Supporting Generalized Context Interactions

Greg Hackmann, Christine Julien, Jamie Payton, and Gruia-Catalin Roman

Department of Computer Science and Engineering
Washington University in St. Louis
Campus Box 1045, One Brookings Drive
St. Louis, MO 63130-4899, USA
{ghackmann, julien, payton, roman}@wustl.edu

Abstract. Context-awareness refers to a computing model where application behavior is driven by a continually-changing environment. Mobile computing poses unique challenges to context-sensitive applications and middleware, including the ability to run on resource-poor devices like PDAs and the necessity to limit assumptions about the underlying network. Though middleware exists to provide context-awareness to applications, they have not been designed with the limitations inherent in dynamic mobile environments in mind. This paper discusses a lightweight approach to context-sensitivity that takes into account these considerations. We explore the use of modularization to tailor service discovery policies for specific applications, as well as leveraging existing language constructs for simplifying the creation and aggregation of different context types. We also discuss a Java implementation of these concepts, along with three sample applications that can automatically propagate changes in context to clients running on devices varying from mobile phones to desktop computers.

1 Introduction

Traditionally, context-aware computing refers to an application's ability to adapt to changes in its environment. For example, calendar or reminder programs [1] use time to display pertinent notifications to users. Tour guide applications [2, 3] display different information based on the user's current physical location. Still other programs implicitly attach context information to data, e.g., to research notes taken in the field [4]. Each of these applications independently gathers context information from the required sensors and tailors the provision of context.

With the increasing popularity of communicating mobile devices, context-aware computing has moved from a target environment of an autonomous device gathering information single-handedly to a sophisticated network of connected devices, all providing context information to each other. This enables powerful context-aware applications that allow complex interactions across a dynamic network of heterogeneous devices. Presenting this context information to software engineers, however, has received little attention. Building existing context-aware applications like those mentioned above required each developer to individualize

his treatment of context information and independently construct mechanisms to monitor and collect the necessary context information.

In this paper, we introduce CONSUL, a middleware solution that simplifies access to context information. By providing abstractions, we allow novice programmers to build applications that utilize context information collected from a heterogeneous environment. We significantly simplify the development task by removing the need to handle the intricate network programming necessary to collect the information and instead present an accessible yet expressive and extensible interface for using context information.

In the next section, we outline the requirements of a context monitoring middleware for dynamic mobile environments. Section 3 examines existing solutions and evaluates them for use in our target environment. Section 4 details the architecture and implementation of CONSUL, and Section 5 discusses three sample applications developed with the middleware. Finally, conclusions appear in Section 6.

2 Problem Definition

Context items are pieces of data sensed about the environment e.g., location, temperature, link latency, etc. The environment is open, meaning hosts contributing context information can join or leave the network at any time. We assume a heterogeneous and dynamic environment containing resource constrained devices such as environmental sensors, cellphones, PDAs, and laptops.

The programming tasks associated with collecting and monitoring context in such an environment can be burdensome. A programmer must identify the desired source, contact the provider and collect the context items, and interpret the information. Typically, these tasks are achieved through the use of network programming mechanisms that require the programmer to know the identity and location of the context provider. In open and dynamic environments, it is often infeasible to rely on such a priori knowledge. Mobility compounds the problem since the movement of context providers requires the programmer to manage network disconnections. In addition, given the wide array of devices available and the multitude of applications that run on them, the collected pieces of context are likely to be captured in diverse formats that require unification. Finally, the set of available context items available is not static; applications continuously inject context items into the operational environment.

We aim to simplify application development by reducing the complexity of handling context collection and monitoring in dynamic environments. We achieve this goal through a middleware that hides the details of these tasks. The following are requirements of such a middleware infrastructure.

- **Decoupled communication.** We must assume no advance knowledge of communication partners.
- **Transparent monitoring of context.** Issues associated with distribution, mobility, and unpredictable connectivity between users and providers of context should be hidden. Moreover, the process of determining how context

changes are presented to an application should be relegated to the infrastructure.
- **Generalized treatment of context.** Context should be generalized so that applications interact with different types of context information in a similar manner.
- **Extensibility.** Given the openness of the environment, the infrastructure should adapt to the inclusion of new context users and providers with little or no intervention from a system administrator.
- **Scalability.** To scale to large networks, a decentralized solution is necessary.
- **Accommodate small devices.** The middleware primitives must have a lightweight implementation to account for resource-constrained participants.

In the remainder of the paper, we examine how current solutions fall short of meeting these requirements and propose a new middleware infrastructure designed to facilitate rapid development of context-aware applications.

3 Related Work

Though current approaches to context-aware middleware contain weaknesses that limit their applicability to dynamic environments, they also contain many desirable characteristics. It is important to identify these beneficial features so that they may be preserved in CONSUL, while at the same time noting their shortcomings so that they can be rectified. In this section, we review examples of systems which support context-aware application development. We do not attempt an exhaustive review, focusing instead on three well-known systems that have interesting features: Stick-e Notes, CALAIS, and the Context Toolkit. For brevity, other such context-aware support systems such as CoolTown [5], Gaia [6], and Confab [7] are not discussed here; the interested reader is referred to their references.

3.1 Stick-e Notes

Stick-e Notes [8, 9] serves as a precursor for many context-sensitive middlewares. Notably, its approach favors ease-of-use, which differentiates it from many other models. In Stick-e Notes, virtual "sticky notes" are attached to physical phenomenon like times, places, and events. The decision of when a note is in context is included within the note itself; the SGML structure of the Stick-e Note includes a section to semantically describe when a note is to be triggered. Exactly what it means to trigger a note, and what that note's contents describe, is left to the discretion of the client application.

This model is unique in that end-users need only basic knowledge of SGML and no programming knowledge to create notes. However, significant trade-offs are made for the sake of ease-of-use. First, the context is determined by the note itself and not the client, which limits flexibility. For example, a note may be triggered when the user enters a certain range of locations. However a user

traveling in a car may wish to trigger notes within a greater range of distances than a user on foot; this is not possible under the Stick-e Notes model, since this decision is not made by the client. In addition, the model provides no way to disseminate these notes; it is simply assumed that the client either has them in local storage or can obtain them using some external mechanism. This limits the applicability of this model to dynamic environments.

3.2 CALAIS

CALAIS [10] offers an alternative for providing location-based context. It provides a platform-independent way for applications to register themselves with sensors, which in turn use callbacks to notify applications of changes in their state. Location information is stored in a central database, which tracks physical objects (like Active Badges [11]) and uses spatial algorithms to determine which room contains these objects. This location information can be automatically delivered to registered applications. A simple language allows individual contexts to be aggregated into more-complex contexts. The use of callback and context aggregation addresses the most serious shortcomings of the Stick-e Notes design by allowing the client to determine context from a number of sources, which automatically notify the client of any change in state.

CALAIS relies extensively on CORBA, which is heavyweight and thus poorly-suited for many mobile devices. Additionally, it is geared for a specific type of contextual information. Finally, the design of the location service necessitates a central server capable of processing complex spatial relationships, which places extra requirements on the network and raises issues of performance in an environment where many objects are being simultaneously tracked.

3.3 Context Toolkit

The Context Toolkit [12] expands on the idea of context-sensitivity while avoiding many of the shortcomings of the Stick-e Notes and CALAIS models. Hooks are provided for automatically discovering varied kinds of resources (known as "widgets") that provide context, and these contexts can be aggregated within the middleware to form more complex contexts. Unlike CALAIS, Context Toolkit does not depend on any specific back-end for communication between devices; by default it uses XML over HTTP for communication, but this can be swapped out to accommodate networks where this is not feasible.

This model is not without its own shortcomings. First, the Context Toolkit middleware is extremely large and complex, which restricts its use to devices capable of handling such a large code library. As noted in [13], this complexity also hinders the task of creating new widgets. Finally, the movement of context aggregation functionality away from the client and into the middleware unnecessarily limits the type of aggregations that can be performed.

3.4 Observations

Despite their shortcomings, these systems identify several desirable characteristics of context-sensitive middleware. These characteristics, further refined in [14], form a list of challenges that must be met when writing such a middleware. First, the context providing infrastructure must work independently of platform and programming language. Second, the system should adapt to the addition, replacement, or removal of resources providing context. Moreover, the system should be able amenable to mobile, dynamic environments, providing the ability to adapt to changing contexts and a mechanism to propagate context changes to applications. Third, the infrastructure must require minimal administration to be able to scale to large numbers of devices. Fourth, context should be treated universally to promote code reuse. Finally, to allow incorporation of resource-constrained devices, the middleware must remain lightweight.

4 A Middleware for Environmental Monitoring

From the above review, we can see that existing solutions fall short of meeting application needs, most specifically in handling the needs of applications on resource-constrained devices operating in highly dynamic networks. To address these concerns, we have developed CONSUL (CONtext Sensing User Library). In the creation of this middleware, the overarching goal was to provide application developers access to context information through a simplified interface that eases the programming task and places the ability to build context-aware applications in the hands of novice programmers. The architecture of the resulting middleware is shown in Figure 1. In the figure, the solid gray components de-

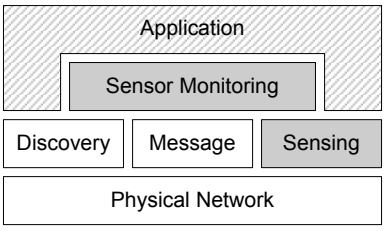

Fig. 1. The architecture of an application using the sensor monitoring and sensing capabilities.

fine CONSUL. The white components we assume to exist, and the cross-hatched component is what an application developer provides. Throughout this section, we will discuss the implementation of the components of this middleware and show how application programmers use the infrastructure to build expressive and flexible context-aware applications.

4.1 Foundational Components

In building a middleware for sensing information from a dynamically changing network, we assumed the existence of several service components. As shown in Figure 1, CONSUL builds on a physical network. This layer of the architecture includes the physical hosts and the connections (wired or wireless) that allow the hosts to communicate. On top of this, our middleware also relies on an existing message passing mechanism.

The final component in Figure 1 that we assume to exist is a component for network discovery that allows a host to find its neighbors. For the remainder of this paper, unless otherwise explicitly specified, we are relying on the simplest discovery mechanism: one that keeps a host informed of all of its one-hop neighbors, i.e., all other hosts in the network that the host can directly communicate with. We have intentionally separated the choice of discovery mechanism from the sensing functionality because different application domains will require different definitions of network neighborhoods. By allowing each application to select its own discovery mechanism, CONSUL remains as flexible and general as possible.

4.2 CONSUL

As shown in Figure 1, two components contribute to providing the environmental monitoring functionality: the sensing component and the sensor monitoring component. Figure 2 shows the internal class diagrams for these two components and how they interact with each other and the application.

Sensing. The sensing component allows software to interface with sensing devices connected to a host. Each of these sensing devices has a corresponding piece of software, a *monitor*. In CONSUL, each monitor extends an `AbstractMonitor` base class that provides unified functionality. In general, each monitor contains its current value in a variable (e.g., the value of a location monitor might be represented by a variable of type `Location`) and contains a method that allows applications to access the value (through the `getMonitorValue()` method). An application can also react to changes in the values by implementing the `MonitorListener` interface and registering itself with the particular monitor. When a programmer extends the sensing functionality to add a new monitor, he must extend the `AbstractMonitor` base class. The extending class must ensure that the monitor's `value` is kept consistent with the current state of the environment. Changes to this variable should be performed through the `setValue()` method in the base class to ensure that any listeners registered for changes to the variable are notified.

Figure 3 demonstrates the programming task through an example class that extends `AbstractMonitor` to collect information from a GPS device. From the perspective of our package, the important pieces are how the extending class interacts with the base class. The details of communicating with a particular GPS device are omitted; their complexity is directly dependent on the particular device and its programming interface.

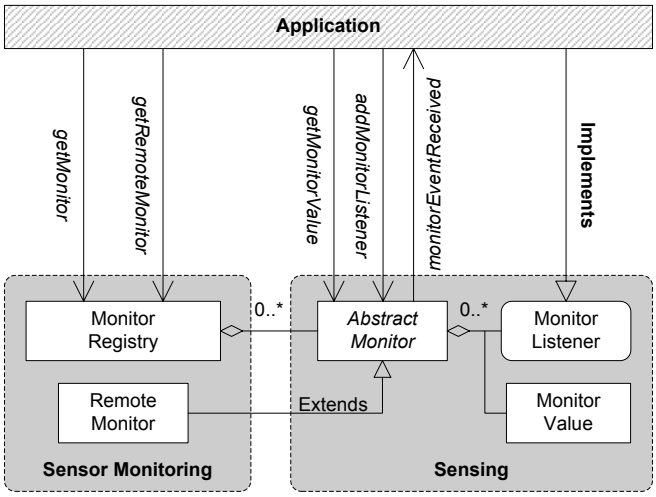

Fig. 2. The internal class diagrams of the sensor monitoring and sensing components of CONSUL

```
public class GPSMonitor extends AbstractMonitor{
  public GPSMonitor(...){
    //call the AbstractMonitor constructor
    super("GPSLocation");
    //set up serial connection to GPS receiver
    ...
  }
  public void serialEvent(SerialPortEvent event){
    //handle periodic events from GPS receiver
    ...
    //turn GPS event into a GPSLocation object
    ...
    //set local value variable, notify listeners
    setValue(gpsLocation);
  }
}
```

Fig. 3. The GPSMonitor Class

To assist application developers with the use of the sensing component of the architecture, we provide several `MonitorValue`s that they can use when building their software monitors or when constructing more complex `MonitorValue`s. These values reside in a library that application developers can add new types to as they create them. For example, the library contains an `IntValue` that can be used for sensors whose state can be represented as a single integer value. There are also aggregate values in the library, e.g., `DateValue`, that build on

```
public class GPSLocation extends ArrayValue {
  public GPSLocation(double latitude, double longitude) {
    super(new IMonitorValue [] {
     new DoubleValue(latitude), new DoubleValue(longitude)
    });
  });
  public double getLatitude() {
    return ((DoubleValue)getValues()[0]).getValue();
  }
  public double getLongitude() {
    return ((DoubleValue)getValues()[1]).getValue();
  }
}
```

Fig. 4. The GPSLocation Class

the simple value types. In addition to being available for developers to use, they also serve as examples for defining new values for new monitors. Figure 4 shows a class that extends `ArrayValue` to aggregate GPS coordinates (represented by `DoubleValue`s).

Sensor Monitoring. The higher-level sensor monitoring component we provide maintains a registry of monitors available on the local hosts and hosts within the neighborhood (as determined by the discovery package). The former are referred to as *local monitors* and the latter as *remote monitors*. As described above, the former are created to make the services available on a host accessible to applications. To monitor context information on hosts in the network neighborhood, the monitor registry creates `RemoteMonitor`s that connect to concrete monitors on remote hosts. These `RemoteMonitor`s serve as proxies to the actual monitors; when the values change on the monitor on the remote host, the `RemoteMonitor`'s value is also updated. This is accomplished within the sensor monitoring component using the `MonitorListener` interface. To gain access to local monitors, the application requests them by name (e.g., "Location") from the registry. The registry returns a handle to the local monitor to the application. To access remote monitors, the application must also provide the ID of the host (which can be retrieved from the particular discovery package in use) and the name of the monitor. When this request occurs (through the `getRemoteMonitor()` method in the registry), the monitor registry creates the proxy on the local host, connects it to the remote monitor, and returns a handle of the proxy to the application. The application can then interact with the remote monitor as if it was a local monitor.

The next section shows how developers use CONSUL by creating monitors, registering them, and interacting with both local and remote monitors.

5 Example Applications

In this section, we present three applications including a stock viewer, a smart room, and support of a mobile communication protocol. For each of these applications we show how using CONSUL extensively simplified the programming task.

5.1 Stock Viewer

In our first example, an application provides stock quotes to handheld devices. Behind the scenes, one or more central servers advertise stock information by acting as monitors. A client running on J2ME-enabled devices automatically discovers these advertised stocks and displays them to the user. The user can then select a stock from the list of monitors and view its current value. As discussed below, implementing the stock ticker using the CONSUL middleware is fairly straightforward and requires minimal "from scratch" coding.

Fig. 5. Left: the client stock ticker on a mobile phone emulator, displaying a list of discovered stock monitors. Right: the client displaying the value of a selected stock.

A stock's value consists of its ticker symbol; current, low, and high dollar values; trading volume; and company name. Creating a custom `StockValue` class simply requires aggregating the predefined `StringValue`, `IntValue`, and `DoubleValue` classes in an `ArrayValue`.

The stock monitor inherits its ability to automatically notify clients of changes from the existing `AbstractMonitor` class. The implementation of the `StockMonitor` class requires only a call to `AbstractMonitor`'s `setValue()` method to update its value and propagate the updated value to clients. The server benefits from similar substantial code re-use. As shown in Figure 6, the

server is implemented using only a few calls to existing classes in CONSUL. The simple code snippet shown assembles a fully-functioning context server from the CONSUL components and the two classes mentioned above. In the code shown, the first three lines start a device-discovery server. Then, a registry is created on a particular port (p) using a simple constructor to allow remote hosts to query local monitors on port p. The final lines are specific to the stock ticker application; they create local monitors for the MSFT, YHOO, and T stock tickers.

```
DiscoveryServer discovery = DiscoveryServer.getServer();
discovery.setProxy(true);
discovery.start();
MonitorRegistry registry = new MonitorRegistry(p);

registry.addMonitor(new StockMonitor("MSFT"));
registry.addMonitor(new StockMonitor("YHOO"));
registry.addMonitor(new StockMonitor("T"));
```

Fig. 6. A Stock Ticker Server

The client application gains most of its functionality from the predefined discovery server and monitor registry components. Once the code for the user interface (UI) has been written, adding all of the functionality to find and update stock context information is almost trivial: four lines of code to begin finding stock servers, six lines to listen to monitors on discovered servers, and two lines to receive updated stock values. The extensive code re-use illustrated above allows rapid development of context-aware applications, shifting development effort away from the back-end and allowing more development time to be focused on the UI.

Though this sample application only includes one client application, the use of CONSUL means that the server is not tied to any specific client. This allows for a wide variety of custom clients that can present stock information in different ways, such as a client that runs in the background and pops up a notification when a specific stock reaches a certain price.

Since CONSUL is extremely lightweight, the client can easily be run on resource-poor devices like mobile phones, provided they have a J2ME runtime and are network-enabled. The stock viewer application bundled with all of the required libraries consumes only 48 kilobytes of storage space. This small size also means that it is feasible to send the application to mobile devices over-the-air and discard it when the user is done.

The use of context-aware middleware places one restriction on the devices being used as clients: they need full IP networking support. In this sample application, the server obtains its stock information from a Web service. Web services have one distinct advantage over full-scale context-aware middleware: the clients

only need basic HTTP networking support as opposed to full IP support. This drawback does not necessarily reflect a general shortcoming of our middleware, but rather raises the issue of how useful context-sensitivity is in this scenario; stock information is widely available from multiple well-known sources on the Internet. Despite our middleware's small footprint, from a practical point-of-view it may still be overkill for retrieving stock quotes. The need for context-sensitive middleware is more pressing when the sources of contextual information are not public or must be discovered at runtime, as in the next example.

5.2 RFID-Activated Smart Room

In our second sample application, a server uses an attached RFID scanner to provide information about who is currently in a room. Information about the presence of users is provided to the server through the use of wearable RFID tags that interact with an RFID scanner. A separate client computer has predefined playlists for each person who might enter the room. This computer uses the continuously-updated list of people in the room to play music using the freely available Winamp media player; it selects music from a "master" playlist made by intersecting the playlists of everyone in the room.

This client computer in turn has a monitor that provides information about the current song being played, which can be displayed on a handheld computer. Another computer combines the MP3 player's context with the RFID context information to serve a Web page with a list of people currently in the room and the music currently playing.

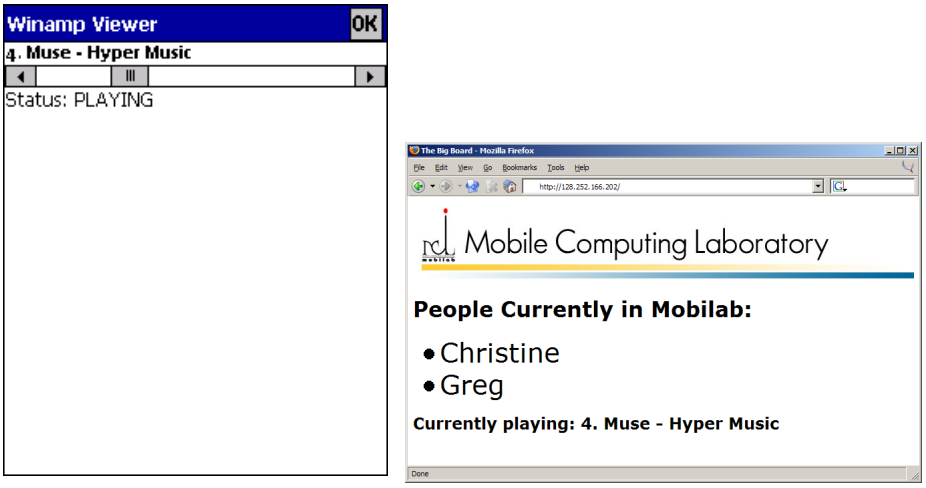

Fig. 7. Left: a client running on a PocketPC showing the current song being played. Right: a Web page showing the current song being played and the people in the room.

Unlike the previous example, these contexts are not publicly available on the Internet nor available from well-known sources, so the use of device discovery is integral to this application. We also expect the playlist to be updated in real time as people enter and leave the room. This means that the client requires "push" service, which CONSUL provides; Web services inherently provide "pull" information only.

The code to add context-sensitivity to the smart room application is very similar to that in the previous section. Thus, for the sake of space we will not reiterate the implementation in detail. Interestingly, the computer running the MP3 player acts both as a recipient of context information (a list of people in the room) and as a provider (the current song being played). Once the RFID client was written, extending it to serve the MP3 player's contextual information required only a single line of code to instantiate a `WinampMonitor` and add it to the same `MonitorRegistry` that was already being used to receive contextual information from the RFID server.

Since all of the context monitors in the room are re-usable components, extending this application is straightforward and transparent to the application. For example, a computer connected to an X10 controller could use the RFID context to automatically turn the lights off when the room is empty.

This application demonstrates that the monitors and values constructed using CONSUL are re-usable components just as CONSUL itself is. Multiple clients use the contextual information provided by the RFID monitor and MP3 player monitor for different purposes, and the Web server drew context from multiple services. The `MonitorRegistry` class handles this transparently to the programmer, so the programmer is simply assembling applications from re-usable components.

Currently, the device discovery mechanism does not allow programmers to search for specific types of devices. Instead, clients interested in context information are given a list of all the devices in the room. They must then collect a list of monitors running on the devices and select one or more monitors based on their names. This requires us to make the assumption that the names of the monitors reflect their function. For example, the clients interested in RFID information search for monitors named "RFID" on discovered devices. This problem can be avoided by replacing the existing device discovery mechanism with a more-sophisticated method of discovering devices.

5.3 Ad Hoc Mobile Communication Protocol

The Network Abstractions protocol [15] provides context-sensitive routing in ad hoc networks. To accomplish this, the protocol requires monitoring the values of sensors on the local host and on hosts that are directly connected in an ad hoc wireless network. This protocol allows an individual application to limit its operating context to a neighborhood within the ad hoc network. To allow the size and scope of this neighborhood (or context) to be determined by the application-specific needs, each application can specify an abstract metric over arbitrary properties of hosts and links in the network. This metric calculates

a logical distance from the application's local host to any other host in the network. The metric includes a bound on allowable distances that restricts the hosts belonging to the neighborhood. As a simple example, an application might want to communicate with all other hosts within three miles.

An implementation of this protocol can benefit from the use of CONSUL in providing access to the properties of hosts and links that define the protocol's metrics. Coupled with a network discovery component that maintains a list of exactly the one-hop neighbors, CONSUL relieves the protocol implementer and user from concerns associated with maintaining a consistent view of the values of the relevant sensors on the local host and remote hosts. For example, when the protocol wants to build a new context that it maintains over time, even as the properties of the network changes, it can use the code in Figure 8 to register itself as a listener for the appropriate monitors. When changes in the monitor values occur, the protocol is automatically notified and can change the structure of the routing paths as needed.

```
ContextMonitorListener cml = new ContextMonitorListener(...);
AbstractMonitor m = registry.getMonitor("GPSLocation");
m.addMonitorListener(cml);

for(int i=0; i<neighbors.length; i++){
   AbstractMonitor m2
            = registry.getRemoteMonitor("GPSLocation", neighbors[i]);
   m2.addMonitorListener(cml);
}
```

Fig. 8. A Portion of the Network Abstractions Protocol using CONSUL

The code in the figure explains how, on behalf of a single application, the protocol uses CONSUL to provide the information needed to build a network abstraction based on relative physical locations. The first line of the displayed code simply creates an instance of a monitor listener (the `ContextMonitorListener`). The protocol then retrieves the local instance of the `GPSMonitor` and adds the created listener to the monitor. This allows the protocol to be notified when the local host's location changes. Because the network abstractions metric is based on the physical distance between hosts in the network, the protocol must also register as a listener for changes in all of the one-hop neighbor's locations. This functionality is contained in the second portion of the code that, for each neighbor (in a list retrieved from a neighbor discovery component), the application adds its listener to the remote location monitor on the neighbor. Not shown in this code snippet is the fact that, when new neighbors are discovered, a listener must be added to their location monitors, and when neighbors move away, the listeners must be removed. Additional code within the listener also handles the

reception of monitor events to adjust the metric values with the locations of the involved hosts change.

5.4 Comparisons and Lessons Learned

These sample applications demonstrate the flexibility of CONSUL. They include components running on desktops with a full J2SE runtime, a PocketPC equipped with a J2ME Personal Profile runtime, an emulated mobile phone with support for J2ME MIDP, and a variety of other mobile devices. This flexibility comes from its small footprint as well as the fact that it relies on no language-dependent features. In comparison, CALAIS and Context Toolkit have large footprints and would very likely not run on smaller devices like PocketPCs or mobile phones.

The applications presented in this section required little programming effort to transform a stand-alone utility or viewer into a context-aware application. This is because CONSUL encapsulates all the functionality needed to find and propagate contextual information across a network. To implement the same applications in Stick-e Notes, additional code would be required to propagate context to clients, since no such mechanism is included in the Stick-e Notes library.

The second sample application demonstrates that CONSUL promotes separation of concerns, modularity, and code reuse. In the smart room application implementation presented here, custom-made monitors and values were effectively separated into re-usable components. A relatively complex smart room was built using simple components, such as an RFID monitor and an MP3 player that used the RFID context to select a playlist. This application also demonstrates that CONSUL promotes the development of extensible context-aware systems.

6 Conclusions

The applications described above highlight the ways that CONSUL fits the requirements outlined previously: portability, adaptability, scalability, and applicability to small devices. Since CONSUL does not rely on any features specific to any version of Java, it has been implemented on platforms ranging from desktop computers to mobile phones. Though only a Java implementation is discussed in this paper, CONSUL could also be re-implemented in any programming language with support for IP networking and threading.

The smart room example highlights CONSUL's use of device discovery to find and replace sources of contextual information. Should the RFID server suddenly disconnect from the network and be replaced, the MP3 player would automatically discover the new server and register itself to begin receiving RFID context from the new source. Though CONSUL provides extensive context-sensitive functionality to its users, its underlying communication mechanism is very simple. Messages and context values are passed between hosts using small packets sent over standard TCP/IP sockets. If we assume that the device discovery mechanism used scales well, then CONSUL scales no worse than any standard client/server application. Finally, CONSUL has an extremely small footprint. We

have even created applications for an emulated mobile phone, whose resources and feature set are even more limited than PDAs.

Nevertheless, CONSUL still has room for further development. Permitting two-way communication between clients and servers would allow clients to take advantage of a wider range of services; for example, monitors can currently advertise the status of a printer but not allow clients to send documents to the printer. A CONSUL interface to Web services would also greatly increase the range of services immediately available to a system. Finally, using information available about the network at runtime (like bandwidth and number of devices) to select between multiple device discovery mechanisms could eliminate even the minimal manual administration involved in selecting a mechanism appropriate for the underlying network.

References

1. Dey, A.K., Abowd, G.D.: Cybreminder: A context-aware system for supporting reminders. In: Proceedings of the 2^{nd} International Symposium on Handheld and Ubiquitous Computing. (2000) 172–186
2. Abowd, G., Atkeson, C., Hong, J., Long, S., Kooper, R., Pinkerton, M.: Cyberguide: A mobile context-aware tour guide. ACM Wireless Networks **3** (1997) 421–433
3. Cheverst, K., Davies, N., Mitchell, K., Friday, A., Efstratiou, C.: Experiences of developing and deploying a context-aware tourist guide: The GUIDE project. In: Proceedings of MobiCom, ACM Press (2000) 20–31
4. Pascoe, J.: Adding generic contextual capabilities to wearable computers. In: Proceedings of the 2nd International Symposium on Wearable Computers. (1998) 92–99
5. Kindberg, T., Barton, J.: A Web-based nomadic computing system. Computer Networks (Amsterdam, Netherlands: 1999) **35** (2001) 443–456
6. Romn, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R.H., Nahrstedt, K.: A middleware infrastructure for active spaces. IEEE Pervasive Computing **1** (2002) 74–83
7. Hong, J.I., Landay, J.A.: An architecture for privacy-sensitive ubiquitous computing. In: Proceedings of the 2nd international conference on Mobile systems, applications, and services, ACM Press (2004) 177–189
8. Brown, P.J.: The stick-e document: a framework for creating context-aware applications. In: Proceedings of EP'96, Palo Alto, also published in *EP–odd* (1996) 259–272
9. Brown, P.J., Bovey, J.D., Chen, X.: Context-aware applications: from the laboratory to the marketplace. IEEE Personal Communications **4** (1997) 58–64
10. Nelson, G.J.: Context-Aware and Location Systems. PhD thesis, University of Cambridge (1998)
11. Want, R., Hopper, A., Falcão, V., Gibbons, J.: The active badge location system. Technical Report 92.1, Olivetti Research Ltd. (ORL), 24a Trumpington Street, Cambridge CB2 1QA (1992)
12. Dey, A.K.: Providing Architectural Support for Building Context-Aware Applications. PhD thesis, Georgia Institute of Technology (2000)

13. Yellin, D.M.: Stuck in the middle: Challenges and trends in optimizing middleware. SIGPLAN Not. **36** (2001) 175–180
14. Hong, J., Landay, J.: An infrastructure approach to context-aware computing (2001)
15. Roman, G.C., Julien, C., Huang, Q.: Network abstractions for context-aware mobile computing. In: Proceedings of the 24^{th} International Conference on Software Engineering. (2002) 363–373