HUMAN-COMPUTER INTERACTION, 2001, Volume 16, pp. 287–303

An Infrastructure Approach to Context-Aware Computing

Jason I. Hong and **James A. Landay**
University of California at Berkeley

ABSTRACT

The Context Toolkit (Dey, Abowd, and Salber, 2001 [this special issue]) is only one of many possible architectures for supporting context-aware applications. In this essay, we look at the tradeoffs involved with a service infrastructure approach to context-aware computing. We describe the advantages that a service infrastructure for context awareness has over other approaches, outline some of the core technical challenges that must be addressed before such an infrastructure can be built, and point out promising research directions for overcoming these challenges.

1. INTRODUCTION

People have always used the context of the situation to get things done. We use our understanding of current circumstances to structure activities, navi-

Jason Hong is a computer scientist with an interest in context-aware computing and multimodal interfaces; he is a PhD candidate in the Group for User Interface Research at University of California at Berkeley. **James Landay** is a computer scientist with interests in informal user interfaces, multimodal interaction, and design tools; he is an Assistant Professor in the Computer Science Division at University of California at Berkeley and co-leader of the Group for User Interface Research.

CONTENTS

gate the world around us, organize information, and adapt to conditions. For example, when we're holding a conversation in a noisy place, we talk louder so that the other person can hear. But when we're in a meeting, we whisper so as not to disturb other people.

Context awareness has also been an integral part of computing. Even simple forms of context, such as time and identity, have been used in a number of meaningful ways. For example, by being aware of the current time, computers can give us reminders of calendar events. By being aware of our identity through logins, computers can personalize the look and feel of our user interface. Different kinds of context can also be used together. For example, computers can tag files with both time and identity, giving us many ways of organizing and finding information created in the past.

Strides in miniaturization, wireless networking, and sensor technologies are enabling computers to be used in more places and to have a greater awareness of the dynamic world they are a part of. In fact, over the past few years, a new class of context-aware applications that make use of these technologies has been developed, showing how computers can leverage even elementary notions of location, identity, proximity, and activity to great effect (e.g., Active Badges, Want, Hopper, Falcao, & Gibbons, 1992; ParcTabs, Want et al., 1995; and Cyberguide, Abowd et al., 1997). The key to these context-aware applica-

tions was that they provided tighter ties between the physical and social worlds we live in and the virtual world in which computers operate.

These prototypes have demonstrated the potential of context-aware applications but have also shown that these kinds of systems are still extremely difficult to design, develop, and maintain. There remain a number of technical challenges that must be overcome before even simple context-aware systems can be widely deployed and realistically evaluated. Some recent research has focused on developing frameworks and toolkits to assist development, including the system for ParcTabs (Schilit, 1995), Stick-E Notes (Pascoe, 1997), and MUSE (Castro & Muntz, 2000). The most complete work in this area has been the Context Toolkit by Dey, Abowd, and Salber (2001 [this special issue]).

Our work takes inspiration from many of the ideas in these projects but recasts them in the light of *service infrastructures*. Our position is that to greatly simplify the tasks of creating and maintaining context-aware systems, we should shift as much of the weight of context-aware computing onto network-accessible middleware infrastructures. By providing uniform abstractions and reliable services for common operations, such service infrastructures could make it easier to develop robust applications even on a diverse and constantly changing set of devices and sensors. A service infrastructure would also make it easier to incrementally deploy new sensors, new devices, and new services as they appear in the future, as well as scale these up to serve large numbers of people. Last, a service infrastructure would make it easier for sensors and devices to share sensor and context data, placing the burden of acquisition, processing, and interoperability on the infrastructure instead of on individual devices and applications.

In the following sections, we discuss the advantages of an infrastructure approach to context-aware computing in more detail, comparing it to some design decisions taken in the Context Toolkit by Dey et al. We then outline some of the core technical challenges involved and describe some promising directions for building such an infrastructure.

2. LIBRARIES, FRAMEWORKS, TOOLKITS, AND INFRASTRUCTURES

Before going on, it's important to make the distinction between different kinds of software support for building context-aware applications. In general, software support for applications can be classified as libraries, frameworks, toolkits, or infrastructures. These approaches are not mutually exclusive: There are cases where it's useful to have all of these.

A *library* is a generalized set of related algorithms. Examples include code for manipulating strings and for performing complex mathematical calculations. Libraries focus exclusively on code reuse. On the other hand, *frameworks*

concentrate more on design reuse by providing a basic structure for a certain class of applications. Frameworks shoulder the central responsibilities in an application but provide ways to customize the framework for specific needs. *Toolkits* build on frameworks by also offering a large number of reusable components for common functionality. So a graphical user interface event dispatching system would be an example of a framework, and a corresponding toolkit would provide buttons, checkboxes, and text entry fields for that framework. The Context Toolkit fits pretty well under this definition of a toolkit, as it offers a framework for sensor-based context-aware applications and provides a number of reusable components.

An *infrastructure* is a well-established, pervasive, reliable, and publicly accessible set of technologies that act as a foundation for other systems. We are most interested in *service infrastructures*, middleware technologies that can be accessed through a network. Any kind of device or application can use these services by adhering to predefined data formats and network protocols. An example infrastructure is the Internet itself. An example service offered by some computers connected to the Internet is the Domain Name System, which converts computer names such as "www.berkeley.edu" into IP addresses such as "128.32.25.12." All an application needs to know to access the Internet is the TCP/IP suite of network protocols. Once a device has been programmed to understand these, any computer connected to the Internet can be contacted from any location in the world. From a developer's standpoint, the rather complex task of communicating with other computers has been reduced to understanding the data formats and protocols used.

This is one of the key insights into the success of the Internet: The fundamental problem of interoperability can be overcome by separating the desired properties and responsibilities into separate layers and by defining good enough data formats and network protocols at each of these layers. The data formats and protocols are more important than any specific library or toolkit that implements them because they enable computers that know nothing of each other to interoperate.

There are several subtle distinctions between having library, framework, or toolkit support for an application versus having infrastructure support. These distinctions can be illustrated by comparing the differences between a CD-ROM of an encyclopedia versus access to Encyclopedia Britannica's Web site. With the CD-ROM, you need a CD-ROM drive, a fast microprocessor, a specific operating system, and a nontrivial amount of disk space to install and run the encyclopedia application. In contrast, with the Web site service, you just need a simple microprocessor, a network connection, and a Web browser. The Web site is "always on," can be accessed by anyone from any device that has a Web browser, and can be periodically updated with new information and new functionality without mailing out new CD-ROMs to everyone. Simi-

larly, we argue that there are several compelling benefits to having a service infrastructure for context awareness over having just library, framework, or toolkit support. These benefits are described in the next sections.

3. ADVANTAGES TO AN INFRASTRUCTURE APPROACH

Software support for context-aware applications has so far focused on general architectures (e.g., the system for ParcTabs; Schilit, 1995) and frameworks and toolkits (e.g., Stick-E Notes, Pascoe, 1997; the Context Toolkit, Dey et al.; and MUSE, Castro & Muntz, 2000), with only basic notions of infrastructures present. We advocate pushing as much of the acquisition and processing of context into the infrastructure as services that can be accessed by any device and any application. Frameworks and toolkits are useful, but we claim that there are even greater advantages to an infrastructure approach, which benefit not only developers but also administrators maintaining the infrastructure and end-users using the infrastructure.

3.1. Independence From Hardware, Operating System, and Programming Language

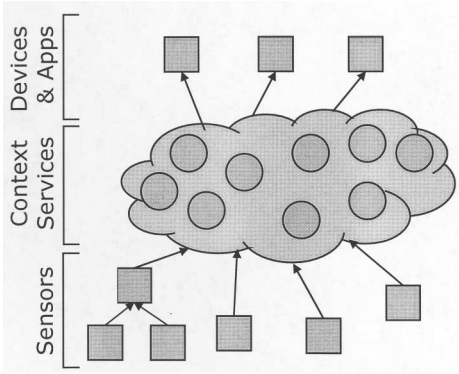
The first benefit of a service infrastructure is that it can be used independently of hardware platform, operating system, and programming language. By using standard data formats and network protocols that can be easily implemented, the infrastructure can support a greater range of devices and applications. This approach makes the infrastructure easier to evolve as new sensors, devices, operating systems, and programming languages appear.

By default, the Context Toolkit uses HTTP as the network protocol and XML as the data format, achieving a certain level of interoperability. The Context Toolkit is also flexible enough to allow different protocols and data formats to be used. This is demonstrated quite nicely by the fact that context widgets have been implemented in C++, Visual Basic, and Python.

3.2. Improved Capabilities for Maintenance and Evolution

A second benefit of a service infrastructure is that sensors, services, and devices can be changed both *independently*, without affecting anything else, and *dynamically*, even while other sensors, services, devices, and applications are running. By providing a middleware layer that presents a uniform level of abstraction, an infrastructure can strictly separate sensors from services and services from devices and applications (see Figure 1). The end result is that the

Figure 1. **An infrastructure for context awareness can provide a middleware layer between sensors on one side and devices and applications on the other. The middleware layer presents a uniform layer of abstraction, making it easier to update individual pieces independently of each other.**



entire system can be incrementally evolved as new sensors, services, devices, and applications appear.

As an analogy, a new computer can be added to the Internet without having to update or restart any other computer. If designed correctly, the same can be true with adding sensors and services to the infrastructure. To add a new sensor, all that is needed is software that connects the sensor to the rest of the middleware. To add a new service, all that is needed is a space in the middleware where services can be uploaded, discovered, and then run when needed. To add a new device, all that is needed is software that understands the protocols and data formats used by the infrastructure.

One positive side effect of this design is that sensors and services can be upgraded in place and be immediately accessible to everyone. For example, suppose we had a service that detected moving objects in a video stream. In the future, faster algorithms will likely be developed for object detection. The service could then be upgraded, and as long as it accepted the same kind of input and provided the same kind of output as before, every application that used this service would still work correctly but also run faster.

In contrast, there is a serious distribution problem with libraries, frameworks, and toolkits. Because the code is stored and run locally on individual devices, every copy of the code has to be updated whenever changes are made.

In this respect, the Context Toolkit exhibits some properties of an infrastructure. New sensors can be added by finding (or creating) the appropriate widget and then writing the code that connects the sensor to the widget. Context widgets can also be upgraded in place by finding the computer running the old widget, stopping the widget, and then running the new one.

As an aside, one problem here is that the Context Toolkit relies on context widgets as the primary abstraction for sensors, but this may not be right in all cases. For example, Active Badges can be used for acquiring both identity and location, but in Figure 2 of Dey et al.'s anchor article the Active Badge is mapped directly to a location widget. To get identity, the Active Badge location (and implicitly the Badge ID) is fed into an interpreter, which returns a user name. This is an awkward mapping because an Active Badge represents both location and identity but can only mapped to one context widget in the Context Toolkit.

Another example where the widget abstraction is insufficient is with smart dust motes (Kahn, Katz, & Pister, 1999). Smart dust motes are wirelessly networked and sensor-based computers with size on the order of tens of millimeters. Motes vary widely in terms of network connectivity, available power, available sensors, and reliability of sensor data. It typically takes a collection of motes to reliably process certain kinds of information, such as temperature and humidity. For this reason, a one-to-one mapping from a mote to a context widget does not make sense, even if it is possible.

In both of these cases, it makes sense to add another layer to separate context data from sensors. This layer would manage and process sensor data before they become context data. Such a layer is not intrinsic to an infrastructure, however, and could be added to the Context Toolkit.

3.3. Sharing of Sensors, Processing Power, Data, and Services

A third benefit of an infrastructural approach is that context-aware devices and applications will be easier to develop and deploy because sensors, processing power, data, and services within the infrastructure can be shared. By sharing sensors, individual devices will not need to carry every type of conceivable sensor to acquire the needed context information. Instead, the burden can be placed on the infrastructure to find suitable nearby sensors. For example, a PDA wouldn't need location sensors if it can simply ask the infrastructure to use neighboring sensors to tell it where it is. A side effect is that applications don't need to be tied to specific platforms just because the platform has specific sensors. If the infrastructure can provide the right kinds of context information, the application can be run on any networked device.

By sharing processing power, devices wouldn't need to have powerful, expensive, and energy-hungry microprocessors. Likewise, by sharing data, devices wouldn't need large amounts of storage. Even though processing power and storage capacity are steadily increasing, there are still many reasons to offload computation and data to the infrastructure. Some algorithms used for context-aware applications, such as speech recognition or image processing,

are computationally expensive and cannot feasibly be run on small devices. In many cases, it makes more sense to have dedicated machines to do this kind of processing. Similarly, it's simply impractical to keep certain kinds of data on individual devices. For example, extremely large data sets (e.g., book ISBN numbers and U.S. zip code numbers) are too large to be feasibly stored on most portable devices. Similarly, highly dynamic data, such as stock prices and traffic information, are updated too often. It makes more sense to keep these kinds of data in the infrastructure than on individual devices. Even personal data can be stored in the infrastructure as long as the data are easily accessible from any device the individual is using. An added benefit of this approach is that devices can be lost or stolen but the data will still be safe.

By sharing services, applications can be smaller and thus easier to store on portable devices. Instead of monolithic and self-contained applications, the bulk of an application's functionality would be in the form of many small services that exist in the infrastructure that applications could simply call. Although there would only be a few services at first, the more applications that are built, the more services there will be that others can use, making it easier to build applications in the future.

This sharing is one philosophical difference between the Context Toolkit and an infrastructure approach. With the Context Toolkit, the implicit desire is to keep the programming model within the current paradigm but add some extensions to handle sensor input. An infrastructure approach differs in two ways. First, put as much functionality as possible into the network to increase utilization, reliability, and sharing, as well as to decrease maintenance. Second, create many small network-based services that can be composed together instead of monolithic applications that reside on devices. It's possible to do this with the Context Toolkit, but it's neither emphasized nor supported.

4. CHALLENGES TO BUILDING A CONTEXT-AWARE INFRASTRUCTURE

Although we have pointed out many advantages to an infrastructure approach for supporting context-aware applications, there are still many technical challenges that must be overcome. We draw special attention to five of these and sketch out some current research and potential directions for addressing each of these issues.

4.1. Defining Standard Data Formats and Protocols

The first challenge lies with designing the data formats and protocols used by the infrastructure. These standards will be the glue that allows the separate pieces of the infrastructure to interoperate. For this reason, they need to be sim-

ple enough that they can be implemented for practically any device and used by any application. A good negative example is the Jini™ coordination framework (Waldo & the Jini Technology Team, 2000). Jini requires clients to have access to a full-fledged Java™ Virtual Machine as well as a large set of Java libraries, seriously restricting the class of devices that can use Jini. To encourage as many people as possible to use the infrastructure, it needs to be as agnostic as possible with respect to hardware platform, operating system, and programming language. The Salutation (Salutation Consortium, 2000) and Universal Plug and Play (Universal Plug and Play Forum, 2000) coordination frameworks are good examples along these lines. The SOAP protocol (Box et al., 2000), currently under consideration as a World Wide Web Consortium standard, is a remote procedure call protocol based on XML and also shows great promise.

Besides being simple, the data formats and protocols also need to be rich enough to cover the diverse range of sensors and assorted types of context. Furthermore, as pointed out by the anchor article, many types of context are inherently ambiguous. The data formats need to address the fact that sensor data are often partial and unreliable, leading to ambiguity in how the context is interpreted. There has been much work in using probabilistic frameworks to model uncertainty, and we believe that this is the most promising approach for modeling context data. One advantage of representing context data probabilistically is that applications can be given a notion of the confidence of the context data before acting on it. Another advantage is that a probabilistic framework makes it easier to fuse context data together to give better results. One example where this is taking place is with MUSE (Castro & Muntz, 2000), which uses Bayesian nets and Hidden Markov Models to model sensor data.

Here, we need to make the distinction between sensor data and context data. Sensor data are unambiguous; however, they have several attributes, including precision, granularity, and accuracy, that affect how they are interpreted as higher level context data. By *precision* we mean the variation in a set of repeated measurements. By *granularity* we mean the smallest unit that can be measured. By *accuracy* we mean the difference between the calculated value and the actual real-world value. To illustrate the difference between these terms, let's suppose we have a global positioning satellite (GPS) device and check it every few seconds. If we are standing still and see that our measured location is jumping around sporadically, then our measurement is not very precise. If we can walk at most a half meter in any direction and not have our measured location change, then our device has a granularity of about 1 m. If we are actually at one location but the measurement says we are 50 m away, then our measurement is not very accurate (for most applications, anyway).

In turn, these attributes affect the interpretation of sensor data as context data. For example, if we have highly accurate GPS location data and want to model the current street we are on, we might say that we are on Street A with

98% confidence and on Street B with 2% confidence. However, if we are using a less accurate system, such as some form of radio triangulation, we might find that we are on Street A with 80% confidence, on Street B with 12% confidence, and on Street C with 8% confidence. The key here is in representing uncertainty in a uniform way and then letting other entities choose what to do. Some applications may choose to use sophisticated probabilistic algorithms, whereas others may simply reject data below a certain threshold.

Another quirk that needs to be modeled in the context data format is that the context may not be available at all. Perhaps the needed sensors may not be at hand, the network is down, a person simply wants to keep certain information private, or the requestor does not have authenticated access to the data. For these reasons, UNKNOWN always needs to be a valid value for all types of context data at all times.

4.2. Building Basic Infrastructure Services

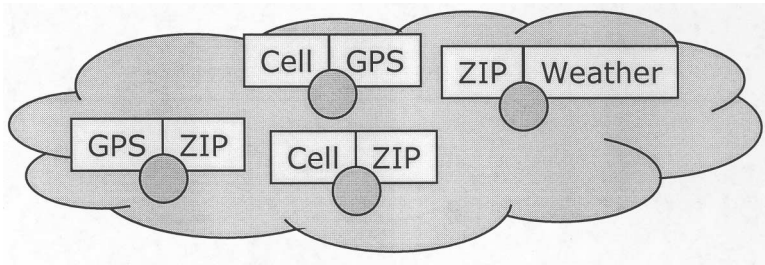
The second challenge to building a context infrastructure lies with designing the services. Some of these context services will be highly application specific and will have to be designed and implemented on a case-by-case basis. However, other context services will be basic enough that they will be an integral part of the infrastructure itself. We describe two such services later: automatic path creation and proximity-based discovery.

Automatic Path Creation

One such service is *automatic path creation.* Previous research on automatic path creation has focused on network protocol and data format interoperability (Kiciman & Fox, 2000; Mao, 2000). However, we believe that automatic path creation can be adapted for context awareness to simplify the task of refining and transforming raw sensor data into higher level context data.

Automatic path creation relies on operators, a special subset of services. Figure 2 shows four operators. The first operator transforms GPS location data to zip code data (denoted GPS à ZIP). The second takes a cellular phone's cell location and calculates the GPS coordinates of the center of the cell (Cell à GPS). Another operator takes cell location, does a lookup to find the zip code, and then returns the zip code. The last operator takes zip code data and returns the current weather conditions for that area. Individually, none of these services are very interesting; however, much like UNIX pipes, they can be chained together into paths to form more interesting services. For example, GPS can be used to retrieve local weather conditions (chaining GPS à ZIP and ZIP à Weather together).

Figure 2. **Operators are a special kind of service that reside in the infrastructure. Operators offer simple services, such as converting GPS data into zip code data. The power of operators comes from the fact that they can be composed into more powerful services.**

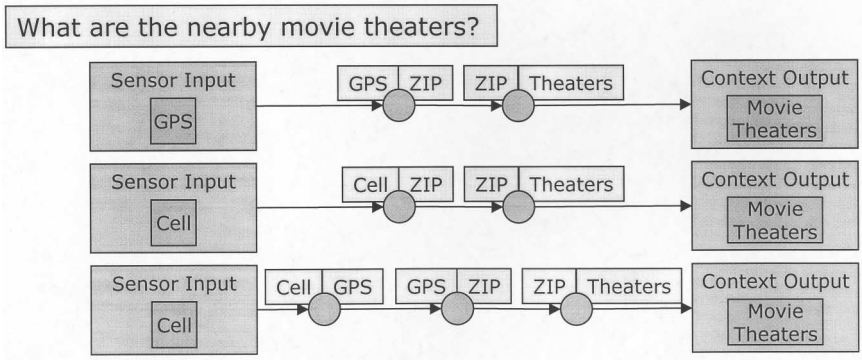


Clearly operators can be combined manually into paths, but the real power comes from the fact that they can be composed automatically based on high-level needs and on whatever resources are available. For example, Figure 3 shows three different ways of answering the question "What are the nearby movie theaters?" In the first case the path goes from GPS à ZIP and ZIP à Movie Theaters. In the second case it goes from Cell Location à ZIP and ZIP à Movie Theaters. In the third case it goes from Cell Location à GPS, GPS à ZIP, and then ZIP à Movie Theaters. With automatic path creation, any of these three paths can be created dynamically on demand depending on what kind of location sensors and services are currently available.

Automatic path creation provides a flexible and high-level abstraction for context awareness. It relieves application developers from having to know about specific sensors and services. Instead, developers only need to worry about formulating the right context query. Furthermore, it provides greater reusability than if the transformations were simply hardwired together as a single monolithic application. For example, now that there is a GPS to zip code operator, all a developer needs to do to get the local traffic report is to create an operator that takes zip code information and retrieves the traffic conditions in that area. Compare this with how context is currently processed in the Context Toolkit. Currently, developers have to manually specify what path context data flow through. Not only is this tedious, it also makes it difficult to acquire context information in changing situations, such as when a person moves from one room full of sensors to another.

The challenge to building an automatic path creation service is fourfold. The first problem is one of engineering. A critical mass of operators must be built before automatic path creation becomes sufficiently useful. The second problem is one of standards. Standard data types need to be developed; otherwise, operators cannot be connected with one another. The third problem is

Figure 3. **Automatic path creation is one basic service that would be provided by a context infrastructure. Given a context query, automatic path creation can incrementally transform raw sensor data into an answer. This figure shows three different paths for computing the answer to the question "What are the nearby movie theaters?" The first case uses GPS sensors. The second and third cases use cell phone location. With automatic path creation, any of these paths can be created on demand based on whatever resources and services are available.**

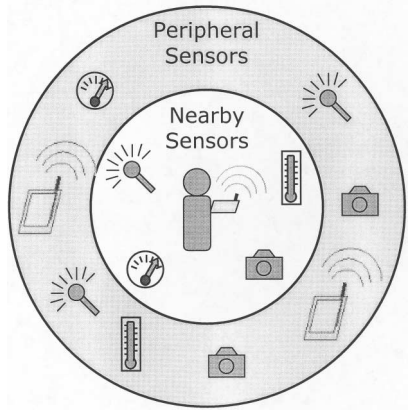


one of building good paths. The system needs a way of selecting a path if there are multiple valid paths. It also needs a way to determine where to run each of these operators. These decisions may also be influenced by performance, quality of service, and uncertainty constraints. The fourth problem is one of representing the context query. The query needs to be rich enough to pose interesting context questions, but also simple enough that it can realistically be understood and processed.

Proximity-Based Discovery

Another basic service is *proximity-based discovery*, which finds all nearby sensors (see Figure 4). Proximity-based discovery is important because many context-aware applications make use of spatial locality. For example, suppose that a meeting capture service wants to know if there is a meeting going on in a given room right now so it can begin recording. Instead of hardwiring the service to use the specific sensors in that room, it can ask the infrastructure to locate the sensors in that room and then use automatic path creation to bridge the gap between the low-level sensor data and the higher level question of "Is there a meeting right now?" Combining proximity-based discovery with automatic path creation makes it easier to design and deploy context-aware applications. Applications don't have to know beforehand exactly which sensors will be used, but as long as the right sensors are there, they will still work correctly.

Figure 4. **Proximity-based discovery is another basic service that would be provided by a context infrastructure. Given a location, the service would find all of the nearby sensors.**

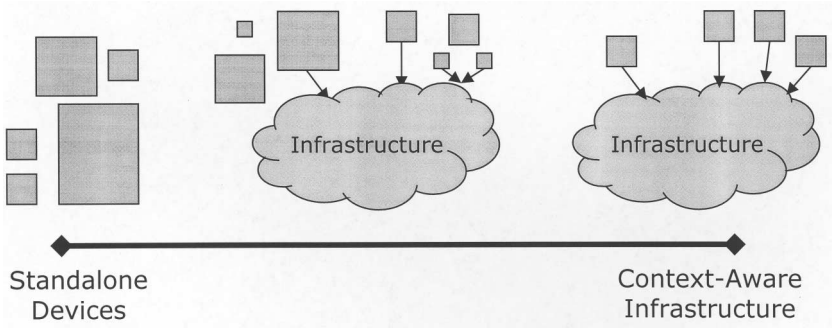


There are three difficulties inherent in proximity-based discovery. The first problem is logistical. Not every sensor will be able to pinpoint its location, meaning that some sensors will have to be manually configured (a tedious proposition). The second problem is representing the location of sensors. For example, a nearby sensor could be "10 meters away," as well as be "on a desk," "in Room 525," "in Building Soda Hall," and "on the campus of the University of California at Berkeley." The representation needs to be flexible enough so that higher level queries can be constructed. The third problem is storing the location of sensors. In some cases, this is obviated by wireless technologies such as Bluetooth™, which can use physical proximity to discover similarly enabled sensors. The trouble is that not all sensors will be wireless, meaning that sensor location data still need to be stored somewhere. Fortunately, this problem can be reduced to just finding the local service that contains the location data, a more tractable problem.

4.3. Apportioning Responsibilities

A third challenge to building a context-aware infrastructure is deciding how to split responsibilities between devices, applications, and the infrastructure. For example, Hinckley, Pierce, Sinclair, and Horvitz (2000) modified a PDA to have tilt, touch, and proximity sensors. The screen would rotate simply by rotating the PDA. Also, the voice recorder would automatically activate if the PDA was tilted upwards, was being touched, and was near something (ideally the user's face). All of these sensor data were processed locally on the device it-

Figure 5. **At one end of the spectrum, context can be acquired and processed entirely in standalone devices. These devices will vary in terms of capability. At the other end, context can be handled entirely by the infrastructure, with extremely simple devices. There is also a large middle ground, where some devices are autonomous and some devices rely on the infrastructure.**



self. However, if it were processed in a context infrastructure, it's likely that the interactivity would be stilted due to network latency.

There are two extremes here (see Figure 5). On one side, we have smart, standalone devices that are completely autonomous and self-contained. On the other, we have a smart infrastructure with extremely simple clients, with all of the work done by the infrastructure. There are clear advantages and disadvantages to both of these approaches. The essence of the problem is finding the middle ground, determining what devices and applications should handle and what the infrastructure should handle. More applications will have to be built before we have a better understanding of the tradeoffs involved.

4.4. Scoping and Access of Sensor and Context Data

A fourth and very difficult challenge to building a context infrastructure lies with scoping and access of both sensor and context data. In other words, who has access to what data? Clearly, the infrastructure needs to be secure against unauthorized access, but it also needs ways to let people introspect, so that they can understand what is being done with the data and by whom. Furthermore, the infrastructure needs to be designed such that privacy concerns are legitimately and adequately addressed. There is a clear mental model with standalone devices and applications: Everything is done locally. However, once we shift to a service infrastructure, the lines become blurred. It's no longer obvious what is being captured, who has access to it, and what is being done with it.

Clearly, access control and encryption will be an important part of any context infrastructure. One possible approach is to follow the model the Web has

taken with digital certificates. Digital certificates are issued by trusted third-party organizations and provide an identity and authentication of that identity. As long as these third-party organizations are trusted, their digital certificates should be too.

4.5. Scaling Up the Infrastructure

A fifth challenge is that of scale. The sheer number of sensors, services, and devices envisioned poses some fundamental engineering challenges. The infrastructure needs to work for large numbers of sensors, services, devices, and people. It also needs to require a minimal amount of administrative effort. Much work has been done in building distributed systems (e.g., CORBA™, Object Management Group, 2001; and DCOM™, Eddon & Eddon, 1998), application servers (e.g., BEA™ WebLogic, BEA Systems, 2001; and IBM® WebSphere, IBM Corporation, 2001), and service infrastructures (including Jini, Waldo & the Jini Technology Team, 2000; Salutation™, Salutation Consortium, 2000; HP® e-Speak, Hewlett-Packard Inc., 2001; and Ninja, Gribble, et al., 2001). However, considerable progress is still needed in these areas before a context infrastructure can be deployed across a wide area.

5. SUMMARY

Although frameworks and toolkits are very useful for building context-aware applications, we believe that there are several compelling advantages to a service infrastructure approach. We have identified three broad benefits. First, because an infrastructure can be neutral with respect to hardware platform, operating system, and programming language, a greater variety of devices and applications can access the infrastructure. Second, the middleware layer decouples the individual pieces of the infrastructure from one another. This allows sensors and services to be upgraded independently of one another and dynamically while the system is still running. Third, devices can be simpler because they can rely on using sensors, processing power, services, and data contained in the infrastructure.

We have also outlined five challenges that must be overcome before a context-aware infrastructure can be built. The first challenge is in designing the data formats and network protocols to be simple enough that they can be implemented on virtually any platform but also rich enough to represent the majority of sensor and context data. The second challenge is in building the basic services in the infrastructure, including automatic path creation and proximity-based discovery. The third challenge is in finding the middle ground between smart devices and smart infrastructures, finding the right balance of responsibilities between the two. The fourth challenge is in scoping of sensor

and context data to ensure security and privacy. The fifth challenge is in building an infrastructure that will scale up gracefully for large numbers of sensors, services, devices, and people. Our group at University of California at Berkeley is currently designing an infrastructure to meet these challenges

NOTES

Background. This essay describes the preliminary work of Jason I. Hong's doctoral dissertation.

Acknowledgments. We thank all of the members of the Group for User Interface Research (GUIR) for their feedback on this essay.

Support. This work is supported in part by DARPA Grant N66001–99–2–8913: The Endeavour Expedition.

Authors' Present Addresses. Jason I. Hong, 525 Soda Hall, Computer Science Division, University of California at Berkeley, Berkeley, CA 94720–1776. E-mail: jasonh@cs.berkeley.edu. James A. Landay, Soda Hall, Computer Science Division, University of California at Berkeley, Berkeley, CA 94720–1776. E-mail: landay@cs.berkeley.edu.

HCI Editorial Record. First manuscript received January 8, 2001. Accepted by Thomas Moran and Paul Dourish. Final manuscript received March 2, 2001. — *Editor*

REFERENCES

Abowd, G. D., Atkeson, C. G., Hong, J., Long, S., Kooper, R., & Pinkerton, M. (1997). Cyberguide: A mobile context-aware tour guide. *ACM Wireless Networks, 5,* 421–433.

BEA Systems. (2001). *BEA WebLogic application servers* [On-line]. Available: http://www.bea.com/products/weblogic

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S., & Winer, D. (2000). *Simple Object Access Protocol (SOAP) 1.1* (W3C Note 08). Cambridge, MA: World Wide Web Consortium. Available: http://www.w3.org/TR/SOAP/

Castro, P., & Muntz, R. (2000). Managing context for smart spaces. *IEEE Personal Communications, 7*(5), 44–46.

Dey, A. K., Abowd, G. D., & Salber, D. (2001). A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction, 16,* 97–166. [this special issue]

Eddon, G., & Eddon, H. (1998). *Inside distributed COM.* Redmond, WA: Microsoft Press.

Gribble, S. D., Welsh, M., von Behren, R., Brewer, E. A., Culler, D., Borisov, N., Czerwinski, S., Gummadi, R., Hill, J., Joseph, A., Katz, R. H., Mao, Z. M., Ross, S., & Zhao, B. (2001). The Ninja architecture for robust Internet-scale systems and services. *Computer Networks, 35*(4), 473–497.

Hewlett-Packard Inc. (2001). *e-Speak: The universal language of E-services* [On-line]. Available: http://www.e-speak.net/

Hinckley, K., Pierce, J., Sinclair, M., & Horvitz, E. (2000). Sensing techniques for mobile interaction. *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology (UIST 2000)*. New York: ACM.

IBM Corporation. (2001). *IBM WebSphere application server* [On-line]. Available: http://www.ibm.com/software/webservers

Kahn, J. M., Katz, R. H., & Pister, K. S. J. (1999). Next century challenges: Mobile networking for "Smart Dust." *Proceedings of ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 99)* [On-line]. Available: http://robotics.eecs.berkeley.edu/~pister/publications/1999/mobicom_99.pdf

Kiciman, E., & Fox, A. (2000). Using dynamic mediation to integrate COTS entities in a ubiquitous computing environment. *Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing (HUC2K)*. Heidelberg, Germany: Springer-Verlag.

Mao, Z. M. (2000). *Fault-tolerant, scalable, wide-area Internet service composition*. Unpublished master's thesis, University of California at Berkeley, Available: http://www.cs.berkeley.edu/~zmao/Papers/techreport.ps.gz

Object Management Group. (2001). *The common object request broker architecture* [On-line]. Available: http://www.omg.org/technology/documents/index.htm

Pascoe, J. (1997). The Stick-e note architecture: Extending the interface beyond the user. *Proceedings of the IUI 97 International Conference on Intelligent User Interfaces*. New York: ACM.

Salutation Consortium. (2001). *Salutation specification*. Available: http://www.salutation.org/ordrspec.htm

Waldo, J., & the Jini Technology Team. (2000). *The Jini™ specifications* (2nd ed.; K. Arnold, ed.). Boston: Addison-Wesley.

Schilit, B. (1995). *System architecture for context-aware mobile computing*. Unpublished doctoral dissertation, Columbia University, New York. Available: http://www.fxpal.xerox.com/people/schilit/schilit-thesis.pdf

Universal Plug and Play Forum. (2000). *Universal Plug and Play device architecture*. Available: http://www.upnp.com/download/UPnPDA10_20000613.htm

Want, R., Hopper, A., Falcao, V., & Gibbons, J. (1992). The Active Badge location system. *ACM Transactions on Information Systems, 10*(1), 91–102.

Want, R., Schilit, B. N., Adams, N. I., Gold, R., Petersen, K., Goldberg, D., Ellis, J. R., & Weiser, M. (1995). Overview of the PARCTAB ubiquitous computing experiment. *Mobile Computing, 2*(6), 28–43.