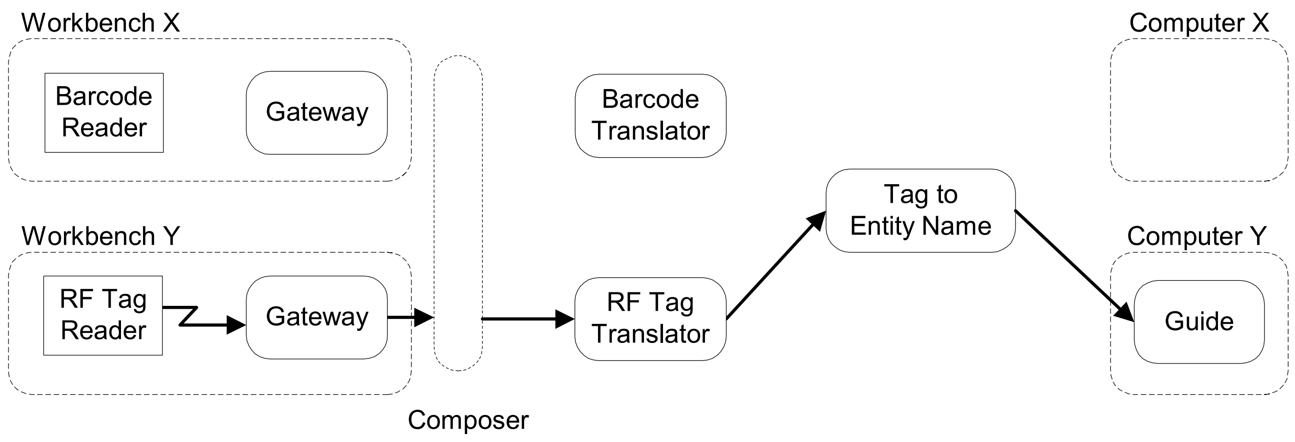
A user starts her experiment session by starting the guide application. The guide application guides the user through the experiment by showing the experiment procedure and recording data generated during the experiment. The guide application migrates from one tablet computer to another following the user as the user moves from one workbench to another. The tag scan events generated at the userís current workbench are translated and delivered to the userís guide application.

The Labscape system was first implemented prior to our work, and we describe the version implemented using our infrastructure. When the user is at workbench X that has a barcode reader, her guide application is running on tablet computer X. A barcode scan event at workbench X is first sent to the barcode translator. Then, the translated data is sent to the next service that converts the tag data into an entity name. The entity name is delivered to the userís guide application on computer X. When the user moves to workbench Y, the user uses an RF tag scanner instead of barcode reader. The RF tag scan event passes through different services from the barcode scan event before delivered to the guide service that has now also migrated to computer Y. The association between the scanning events and the user is provided by the location service that detects the user through their badge at the workbenches corresponding to the reader.

In this example, the user uses two different types of sensors (barcode reader and RF tag reader) to generate the same type of data (tag data). The guide application cannot anticipate which reader the user is going to use until the user actually uses the reader. Other service composition schemes that compose services together before sending or receiving data do not work well in



(a) At workbench X, the user uses a barcode reader to get an entity name.



(b) The user has moved to workbench Y and now uses an RF tag reader.

Figure 3. Dynamic composition in Labscape.

this situation. In our architecture, applications do not have to worry about this kind of dynamic change that is the norm in ubiquitous computing environments. The infrastructure handles this dynamic change by composing services at the moment the data is generated. Another interesting point is that the guide application is migrating to another computer while it is still running. Because service composition and communication is done in an event-driven fashion without explicit connections in our architecture, there is no need for reconnecting broken connections.

6 Discussion

6.1 Configurability

In other systems, an application first establishes a connection to the device it wants to use. Then, the application communicates with the device through the connection. To make applications be able to locate and connect the devices, they are required to be configured ñ getting a network address and setting up appropriate parameters for the network ñ and registered with the discovery service. In our architecture, however, by using the event-driven communication model and making sensor data self-describing, we could eliminate the need for configuring sensors. Sensors do not have to configure themselves nor do they need to establish connections directly to their proxies or other applications. They can start generating and forwarding data without any pre-registration or configuration. So, adding a new sensor is as easy as just turning it on. This feature is especially useful in environments with lots of small wireless sensors. Updating configuration information by hand for each of them is practically infeasible when their numbers reach hundreds or more and we want to consider them as consumer devices.

Configuring software services is also an important issue, and adding, removing or upgrading software services should be simple, too. In our architecture, services are dynamically composed in a per message basis, and there is no connections established between services and application. Thus, adding, removing, or upgrading services does not break any other existing services or require stopping other services.

6.2 Evolvability and Portability

Ubiquitous computing environments have different types of sensors and devices, and the applications written for one environment are often hard to port to another. Introducing new types of sensors into an environment may cause the applications to stop running. That is because the application is written to use specific types of sensors and devices. In our architecture, applications do not specify which type of sensors to use. Instead, they specify high-level data types [1], and the intermediate services in the environment transform the specific sensor data to the high-level data types usable by applications. By separating environment-dependent information from the applications, we make the applications more portable and more adaptable thereby supporting system evolution.

6.3 Support for Dynamic Rebinding

Ubiquitous computing applications often need to change their data sources as the situation changes. It requires that the applications keep monitoring the situation and change the connections to the sensors. In our architecture, the infrastructure does this job for applications by using the event-driven communication model and using dynamic attributes in the discovery process. Applications just specify their interests that include both static and dynamic attributes of the data sources, and the infrastructure handles the rebinding process and delivers the correct sensor data to the applications.

6.4 Performance

We have measured the performance overhead of our composition infrastructure. We have used desktop machines with Intel Pentium III processors at 1.2 GHz, 512 MB of RAM, and running Windows 2000. First, we measured the overhead of computing paths. In Table 1, the path computing time is the time spent by the composer to find paths for an incoming sensor data message. The composer latency is the time for a sensor data message to travel from a gateway to the first service in the path generated by the composer. So, the composer latency is the sum of the path computing time and the communication latency. From our experiences on several ubiquitous computing applications including the two described in section 5, we have found that in most cases the length of the path is less than 4 and that the number of services in an environment is usually not greater than hundred. In the test set, the graph formed by the intermediate services is a tree with depth of 4, and the length of the path generated by the composer is also 4. As the result shows, the composer latency is dominated by the communication overhead, and finding paths does not take appreciable time. Also, the number of services registered in the composer does not affect the time taken for finding paths. So, the composer overhead is not too large and can be considered as adding approximately one more hop to the message path.

Table 1. Composer overhead.

Number of services registered	Path computing time	Composer latency
50	2.3 ms	6.7 ms
100	2.5 ms	7.3 ms
150	2.4 ms	7.0 ms
200	2.9 ms	6.6 ms

We also measured the latency of our composition infrastructure and two alternatives that bound its performance ñ direct connection and the pure publish/subscribe model. In the best case, using a direct connection, each service is pre-configured and knows the next service to which it is to forward messages without referring to the path information in the data message, and each message visits the minimum number of nodes before it gets to the final application. In our infrastructure, a data message first visits the composer before traveling to the final application. So, each data message visits one more service than in the direct connection case. And, each service needs to process the path information in the message. In the pure publish/subscribe model, a message is always sent to the central message queue that dispatches each message to its receivers. So, each message visits twice as many nodes as in the direct connection. Table 2 shows the latencies for each case. The composer overhead and path processing overhead is relatively large when the path length is short and message size is small. However, the overhead is mitigated as the path length becomes longer and the message size becomes larger. The publish/subscribe model needs more than twice the time of the direct connection approach in every case, as expected. The composition infrastructure has more than 20% performance overhead over direct connection. But, the ease of configuration and management of services it provides seems more than worth the cost for most non-real-time applications.

Table 2. Latencies for 3 message delivering mechanisms.

Method	*10-byte message*		*5000-byte message*	
	2 hop	*4 hop*	*2 hop*	*4 hop*
Direct	10.5 ms	21.8 ms	145.8 ms	298.7 ms
Composition	19.1 ms	34.5 ms	221.4 ms	365.6 ms
Pub/sub	24.5 ms	50.7 ms	300.1 ms	640.8 ms

6.5 Application Experience

The comparison of the two different implementations ñ one using our infrastructure, the other not ñ of each of the two applications in section 5 shows the usefulness of our infrastructure in ubiquitous computing environments.

In the original version of the PlantCare application, direct connection is used for connecting services, that is, each service knows the next service to which it is to forward its messages. Some of the services use the discovery service, but they use it just for mapping servicesí names into servicesí addresses. This severely limits the systemís flexibility in that whenever a service is added, the services that will need to use it must be explicitly modified. The new implementation using our infrastructure solves this problem, and we can add or remove services at any time without affecting other services. The connections between them are handled completely by the composer.

In the original version of the Labscape application, the components that process sensor data are integrated into the guide application. Sensor data generated by barcode readers and RF tag scanners is directly delivered to the guide application. In this implementation, adding a new type of sensor (tag scanner) requires the guide application to be rewritten so that it can understand the data generated by the new type of tag scanner. In the implementation using our infrastructure, the guide application does not have to be modified at all. All that is required is starting a new translator service that can translate the new type of data into tag data that the composer with then automatically routes (along with its association parameters) to the guide application.

7 Conclusion and Future Work

Ubiquitous computing environments consist of large numbers of small wireless sensors and dynamically composed software services. As the number of sensors and software services increases, it becomes more difficult to configure and manage them efficiently. In addition to that, ubiquitous computing applications want to access sensors based on the dynamic attributes of the sensors and rebind to different sensors as the situation changes. In this paper, we described a new approach for configuring and managing these sensors and services that efficiently supports dynamic rebinding between applications and sensors. Event-driven communication and self-describing sensor data enable small wireless sensors to be integrated into the environment without any explicit configuration. Dynamic service composition that occurs on a per message basis makes management of software services easy and simple. And, the use of dynamic attributes in the discovery process coupled with event-driven communication supports dynamic rebinding efficiently.

The next step is to research the service description methods. In our prototype implementation, we use the input data-type and output data-type for describing a service. This has worked well for the applications we have implemented up to this point. However, the input data-type and output data-type are not enough for describing the semantics of more complex services or those that require the aggregation of data from a collection of sensors. Therefore, we are investigating more expressive description methods for describing the semantics of more complex services.

In the current architecture, each message carries its list of services through which it must travel. Instead of a simple list of services, we want to add logic to the path information so that the composition can be optimized. This idea comes from the active networking community [28], and we are exploring the possibility of applying this idea to our infrastructure.

Also, we are investigating how to manage the large number of services in the environment: starting the necessary services on appropriate machines to enable an application on demand; stopping unused or obsolete services; and adjusting workloads to manage storage, performance, and bandwidth requirements and constraints.

References

1. William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley, The design and implementation of an intentional naming system. In 17th ACM Symposium on Operating Systems Principles (SOSP), Kiawah Island, SC, December 1999.
2. Larry Arnstein, Chia-Yang Hung, Robert Franza, Qing Hong Zhou, Gaetano Borriello, Sunny Consolvo, and Jing Su. Labscape: A Smart Environment for the Cell Biology Laboratory, IEEE Pervasive Computing Magazine, vol. 1, no. 3, July-September 2002.
3. Barry Brumitt, Brian Meyers, John Krumm, Amanda Kern, and Steven Shafer. EasyLiving: Technologies for Intelligent Environments. In Second International Symposium on Handheld and Ubiquitous Computing 2000 (HUC2k), Bristol, England, September 2000.
4. Fabio Casati, Ski Ilnicki, LiJie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. Adaptive and dynamic service composition in eflow. Technical Report, HPL-200039, Software Technology Laboratory, Palo Alto, CA, March 2000.
5. Dipanjan Chakraborty, Filip Perich, Anupam Joshi, Timothy Finin, and Yelena Yesha. A reactive service composition architecture for pervasive computing environment. In 7th Personal Wireless Communications Conference. 2002.
6. Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. An Architecture for a Secure Service Discovery Service. Fifth Annual International Conference on Mobile Computing and Networks (MobiCom '99), Seattle, WA, August 1999.
7. David Garlan, and Mary Shaw. An Introduction to Software Architecture. Technical Report, CMU-CS-94-166, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1994.
8. C. Gray and D. Cheriton. Leases: An efficient fault tolerant mechanism for distributed file cache consistency. In 12th ACM Symposium on Operating Systems Principles (SOSP), 1989.
9. Steve. D. Gribble, Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, and B. Zhao. The Ninja Architecture for Robust Internet-Scale Systems and Services. Computer Networks, vol.35, no. 4, March 2001.
10. Nicholas Hanssens, Ajay Kulkarni, Rattapoom Tuchinda, and Tyler Horton. Building Agent-Based Intelligent Workspaces. In International Workshop on Agents for Business Automation. Las Vegas, NV, 2002.
11. Christopher K. Hess, Manuel Rom· n, and Roy H. Campbell. Building Applications for Ubiquitous Computing Environments. In International Conference on Pervasive Computing (Pervasive 2002), Zurich, Switzerland, August 2002.
12. Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for network sensors. ASPLOS 2000.
13. Brad Johanson, Armando Fox, and Terry Winograd. The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms. IEEE Pervasive Computing Magazine, vol. 1, no. 2, April-June 2002.
14. Emre Kiciman, and Armando Fox. Using Dynamic Mediation to Integrate COTS Entities in a Ubiquitous Computing Environment. In Proceedings of the Second International Symposium on Handheld and Ubiquitous Computing 2000 (HUC2k), Bristol, England, September 2000.
15. Tim Kindberg, and Armando Fox. System Software for Ubiquitous Computing. IEEE Pervasive Computing Magazine, vol. 1, no. 2, April-June 2002.
16. Anthony LaMarca, David Koizumi, Matthew Lease, Stefan Sigurdsson, Gaetano Borriello, Waylon Brunette, Kevin Sikorski, and Dieter Fox. PlantCare: An Investigation in Practical Ubiquitous Systems. In Proceedings of the Fourth International Conference on Ubiquitous Computing, October 2002.
17. Anthony LaMarca, David Koizumi, Matthew Lease, Stefan Sigurdsson, Gaetano Borriello, Waylon Brunette, Kevin Sikorski, and Dieter Fox. Making Sensor Networks Practical with Robots. In International Conference on Pervasive Computing (Pervasive 2002), Zurich, Switzerland, August 2002.
18. David Mennie, and Bernard Pagurek. An Architecture to Support Dynamic Composition of Service Components. In 5th International Workshop on Component-Oriented Programming, WCOP 2000, Cannes, France, June 2000.

19. Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The Information Bus ñ An Architecture for Extensible Distributed Systems. In Proceedings of the fourteenth ACM Symposium on Operating Systems Principles (SOSP), North Carolina, December 1993.

20. Joseph A. Paradiso, Mark Feldmeier. A Compact, Wireless, Self-Powered Pushbutton Controller. In Proceedings of the third International Conference on Ubiquitous Computing, Atlanta GA, Sept. 2001.

21. Rain. http://seattleweb.intel-research.net/projects/rain/.

22. Nathan S. Shenck, Joseph A. Paradiso. Energy Scavenging with Shoe-Mounted Piezoelectrics. IEEE Micro, Vol. 21, No. 3, May-June 2001.

23. Joao P. Sousa, and David Garlan, Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. In Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture 2002, Montreal, August 2002.

24. R. Srinivasan. Remote Procedure Call Protocol Specification Version 2, Internet Engineering Task Force, RFC no. 1831, August 1995.

25. Sun Microsystems Corporation. Java Remote Method Invocation Specification, Rev 1.7, Palo Alto, California, December 1999.

26. Sun Microsystems Corporation. The Jini Architecture Specification, Ver 1.2, Palo Alto, California, December 2001.

27. Sun Microsystems Corporation, Jini Device Architecture Specification, Rev 1.2, Palo Alto, California, December 2001.

28. David L. Tennenhouse, Jonathan M. Smith, W. D. Sincoskie, David J. Wetherall, and Gary J. Minden, A Survey of Active Network Research, IEEE Communications Magazine, Vol. 35, No. 1, pp80-86. January 1997.

29. Universal Plug and Play. http://www.upnp.org/.