Technology Challenges for Building Internet-Scale Ubiquitous Computing

Tatsuo Nakajima, Hiro Ishikawa, Eiji Tokunaga
Department of Information and Computer Science, Waseda University
3-4-1 Okubo Shinjuku Tokyo 169-8555, JAPAN
Frank Stajano
Toshiba Corporate R&D Center, Kawasaki, JAPAN and
Laboratory for Communications Engineering, University of Cambridge, UK

Abstract

In the future, many of the physical objects that surround us will be augmented with microprocessors and wireless transceivers. They will communicate with each other, quietly monitor our daily activities and notify us of important events. Any piece of relevant information on the Internet will be available to us anytime and anywhere. We call this scenario the *Internet-scale ubiquitous computing environment*.

In our current lifestyle, one of the most precious and scarce resources is human attention. Internet-scale ubiquitous computing environments have the potential to make our daily life more comfortable and attractive because they will allow us to focus only on the essential tasks, letting technology take care of itself.

However, building Internet-scale ubiquitous computing environments requires us to address a variety of technical issues. In particular we have to take into account the software development costs to build such an attractive future. As a road to developing the necessary software in a timely fashion, we propose to base the system on existing infrastructural software components such as Linux, Java and CORBA.

In this paper we identify several important architectural problems that builders of Internet-scale ubiquitous computing environments will have to address, and we present some recommendations on the basis of our analysis and implementation experience.

1 Introduction

The coming age of *ubiquitous computing* [1, 3, 4, 43, 21, 64] promises to change our lifestyle in significant ways. Everyday objects, from pens to shoes, will become "smart" and will contain embedded processors; they might monitor our behaviour, react to it and adapt their functionality to the personal preferences of the current user [16, 15, 20, 28, 63]. Traditional appliances such as television sets, microwave ovens and refrigerators, which already contain digital electronics, will become more intelligent and communicate with each other. And we will be carrying or wearing various mobile gadgets with a wireless connection to the Internet.

Many research groups have been involved in the design and implementation of ubiquitous computing environments. We expect that, in the future, these environments will be connected to the Internet: any appliance in the world will be able to communicate with any other. If we wish to, we will be able to ask this globally interconnected system to follow us wherever we are and intelligently monitor and augment our behaviour according to our personal choices and preferences. Devices and appliances everywhere will reconfigure themselves and interact with us in familiar ways as if they were the ones we have at home.

Sophisticated enterprise networks, such as online Internet-based billing systems, have already been built. We must therefore envisage the integration between ubiquitous computing applications and existing distributed systems on the Internet. We shall call the resulting computing environments *Internet-scale ubiquitous computing* (InterUbicomp). These environments are characterized by *ubiquitous computers*, *ubiquitous information* and *ubiquitous networks*. Physical spaces will be augmented by computer-generated information. InterUbicomp will bring along an evolution of our lifestyle, culture and society. But we will need to solve many technical problems in order to transform this vision into a reality.

In this paper we present several technical challenges for the implementation of InterUbicomp. Infrastructural software components such as operating systems and middleware will have to run on a variety of heterogeneous hardware platforms. It is unrealistic to reimplement such components from scratch for each target hardware platform, because the development of such system software is extremely expensive. Our research focuses on software portability to reduce the development cost, and on adding advanced features to existing software components without modifying them (note that we are not only talking about applications but also, and particularly, about system software). We also believe that InterUbicomp software should be very robust: the more these systems become pervasive in our lives, the more serious the disruption will be if they do not work as they should.

This paper gives an overview of what we consider to be the main system-level challenges in the implementation of InterUbicomp environments. We have started a couple of practical projects in this area; this paper describes the general framework in which they fit. Based on our implementation experience, we point out our preferred solutions to the main technical challenges in this field.

The remainder of this paper is structured as follows. In Section 2, we give a more detailed description of Internet-scale ubiquitous computing. Section 3, which is the main part of the paper, presents several technical challenges for InterUbicomp implementation. We categorize the challenges into software design issues, system issues, and survivability issues. Section 4 reports on the current status of our work. We describe the role of object-orientation in Section 5. Finally, in Section 6, we draw some conclusions.

2 Internet-Scale Ubiquitous Computing

2.1 Overview of Internet-Scale Ubiquitous Computing

The locution "ubiquitous computing" was introduced by Weiser [64] to describe a scenario in which, literally, computing is everywhere. This should not be taken in the narrow-minded sense of "a computer on every desk", but in the rather subtler one of computers becoming embedded in everyday objects and augmenting them with information processing capabilities. This embedding would be discreet and unobtrusive: the computers would disappear from our perception, leaving us free to concentrate on the task at hand—unlike today, when a majority of users perceives computers as getting in the way of their work.

Part of this vision is already becoming a reality, in so far as most of us have already lost count of the number of gadgets we own that contain a microprocessor. Other parts of the vision, though, such as usability and universal interoperability, are still far away. Further developments are also envisaged: in the decade since the publication of Weiser's visionary article, the popularity of the World Wide Web has exploded, and it is easy to forecast that any ubiquitous computing infrastructure we will build in the future will perforce include global Internet connectivity. Moreover, the ongoing miniaturization of electronics, together with advances in power saving and wireless communication systems, has brought us to the stage where most of us are happy to *carry* computing and communication devices all the time. The most popular item for the general public is by far the cellular phone, but many also enjoy PDAs and digital music players. Mobility is now another important aspect of the ubiquitous computing scenario, and will be so even more as we evolve from gadgets we carry to gadgets we wear.

The next generation of home appliances is also going to be enhanced with communication and sensing capabilities. You can already buy CD players that connect to the Internet to look up the artist name and track titles for any CD you feed them; but this style of interaction will soon extend to non-electronic goods. The refrigerator will know that you are running out of milk and it will send a message to your phone when you are on the way to the shops—or it might reorder it directly by contacting the supplier through the Internet. The packaged food you wish to reheat will transmit the correct time and power settings to the microwave oven. And the washing machine will ask for confirmation before proceeding if it spots a white lace garment in a load of blue jeans.

There will be ubicomp-enhanced appliances not only in our homes but also in shared spaces such as offices, shops, restaurants and public transport vehicles such as trains. They will be connected to the Internet and might act as gateways (as well as direct interlocutors) for our mobile and wearable devices. We will be able to share and retrieve information wherever we are.

Devices and applications will become more responsive and friendly by personalizing their behaviour according to the preferences of the person who is using them. The capability of these systems to sense some aspects of their environment will make them adapt to the current situation. Location and environmental information, user identity, even user mood are some of the possible inputs to such systems [1, 45] that will allow applications to change their behaviour to fit the circumstances. New interaction techniques will become possible [49, 53, 24, 22, 19]; a variety of sensors will monitor our behavior, and improved models of the real world will provide better awareness of the context for the computing systems [18, 11, 7, 59, 5, 17].

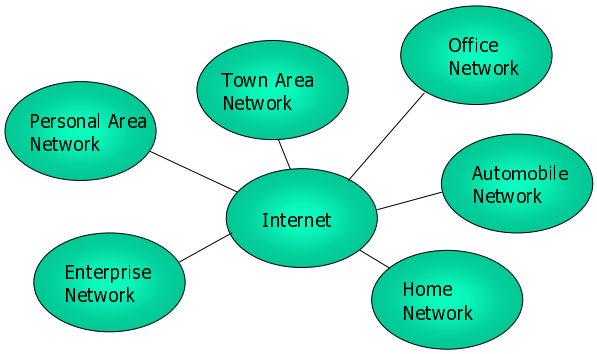

Figure 1: Internet-Scale Ubiquitous Computing

In this paper we define Internet-scale ubiquitous computing (InterUbicomp) as environments which embed computing, communication and sensing facilities and that are interconnected on a global scale, as shown in Figure 1. Any physical space can be augmented by embedding computers in it. Examples include the personal space (for body-worn devices), home spaces, vehicle spaces, office spaces, and university campus spaces. Public spaces such as airports, stations, bus stops, and public transport will also be globally connected, and different spaces will have different characteristics. For example, a personal space may contain a cellular phone, a PDA and an MP3 player that are connected by body networks. (Later these may be complemented by smart clothing.) Usually, appliances used in the personal space are small, battery-powered and wireless. On the other hand, in a home space, audio and video appliances are connected by wired high speed networks and are endowed with generous resources in terms of CPU, memory and energy.

¿From this we can see that, in InterUbicomp, one of the most important issues that needs to be taken into account is extreme heterogeneity. Future appliances will run on many different processors, will be equipped with a variety of I/O devices and will be connected by various types of networks and protocols. Therefore, it is necessary to take into account such extreme heterogeneity when designing software for *InterUbicomp*.

2.2 Requirements for Internet-Scale Ubiquitous Computing

The following requirements are considered as architectural guidelines to build InterUbicomp.

- Extreme Portability
- Uniform Behaviour
- High Level Abstraction
- Survival Systems

The software for InterUbicomp (both applications and middleware) should be portable because the environments are extremely heterogeneous, but writing good middleware is difficult, so we do not expect to reimplement it on every platform. However, while the API stays the same, the implementation (for example the resource management of a middleware component) should be optimized by tuning it to the characteristics of applications and platforms.

The environments should allow a user to behave in a uniform way when s/he moves to any spaces such as his/her house, an airport, and an office. For example, a user should control a television in the same way in any places such as his/her house or an airport. The uniform behavior is very important to design the user interface for InterUbicomp.

Also, the system software for building InterUbicomp should provide high level abstraction to allow us to build application programs easily. The existence of high level models is very important to build application programs in a systematic way since the lack of a suitable abstraction makes the structure of the application programs ad-hoc.

Lastly, InterUbicomp should survive against security attacks, system crashes, and natural disaster since our daily life will heavily rely on the environments. However, it is very difficult that a usual programmer takes into account survivability issues when implementing programs. Therefore, it is important to develop a methodology to add survivability to an existing programs.

We believe that the above requirements are very important to make future appliances at the low cost. Future appliances should be customized to respective persons since a person in future environments thinks that things attractive for him/her have high valuable.

3 Research Topics for Internet-Scale Ubiquitous Computing

In this section, we describe several technical challenges to implement InterUbicomp. The technical challenges are classified into three categories. The first category discusses software design issues. Especially, we describe problems to build software in extremely heterogeneous environments. The second category discusses high level abstraction to build complex ubiquitous computing applications in an easy way. The third category shows survivability issues to build systems that can survive against various security attacks and internal software bugs.

3.1 Software Design Issues

In this section, we describe several software design issues to develop software for InterUbicomp. As described in the previous section, portability and survivability are key issues to develop software for Internet-scale ubiquitous computing environments. We show several fundamental problems to build software in InterUbicomp.

3.1.1 Transparent QOS Evolution

As described in the previous section, we need to reuse a variety of COTS software components to realize InterUbicomp. However, these components must take into account various properties such as security, reliability, and predictability, but not all programmers have the required specialistic know-how in those areas. Therefore, these properties may be sometimes ignored in COTS software components.

One solution is to add these properties into COTS software components by post hoc. This means that a COTS component is transparently translated to a component that takes into account non functional properties such as security, reliability and predictability. The solution may not provide these properties completely, but the approach improves the quality of software drastically.

We are considering several approaches to realize the goal. One approach is to use aspect-oriented programming (AOP) techniques [2, 25, 27, 42] to add these non functional properties. When using AOP techniques, these properties are defined as aspects, and the aspects are merged into base COTS software components. In the most favourable cases, this can be achieved by recompiling the components using an AOP-aware compiler, but without substantial modifications to the source code. Therefore, this enables the COTS software components to be adapted according to the characteristics of underlying platforms. Also, there are several proposals to translate Java binary codes that are used for adding these properties by post hoc [8].

However, we have to take into account several issues to overcome the problems of current proposals. The first issue is whether the QOS evolution can be achieved transparently from a client program. If some assumptions of the components are changed by adding these features, the correctness of a client application may be violated. We need to define rigorous API semantics for middleware components, and need to check the assumptions when adding the features. The second issue is that current AOP techniques require to understand the internal structure of the base COTS components to define aspects. However, it is not easy to understand complex COTS software components such as Linux, Java and CORBA. It is important to export the high level abstract structure of the components to define various aspects.

The other approach to add non-functional properties transparently is the resource kernel [41, 46]. The resource kernel monitors the resource utilization of all applications, and enforces their behaviors if they violate the QOS requirements specified by them. The approach enables a traditional operating system to be converted to a real-time operating system by adding the resource kernel to a non real-time operating system. The resource kernel provides several primitives to control resources explicitly, therefore an application should be modified to invoke the primitives if it is required to make the application's behavior predictable. The primitives are considered as meta-interface [26] to control internal resources such as CPU, memory and network bandwidth. This seriously affects the portability of an application as described in the next section.

3.1.2 Portability Issues

In InterUbicomp, we need to take into account a variety of platforms. If we like to use COTS software components, it is important to consider how to exploit advanced characteristics provided by these underlying platforms. This requires to add meta level interface [26] or QOS parameters to control the internal algorithms of the COTS software components. However, it is not easy to export generic interface to control underlying platforms because the generic interface usually hides some low level characteristics of the underlying platforms. We believe that it is desirable that the interface should be customized to respective underlying platforms for enabling us to use the full power of the platforms. Our research aims are how to provide platform specific meta interface or QOS parameters in a systematic way, and how to build portable applications that access the platform specific meta interface or QOS parameters. We are considering to exploit AOP techniques and design patterns to implement COTS software components.

When building software, we need to take into account various tradeoff among many metrics. For example, a programmer needs to consider several metrics such as timeliness, precision, accuracy, and consistency to build mobile applications [34, 33]. It is impossible to satisfy all requirements so he/she must consider which requirements we need to focus on. The decision affects the program's structure dramatically.

For example, distributed applications should take into account three metrics: consistency, availability, and network partition [14]. If we like to improve the application's availability, we need to select either consistency or network partition. If we need to assume that network partition may occur, it is impossible to ensure complete consistency, and it is desirable to adopt optimistic protocols to satisfy consistency. On the other hand, if we require strong consistency, we need to assume that network partition will occur.

Building portable software requires to make its assumptions explicit because the assumptions are necessary to ensure the correctness of a program. Combining several components that provide different assumptions to their client programs may allow us to use the combined component as one that supports a wide range of assumptions. This requires us to switch components when the assumptions change. However, for the duration of the change, the component may cause inconsistency, so it is important that the change be executed atomically.

3.1.3 Robust Complex Systems

Platform software components to realize InterUbicomp are very complex, and it is very hard to build such complex software components. We need to consider several issues to design robust software components to reduce the development cost.

A programmer needs to choose a software platform to develop his program, and takes into account various tradeoff among the platforms. For example, it is easy to build a multi-threaded program in Java, but the behavior may not be predictable due to the unpredictability in Java virtual machines. On Linux, an efficient and predictable multi-threaded program can be written, but we may have a lot of serious bugs due to manual memory management. Moreover, if an application requires more severe timing requirements, we may adopt RT-Linux that implements Linux on a small real-time operating system, and we implement timing critical programs on the small real-time operating system.

We believe that multi-layered platforms are preferable to build complex software components, and we can choose a suitable software platform to develop our program. In the approach, we configure these platforms in a layered fashion. This structuring is very natural in usual cases. For example, the Java platform is able to run on the Linux platform. If predictability is more important, or the development cost can be reduced by using existing software, we will choose the Linux platform, but the Java platform is desirable if productivity is more important. In our project, we are working on a uniform component framework that enables components on different platforms to communicate with each other without considering on which platform a target program is executed.

Also, it is important to take into account the behavior of programs explicitly. We believe that predictability is a key issue to build robust software. For example, let us assume that we like to implement a fault-tolerant system in Java. When designing fault-tolerant software, it is required to detect faults using timeouts. However, the garbage collector in a Java virtual machine may unpredictablly block a process for such as long time that an extraneous timeout occurs. The process will therefore erroneously believe that the subsystem being monitored has failed and will start a complex recovery procedure. To define a correct timeouts value, the behavior of COTS software components should be predictable. In our approach, we specify the assumptions of software explicitly, and the assumptions contain the desirable predictable behavior of the software.

3.1.4 Rigorous API Semantics

COTS middleware components to implement InterUbicomp run on various platforms. Therefore, if rigorous API semantics[1] are not defined, it is impossible to build robust and survival software components.

In traditional programs, we do not specify their assumptions explicitly. However, the assumptions are necessary to check whether the programs can be executed correctly on their platforms. Also, it is important that an application specify its requirements explicitly. Thus, it is possible to ensure whether it runs correctly. Also, it may be possible to choose an appropriate middleware component that satisfies the requirements by examining middleware component's assumptions. However, we found that it is important to assume predictable behavior when building these COTS middleware components [35, 32]. For example, our mobile middleware adapts its behavior according to the characteristics of operational environments, but it is not easy to adapt the components correctly because underlying software such as operating systems and networks do not provide predictable behavior, thus checking the assumptions of the components is not easy.

Ubiquitous computing applications require us to compose several existing services or appliances to define a new service. Also, applications can change their behaviors according to operational environments. If these behaviors are not predictable, it is impossible to define the assumptions of the composed service. For example, the failure semantics of composed services should be defined rigorously because it is impossible to write a robust program with underlying services whose failure semantics is obscure. Also, this kind of predictability is important to develop context-aware applications that do not behave in an unexpected way.

3.1.5 High Level Abstraction

For building complex systems, abstraction is a very powerful tool to deal with the complexities. There are a lot of places where using abstraction is effective. For example, home computing applications require to access various home appliances such as televisions and video cassette recorders. The applications require to access these appliances without taking into account the difference among implementations to build portable applications. Also, ubiquitous computing applications require various new techniques such as service integration and context awareness. These techniques require good abstraction to build well structured applications.

Also, complex software usually adopts various techniques such as optimization and adaptation. Ad-hoc uses of the techniques make the structure of programs very unclear. For example, by inserting a lot of "if" statements to check the current situation, we can build context-aware applications. However, the program is very difficult to modify when the program needs to consider another situation. We need good abstraction to support adaptive context-aware applications. Similarly, adopting aggressive optimization makes programs less portable since the optimization may not be effective in another situation. We believe that using better abstraction makes it possible to deal with optimization in a portable way.

Finally, domain specific languages that provide high level abstraction are very useful to build complex software. For example, in [62], their paper shows the effectiveness of domain specific languages to verify the complex cache coherence protocol of their distributed file system. Also, in [31, 60], high

[1] It is very important that the API also should be implementable. In some situations, desirable API semantics is not implementable. For example, the local procedure call semantics is not implementable in distributed systems.

level abstraction provided by domain specific languages are very useful to develop system software. We believe that there are a lot of areas where domain specific languages are useful such as to describe the composition of services, the adaptation of applications, and the management of context information.

3.2 System Issues

InterUbicomp requires several high level abstractions to provide advanced features. In this section, we discuss five topics that we are interested in.

3.2.1 Service Integration

In InterUbicomp, a variety of services are installed in a variety of places. The services are offered by some appliances or servers. One of the most important issues is to develop a new service from existing services [47][2]. For example, the combination of Internet streaming services and television enables us to watch Internet TV on our television. Traditionally, home appliances provide a fixed set of services, but adding a new service requires to buy another appliance. Our approach enables us to use existing appliances by composing the new service with the appliance.

The service integration is very promising, but it is very difficult to achieve because each service requires to offer clean interface. Also, it is important how to specify the composition of services. We are considering several types of service compositions. The first one is the functional composition that is used to compose several devices. For example, a television can be considered as a composition of a tuner, an amplifier, a display and a speaker. The second composition is the presentation composition that is used to compose several presentations. For example, the graphical user interface of a television and a video cassette recorder can be combined as one user interface to control the appliances as one appliance. By clearly separating the functional composition and the presentation composition, it is possible to build complex services from existing services in an easy way.

Each service that takes into account the composition requires programmers to specify input/output ports explicitly. This scheme enables us to compose services. Also, a composed service can be used to create a new service by composing it with other services. The connectivity of the port is determined by a protocol. The scheme is proposed by the ROOM modeling language [51], and we believe that component composition is very similar to service integration defined by us. Secondly, we need a way to specify the composition of services. A domain specific language to specify the composition is desirable. As described in the previous section, it is important to define the rigorous semantics and the behavior of the composition, and the semantics provides by the language should be implementable for any services and any configuration because ad-hoc composition may violate our expectation.

3.2.2 Context Awareness

Ubiquitous computing applications need to adapt their behaviors according to environmental information such as location, emotion and preference. The most important point is how to model environmental information and access the model from an application. We call the model the *world model*. The real world is represented in computers as the world model. The model should represent every aspect of our world. Also, the model should represent various information about systems such as network topologies, traffic information and machine configurations. The information can be used to optimize the behavior of an application.

However, our question is how to represent the complete world model. We have found that it is not easy to model our real world completely. The model may be too complex to be used from usual applications. Therefore, it is desirable to model necessary information for respective applications. In our project, we are working on two issues for building practical context aware applications.

The first issue is to construct the world model in a modular way. In our approach, the world model is divided into several domains. Each domain represents a part of the world model. For example, one domain represents location information of users in an office, and another domain represents the emotion of a user. Each domain has a name to identify it. We are currently designing the naming scheme to identify contexts. Also, it is important to compose several domains as one domain. Therefore, each user can construct a domain representing a suitable world model for his application.

The second issue is how to build context-aware applications. Our approach enables us to specify a suitable world model for respective situations. However, if a component represents a context explicitly, different situations require to modify programs. Our approach is to specify context information as an aspect, and the aspect is merged to an application at runtime. Therefore, it is possible to represent how to adapt programs according to context in each case.

3.2.3 Application Specific Networks

A variety of appliances such as home appliances and personal appliances will be connected to the Internet in the near future, and the number of appliances running on the Internet will be extremely large. The Internet provides connectivity among the computers executing various services and applications for providing new advanced services to us. However, for connecting these appliances and services, the traditional IP protocol is not sufficient because these future appliances may require their own standard protocols like Jini and HAVi. Thus, the connectivity cannot be realized without protocol translation. Also, these appliances may require their own naming schemes, routing algorithms, and mobility/multicast supports. Therefore, it is preferable that this functionality be customized according to the respective applications. Since the functionality that can be added at the IP level is too generic for most applications, we believe that future networks should be customized according to the characteristics of their respective applications.

In our project, we propose a virtual overlay network for integrating networked home appliances [40]. Virtual overlay networks enable us to build new application specific networks on existing networks. Our virtual overlay network is specialized to access a variety of appliances on home networks from personal or home appliances on other home networks. Application level gateways in our virtual overlay network convert between home network protocols such as Jini and HAVi and the HTTP protocol. Each appliance is identified with a URL. The interpretation of URL is processed in the application level gateways. Our definition of URL can contain some variables whose values are substituted in application level gateways. We believe that the networks architecture will become a base to compose several appliances on different home networks.

[2] In [6], Buxton said that the real value lies in the symbiotic relashionship among a suite of applications, rather than the value of any one "killer" application. Hence, approaches that are limited to individual applications, or applications in isolation, run the risk of missing the target. If true, then the consequence is that a far more holistic approach must be taken.

3.2.4 Seamless Spaces

In InterUbicomp environments, a user may have to control the same kind of appliances in many different places; for example, we may find televisions at home and in public spaces, all of them with different user interfaces. Firstly, we want each user to be able to control those different applications in a common way, for example from his/her PDA. Secondly, we do not want to tie down the user to a specific controller: s/he should have the choice of, say, controlling the appliances by voice if he hands are busy. Other users may choose other interaction devices as their preferred controllers. Everyone will have a familiar and uniform way to control appliances regardless of their make and location. There may still be higher level constraints: a user needs to behave differently in a personal space and in a public space, because the public space is shared with many unknown people. But, as far as the technical aspects are concerned, the user should be able to behave uniformly without having to take into account the particular space in which s/he is.

In our project, we define the universal interaction protocol between interaction devices and various applications and services [38, 39]. Currently, our universal interaction protocol is based on bitmap images and keyboard and mouse events. In our system, any graphical user interface can be shown on any output device that has a display. The size and color are converted according to respective interaction devices, then our system can generate GUI on any output devices. Also, events generated by input interaction devices are converted into keyboard and mouse events, and the events are transferred to applications. We are currently considering better universal interaction protocols because the current protocol is too low level, and is difficult to customize the user interface according to a user's preference. However, our approach shows that the effectiveness of our approach, and all appliances can be controlled in the same way. In our project, we think that "Seamless spaces" is one of the most important visions to imlement InterUbicomp environments. The vision will avoid to waste human attention that is a precious resource in our life, and will make our life more confortable and attractive.

In a similar way, we like to communicate with other persons using the same method regardless of where they are. In the system, any communication styles are converted to other communication styles. Thus, any communication devices can be used to communicate with each other by converting media formats. The approach is very similar to the *Universal Inbox project*(University of California at Berkeley) [47].

3.2.5 Peer-to-Peer Systems

In the future, appliances will be connected to the Internet, and will communicate with each other. The appliance may have various information and a user likes to find expected information on any appliances. Also, these appliances may monitor various information, and notify some events to a user who has an interest in the event. Also, smart dust [23] monitors and stores any information in the world, and the information can be available for any persons.

Currently, peer-to-peer systems are popular to share MP3 files such as Napster and Gnutella. In InterUbicomp, we require large scaled peer-to-peer storage and event systems [48, 58, 29, 50] for storing and delivering various information monitored by a variety of sensors. These systems require new techniques that integrate routing and naming. A network connecting these appliances is constructed at the application level. This means that each application forwards a message to near appliance, and the appliance delivers the message towards a destination appliance. However, personal appliances are moved with their users so the networks are reconfigured dynamically [13]. Therefore, we consider that the combination of a peer-to-peer system and an ad-hoc network is very important to support InterUbicomp. The combination provides new challenges for building future peer-to-peer systems and enables us to create new services.

We are interested in connecting a lot of personal area networks, and these personal appliances are integrated in a peer-to-peer system. We call the peer-to-peer system *wearable home*. This means that a user carries most of portable home appliances with him/her, and these appliances can be used at anytime and anywhere. Each person's wearable home is connected with each other in an ad-hoc way. This organizes a virtual family to share their spaces temporally. If a person takes a photo that is of interest to other persons, an event is delivered to the persons. Also, someone's information can be retrieved by other persons who have an interest in the information.

3.3 Survivability issues

Once ubiquitous computing evolves from the research phase into the deployment phase, reliability is going to be a major concern. The authors of [12] define the related term of "survivability" as the quality of a system that works correctly even in the face of attack, failure or accident. The more the ubiquitous computing infrastructure becomes pervasive, and the more we rely on it, the greater the impact of any failures will be.

We have to face two fundamentally different causes of trouble: accidental and intentional ones—in other words, bugs and attacks. We shall look at these in the next two sections.

3.3.1 Security

For ubiquitous computing, the security property of greatest relevance is probably going to be availability—ensuring that the system performs as expected in response to requests from legitimate users. This property is threatened by *denial of service* (DOS) attacks, in which a malicious principal intentionally exhausts some resource of the system. Classical examples of denial of service include the exhaustion of a machine's computing cycles, the exhaustion of the capacity of a communication channel and the exhaustion of the available mass storage space.

Sceptical observers sometimes question the plausibility of such threats, on the grounds that they offer little or no advantage to the attacker; but to assume that only attacks backed by a rational motivation will be performed is very naive, as viruses have amply demonstrated. There are several lessons to be learnt from the conventional personal computer environment. One is that pointless vandalism is a sufficient "reason" for many devastating attacks. A second one is that it is dangerous and unwise to deploy an infrastructure in which untrusted code is automatically executed without checks or restrictions: the number of email viruses that have spread thanks to Microsoft Outlook in recent years is a testimony of the dangers of ignoring caution in order to provide fancier functionality. And, although the particular case of Outlook is a textbook example of the wrong way to address the issue, it is certainly true that the tension between security (which requires lots of checks and authorizations) and usability (which requires things to "just work" automatically, without the constant need for extra input from the poor user) is going to be one of the most difficult trade-offs for ubiquitous computing, especially given the extra threats introduced by the Internet-scale dimension.

A related lesson from the virus and trojan wars is that disabling the system under attack may not be the primary objective. Sometimes malicious code will infect a system but remain hidden, waiting for a wake-up call from a coordinating master that will designate the real target. At that point all the deployed copies of the malicious software will attack that target simultaneously, performing a so-called distributed denial of service (DDOS) attack from the infected machines of otherwise innocent users. In designing the software infrastructure for Internet-scale ubiquitous computing it is therefore important to recognize these threats as highly likely and to include from the start the appropriate countermeasures. Of course not all attacks will be easily prevented (in particular there is little that the individual node can do to avoid a DDOS attack on its communication channel) but we should at least learn from existing failure modes to avoid repeating the same mistakes in the ubiquitous computing context. While it may be hard to do much against a concerted DDOS attack from network nodes outside our control, we should at least make it difficult to subvert and use our own nodes as accomplices in the attack.

In the light of this, the execution of foreign code should be treated with great care. The Java model of sandboxing is certainly a good first step, and a definite improvement on that of Outlook in which everything gets executed with no questions asked, but the full picture should also take into account some more subtle cases. These include updates to the system software of the device (which can't be sandboxed since they need to run with full privileges and full access to the machine) and code that gets executed for "unofficial" reasons such as the strategic exploitation of bugs (buffer overflow [10, 56] being a notorious case).

Going back to resource exhaustion, the ubiquitous computing context also introduces previously unknown threats: among the most interesting is the *sleep deprivation torture*. Mobile devices are usually battery-powered, but in most cases their power consumption is such that, if run continuously, their battery would only last for a few days. To achieve a more reasonable lifetime, they spend most of their duty cycle in a quiescent state ("sleep"). If an attacker manages to keep them awake with irrelevant queries, he can exhaust the battery energy and walk away, leaving the device disabled.

Another security issue of relevance in any distributed system is authentication—a precondition for any other security property since we cannot grant a principal the correct set of rights if we are not sure whether we are talking to the expected one or to an impersonator. The ubiquitous computing case is interesting because the traditional authentication strategies of distributed systems fail in the ad-hoc networking environment, where we cannot assume permanent connectivity to a central server. This means we cannot use an Authentication Server to distribute Kerberos-style tickets; and also that, while we can still verify public key certificates offline, we cannot ensure timely revocation if there is no guarantee of online connectivity. We have developed a security policy model that addresses this problem by establishing a local bond between a master and a slave device. The master may also specify which other devices the slave is allowed to contact, and there is provision for transferring authority to another master [54, 55, 57].

3.3.2 We Have to Live with Software Bugs

Our software usually contains a lot of bugs, and it may not work correctly in some situations. As described above, this may disturb our life seriously. For example, Linux operating systems report several bugs in every release. Also, COTS middleware such as CORBA and Java contains a lot of bugs, but these bugs may not be removed because additional functionalities are introduced in every release.

Also, it is very difficult to avoid security bugs because it requires a wide range of knowledge about security issues, and building fault tolerant software requires a wide range of knowledge about reliability. Usual programmers do not have these knowledge so it is usual to contain security and reliability bugs in our software.

The above discussion is a negative opinion to realize ubiquitous computing environments. Moreover, in InterUbicomp, the environments require extreme heterogeneity, but software will be reused on a variety of platforms to reduce development cost.

We are considering several approaches to solve the above problems. In the first approach, we like to make the assumption of software clear, and a client's and a platform's requirement should be also clear. This makes it possible to check whether a program works correctly in every situation. In our approach, we provide several implementations that offer different assumptions. Our system chooses an implementation that ensures its assumptions. To specify an assumption correctly, a program needs to behave in a predictable way. However, building predictable software is very hard, and it may contain predictability bugs that may violate the specified assumption. Therefore, a mechanism to recover from predictability bugs is necessary to build survival systems.

We are also considering to take into account a variety of bugs in software. Especially, we will focus on security bug, reliability bugs, and predictability bugs. We call this survivability bug tolerance. Survivability bug tolerance enhances the survivability of a flawed system by post hoc dealing with system's security flaws, prediction errors, and a variety of program errors [9]. For example, software fault isolation [61] detects memory leaks, stack guards [10] detects buffer overflows vulnerability, and performance assertion detects performance errors [44]. Also, transaction's all or nothing properties may help to make our system survival [52]. Our project likes to extend the previous approaches to detect a variety of bugs, and avoid the disaster of our life.

4 Current Status

Currently, we are working on building several infrastructure components and prototype applications for realizing InterUbicomp. Now, we are developing HAVi [30] based home appliances [39, 37], a user interface system that realizes universal interaction [38], a network system for integrating home networks [40]. Also, we are working several middleware to speak the X.10 protocol and a toolkit for building continuous media applications. These components will become platform software components for building our ubiquitous computing environments.

These middleware components are developed on multilayered platforms, and a HAVi middleware component and our network system enable us to develop an application that composes several home appliances. We are also working on a couple of infrastructure software. For example, we are working on embedded Linux for building a variety of appliances.

Now, we are starting two new projects. In the first project, our Java virtual machine provides platform specific interface that exports the characteristics of underlying platforms directly. Also, we are designing a Java translator that enables us to add timing constraints without modifying base software. The project also focuses on how to make the behavior of Java virtual machine predictable.

In the second project, we are working on the composition of several appliances such as HAVi appliances, X10 devices and Internet services such as the VoIP service. In the project, each appliance provides the assumption for composing other appliances. The composition is specified by a special language that describes the composition of appliances. The language provides a syntax to specify these assumptions explicitly.

5 Role of Object-Orientation

In this section, we describe four topics about the role of object-orientation for realizing InterUbicomp.

The first topic is about distributed object-oriented technologies for building InterUbicomp. There are a couple of standard specifications for building distributed systems. Especially, CORBA and Java RMI are widely adopted. Distributed objects provide us a unique view to access various programs. However, the homogeneous approach is not suitable to build InterUbicomp because unique view requires a single naming scheme, and the approach cannot support applications specific naming scheme. In the future, we need to deal with a huge number of objects. For example, different content types require their own naming scheme. Also, single communication style cannot be enough to support various types of applications. We believe that a traditional object model is too simple to support various types of objects in Internet-scale ubiquitous computing.

The second topic is about object-oriented design. Advanced object-oriented design techniques such as aspect-oriented programming and design patterns are very useful to build robust and portable software. One of the problem of the object-oriented design is that objects hide their implementations. The role of abstraction is fundamental to build portable programs. However, the abstraction makes the performance of a program slow. In [26], Kiczales has proposed open implementation that exports API for controlling the implementation. The problem is that we need different meta interfaces for different implementations. Thus, the meta interface decreases the portability of a program. We need to reconsider the role of abstraction for designing programs.

The third topic is about component models. The component models are very important to build large-scale software by composing several components. Especially, if there are multiple platforms, components on different platforms should be communicated using the same mechanism. Thus, the component model can be implemented on any platforms, and it should be easy to reimplement the component on other platforms.

The fourth topic is about Java. A program written in Java is portable on various platforms. However, the implementation may cause a lot of problems. For example, if we like to write fault-tolerant software on Java, the timeout should be correctly detected. However, garbage collection may delay to check whether a site is alive or not. Then, it may increase the mistake to detect failures. We believe that we need to ensure predictable behavior to implement some functionalities.

In the future, we will seek object-oriented abstraction that is useful to build InterUbicomp, but we believe that object-orientation is very important to build software on extremely heterogeneous environments.

6 Conclusion

In this paper we have described a number of technical challenges that need to be addressed in order to build InterUbicomp systems.

In the InterUbicomp scenario users will be completely surrounded by computers. They will depend on them to the same extent that, in modern societies, people depend on electricity. Nobody would however accept a situation in which the comfort of one's lifestyle depends on software as unreliable, crash-prone and insecure as that which drives today's personal computers. We need a major engineering effort to improve software robustness, and we believe this to be perhaps the most important challenge for InterUbicomp. We must strive to build systems that will survive malicious attacks, environmental failures and, not least, any leftover bugs.

Building robust software will require us to revisit, update and unify several well-known computer science research themes. For example, during software development we need to reconsider abstraction, decomposition and transparency. For distributed software we need to consider the interaction of naming and routing. InterUbicomp will see the deployment of large numbers of net-connected appliances, so we expect the new peer-to-peer networking strategies to grow in importance and popularity.

We must also acknowledge that it is impossible to anticipate all future requirements when developing software. We need to take into account the so-called "non functional properties" such as security, predictability, scalability and reliability. We need to develop techniques to add these properties transparently as aspects, because the implementation of these non functional properties requires specialized know-how that is often quite independent of that required to develop the functional parts.

Finally, in order to simplify the process of developing ubiquitous computing applications, we need to provide high level abstractions [36] coupled with clear and unambiguous API semantics.

References

[1] G.D. Abowd, E.D. Mynatt, "Charting Past, Present, and Future Research in Ubiquitous Computing". *ACM Transactions on Computer-Human Interaction*, 2000.

[2] M. Aksit, B. Tekinerdogan, "Aspect-Oriented Programming Using Composition Filters". Position Paper for the Aspect Oriented Programming Workshop, Springer-Verlag, LNCS 1543, 1998.

[3] Project Aura - Distraction-free Ubiquitous Computing, Carnegie Mellon University, http://www.cs.cmu.edu/~aura/.

[4] G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. Sussman, D. Zukowski, "Challenges: An Application Model for Pervasive Computing". In "Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking", 2000.

[5] B. Brumitt, J. Krumm, B. Meyers, S. Shafer, "Ubiquitous Computing and the Role of Geometry". *IEEE Personal Communications*, August 2000.

[6] B. Buxton, "Integrating the Periphery and Context: A New Taxonomy of Telematics". In "Proceedings of Graphics Interface '95", 1995.

[7] K. Cheverst, N. Davies, K. Mitchell, A. Friday, "Experiences of Developing and Deploying a Context-Aware Tourist Guide: The GUIDE Project". In "Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking", 2000.

[8] S. Chiba, "Load-time Structural Reflection in Java". In "Proceedings of ECOOP 2000 – Object-Oriented Programming", LNCS 1850, Springer Verlag, pp. 313–336, 2000.

[9] C. Cowan, C. Pu, H. Hinton, "Death, Taxes, and Imperfect Software: Surviving the Inevitable". In "Proceedings of the New Security Paradigms Workshop", 1998.

[10] C. Cowan, P. Wagle, C. Pu, S. Beattie, J. Walpole, "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade". In "Proceedings of DARPA Information Survivability Conference and Expo (DISCEX)", 2000.

[11] A.K. Dey, G. Abowd, D. Salber, "A Conceptual Framework and a Toolkit supporting the Rapid Prototyping of Context-Aware Applications". *Human-Computer Interaction (HCI) Journal* 16, 2001.

[12] R.J. Ellison et al., "An Approach to Survivable Systems", Carnegie Mellon University, CERT Coordination Center, 1999.

[13] D. Estrin, R. Govindan, J. Heidemann, "Scalable Coordination in Sensor Networks". In "Proceedings of Mobicom99", 1999.

[14] A. Fox, S.D. Gribble, Y. Chawathe, E.A. Brewer, "Cluster-Based Scalable Network Services". In "Proc. 1997 Symposium on Operating Systems Principles (SOSP-16)", St-Malo, France, Oct. 1997.

[15] N. Gershenfeld, *When Things Start to Think*. Owl Books, 2000.

[16] H.-W. Gellersen, A. Schmidt, M. Beigl, "Adding Some Smartness to Devices and Everyday Things". In "Proceedings of the Third Workshop on Mobile Computing System and Applications", 2000.

[17] A. Harter, A. Hopper, P. Steggles, A. Ward, P. Webster, "The Anatomy of a Context-Aware Application". In "Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking", 1999.

[18] A. Harter, A. Hopper, "A Distributed Location System for the Active Office". *IEEE Network Magazine*, 8(1), January 1994.

[19] T. Hodes, R.H. Katz, "A Document-based Framework for Internet Application Control". In "Proceedings of the Second USENIX Symposium on Internet Technologies and Systems", 1999.

[20] A. Hopper, "Sentient Computing", The Royal Society Clifford Paterson Lecture, 1999.

[21] InfoSphere Project - Smart Delivery of Fresh Information, http://www.cc.gatech.edu/projects/infosphere/.

[22] H. Ishii, B.Ullmer, "Tangible Bits: Towards Seamless Interfaces between People, Bits and Atoms". In "Proceedings of Conference on Human Factors in Computing Systems", 1997.

[23] J.M. Kahn, R.H. Katz, K.S.J. Pister, "Mobile Networking for Smart Dust". In Proceedings of Mobicom 99, 1999.

[24] N. Khotake, J. Rekimoto, Y. Anzai, "InfoStick: an interaction device for Inter-Appliance Computing". In "Proc. Workshop on Handheld and Ubiquitous Computing (HUC'99)", 1999.

[25] G. Kiczales et al., "Aspect Oriented Programming". In "Proceedings of the European Conference on Object-Oriented Programming", Springer-Verlag, 1997.

[26] G. Kiczales, J. Lamping, C. V. Lopes, C. Maeda, A. Mendhekar, G. Murphy, "Open Implementation Design Guidelines". In "Proceedings of the 19th International Conference on Software Engineering (ICSE)", 1997.

[27] G. Kiczales et al., "An Overview of AspectJ". In "Proceedings of the European Conference on Object-Oriented Programming", Springer-Verlag, 2001.

[28] T. Kindberg et al., "People, Places, Things: Web Presence for the Real World". In "Proceedings of the Third Workshop on Mobile Computing System and Applications", 2000.

[29] J. Kubiatowicz et al., "OceanStore: An Architecture for Global-Scal Persistent Storage". In "Proceedings of ASPLOS 2000", 2000.

[30] R. Lea, S. Gibbs, A. Dara-Abrams, E. Eytchson, "Networking Home Entertainment Devices with HAVi". *IEEE Computer* 33(9), 2000.

[31] F. Merillon, L. Reveillere, C. Consel, R. Marlet, G. Muller, "Devil: An IDL for Hardware programming". In "Proceedings of OSDI 2000", pp. 17–30, San Diego, October 2000.

[32] T. Nakajima et. al., "Integrated Management of Priority Inversion in Real-Time Mach". In "Proceedings of International Conference on Real-Time System Symposium", 1993.

[33] T. Nakajima, "A Framework for Building Adaptive Continuous Media Applications using Service Proxies". In *Handbook of Multimedia Computing*, CRC Press, 1998.

[34] T. Nakajima, "A Framework for Building Environment-Aware Software". In "Proceedings of International Symposium on Object-Oriented Real-Time Distributed Computing", 1999.

[35] T. Nakajima, "Practical Explicit Binding Interface for Supporting Multiple Transport Protocols in a CORBA system". In "Proceedings of International Conference on Network Protocols", 2000.

[36] T. Nakajima, "Towards Universal Software Substrate for Distributed Embedded Systems". In "Proceedings of International Workshop on Object-Oriented Reliable Distributed Systems", 2001.

[37] T. Nakajima et al., "A Framework for Building Audio and Visual Home Appliances on Commodity Software". In "Proceedings of the IASTED International Conference on Internet, Multimedia Systems, and Applications", 2001.

[38] T. Nakajima, A. Hasegawa, "Universal Interaction with Home Appliances using Stateless Thin-Client Architecture". In "Proceedings of the 2nd International Workshop on Ubiquitous Computing and Communication", 2001.

[39] T. Nakajima, "System Software for Audio and Visual Networked Home Appliances on Commodity Operating Systems". In "Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms - Middleware 2001", 2001.

[40] T. Nakajima, D. Ueno, E. Tokunaga, H. Ishikawa, I. Sato, H. Aizu, "A Virtual Overlay Network for Integrating Home Appliances". In "Proceedings of the International Symposium on Applications and the Internet", 2002.

[41] S. Oikawa, R. Rajkumar, "Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior". In "Proceedings of the IEEE Real-Time Technology and Applications Symposium", 1999.

[42] H. Ossher, P.L. Tarr, "Hyper/J: Multi-Dimensional Separation of Concerns for Java". In "Proceedings of the International Conference on Software Engineering", 2000.

[43] Oxygen Project, Laboratory for Computer Science, MIT, http://oxygen.lcs.mit.edu/.

[44] S.E. Perl, W.E. Weihl, "Performance Assertion Checking". In "Proceedings of ACM SOSP'93", 1993.

[45] R. Picard, *Affective Computing*. The MIT Press, 1997.

[46] R. Rajkumar, K. Juvva, A. Molano, S. Oikawa, "Resource Kernels: A Resource-Centric Approach to Real-Time Systems". In "Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking", 1998

[47] B. Raman, R. Katz, A. Joseph, "Universal Inbox: Providing Extensible Personal Mobility and Service Mobility in an Integrated Communication Network". In "Proceedings of the Third Workshop on Mobile Computing System and Applications", 2000.

[48] S. Ratnasamy, P. Francis, M. Handly, R. Karp, S. Shanker, "A Scalable Content-Addressable Network". In "Proceedings of SIGCOMM'01", 2001.

[49] J. Rekimoto, M. Saitoh, "Augmented Surfaces: A Spatially Continuous Workspace for Hybrid Computing Environments". In "Proceedings of CHI'99", 1999.

[50] A. Rowstron, P. Druschel, "Past: Persistent and Anonymous Storage in a Peer-to-Peer Networking Environment". In "Proceedings of the 8th Workshop on Hot Topics on Operating Systems", 2001.

[51] , B. Selic, G. Gullekson, P. T. Ward, *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

[52] M. Seltzer, Y. Endo, C. Small, K. Smith, "Dealing with Disaster: Surviving Misbehaved Kernel Extensions", In "Proceedings of USENIX Operating System Design and Implementation", 1996.

[53] I. Siio, T. Masui, K. Fukuchi, "Real-world Interaction using the FieldMouse". In "Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'99)", 1999.

[54] F. Stajano, R. Anderson, "The Resurrecting Duckling: Security Issues in Ad-Hoc Wireless Networks". In "Security Protocols, 7^{th} International Workshop, Proceedings", LNCS 1796, Springer-Verlag, 1999, pp. 172–182, http://www-lce.eng.cam.ac.uk/∼fms27/duckling/.

[55] F. Stajano, "The Resurrecting Duckling—What Next?". In "Security Protocols, 8th International Workshop Proceedings", LNCS 2133, Springer-Verlag, 2001, pp. 204–214. http://www-lce.eng.cam.ac.uk/∼fms27/papers/duckling-what-next.pdf.

[56] F. Stajano, H. Isozaki, "Security Issues for Internet Appliances". In "Proceedings of the IEEE International Workshop on Linux and Internet Appliances", 2002 (to appear).

[57] F. Stajano, *Security for Ubiquitous Computing*. John Wiley & Sons, 2002 (to appear).

[58] I. Stoica et al., "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications". In "Proceedings of SIGCOMM'01", 2001.

[59] T.-L. Pham, G. Schneider, S. Goose, "A Situated Computing Framework for Mobile and Ubiquitous Multimedia Access using Small Screen and Composite Devices". *ACM Multimedia*, 2000

[60] S. Thibault, R. Marlet, C. Consel. "Domain-Specific Languages: from Design to Implementation - Application to Video Device Drivers Generation". *IEEE Transactions on Software Engineering* 25(3), 1999.

[61] R. Wahbe, S. Lucco, T. Anderson, S. Graham, "Efficient Software-Based Fault Isolation". In "Proceedings of 14th SOSP", 1993.

[62] R. Wang, T. Anderson, M. Dahlin, "Experience with a Distributed File System Implementation". *Software Practice and Experience*, 1998.

[63] R. Want, B. Schilit, N. Adams, R. Gold, K. Petersen, J. Ellis, D. Goldberg, M. Weiser, "The ParcTab Uniquitous Computing Experiment". Technical Report CSL-95-1, Xerox Palo Alto Research Center, 1995.

[64] Mark Weiser, "The Computer for the 21st Century". *Scientific American* 265(3), 1991.