JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 5, No. 2, June 2011, pp. 141-149

Ubiscript: A Script Language for Ubiquitous Environment

Minkyu Lee and Dongsoo Han*
Department of Computer Science, Korea Advanced Institute of Science and Technology, Daejeon, Korea
niklaus@kaist.ac.kr, dshan@kaist.ac.kr

Abstract

Many distributed and heterogeneous services and devices are accessible in ubiquitous computing environment, so interoperating those services and devices is one of the key tasks in implementing ubiquitous applications. We used to use script languages in integrating such interoperating components and services. However currently available most script languages are not suitable for ubiquitous environment because there are so diverse forms of interoperation targets such as service objects, web, legacy objects and programmable devices. So it is worthwhile designing a new script language well-suited to ubiquitous environment. In this paper, we propose a new script language, called Ubiscript, for the ubiquitous environment. We develop and adopt several unique language features such as remote scope, multiple contexts, web and legacy objects, remote exception handling, etc. in Ubiscript to overcome the limitations of conventional script languages. In this paper, we also describe the implementation of Ubiscript and its runtime system. A couple of ubiquitous applications were developed in Ubiscript, and the applications are tested on the runtime system. According to our experiences and evaluation, Ubiscript turned out to have a high potential in its expression power and contribution to improving ubiquitous application developers' productivity.

Category: Ubiquitous computing

Keywords: Ubiquitous computing; Script language; Mobile code

I. INTRODUCTION

The ubiquitous computing [1] assumes that there are many distributed and heterogeneous devices, appliances, sensors in our everyday life environment such as home, office, car, hospital, and so on. These are interconnected via networks and interoperate seamlessly with each other in ubiquitous applications to provide convenient services for users. Let's consider the following morning scenario in a smart home: Tom is a business man and he has to go to his office by 9 o'clock at every weekday. At 7 o'clock every weekday morning, digital alarm clock in his bedroom starts ringing to wake him up. In his living room, a large liquid crystal display (LCD) panel on the wall displays the weather information of the day such as temperature, humidity, the possibility of rain, and a television shows today's morning news. If the temperature is below the preset minimum temperature, the air-conditioner in his car is automatically turned on to heat up the car. The first appointment location stored in his calendar software in the computer at his office is automatically set up as the destination in the navigation system of his car.

In order to implement this scenario we need to consider many technologies such as context-awareness and service discovery, but we will focus on how to integrate services and devices in ubiquitous environment. In the above scenario, we can figure out four different kinds of interoperation. The first one is interoperation with service objects. In order to control a device, we need a service object implemented in a middleware such as UPnP [2], Jini [3], service location protocol (SLP) [4], and so on. The second one is interoperation with web. We can acquire diverse information such as weather, traffic, geography and currency, so it is an important issue to access information on the web in ubiquitous computing environment. The third one is interoperation with legacy objects. Since we always live together with legacy systems even when ubiquitous computing vision is realized in

Open Access 10.5626/JCSE.2011.5.2.141 http://jcse.kiise.org

Received 22 February 2011, Accepted 21 March 2011
*Corresponding Author

 pISSN: 1976-4677 eISSN: 2093-8020

the future, so integrating with legacy systems is crucial in real situations. The last one is interoperation with programmable devices. In ubiquitous environment, there are many programmable devices such as television, refrigerator and navigation system. They can execute a program by themselves because they have own processor, memory and networking capability. Providing just service objects is not enough to control them. For example, if a graphical user interface is necessary on a television, we cannot implement such service just calling the operations of service objects. We must prepare a separated program and deploy it on the device and then execute it. That is, we need an interface to execute a separate program on a programmable device.

In this paper, we introduce Ubiscript, a script language designed to meet the requirements for ubiquitous computing environment. The Ubiscript language is an object-based, interpreted script language syntactically similar to ECMAScript [5]. The several new characteristics are embodied in Ubiscript language. Those are remote scope, web and legacy objects, multiple contexts, remote exception handling, etc. Remote scope is a linguistic element to naturally express the nature of ubiquitous computation. It can be simply understood as the extension of scope concept of general programming language theory for the application of the remote devices. With remote scope, we can write a ubiquitous application that is executed on distributed multiple computing devices in a single source file. Another key characteristic of Ubiscript is multiple contexts. In ubiquitous computing environment, many perspectives should be accommodated on a single machine. For example, a television can be integrated in applications such as home banking, healthcare, crime prevention, shopping and so on. Each application requires the television to provide different contexts (or computational environments) that consist of objects, functions, variables, and security policies. In Ubiscript, multiple contexts which have a unique uniform resource identifier (URI) can be defined in a runtime system. In addition, web and legacy objects are provided for interoperation with web and legacy components, and remote exception handling is provided for adapting device failures.

We implemented Ubiscript and its runtime system. A couple of ubiquitous applications were developed in Ubiscript, and the applications are tested on the runtime system. According to our experiences and evaluation, Ubiscript turned out to have a high potential in its expressive power and contribution to improving the productivity of ubiquitous application developers. The rest of the paper describes Ubiscript language, its runtime system with detailed explanation of key characteristics and the implementation of Ubiscript, and an experimental ubiquitous application.

II. Language Requirements

As the first step to design a new script language, we identified six requirements from the limitations of traditional script languages in the context of ubiquitous environments. The requirements are summarized in the below.

A. Dynamic Service Invocation

There are two ways to invoke a service in system programming languages. The first way is using stub code. To enable programmers to call a remote method in the manner as a local method, we need a stub code which encapsulates networking and marshalling tasks. In the case of a web service, we generate a stub code from web services description language (WSDL). The second way is calling a service dynamically. In order to call a service dynamically, we need to marshall parameters and unmarshall results manually. We want to achieve the advantages of both of the ways: allowing a programmer to call a service like a local call without generating and incorporating a stub code. This is applicable to legacy and web services as well as ubiquitous services. It enables a programmer to integrate many services easily and rapidly.

B. Code Mobility

Code mobility is to support one for showing a graphical user interface (GUI) in a remote device. Displaying today's morning news in a television in the above morning scenario is the case for code mobility. Implementing the GUI as a pre-deployed service is not flexible and practical because the service is better to be deployed and invoked at runtime. A script program, which dynamically builds a GUI via network at runtime, is much more convenient to provide such services.

C. Distributed Logic

Typically, a service scenario in ubiquitous environment integrates diverse forms of devices and services. So, it is necessary to specify multiple mobile and stationary codes in a single source text because we usually specify a task in a single source text in a script language in general. According to Banavar and Bernstein [6], ubiquitous applications are not regarded as pieces of software targeted to a particular device or environment but rather higher-level description of tasks a user needs to perform.

D. Security

If a distrusted program sneaks into and controls remote devices, specially the devices at your home, it may result in severe privacy and security problems. Therefore only authorized and trusted programs must be allowed to access devices that you are using for your conveniences.

E. Failure Handling

Ubiquitous environment frequently changes. A device may newly join and a device may accidentally be inaccessible. A program must be able to adapt to this changes with flexibility. Consider the situation that if you need print a document and suddenly the connected printer is in failure, but other alternative devices (e.g., fax or plotter) may be able to do the work instead. In order to adaptively cope with this failure situation, we need facilities to catch and handle the failures.

F. Concurrency

In ubiquitous environment, more than one device often works concurrently due to the nature of the task or to enhance the performance. For example, a micro-oven can scald a dish

while a television is delivering today's news. Therefore, a new script language needs language facilities to specify concurrent jobs in a single source text.

III. THE UBISCRIPT LANGUAGE

In the previous section, we described some essential features and requirements of a new script language in ubiquitous computing environments. We designed a new script language with some advanced ubiquitous programming facilities, which are missing in conventional general purpose script languages, is desirable for the support of ubiquitous application development. In this section, we describe the new language, called Ubiscript, in detail.

A. Language Overview

Simplicity is one of the most important design principles of Ubiscript language. To achieve this, we minimally add new linguistic elements to a conventional general script language in meeting the requirements for ubiquitous computing environments. Ubiscript is based on the syntax and semantics of ECMAScript [5], and it adopts mobile code technology to embody the ubiquitous computation model.

B. Remote Scope

In a programming language, the scope of a variable is the range of statements in which the variable is visible [7]. For example, the scope of a variable declared in a function body is the statements inside the function body. In traditional languages, the scope is limited in a single computing machine, but we need to expand the scope concept to the ubiquitous computing environment. We define that remote scope of a variable is the range of remote statements in which the variable is visible. Here, remote statements are the statements located in remote machines. The remote scope is a part of the range of the whole scope. In order to express ubiquitous computation naturally, we need to expand the scope of variables in remote devices, so as to access the remote variables in local statements.

Let's consider an example program in Fig. 1. Machine A has a program text to be executed, and machine B has an environment for the program text. A variable `p` in machine A refers to the environment in machine B. The `on` clause provides a block with expanded scope for each variable declared in environment `p`. This range inside of `on` block is the remote scope of the environment `p`. More specifically, the scope of all variables in the environment `p` is expanded to the range of the `on` block. Now guess, which value will be printed on machine A? The answer is `30` because the variable `y` and `z` in the machine B are visible in the `on` block and the variable `x` is visible in all the rest program text. The remote scope mechanism is implemented by mobile code technologies [8], so the statements inside `on` block is actually moved to the remote machine and then executed on the machine. We explain the details of the mechanism later.

The morning scenario described earlier can be briefly specified in Ubiscript as shown in Fig. 2. Each device, such as television, car, and PC provides an environment. To execute some

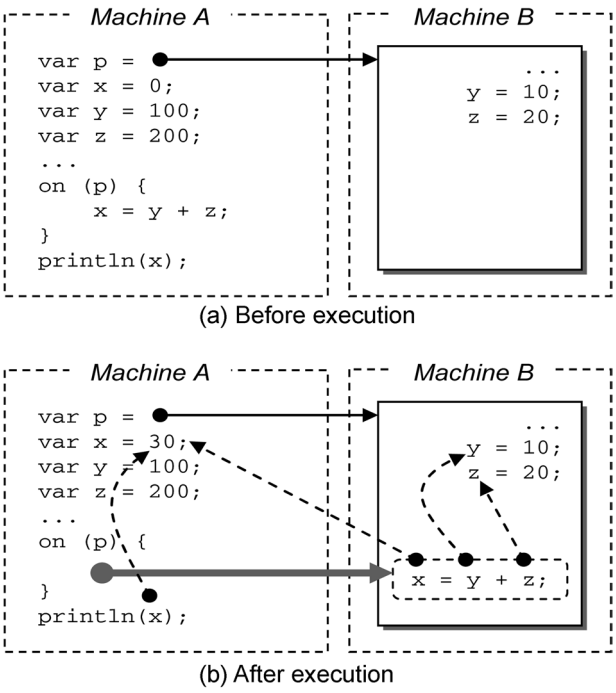

Fig. 1. An illustration of remote scope mechanism. (a) shows status of machine A and B before execution of the code, and (b) shows the status of the two machines after execution of the code.

```
var tv = new Place("http://tv.myhome.kr/gui");
var car = new Place("http://mycar.kr/ctrl");
var pc = new Place("http://pc.myoffice.kr/");
var weather =
new WebObject("http://weather.kr/ws");
var task = null;
on (tv) {
  var player = new MultimediaPlayer();
  player.setURL("mms://video.news.com");
  player.start();
}
on (pc) {
  var cal = new CalenderManager();
  task = cal.getToday().getTasks();
}
on (car) {
  if (weather.getTemperature() < 5) {
    var air = new AirConditioner();
    air.setExpectedTemperature(24);
  }
  var nav = new Navigation();
  nav.setDestination(task[0].location);
}
```

Fig. 2. Morning scenario example in Ubiscript.

statements in an environment, we need a reference to the environment. In Ubiscript everything that occupies memory is considered as object, so the reference is also represented by an object. We call it a place object. A place object can be obtained by calling Place constructor with the device's uniform resource identifier (URI).

The `on` block requires a place object where the statements inside the block to be executed. When an `on` block starts execution, the statements inside the block are encoded and transmitted to the environment pointed to by the place object, and

evaluated at runtime in the environment. To implement the remote scope mechanism, we need to address two issues. One is how to transmit the part of the code running in the runtime system to the remote device, and the other one is how to deal with the variables declared in local environment but referenced by the code running in a remote device such as the variable x in Fig. 1.

Since Ubiscript is an interpreting language, an abstract syntax tree (AST) is constructed from the program and then each node is visited and evaluated one by one by traversing the tree. On visiting an on block node in the traversing, the child nodes are transformed into the original source text and then the encoded source text is sent to the remote device at runtime. The execution is suspended until the execution of source text sent to the remote device is finished.

However there still remains a problem. Recall the example illustrated in Fig. 1. The statement "x = y + z" inside the on block is encoded and sent to machine B, and then the statement is decoded and executed in the environment p. The variable x is used in the statement but the variable with name x is not reserved in environment p. It is natural to consider that variable x refers to the one in machine A. To realize this, all the free variables of the statements in an on block should be converted to network references, and encoded together with the statements. The machine receiving the encoded codes and network references, decodes the network references for the variables that are not declared in its environment. In the case of Fig. 1, variable x, y and z are converted to network references and transmitted, but only variable x is activated in the environment p because variable y and z are declared in the environment. After the transmission phase, the transmitted statements are evaluated in the environment.

C. Multiple Contexts

The devices embedded in our life environment are not used only for a single purpose. The television can not only be used to watch TV programs, but also it can be used for security, healthcare, entertainment, etc. The display on a refrigerator can show the expiration dates of foodstuffs, and it also can be used for an announcement board for family members. That is, the utilization of a device changes according to applications in ubiquitous environment. So a device should support multiple utilization purposes. Moreover different utilization purposes may require different trust levels. The refrigerator should be protected from applications that have permissions only on showing simple information on the display, accessing to the information of the foodstuffs inside. Different security policies are needed to be flexibly specified in each specification of utilization purpose for a device.

In order to support multiple utilization purposes for a device, Ubiscript language provides facilities to support multiple contexts. A context is used to specify a utilization purpose. A context has a unique path name, its own security policy, and properties. The elements of a security policy are summarized in the below:

1) Authentication: It qualifies the applications allowed to access to the context by defining the accounts. Each account has username and password. It should be delivered to the developers of the qualified applications.

2) Authorization: It manages the objects that are allowed to access from the qualified applications. Ubiscript can load the libraries selectively which contains various objects at runtime. We can restrict the libraries that can be loaded by the applications.

3) Firewalling: It restricts accesses from external hosts according to domain patterns. This approach allows the accesses only from the designated hosts, so it can prevent access to secure devices from the applications of unidentified hosts.

```
...
<context path="/sandbox">
  <account
    username="guest"
    password="827ccb0eea8a706c"/>
  <library file="/lib/std.ubs"/>
</context>
<context path="/full">
  <account
    username="admin"
    password="c2a39d892bb71fa3"/>
  <library file="/lib/*.ubs"/>
  <property
    name="TEMP"
   value="/SYSTEM/TEMP"/>
</context>
<context path="/inform">
  <account
    username="scott"
    password="96e79218965eb72c"/>
  <library file="/lib/std.ubs"/>
  <library file="/lib/gui.ubs"/>
</context>
<context path="/vendor">
  <account
    username="vendor"
    password="c6d1a3290314b34c"/>
  <library file="/lib/std.ubs"/>
  <library file="/lib/tvlog.ubs"/>
  <domain pattern="220.69.180.*"/>
</context>
...
```

Fig. 3. An example of multiple contexts for a television.

Besides, restricting session count and logging for the transmitted execution code can augment safety of each device.

An example context definition for a television is presented in Fig. 3. While the "/sandbox" context is configured to prevent damages from any application, the "/full" context is configured to allow accession from the applications with only full privileges. Since the "/inform" context is configured to permit accession only to standard and GUI objects, it is not allowed to read information of the television, or to access to memory or file, or to make network connections. The "/vendor" context is configured for the vendor of the television, so only the applications on vendor's IP addresses are allowed.

D. Web and Legacy Objects

Many useful services published in web and UPnP services are implemented as web services based on simple object access protocol (SOAP) protocol. If one wants to call a Web Service in system programming languages such as C++ or Java, we need to generate a stub from WSDL or write a bunch of marshalling and unmarshalling code manually. In addition, there are many legacy services written in Java or ActiveX. Jini services are written in Java and most of components in Windows platform are written in COM/ActiveX. In order to support dynamic interoperation

with those services, Ubiscript provides three kinds of objects. They are `WebObject`, `JavaObject`, and `ActiveXObject`. Each object can be created by a constructor and accessed in the same way as a plain Ubiscript object.

E. Remote Exception Handling

When a particular device fails during the execution of a task, the task is usually performed by substituting an alternative device for the failed device. To support this, we provide a means to catch and handle an exception raised from a remote device. In Ubiscript, the catching and handling exceptions raised inside of mobile code to be executed in a different device is specified in the following way:

```
try {
  on (p) {
    doSomething(); // exception occur
  }
} catch (e) {
  // handle exception e.
}
```

In the above code, the statement inside on block is executed in p environment of a particular device. When an exception is raised during the execution of the doSomething() statement, the exception e is propagated to the original device via network and handled in the original device.

F. Thread Object

Ubiscript provides a basic thread object and the usage of thread object is the same as that of other thread-enabled languages such as Java. We can specify asynchronous tasks by simply mixing thread object and remote scope like the code in the below:

```
on (p) {
  var thread = new Thread(
    function() {
      // asyncronous job
    }
  );
  thread.start();
}
```

G. Design Rationale

We developed Ubiscript's key features to satisfy the requirements of script languages for developing ubiquitous applications. For example, the notion of remote scope effectively achieves the design goals like code mobility, distributed logic, failure handling, and concurrency. The remote scope mechanism enables us to specify the logics of incorporating services and programmable devices in a single program text. That is, it achieves code mobility and distributed logic requirements. In addition, we can mix the remote scope with conventional thread model and exception handling mechanism to achieve concurrency and failure handling requirements in ubiquitous environment. Due to web and legacy objects, we can dynamically access service objects without gene-rating any stub codes or writing mashalling codes manually to meet the dynamic service invocation requirements. Multiple contexts, which allow programmers to specify different security policies for contexts in various environments, meet the security requirement to some degree. Each context has its own accessible objects with a security policy pertaining to the context, and thus the objects are protected from the applications missing a permission to access to the context.

IV. IMPLEMENTATION

Ubiscript runtime system (i.e., Ubiscript interpreter) is written in Java language, so we can deploy the system on any devices as long as Java Virtual Machine (JVM) is available. Prior to the execution of a program written in Ubiscript, the runtime system should be installed on the device where the program to be executed, and the devices specified by place objects of the program. HTTP GET/POST protocol is used for the transmission of the commands and program texts, and ANTLR parser generator tool [9] is used in implementing the parser for Ubiscript language.

Meanwhile, the runtime system provides five core commands in the below:

GET_SESSION: This command requires a context path name as an input parameter. It returns a reference to a new session for a specified context.

EXECUTE_CODE: This command requires a session reference, encoded program text to be executed, and encoded network variables for input parameters. It returns the result of the execution of the program text. If an exception is raised, the exception is propagated to the sender at runtime.

GET_VALUE: This command requires a session reference and a network variable to be read. It returns the value of the network variable.

PUT_VALUE: This command requires a session reference and a network variable to be modified. It returns a result of modification of the variable. If some problem occurred to modify the variable, it propagates an exception.

PROCEDURE_CALL: This command requires a session reference, a reference to a function, and marshaled arguments. It calls the function with the arguments, and returns the result. If an exception is raised during the function call, the exception is propagated to the caller runtime system.

In order to execute a program text at runtime, at first, the runtime system invokes GET_SESSION command to get a reference to a session, and EXECUTE_CODE command is applied to the program text with the reference of the session. Then, the runtime system parses the program text and constructs an AST, and executes nodes in the AST while traversing the AST. On visiting an on block in the traversing, call GET_SESSION and EXECUTE_CODE command to execute the statements in the block on the remote machine specified in the place object of the on block. The runtime system restores the network variables on the session's environment, parses and executes the program text received through EXECUTE_CODE command. During the program execution three kinds of cases may occur. The first case is reading values of network variables. GET_VALUE command is used to read the network variables. The second case is writing values to network variables. PUT_VALUE command is used to write values the network variables. The last case is calling functions referenced by network variables. PROCEDURE_CALL command is used to call remote functions on the runtime system with arguments to be passed, and to receive the result of the function call and continue the execution.

V. EXAMPLE APPLICATIONS

A. UbiCafe

When a customer carrying a mobile device (smart phone, personal digital assistant [PDA], or laptop, etc.) enters a self-service cafe, an application for the cafe is automatically sent to and executed on the customer's mobile device. The application shows a menu for coffee and waits for orders. The barista of the café can check the orders through a desktop computer in the café, and can notify to the customers when the order is ready. The notification dialog is pop up on the customer's mobile device, and then the customer picks up his/her coffee.

We implemented the UbiCafe application described above by using Ubiscript language on the runtime system. Firstly, we adopted radio-frequency identification (RFID) technology to recognize when the customer's mobile device enters into a cafe. RFID tags are attached to the tables in the café and the RFID reader is linked to the customer's mobile device. The customer's mobile device prototype and graphical user interface are presented in Fig. 4. The UbiCafe application is executed on the mobile device when a customer brings the RFID reader, connected to the mobile device, close to the RFID tags. Since memory resources of the tag is limited to store the UbiCafe application, the application is placed in a separated application server and only the unique ID of the application is stored in the tag. The mobile device reads the application ID from the tag and connects to the application server for the download of UbiCafe application to be executed on it. The graphical user interface of UbiCafe application is fully written in Ubiscript. Without Ubiscript, we have to install a particular mobile agent middleware to a mobile device and implement the UbiCafe application as a mobile agent which is capable for presenting graphical user interfaces and interoperating with web services. That is not easy work, but we have done it simply in a single source text by a short scripting.

B. Smart Presenter

When somebody wants to make a presentation, either the notebook of the person must be connected to a projector by a cable or the presentation file must be stored into the computer to which the projector is already connected. Downloading the file from the Internet could be another way for this. Now suppose that there are many presenters, it is very inconvenient and cumbersome. Smart Presenter provides several functionalities for this situation. The first function is to enable presentation

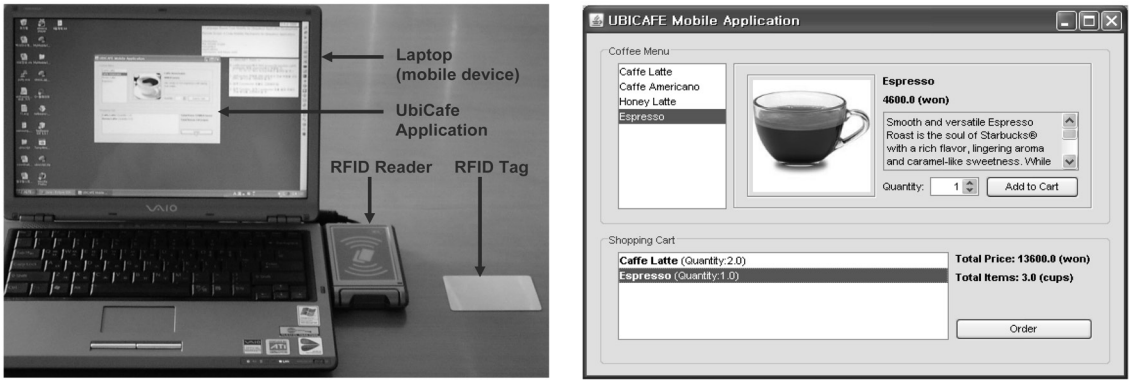

Fig. 4. UbiCafe application. The left picture is hardware prototype of customer's mobile device for UbiCafe application, and the right snapshot is a graphical user interface for a customer.

Fig. 5. Snapshots of Smart Presenter implemented on a smart room. The left picture shows a presentation situation and the right picture shows a situation that a presenter approximate the whiteboard.

without cable connection, file copy, or even file download. The second function is that it automatically adjusts brightness of the light in the meeting room. When the presentation is started, the illumination of the room is decreased and when the presentation is finished, the illumination is increased automatically. The final function is detecting the presenter's location so that the light for a whiteboard automatically turned-on when the presenter approaches the whiteboard.

We implemented this scenario in an experimental smart room which has various sensors, devices and a context server as presented in Fig. 5. The context server collects context information from installed various sensors. When a particular event or situation is detected based on context information, the context server executes a specific task such as turning-on the lights or decreasing volume of the television. For this context-awareness capability, we adopted a context-aware middleware Active Surroundings [10]. In order to implement Smart Presenter scenario on an existing context-server, we identified two major situations that can be detected by sensors in the room. Those are when a presentation is started and when a presenter approaches the whiteboard. When a presenter started a presentation by using his or her laptop, the presentation slides should be casted on the screen through the projector in the room and the lights on room should be turned off. When a presenter approaches the whiteboard during a presentation, the light on the top of whiteboard area should be turned on.

We implemented the task part of the two situations in Ubiscript. The situations are decided by the context-aware middleware, and then the middleware triggers the task scripts. The task script for the first situation creates a Power Point object in the PC connected to the projector and then opens a presentation file in the presenter's remote laptop and invokes a Web Service to turn off the lights in the room.

VI. RELATED WORK

Some languages have been developed for ubiquitous computing or applications. Indus [11] is an object-oriented language for ubiquitous computing. It provides concurrently executing agents and coordination to support development of ubiquitous applications. Indus tries to address similar issues of this paper, but it differs from our work in the following aspects. Indus is developed based on agent and coordination models so a developer has to learn about the models of agent and coordination, while Ubiscript doesn't require any agent and coordination models so a developer can learn faster and easier. Note that Ubiscript is a script language. Moreover, Indus compiles to native code at compile-time, on the other hand Ubiscript interprets code at runtime. In that sense, Ubiscript is competitive in rapid development situations against Indus. There are languages dealing with different issues from those of Ubiscript. ContextL [12], context-oriented programming (COP) [13] and context-aware aspect [14] propose languages or language features for context-oriented programming. They developed mechanisms to separate context-dependent codes from a program. By doing so, a developer can easily add and manage codes for new context and context-awareness. Since the issues they are dealing with are completely different from ours, our work can be compensation for theirs and vice versa.

Remote scope is one of the key notions and mechanisms developed for Ubiscript language. Remote scope is based on mobile code technology [8] in which a program code is sent to across a network and executed on remote machines. Telescript [15] and AgentTCL [16] are good cases supporting code mobility. The mobile code is natural to ubiquitous computing that requires dynamic discovery and integration of distributed devices. Mobile agent is a kind of the mobile code technology and there are several works [17, 18] adopting mobile code for ubiquitous environment. However, Ubiscript takes a language-based approach for code mobility. There are two similar approaches to the mechanism of remote scope. Remote evaluation (REV) [19] supports the evaluation of a procedure in remote machine by remote evaluation. For this, the procedure to be evaluated in a remote machine is transmitted to the remote machine with arguments via network. Then the procedure is evaluated and the result is sent back. If the procedure has free variables, the free variables are provided in the remote machine or those are eliminated by expanding code portion to be transmitted. The second one is distributed scope of Obliq [20] language. Obliq provides a traditional remote procedure call model except that a procedure can be passed as an argument of a remote procedure to support mobile computation [21]. The arguments for the migrated procedure are provided inside of the remote procedure. If the procedure has free variables, the variables are converted to network references and transmitted with the procedure. It is called as closure mobility. In the Ubiscript, a block is transmitted rather than a procedure and the free variables are bound through a lookup-chain at runtime. We call it lookup-chain mobility in which we try to find the free variables in the remote machine first and then find in the original. Remote scope mechanism is quite useful in ubiquitous environment because multiple programmable devices can be easily coordinated and some resources can be shared among them through remote scope. In REV, it is difficult to share a resource among mobile codes because arguments for a procedure to be transmitted are also encoded to be transmitted to a remote machine. In Obliq, it is difficult to write multiple mobile codes in a single source text because arguments for a

Table 1. A comparison on code mobility mechanisms of Obliq, remote evaluation (REV), and Ubiscript

	Ubiscript	REV	Obliq
Terminology	Remote scope	Remote evaluation	Distributed scope
Unit of mobility	Block	Procedure	Procedure
Free variable binding policy	Lookup-chain mobility	Closed-code mobility	Closure mobility
Contents to be transmitted	Block + lookup-chain	Code portion + arguments	Closure (procedure + references to free variables)

procedure to be transmitted should be provided in a remote machine manually. A comparison on mobile code mechanisms are summarized in Table 1.

VII. DISCUSSION

Ubiscript is not a full-fledged language to cover all the requirements for ubiquitous applications but an experimental language to test the effectiveness of the new language features proposed in the language. Firstly, Ubiscript is useful in scripting actions associated with contexts in context-aware applications as illustrated in the example of Smart Presenter. Although a context-aware middleware is capable of collecting context information and reasoning specific situations, but an action triggered by the change of context should be written in a specific programming language. Ubiscript is appropriate in specifying actions associated with the change of context because the actions are mostly specified by interoperating and coordinating various services and devices. Ubiscript is also appropriate for the control of programmable devices. Simple devices such as a light, a clock, a door-lock can be controlled by calling procedures. However complex devices such as a television, a navigation system, a refrigerator require a separate and well-equipped control language. Since the devices usually have own processor and network capability, Ubiscript interpreter can be embedded in the device.

VIII. CONCLUSION

Ubiquitous applications integrate services on distributed and heterogeneous devices in ubiquitous computing environments. However conventional general purpose script languages have limitations to support various kinds of interoperation. In this paper, we have proposed a script language Ubiscript suitable for ubiquitous environment. It incorporates several language features such as remote scope, multiple contexts, remote exception handling, concurrency to overcome the limitations of current conventional script languages such as Tcl, Perl, Javascript, in the context of ubiquitous computing. By implementing experimental ubiquitous applications UbiCafe and Smart Presenter, we confirmed that the language dramatically reduces development time and efforts, so we can conclude it is much more suitable for ubiquitous environments than conventional script languages and system programming languages.

REFERENCES

1. M. Weiser, "The computer for the 21st century," *Scientific American*, vol. 265, no. 3, pp. 94-104, Sep. 1991.
2. B. A. Miller, T. Nixon, C. Tai, and M. D. Wood, "Home networking with universal plug and play," *IEEE Communications Magazine*, vol. 39, no. 12, pp. 104-109, Dec. 2001.
3. K. Arnold, The Jini Specification, Reading, MA: Addison Wesley, 1999.
4. E. Guttman, "Service location protocol: automatic discovery of IP network services," *IEEE Internet Computing*, vol. 3, no. 4, pp. 71-80, Jul./Aug. 1999.
5. ECMA General Assembly, ECMAScript Language Specification, 3rd ed., Standard ECMA-262, 1999.
6. G. Banavar and A. Bernstein, "Software infrastructure and design challenges for ubiquitous computing applications," *Communications of the ACM*, vol. 45, no. 12, pp. 92-96, Dec. 2002.
7. D. A. Watt and W. Findlay, Programming Language Design Concepts, Hoboken, NJ: John Wiley & Sons, 2004.
8. A. Fuggetta, G. P. Picco, and G. Vigna, "Understanding code mobility," *IEEE Transactions on Software Engineering*, vol. 24, no. 5, pp. 342-361, May. 1998.
9. T. Parr, The Definitive ANTLR Reference: Building Domain-Specific Languages, Raleigh, NC: Pragmatic Bookshelf, 2007.
10. D. Lee, "Active surroundings: a group-aware middleware for embedded application systems," *Proceedings of the 28th Annual International Computer Software and Applications Conference*, Hong Kong, China, 2004, pp. 404-405.
11. K. Borah, "Indus: an object oriented language for Ubiquitous computing," *ACM SIGPLAN Notices*, vol. 41, no. 2, pp. 18-24, Feb. 2006.
12. P. Costanza and R. Hirschfeld, "Language constructs for context-oriented programming: an overview of ContextL," *Proceedings of the Dynamic Languages Symposium*, San Diego, CA, 2005.
13. R. Keays and A. Rakotonirainy, "Context-oriented programming," *Proceedings of the 3rd ACM International Workshop on Data Engineering for Wireless and Mobile Access*, San Diego, CA, 2003, pp. 9-16.
14. E. Tanter, K. Gybels, M. Denker, and A. Bergel, "Context-aware aspects," *Software Composition, Lecture Notes in Computer Science*, vol. 4089, Heidelberg: Springer Berlin, pp. 227-242, 2006.
15. J. E. White, Telescript Technology: The Foundation for the Electronic Marketplace, White Paper, Sunnyvale, CA: General Magic Inc., 1994.
16. D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawaa, and G. Cybenko, "Agent TCL: targeting the needs of mobile computers," *IEEE Internet Computing*, vol. 1, no. 4, pp. 58-67, Jul./Aug. 1997.
17. N. Hanssens, A. Kulkarni, R. Tuchida, and T. Horton, "Building agent-based intelligent workspace," *Agents for Business Automation (ABA) Conference Proceedings*, Las Vegas, NV, 2002, pp. 675-681.
18. K. Kangas and J. Roning, "Using mobile code for service integration in ubiquitous computing," *Proceedings of the 5th Mobile Object Systems Workshop*, Lisbon, Portugal, 1999.
19. J. W. Stamos and D. K. Gifford, "Implementing remote evaluation," *IEEE Transactions on Software Engineering*, vol. 16, no. 7, pp. 710-722, Jul. 1990.
20. L. Cardelli, "Language with distributed scope," *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, CA, 1995, pp. 286-297.
21. L. Cardelli, "Mobile omputation," *Mobile Object Systems Towards the Programmable Internet, Lecture Notes in Computer Science*, vol. 1222, Heidelberg: Springer Berlin, pp. 1-6, 1997.

Minkyu Lee

Minkyu Lee is a PhD student in computer science at the Korea Advanced Institute of Science and Technology (KAIST). His research interests include location-based services, collective intelligence, and context-aware computing. He received his master's degree in information and communications engineering from the KAIST. Contact him at niklaus@kaist.ac.kr.

Dongsoo Han

Dongsoo Han is a professor in the Department of Computer Science at the KAIST. His research interests include mobile computing, bioinformatics, and healthcare services. He received the PhD degree in the information science from the Kyoto University in 1996. Contact him at dshan@kaist.ac.kr.