Toward a General Software Infrastructure for Ubiquitous Computing

This general software architecture is designed to support ubiquitous computing's fundamental challenges, helping the community develop and assess middleware and frameworks for this area.

Cristiano André da Costa
Federal University of Rio Grande do Sul

Adenauer Corrêa Yamin
Catholic University of Pelotas

Cláudio Fernando Resin Geyer
Federal University of Rio Grande do Sul

"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it."[1] Mark Weiser's visionary statement summarizes what's expected from pervasive or ubiquitous computing (ubicomp): user access to the computational environment, everywhere and at all times, by means of any device. The difficulty lies in how to develop applications that will continually adapt to the environment and remain working as people move or change devices.[2]

The more traditional mobility goal of providing computation "all the time, everywhere"[3] is considered a reactive approach to information access. However, it represents a proactive step toward ubicomp. For this purpose, we need a new class of software, but the limited number of languages and tools available still hinders this field's development.[4]

Ubiquitous applications need middleware to interface between many different devices and end-user applications.[3] The aim is to hide environment complexity by isolating applications from the explicit management of protocols, distributed memory access, data replication, communication faults, and so on. Middleware can also solve heterogeneity problems related to architectures, operating systems, network technologies, and even programming languages, promoting their interoperation. On the other hand, a framework is an environment, comprising APIs, user interfaces, and tools, that simplifies software development and management in a specific domain. We can use frameworks to develop middleware and to build software that runs on that middleware.

Our proposed general architectural model for ubicomp supports frameworks and middleware while considering all the challenges we believe significant in the field. Here, we highlight the numerous requirements that are essential to the area and that software infrastructure should cover.

Ubiquitous computing challenges

Previous studies present issues that are unique or still open in ubicomp (see the "Related Work in Ubiquitous Computing" sidebar). Table 1 summarizes the main issues.

Heterogeneity is a concern derived from distributed systems. Ubicomp software must hide infrastructure differences from users and manage the required conversions from one environment to another, addressing protocol mismatches. In this scenario, developers using a device-independent approach have to create application logic only once.

Published by the IEEE CS ■ 1536-1268/08/$25.00 © 2008 IEEE

Related Work in Ubiquitous Computing

Scalability, heterogeneity, integration, invisibility, context awareness, and context management are all challenges to be addressed, according to Debashis Saha and Amitava Mukherjee.[1] Except for integration, which we discuss indirectly as part of spontaneous interoperation and integration in invisibility, we include all these aspects in the article and in table 1.

Tim Kindberg and Armando Fox base their work on two fundamental characteristics: physical integration and spontaneous interoperation.[2] They also emphasize some common areas in ubicomp scenarios, all directly or indirectly discussed in our proposed model.

The article by Guruduth Banavar and his colleagues at IBM envisions a device-independent application-development process with a highly dynamic load-time system that embraces discovery, negotiation, and dynamic selection of presentation.[3] This model at execution involves dynamic resource sharing, application migration, and failure detection and recovery. Of these, data sharing is the only feature that we don't list; instead, we consider it to be a part of (logical) mobility.

Eila Niemelä and Juhani Latvakoski propose interoperability, heterogeneity, mobility, survivability and security, adaptability, self-organization, and augmented reality with scalable content.[4] Their concept of self-organization amplifies the idea of adaptation by adding a virtual context to the one sensed by users.

Robert Grimm and his contemporaries at the University of Washington suggest three "fault lines" for ubicomp: transparency, heterogeneity, and the use of a single abstraction for data and code.[5] To address this last issue, they recommend keeping data and functionality separate. We don't tackle this in this article, but satisfying this condition would be possible using a different data representation, such as tuples.[6]

An article by Intel researchers Roy Want and Trevor Pering proposes power management, discovery, user interface adaptation, and location-aware computing.[7] We don't directly consider power management in this article; as an alternative, we present the more general issue of context management. The same applies to location awareness.

Martin Modahl and his colleagues propose a taxonomy for the building blocks of a software infrastructure called UbiqStack.[8] It has five subsystems: registration and discovery, service and subscription, computation sharing, context management, and data storage and streaming. The first four categories correspond roughly to our more generic discovery, interoperation, cyber foraging, and adaptation. We don't address the fifth one directly, but we believe our proposal is more comprehensive because we allow for several other categories. Of the research projects mentioned here, Modahl and his colleagues' work is the only one that proposes a software architecture for ubicomp, although Banavar and colleagues offer a new application model considering its life cycle.[3]

REFERENCES

1. D. Saha and A. Mukherjee, "Pervasive Computing: A Paradigm for the 21st Century," *Computer*, vol. 36, no. 3, 2003, pp. 25–31.
2. T. Kindberg and A. Fox, "A System Software for Ubiquitous Computing," *IEEE Pervasive Computing*, vol. 1, no. 1, 2002, pp. 70–81.
3. G. Banavar et al., "Challenges: An Application Model for Pervasive Computing," *Proc. 6th Int'l Conf. Mobile Computing and Networking* (MOBICOM 00), 2000, ACM Press, pp. 266–274.
4. E. Niemelä and J. Latvakoski, "Survey of Requirements and Solutions for Ubiquitous Software," *Proc. Mobile Ubiquitous Computing Conf.*, ACM Press, 2004, pp. 71–78.
5. R. Grimm et al., "Systems Directions for Pervasive Computing," *Proc. 8th Workshop Hot Topics in Operating Systems* (HOTOS VIII), 2001, IEEE CS Press, pp. 147–151.
6. R. Grimm et al., "System Support for Pervasive Applications," *ACM Trans. Computer Systems*, vol. 22, no. 4, 2004, pp. 421–486.
7. R. Want and T. Pering, "System Challenges for Ubiquitous and Pervasive Computing," *Proc. 27th Int'l Conf. Software Eng.* (ICSE 05), 2005, ACM Press, pp. 9–14.
8. M. Modahl et al., "UbiqStack: a Taxonomy for a Ubiquitous Computing Software Stack," *Personal and Ubiquitous Computing*, vol. 10, no. 1, 2006, pp. 21–27.

Another related issue inherited from distributed systems is *scalability*. Ubicomp systems will likely involve countless users, devices, applications, and communications on an unprecedented scale. We must avoid centralized solutions, reduce distant interactions, and prevent bottlenecks.

Sometimes the system can't execute according to functional specifications. Additionally, problems related to misspecifications might arise. Such situations lead to failures. Avoiding failures that are more frequent and more severe than what is acceptable leads to *dependability*, a concept that integrates the attributes of availability, reliability, safety, integrity, and maintainability. The term *pervasive dependability* refers to these needs in the scope of ubicomp.[5]

Security is a concept strictly related to dependability. A system is secure if measures exist to ensure availability, integrity, and confidentiality. We could also use many distributed-systems security mechanisms in ubicomp, but they must be lightweight to preserve both

TABLE 1
Ubiquitous computing issues and challenges.

Issue	Alias	Focus area	Motive
Heterogeneity		Distributed systems	• Allowing a variety of services • Providing different types of devices, networks, systems, and environments
Scalability	Localized scalability*	Distributed systems	• Enabling large-scale deployments • Increasing the number of resources and users
Dependability and security	Fault tolerance†	Mission-critical and distributed systems	• Avoiding failures that are more frequent and more severe than acceptable • Providing availability, confidentiality, reliability, safety, integrity, and maintainability
Privacy and trust		Internet and mobile computing	• Protecting against bad use of personal data • Defining the trustworthiness of interacting components
Spontaneous interoperation	Volatility	Mobile computing	• Allowing interaction with a set of components that can change both identity and functionality • Permitting association and interaction
Mobility	Follow-me applications	Mobile computing	• Providing application and data access anywhere, anytime • Enabling the user environment to go along with the user
Context awareness	Perception	Mobile computing	• Perceiving the user's state and surroundings • Inferring context information
Context management‡	Smartness, masking uneven condition, adaptability	Mobile computing	• Modifying system behavior based on perceived context information • Adapting to the current situation
Transparent user interaction	HCI**	Ubiquitous computing	• Merging the user interface with the real world • Letting users focus on tasks with minimal distraction
Invisibility	Ubiquity, pervasiveness	Ubiquitous computing	• Letting users focus on tasks, not tools • Making computers disappear in the background

* Physical distance is a significant issue in pervasive computing: we must consider the important role that local interactions play.
† This term is more restrictive than "dependability," which community use is converging on.
‡ Some authors consider context management a part of context awareness.
** This term is used in a more general sense.

the spontaneity of interactions and the limitations of some devices.[6]

Moreover, *privacy*—guaranteeing how such information will be used or passed on—will be extremely difficult. Another associated challenge is *trust*, which should be considered in this kind of heterogeneous, dynamic scenario. Since there's neither a fixed infrastructure nor a specific domain, we must use a trust management system to measure how much information should be disclosed.[7]

Spontaneous interoperation is the bringing together of constantly changing components from several devices, enabling reciprocal communication.[8] We need this spontaneity because of the volatile nature of ubicomp, whose components are in continual motion and interacting with different sets of services.

Another challenge, *mobility*, provides access to applications and data wherever users go and however they move. Mobility can be either physical (related to equipment or users) or logical (related to code or data). Applications should be able to move from one device to another, and data access should be maintained ("follow-me" applications).[9]

Mobile computing has also introduced the idea of *context awareness*—that is, inferring context to supply information or services in the case of limited or intermittent availability.[10] Context awareness is broader in ubicomp than in mobile computing, as devices must sense changes and software should act proactively.

Context management is action in response to sensed data, adapting services to environmental changes. It can also expand devices' capacity by using available resources in the current context.

HCI design is also a significant factor. As computers become "smarter," HCI's intensity and quality are bound to increase.[3] The focus on user interfaces evolved from software design, but it acquired a different meaning after mobile computing and new modes of interaction emerged. Another issue is the merging of user data with the real environment, redirecting attention to *transparent user interaction*.

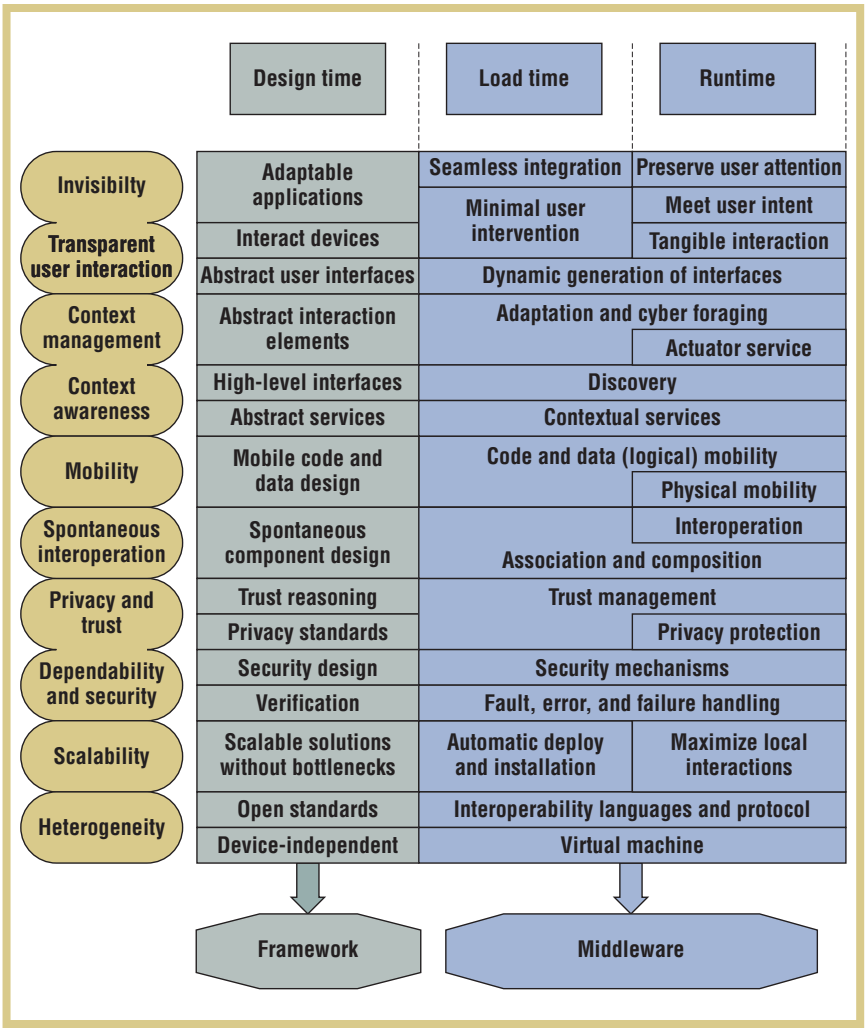


Figure 1. General architectural model for ubiquitous computing.

The last issue, *invisibility*, is directly related to ubicomp itself. It's about keeping user focus on the task rather than the tool.[1] To fulfill this vision, software must satisfy user intent by helping (not obstructing) it. Software should learn with the user and, in some cases, let the user change preferences, interacting, as Mahadev Satyanarayanan suggested, "almost at a subconscious level."[11]

Proposed model

Figure 1 presents the general infrastructure model we propose, including each issue we highlighted in table 1 and the corresponding characteristics that should be available to address it. The structure is then divided considering application life cycle (design time, load time, and runtime).[12] Design time is when the application is conceived, extended, or maintained. At load time, applications are loaded onto specific devices. At runtime, the user executes and uses applications.

Each row in the figure presents a challenge (in an oval box on the far left) and the essential characteristics to be addressed at design time, load time, and runtime, respectively. Some challenges, such as "privacy and trust" and "dependability and security," are more closely related than others (there's no horizontal line separating these ovals). In this situation, we can define dependability as the ability to deliver services that we can justifiably trust. Moreover, to attain privacy protection, collected personal data should be secure. Close dependence also involves context management and context awareness, and invisibility and transparent user interaction, making it difficult to draw an exact borderline.

The issues' order in the figure doesn't imply a layered model, in which each tier depends on the services provided by the other. From the bottom of the figure up, services are organized from lower level to higher level. The challenges that distributed systems already tackle are at the bottom, the issues related to mobile computing are in the middle, and the challenges that arise with ubicomp are at the top.

A framework can provide the abstractions ubicomp needs at design time. The design-time column shows all the characteristics of this stage. The same applies to load time and runtime. However, to provide the characteristics required in these stages, we suggest using middleware. Let's take a closer look at each row of the general architectural model.

Heterogeneity

Several levels of heterogeneity exist, in both hardware (including networks, devices, screen sizes, and power capability) and software (including languages, component models, and structures). To facilitate the bridging between heterogeneous systems, we should use open standards, with published interfaces and standardized communication mechanisms, enabling easier system extension and reimplementation.

Also, frameworks for device-independent projects can make it possible for different hardware, even from diverse vendors, to use the same source code, sometimes with little alteration. Thus, we can keep the developed application almost unmodified, limiting change to device drivers or to the framework itself.

The current solution to heterogeneity is to use middleware with a common,

integrated API and a unified binary format. This binary file should run on a virtual machine, such as Java, that would be available on all platforms. However, different device capabilities mean that we can't always employ the same virtual machine, run the same binary code, or expect the available features to remain unchanged. For instance, Java has different virtual machines for mobile devices and PCs. Nevertheless, using a virtual machine reduces the cost of heterogeneity because fewer changes are needed compared to languages that generate specific machine codes.

Finally, we must focus on components' interoperability, the "ability to understand the exchanged information and to provide something new originating from the exchanged information."[13] Interoperability languages such as XML are commonly used, making it possible to represent data in a standard, structured form, more portable between applications. In other cases, software converts source data into a format that's both expected and transparent to the user. However, differences might occur between the source and destination versions. Besides, protocols that can negotiate services and resources between applications and devices must be available, allowing integration during load and execution.

Scalability

To address the problem of scalability, we must develop software that considers the abundance of users, interactions, components, and devices, avoiding centralized solutions and bottlenecks. Applications should be automatically loaded and managed at load time. Besides, whenever a new application is available, it should be automatically deployed and installed, because manual software distribution and installation for each device would be impractical.

During execution, we should reduce interaction with distant resources. This idea, *localized scalability*,[11] should be a ubicomp goal even if it conflicts with the current guideline of network transparency (in which local and remote resources are accessed with identical operations, their physical location notwithstanding). We should consider resources' location and give priority to local interactions over distant ones.

Dependability and security

In the scope of ubicomp, reliability, availability, and safety must be maximized. Minimizing the cost of maintainability and the effort to preserve integrity is also vital. In terms of security, we must deal directly with the attribute of confidentiality but also with availability and integrity.

During application development, verification could diagnose and remove faults.

The failure-detection and recovery strategies we use today (such as checkpointing, compensation, isolation, or reconfiguration) could be applied to ubicomp as well. Because applications execute in environments and there's always a context involved, requirements differ from those of traditional computing. Also, devices are a means of access to applications, but some device failures might not be specified in the application or middleware. Besides device and application failure, we should also consider network and service failure.

We ought to differentiate failures (situations requiring detection and recovery mechanisms) from system changes (situations where adaptation takes place). To have an adaptable system, we must specify which types of changes will cause adjustments, even though we can't predict all possible situations. Sometimes, unpredicted change occurs, or the system might generate unspecified results. We should detect and recover these examples of failures (no adaptation is possible). Also, we shouldn't consider disconnections as failures but rather as part of the system specifications, treating them with adaptation mechanisms.

We must design a ubiquitous system's security with certain characteristics in mind:[14]

- *User centricity*. Users should be able to circumvent security mechanisms that are discordant with common practices.
- *Context mechanisms*. The security mechanism should be near the activity in which it makes sense.
- *Selection*. Users should be able to understand and manage the employed solutions. Only in this way can they choose a suitable mechanism according to the security needed in each action and context.

Security mechanisms should scale to devices with limited resources, expect lack of knowledge, and allow dynamicity of mobility.[7] For instance, user authentication through login and password wouldn't be feasible for every device. We need other methods; for

To have an adaptable system, we must specify which types of changes will cause adjustments, even though we can't predict all possible situations.

example, the system could exploit biometric information or authenticate on the basis of people's locations.

Privacy and trust

Privacy and trust relate directly to security concerns. We treat them separately from dependability and security because of their magnitude in ubicomp. Although we try to deal with privacy through legislation, we should also apply technology because of the risk of a user exposing too much personal information to an environment. The user might even be unaware of the surveillance. Moreover, the amount and accuracy of sensor-collected data will likely increase as ubicomp advances. Furthermore, privacy protection is particularly difficult in ubiquitous systems because of location sensitivity. The context-aware mechanism of sensing the exact user location could be exploited for tracking purposes. With this mechanism, we can infer users' movements and activities, associating them with their personal information.

During design, we should apply privacy standards. Each standard, enforced by jurisdiction and market, comprises a group of procedures that we should observe in data collection.[7] During the execution phase, we should employ protection mechanisms to realize these standards. For instance, data could be accumulated anonymously or deleted after a period of time.

Trust management can establish the reliance on exchanged information and ensure only authorized users can access that information. The difficulty lies in precisely defining an interacting entity's trustworthiness and granting permissions on the basis of that decision. In some cases, little or no evidence is available about an entity and, as in our daily trust decisions, it's more of a subjective notion. Apart from being subjective, trust is nonsymmetric (two interacting components have different degrees of trust in each other), situation-specific (dependent on context), dynamic (increasing or decreasing over time), and inherently associated with risk (no reason to trust if risk isn't involved).[15] Because of these, there should be trust-reasoning support. This reasoning analysis is made on the basis of available information and considering the various aspects of trust. Solutions for uncertainty should also be present.

The difficulty in trust management lies in precisely defining an interacting entity's trustworthiness and granting permissions on the basis of that decision.

Spontaneous interoperation

The first step is to design spontaneous components—that is, entities that support frequent change among communicating partners and that can easily interact with others. To accomplish this design, we need a dynamic environment with assorted infrastructures and partners. The framework can facilitate the development of spontaneous components and provide a generic interface, which will be combined to create specific entities during execution. Ideally, we should specify components using a uniform description language and then build them independently of context.[13]

During execution, components associate with each other. Association is the logical relationship established between components that allow interactions; we call these interactions interoperation.[6] When we assess association, three points are important:

- scale—efficiently choosing components to associate in a scenario with various possible partners;
- scope—defining the extent to which components must be considered and including all possible partners; and
- boundary principle—considering the physical limits (or other criteria) when defining the scope of association.[6,8]

We can also use discovery services (in this article, a context-awareness characteristic) as part of the association solution.

Interoperation depends on the communication models employed. In ubicomp, we tend to use models based on event systems or tuple spaces because of the asynchronous nature of the former or the ease of development and inherent persistence of the latter. Occasionally, both models are used in the same middleware. Conversely, we can apply other forms of communication such as message passing, remote invocation, or agent systems.

Composition is a special case of association in which external components control internal ones; all interoperation passes through those external components, redirecting or modifying the association. Composition facilitates adaptation and mobility. Each device can have a specific component nesting all others and making all the required changes to their specific interfaces and capabilities. When a component migrates from one device to another, it enters in the specific device components and continues to issue the same set of operations. The adaptation process is up to each device's outer component, as is the redirection

of messages or events arriving after an inner component has migrated.

Mobility

In ubicomp, users change devices frequently, but user applications and data must always be available. This means that the environment should migrate from one device to another. Besides, migration also helps in reducing communication costs and preventing disconnection.

In ubicomp, users change devices frequently, but user applications and data must always be available.

To support code migration during load and runtime, components must be designed with mobile technology. We can obtain this by using languages and systems compatible with code mobility. During execution, middleware has to deal with the mobile component and manage migration. To achieve this, the middleware should be aware of the network and not treat it in a transparent manner.

We must also address data mobility. We can't always employ remote data access, owing to the possibility of disconnection or deficiency of resources. In these situations, we could move or copy data to different locations, provided we pay attention to data coherence and synchronization. Also, specific applications or hardware might require conversion between different formats.

Besides logical mobility, we need to consider physical mobility. As people move, the devices in use will change their network addresses. This is because they will be communicating with different access points and being assigned to different IP addresses. The DHCP (Dynamic Host Configuration Protocol) provides this dynamic acquisition of addresses, allowing devices to maintain service access, regardless of location. However, it might be difficult for other components to interoperate with those devices, because the IP routing mechanism is based on fixed locations and might lose packets when addresses change. In addition, their updating on the DNS is slow, due to extensive use of cache.

To support physical mobility, we can employ a location management strategy. Conceptually, this strategy consists of two operations: search, which a node invokes when it needs to communicate with a mobile device; and update or registration, which the mobile node performs to inform its current location.[16] Another crucial concern is ensuring that a mobile node remains connected while moving from one scope to another. This *handoff* involves deciding when to change to a new scope, selecting it, acquiring resources, and rerouting packets to the new location.[16]

Context awareness

To be ubiquitous, middleware must use relevant information and services available in the surroundings. Discovery is the component that detects services and devices in the current context, while sensors infer the significant information that the context manager can use to reason about actions to take. Adding context awareness to middleware increases device usability and allows better user interaction.

We need framework support to assist the implementation of context-aware applications. In particular, we need a set of abstract services that programmers can employ when building their components, and we need high-level interfaces that hide specific devices or sensor details from the user.[10] Thus, we can split the acquisition of context from its use, which is one of the most important issues toward a more disseminated use of context.[17]

To manage this contextual information, middleware must provide at least four categories of contextual services:

- context subscription and delivery—a service that can notify a component when an event occurs;
- context query—a mechanism to find a suitable information or service;
- context transformation—the conversion of low-level data into high-level information; and
- context synthesis—the aggregation of context information to generate a more precise or detailed context.[10,16]

These services can supply contextual information to applications. Context management can be further improved by offering various imperceptible layers of interpretation, such as transformation and synthesis; by using distributed sensors transparently; by making context acquisition constantly available; and by storing context and history.[17]

We also need dynamic resource discovery (a mechanism to dynamically locate and enumerate resources) available in the environment or matching certain requirements.[18] A resource could be a service, application, device, or any other component. Requirements are sets of specifications or characteristics to which the needed resource must comply.

Many resource-discovery systems exist today with different purposes and design. However, when applied to ubicomp, these approaches have some limitations—for example, in terms of

their interoperability, integration with users, and scalability.[18] We desire a system that doesn't need manual or static configuration and that can find required resources in every environment at any time.

Context management

By detecting context, we can affect system behavior. This change can be made by adapting the system to the new conditions or augmenting the available resources to compensate for the lack of some feature. Another possibility is changing the context using actuators—that is, software-controlled devices that affect the real world. An actuator can activate a device, alter a physical condition such as temperature or luminosity, or execute a logical action (such as loading code, altering parameterization, or moving components). To support this management, we need abstract interaction elements in design time. We can also use these elements during execution, according to context.

Adaptability is a central concept in ubicomp. Adaptation consists in adjusting aspects of applications to changes in operating environments. The most common use of adaptation is in resource-aware applications, when there is a significant difference between resources presented in the environment and those needed.[9] These resources could be, among others, network bandwidth, energy, storage space, or computing power. Some approaches to resource adaptation include fidelity reduction, QoS systems, or the suggestion of corrective actions.[11] The first method consists in changing the application to a minimal use of limited resources. The second keeps a certain resource at a satisfactory level. The last one relies on user intervention to make the desired resources available.

Adaptation is important to other kinds of applications besides resource-aware ones: location-aware applications need to consider physical location; context-aware applications use sensors or monitors to infer state and choose a strategy; and situation-aware applications use the most general form of adaptation, perceiving other nearby applications and their usage context.[9] In the latter case, adaptation takes place depending on usage context and user preferences, since adaptation decisions are external to applications.

A special case of adaptation is cyber foraging. Mobile devices usually have limited capabilities, such as processor power, memory, and battery life. With those constraints, it's sometimes difficult to satisfy the user's computational needs. To minimize this problem, we can use nearby machines as computing and data-staging servers, thus augmenting capability.[11] Cyber foraging means sharing or dividing code or data among servers and mobile devices, which middleware can do automatically during load- and execution-time. Alternatively, it could be user-initiated—for instance, when anticipating changes in connectivity or device.

Servers used to augment capabilities of mobile devices are sometimes called *surrogates*.[19] These surrogates may employ encryption algorithms in stored data. Thus, the users of these servers can't access information saved there.

Transparent user interaction

We should design device-neutral applications—that is, we shouldn't start with the presentation and then build up the programming logic from that.[12] To accomplish this, during design, we can define abstract user interfaces and predict different types of interaction so that deciding which interface to use can be postponed until execution. Another option is to dynamically generate the interfaces during execution on the basis of abstract definitions, specific device features, and contextual information. This option requires less effort during design and tends to consume more processor power and communication latency during execution. However, it facilitates the use of contextual data.

Generating interfaces suited to each specific device eases the design of transparent user interaction. These interfaces must consider the most natural form of interaction for those specific devices, and also contextual information and user behavior (such as preferences and history needs).

A broader concept wouldn't focus only on the human-computer interface of devices but rather on designing the physical interaction itself. This idea leads to tangible interaction and its use in the scope of ubicomp. The idea of tangible interaction is to create a richer interaction experience by coupling digital information with physical artifacts, using the human body as an interface and combining real objects and devices with computers in interactive spaces.[20] The challenge consists in creating interfaces seamlessly integrated with the real world and considering social, personal, and emotional human experience. Finally, to achieve a proper transparency, people should

We desire a system that doesn't need manual or static configuration and that can find required resources in every environment at any time.

be able to focus on their task intuitively, with minimal involvement in system issues.

Invisibility

The first step toward an invisible system is to design adaptable applications. We need framework support that eases this development, following the goals of disappearing computing and of keeping the user focused on the task. At runtime, we require uninterrupted use, with minimal user intervention. For instance, disconnection periods could occur in mobile devices. Actually, the system must mask this disconnection by keeping services uninterrupted and still satisfy the user's needs, maybe with some degradation.

An important characteristic toward invisibility is seamless integration. This requires much effort from middleware and the careful development of each system element, considering many aspects presented on the other layers of the proposed architecture. Guruduth Banavar and his colleagues propose a task-based model that links the abstract interaction to the application logic.[12] This model facilitates integration, since tasks are highly abstract and can be used at load- and runtime to build systems with other applications, services, and capabilities that are available in the pervasive environment. This can bring the notion of a task-aware system.[19]

To be invisible during runtime, a system must act unobtrusively, meeting the user's expectations without human intervention. Debashis Saha and Amitava Mukherjee affirm that "humans can intervene to tune smart environments when they fail to meet user expectations automatically."[3] We can anticipate user needs by capturing user intent. We should also preserve user attention. The user is the most important resource in a system,[19] and keeping him or her focused on the task can foster invisibility.

It's still difficult to find a software infrastructure that has all the necessary characteristics presented here. In the past, projects such as Aura,[19] CoolTown, [6] Gaia,[4] One. World,[2] and ISAM[21] tried to accomplish many aspects of ubicomp. However, it's hard to address several open research topics in one project. The tendency today is to provide middleware or frameworks for specific issues. In spite of this tendency, we think that a general infrastructure model for software can help to develop pervasive middleware or frameworks. We trust that this model could also be useful as a standard for assessing proposals and suggesting needed features. To fulfill Weiser's vision, future ubiquitous infrastructures should, as this model proposes, seamlessly integrate many different challenges. P

> The user is the most important resource in a system, and keeping him or her focused on the task can foster invisibility.

REFERENCES

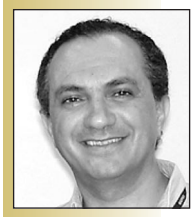
1. M. Weiser, "The Computer for the Twenty-First Century," *Scientific Am.*, vol. 265, no. 3, 1991, pp. 94–101.
2. R. Grimm et al., "System Support for Pervasive Applications," *ACM Trans. Computer Systems*, vol. 22, no. 4, 2004, pp. 421–486.
3. D. Saha and A. Mukherjee, "Pervasive Computing: A Paradigm for the 21st Century," *Computer*, vol. 36, no. 3, 2003, pp. 25–31.
4. M. Román et al., "A Middleware Infrastructure for Active Spaces," *IEEE Pervasive Computing*, vol. 1, no. 4, 2002, pp. 74–73.
5. C. Fetzer and K. Högstedt, "Challenges in Making Pervasive Systems Dependable," *Future Directions in Distributed Computing*, A. Schiper et al., eds., Springer, 2002, pp.186–190.
6. G. Coulouris et al., "Mobile and Ubiquitous Computing," *Distributed Systems: Concepts and Design*, 4th ed., Addison-Wesley, 2005, pp. 657–719.
7. P. Robinson et al., "Some Research Challenges in Pervasive Computing," *Privacy, Security and Trust within the Context of Pervasive Computing*, P. Robinson et al., eds., Springer, 2005, pp. 1–16.
8. T. Kindberg and A. Fox, "A System Software for Ubiquitous Computing," *IEEE Pervasive Computing*, vol. 1, no. 1, 2002, pp. 70–81.
9. I. Augustin et al., "Towards Taxonomy for Mobile Applications with Adaptive Behavior," *Proc. 20th Int'l Symp. Parallel and Distributed Computing and Networking* (PDCN 02), ACTA Press, 2002, pp. 224–228.
10. A. Dey, "Understanding and Using Context," *Personal and Ubiquitous Computing*, vol. 5, no. 1, 2001, pp. 4–7.
11. M. Satyanarayanan, "Pervasive Computing: Vision and Challenges," *IEEE Personal Comm.*, vol. 8, no. 4, 2001, pp. 10-17.
12. G. Banavar et al., "Challenges: An Application Model for Pervasive Computing," *Proc. 6th Int'l Conf. Mobile Computing and Networking* (MOBICOM 00), 2000, ACM Press, pp. 266–274.
13. E. Niemelä and J. Latvakoski, "Survey of Requirements and Solutions for Ubiquitous Software," *Proc. Mobile Ubiquitous Computing Conf.*, ACM Press, 2004, pp. 71–78.
14. P. Dourish et al., "Security in the Wild:

User Strategies for Managing Security as an Everyday, Practical Problem," *Personal and Ubiquitous Computing*, vol. 8, no. 6, 2004, pp. 391–401.

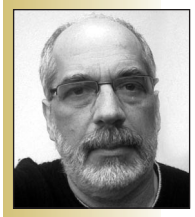
15. V. Cahill et al., "Using Trust for Secure Collaboration in Uncertain Environments," *IEEE Pervasive Computing*, vol. 2, no. 3, 2003, pp. 52–61.
16. F. Adelstein et al., *Fundamentals of Mobile and Pervasive Computing*, McGraw-Hill, 2005.
17. A. Dey et al., "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Application," *HCI J.*, vol. 16, nos. 2–4, 2001, pp. 97–166.
18. F. Zhu, M. Mutka, and L. Ni, "Service Discovery in Pervasive Computing Environments," *IEEE Pervasive Computing*, vol. 4, no. 4, 2005, pp. 81–90.
19. D. Garlan et al., "Project Aura: Toward Distraction-Free Pervasive Computing," *IEEE Pervasive Computing*, vol. 1, no. 3, 2002, pp. 22–31.
20. E. Hornecker, "A Design Theme for Tangible Interaction: Embodied Facilitation," *Proc. 9th European Conf. Computer Supported Cooperative Work* (ECSCW 05), 2005, Kluwer, pp. 23–43.
21. I. Augustin et al., "ISAM, Joining Context-Awareness and Mobility to Building Pervasive Applications," ch. 4, *Mobile Computing Handbook*, M. Ilyas and I. Mahgoub, eds., CRC, 2004, pp. 73–94.

the AUTHORS

Cristiano André da Costa is an associate professor at the University of Vale do Rio dos Sinos and a doctoral candidate at the Federal University of Rio Grande do Sul. His research interests include software infrastructure for ubiquitous computing, context awareness, distributed systems, and operating systems. He received his MSc in computer science from the Federal University of Rio Grande do Sul. He's a member of the IEEE, the ACM, and the Brazilian Computer Society. Contact him at Instituto de Informática, Universidade do Vale do Rio dos Sinos (UNISINOS), Av. Unisinos 950, 93022-000, São Leopoldo, RS, Brazil; cac@unisinos.br.



Adenauer Corrêa Yamin is an associate professor in the Computer Science Department at the Catholic University of Pelotas and works on the technical staff of the Informatic Center of Federal University of Pelotas. His research interests include ubiquitous, grid, parallel, and distributed computing. He obtained his PhD in computer science from the Federal University of Rio Grande do Sul. He's a member of the ACM and the Brazilian Computer Society. Contact him at Universidade Católica de Pelotas, Rua Félix da Cunha 412, Pelotas, 96010-000, RS, Brazil; adenauer@ucpel.tche.br.



Cláudio Fernando Resin Geyer is an associate professor at the Informatics Institute of the Federal University of Rio Grande do Sul. His research interests include ubiquitous computing, parallel and distributed computing, grid computing, and distributed objects. He received his PhD in informatics from the Joseph Fourier University. He's a member of the ACM and the Brazilian Computer Society. Contact him at Universidade Federal do Rio Grande do Sul, Av. Bento Gonçalves 9500, Porto Alegre, 91501-970, RS, Brazil; geyer@inf.ufrgs.br.