# Using JIT Compilation and Configurable Runtime Systems for Efficient Deployment of Java Programs on Ubiquitous Devices

Radu Teodorescu and Raju Pandey

Parallel and Distributed Computing Laboratory
Computer Science Department
University of California, Davis
{teodores,pandey}@cs.ucdavis.edu

**Abstract.** As the proliferation of ubiquitous devices moves computation away from the conventional desktop computer boundaries, distributed systems design is being exposed to new challenges. A distributed system supporting a ubiquitous computing application must deal with a wider spectrum of hardware architectures featuring structural and functional differences, and resources limitations. Due to its architecture independent infrastructure and object-oriented programming model, the Java programming environment can support flexible solutions for addressing the diversity among these devices. Unfortunately, Java solutions are often associated with high costs in terms of resource consumption, which limits the range of devices that can benefit from this approach. In this paper, we present an architecture that deals with the cost and complexity of running Java programs by partitioning the process of Java program execution between system nodes and remote devices. The system nodes prepare a Java application for execution on a remote device by generating device-specific native code and application-specific runtime system on the fly. The resulting infrastructure provides the flexibility of a high-level programming model and the architecture independence specific to Java. At the same time the amount of resources consumed by an application on the targeted device are comparable to that of a native implementation.

## 1 Introduction

As a result of the proliferation of computation into the physical world, ubiquitous computing [1] has become an important research topic. The goal of ubiquitous computing is to bridge cyberspace and physical space. The Internet provides a uniform mechanism for accessing and manipulating information stored on conventional computer systems. Until recently, the Internet was accessible to common users primarily through desktop computers. Now several devices (including cell phones, PDAs and messaging devices) provide users some access to the Internet.

G. D. Abowd, B. Brumitt, S. A. N. Shafer (Eds.): Ubicomp 2001, LNCS 2201, pp. 76-95, 2001.

Ubiquitous computing research envisions extending this accessibility to all ubiquitous devices in a uniform and transparent manner. Thus, in the future an Internet user could check in real-time the weather conditions provided by a network of intelligent sensors distributed worldwide and connected to the Internet; the processors embedded in a car would be enabled to send a notification when the car needs an oil change; or one could check from his workplace whether all the dangerous electrical appliances are turned off at their home. However, in order realize this vision, a software infrastructure must exist that integrates ubiquitous devices into a general distributed system framework, and allow one to interact with and manipulate them in a transparent manner.

Ubiquitous computing systems introduce a new class of hardware devices that may no longer be compatible with the traditional desktop model. Thus, in addition to the traditional problems raised by a distributed system, system designers must take the following characteristics of ubiquitous devices into account:

- **Functional heterogeneity**. In traditional systems the interoperability problem primarily relates to the differences in specific implementation choices (for instance, differences in instruction set, operating system and data representation). In the ubiquitous computing environment, on the other hand, since the nodes implement different computational behaviors, they introduce a new type of heterogeneity, which we call *functional heterogeneity*. Thus, a software infrastructure must not only deal with *how* a node provides certain functionality but *whether or not* it provides a specific functionality.
- **Resource limitations**. Ubiquitous computing devices also have limited computing capabilities due to the concerns about power, size and price. Thus, the footprint of a traditional operating system or language runtime system may be too large for these devices.

Thus, software infrastructure solutions must implement a more complex interoperability problem, while meeting stricter resource limitations. Overcoming the heterogeneity problem requires considerable computation overhead even in traditional distributed systems. Since functional heterogeneity includes the traditional heterogeneity problems and adds new ones, it is expected that the mediation overhead would accordingly be higher. On the other hand, the resource limitations restrict the amount of computation that can be dedicated to solving the heterogeneity problem. Traditional middleware solutions (such as MPI, CORBA, and RM) typically have heavyweight implementations. Each device integrated in such a system must be able to support a substantial runtime system overhead. This limits [28] the approach to only small number of devices[30,31,32].

The primary goal of our research is to make integration of ubiquitous devices within an overall distributed system framework as transparent as possible. This means that end users should be able to continue to use high level programming languages and tools for developing distributed programs. Further, their ability to deploy distributed programs on these devices should not be constrained by the resource limitations of the devices. In this paper, we present an infrastructure, called *JUCE* (*Java for Ubiquitous Computing Environments,* that addresses these concerns. JUCE supports the Java programming environment for developing distributed programs. It addresses the functional heterogeneity problem by transparently translating Java

program components into device-specific programs, and migrating them to the devices. JUCE addresses the resource limitations constraints by dynamically constructing an execution environment that is customized to include only those runtime services needed for an execution of an application on a ubiquitous device.

The JUCE architecture restructures the traditional JVM by introducing two new concepts: **Remote Just-In-Time compiler (R-JIT)** and **Configurable Runtime System (C-RTS)**. R-JIT exploits the fact that much of the overhead associated with Java byte-code processing (for instance, code verification or just in time compiling) can be relocated to a remote host from the device executing the code. Thus, the code targeting an embedded device is compiled just in time on a remote host and migrated to the device in the native code format. This means that a resource-limited device can efficiently execute a Java application while a more powerful node supports the actual Java-specific overhead. The C-RTS has a modular structure that allows JUCE to adapt its configuration according to the resources availability and application requirements. Thus, in JUCE, the different RTS modules are loaded as required by the application. In the traditional approach the Java VM is entirely loaded before starting the execution of the application.. This way a device running a Java application has to deal only with the overhead produced by the RTS services currently required by the application.

We have implemented the JUCE infrastructure and a Linux-based application that emulates ubiquitous devices. We have run several experiments that analyze the overall behavior of the JUCE infrastructure. The experiments show that the infrastructure provides the flexibility of a high-level programming language. At the same time the resource consumed by an application on a targeted device is comparable to that of a native implementation. Further, the resources required by runtime services can be scaled up or down dynamically according to the needs of applications.

This paper is organized as follows: In Section 2 we describe the JUCE architecture in detail. In Section 3, we present quantitative and qualitative analysis of the architecture. Section 4 presents a brief survey of the related work. Finally, we conclude with a brief summary and a description of the future work in Section 5.

## 2 System Architecture

We now describe the JUCE system architecture. First we provide an overall view of the architecture. We then describe a configurable runtime system that spans across hosts and embedded devices.

### 2.1 Overview

The goal of this research is to integrate physically distributed embedded devices within a distributed computing framework that includes general-purpose hosts. Thus, we assume a generic distributed system architecture that includes conventional hosts, including desktop computers and server machines, and ubiquitous mobile and embedded devices. We also assume that the hosts and the devices can communicate

with each other through various network infrastructures. JUCE exploits the additional processing power available at general-purpose hosts to compensate for the limitations of embedded devices.

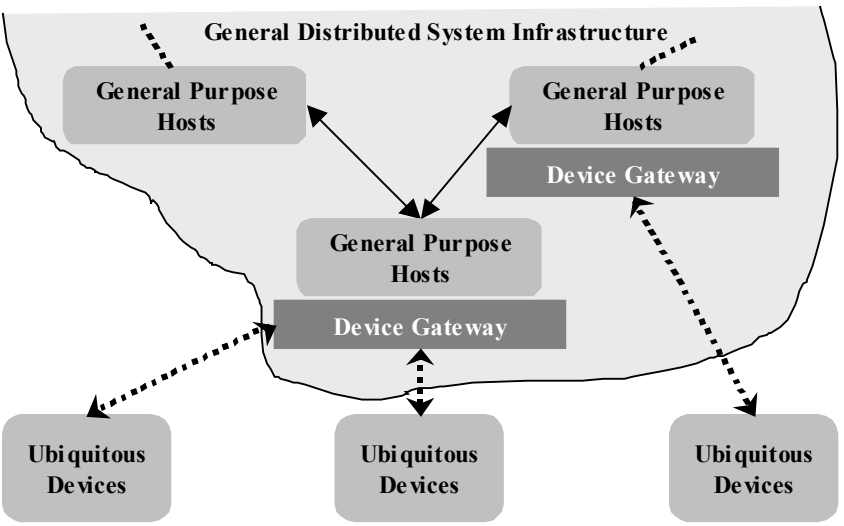


**Fig. 1.** Distributed System Architecture with Gateways

The JUCE programming environment aims to hide the heterogeneity among the general purpose systems and embedded devices by a single distributed programming model, based on Java and MAGE [9]. Users write programs in Java and use high-level Java-based APIs for accessing services provided by specific devices. As these programs are run, the JUCE runtime environment manages the execution of programs on hosts and devices. As shown in Figure 1, the JUCE runtime environment is divided into two kinds of computational hosts: hosts that run the JVM (called *JUCE gateways*), and *embedded devices*[1]. A JUCE gateway adapts application and runtime system code (represented as Java byte code) for specific devices. It first intercepts Java byte code intended for an embedded device. It then arranges for generation of code for the specific device. It also determines a runtime system configuration that is needed for an execution of the code. It then migrates both the runtime system and the application code to the specific device. During the execution of the program on the remote device, the JUCE runtime environment also interacts with the device and sends necessary supporting runtime routines and program objects. It is also the gateway's role to capture any abnormal behavior at the device and formulate it in terms of Java exceptions. The embedded devices implement a small runtime environment that evolves to adapt to the runtime requirements of a program.

Next we describe the JUCE gateway and the runtime in detail.

[1]We use the term embedded device to refer to hosts that cannot run a full JVM due to resource constraints

### 2.2 JUCE Gateway Implementation

We now look at the details of the JUCE gateway. There are two components of the JUCE infrastructure: **Remote Just-in Time** (R-JIT) compiler and **Configurable Runtime System** (C-RTS). The R-JIT compiler translates application Java byte code into device-specific native code. The resulting native code application can be executed directly on the embedded device. The C-RTS provides high level services for the native code (such as dynamic linking, virtual method invocation, memory allocation, exception handling and synchronization).

### 2.3 Remote JIT Compiler

The RJIT compiler converts Java byte-code into native code, which is transmitted to the embedded device. The RJIT compiler performs the required code verification to guarantee that the generated code preserves Java safety and security properties. We have implemented the remote JIT compiler by modifying the Open JIT compiler [9]. The OpenJIT compiler is a reflective JIT compiler. It provides a clean separation between the Java front end and the back-end target machine. The JIT compiler can be easily extended to a new device by implementing a corresponding back-end. The R-JIT compiler is extensible as well in that a gateway can be transparently extended to control a new class of embedded devices by dynamically downloading a new backend. The JUCE infrastructure can also be similarly extended dynamically over newly discovered devices.

### 2.4 The Configurable Runtime System (C-RTS)

The C-RTS supports a set of libraries that implement several services for the compiled code. For scalability purposes, the C-RTS has a component-based structure. Each component corresponds to a specific runtime service provided by the Java Virtual Machine. The C-RTS is partitioned between a gateway and an embedded device. This partitioning is dynamic, and depends on the nature of the device, resource availability, frequency of usage and efficiency. Heavyweight RTS components that are seldom required are placed on the gateway machine, whereas frequently utilized RTS components are placed on the embedded device. The partitioning of the virtual machine between the gateway and device varies according to the application requirements and underlying device configuration. The JUCE architecture divides the overall C-RTS components into three separate modules: **micro-kernel**, **Gateway Resident RTS (GR-RTS)**, and **Device Resident RTS (DR-RTS)**. The goal of this partitioning is to reduce the amount of processing done at the embedded device task by pushing the resource-expensive computation on the gateway. The division is based on the observation that a large portion of the Java RTS functionality (such as class initialization or dynamic linking) is used sparingly during a program execution (for instance only for class initialization). We described each module in detail below:

The primary component of the C-RTS is a **micro-kernel**. The micro kernel forms the basis for dynamically bootstrapping the rest of the runtime system. A device, thus, must be able to host a micro-kernel in order to be part of the overall distributed

runtime infrastructure. The micro-kernel (**μK**) implements the ability to connect with a gateway through a communication medium, download a block of code from the gateway, execute the downloaded code locally, and return the result back to the gateway. Note that while code mobility has been traditionally handled as a high level feature, it is a fundamental element in the JUCE architecture.

The micro-kernel supports a limited subset of Java. This subset includes primitive data types (such as **int, char** and **float**)[2], control flow constructs (such as **if, while** and **for**) and *static native* methods. In the absence of a real Java VM, the micro-kernel only supports the execution of a single static method. It does not support nested method invocations or structured data types.

The programming model supported by the micro-kernel is used as the starting point for implementing the actual runtime system. From the micro-kernel perspective, the rest of the C-RTS is just another migrating application that needs to be executed. Thus, whenever the micro-kernel programming model is not expressive enough for an application, the required components of the rest of the C-RTS are migrated to the embedded device in order to provide the required mediation layer.

**GR-RTS** stores the *constant pool table* associated with a Java program. The Java runtime system uses the constant pool for dynamic runtime linking. The table contains a set of symbolic references that are resolved during program execution. As the Java interpreter comes across an unresolved symbol, it uses the dynamic class loader to map the symbol to a specific program entity. It then replaces all occurrences of that symbol with its direct reference. Accordingly, the application running on the embedded device generates a remote invocation on the gateway the first time it comes across a symbol**.** The gateway resolves the symbol for the device, and downloads the native code on the device. The device can then continue to execute the method. Thus, after each symbol has been resolved at least once, a program can run autonomously on the embedded device without requiring any additional information from the gateway.

**DR-RTS** manages class information and the code required by the embedded device. Unlike the traditional Java approach, where code migration is done at the class level, JUCE uses methods as the basic code migration unit. This allows the gateway to export only those methods that are currently required, thereby reducing the memory usage on the embedded device. Clearly, there is a tradeoff between memory space and how long it takes to download different components. The gateway can provide a balance between the two through the knowledge of the resource availability on a remote device.

In order to further minimize the overhead on the embedded device, the Device-resident RTS is split into a Dynamic-RTS (D-RTS) and a Static-RTS (S-RTS). The Static-RTS represents the code that is permanently loaded on the embedded device while the Dynamic-RTS is written in Java and is loaded from the gateway during execution according to application needs. Thus, each component of the DR-RTS is divided into a dynamic component and a static component. A resource-thrifty design

---

[2] The number of supported primitive types reflects the types supported by the underlying hardware while the rest of the Java types are implemented on top of them by the RTS components. For instance, the OpenJIT X86 implementation invokes JVM for operations with **double types.**

of the local RTS components places all the implementation logic in the dynamic side such that the static side code is reduced to a minimum set of low-level primitives (for instance, direct memory access, direct access to communication I/O ports, and interrupts handlers).

S-RTS mediates between the D-RTS and the device. The JIT compiled application, running on the device, makes calls to a standard set of runtime routines. These routines correspond to the services performed by a JVM (such as method resolution and invocation, memory allocation and exception handling). A typical lightweight S-RTS implementation redirects these calls to the corresponding D-RTS component.

The concrete implementation of the S-RTS varies according to the underlying architecture. The set of primitives provided by the S-RTS also differs between different devices down to the extreme cases in which no primitives are implemented. Of course, the availability of the high-level features provided by the D-RTS is limited by the set of primitives present on each machine. For instance, if the S-RTS does not provide mechanisms for saving and restoring context information associated with an execution, it will be difficult for the D-RTS to implement multithreading support. Thus, the resources available on a device inherently limit the complexity of an application that can be executed on the device.

In order to maximize the range of applications that can be executed on an underlying device, the S-RTS must support suitable programming features for machine deployment and must be lightweight at the same time. Because a solution to the S-RTS design problem is particular to each type of device, the JUCE architecture does not impose a standard design. However, we believe that a static RTS should provide several services: The first is the ability to provide a memory model for storage and access. A simple approach is to use Java byte arrays to model memory. The S-RTS should also provide access to the underlying device registers, which can be used to control low level runtime structures (such as system stack). Since applications must communicate with the gateway, the S-RTS also needs to provide control over I/O ports for communication with the gateway. In addition these low level primitives, our implementation of S-RTS includes several high level services, including support for method retrieval from a gateway, method invocation, and code patching to allow an application to change the method code to point to the actual symbol's value after symbol resolution.

Note that many of the primitives can be written in Java using the low level primitives. For instance, *method invocation* can be expressed as a sequence of register operations. However, such an implementation tends to be considerably more inefficient than a corresponding native one. Further, given the frequent usage of these methods, the native approach seems more appealing. Also, since these services are likely to be required by most applications, there is no real benefit in having them implemented as dynamic components.

The **Dynamic Runtime System (D-RTS)** consists of a set of components corresponding to the various services provided by the RTS (such as object creation, method invocation, symbols resolution and exception handling). Each dynamic component is written in Java and is loaded on the embedded device through the R-JIT compiler in the same way as other application components. The implementation of a dynamic component is defined by establishing an *implementation hierarchy* among the components. Thus, each component implementation can utilize the RTS services

provided by the D-RTS components at a lower level. The implementation of the components at the lowest level of the hierarchy deals only with abstractions implemented by the S-RTS and micro kernel. The higher-level components provide increasingly more complex features such as the notion of objects, object creation, exception handling, garbage collection and synchronization. Figure 2 shows the dependencies among several RTS components. The *method invocation* component is used to retrieve a specific method from the gateway and execute it locally. The *symbol resolution* function performs the conversion between symbolic references used in the compiled code and real references. *Virtual method invocation* component first identifies the actual method that needs to be invoked (using the *symbol resolution* service) and then invokes it using the *method invocation* service. Creation of an object takes place in two steps: first the RTS allocates memory for the object. It then invokes the constructor. The allocation may require *symbol resolution* in order to find information about the class such as object size. The second step is handled implicitly as a virtual method invocation.

The Java classes implementing the dynamic components are stored in a *Dynamic Runtime System Repository* either locally or on a remote machine, and can be accessed by the gateway. This architecture allows us to efficiently upgrade RTS components. In addition, the approach allows alternate implementations of the components for different devices. For instance, we can implement device-specific memory allocators and garbage collectors that exploit the properties of the underlying device to provide a more robust, flexible and efficient implementation

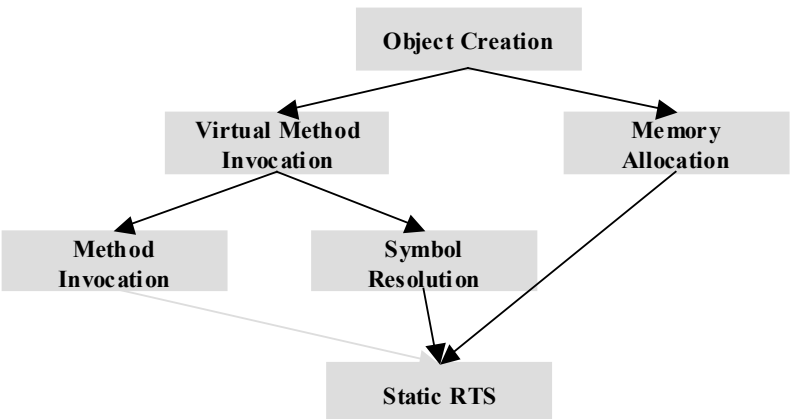


**Fig. 2.** Partitioning of Dynamic RTS

## 2.5 Device API

While the JUCE gateway and the C-RTS provide general purpose Java support, a device also exports an API that enables applications to access device-specific services. The JUCE device API implementation follows the same design principles as

the Device-resident RTS. It is split into two: a low level static (Native) API hardcoded for the device, and a dynamic API written in Java that can be loaded on demand. Figure 3 summarizes the overall JUCE architecture:

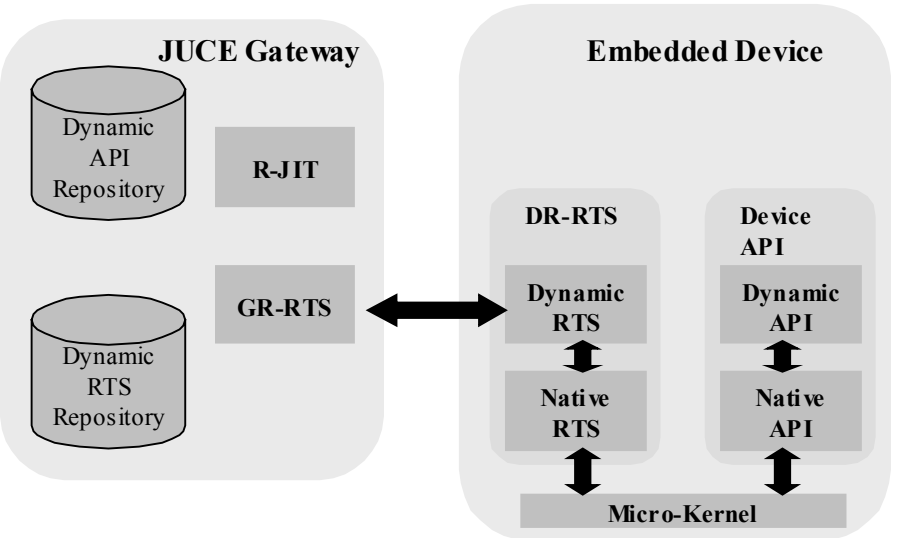


**Fig. 3.** Overall architecture of JUCE infrastructure

## 3 Analysis

In this section, we analyze the characteristics of the JUCE architecture. We first experimentally evaluate how the JUCE architecture can dynamically configure both applications and the runtime components in order to meet the resource limitations of an embedded device. We then examine the security implications of the proposed architecture. Finally we discuss the autonomy of devices with respect to JUCE gateways.

### 3.1 Experimental Analysis

The central goal of the JUCE architecture is to facilitate each application to dynamically choose the right balance between the supported features and resources requirements. This means that an application should incur runtime system overhead for only those features that it uses. Also, it should be able to adapt its behavior according to runtime resources availability.

In this section, we describe three experiments that highlight the tradeoffs among resources availability, performance and features at three different levels. We have used memory as the primary resource benchmark and the basis for adaptation in the experiments. In the first experiment, we have implemented an application that requires very little resources, and can be implemented by the micro-kernel directly. The second experiment outlines the way in which specific components of the runtime

services are incrementally migrated to a remote device based on application needs and the cost of each individual service. The final experiment highlights how an application can dynamically partition itself between a gateway and a remote device in response to resource constraints at a device.

### 3.1.1 Experimental Methodology

The experiments were run on an embedded device emulator. The emulator is written in C, and runs on Linux. It provides the default functionality of the JUCE micro kernel in the context of an X86 architecture. It listens on a TCP/IP port for incoming JUCE code packages and executes them as they are received. The emulator can be configured to support different sets of device APIs, which are implemented as Linux dynamic link libraries. The libraries are used to emulate the static API and the static runtime system.

The emulator has limitations in that it cannot be used to measure the size of the micro-kernel or the system implementing the static API. The micro-kerne uses heavy weight Linux libraries for implementing its functionalities. For instance, the communication between gateways and embedded devices is currently implemented using TCP/IP. Thus, the sizes of the micro-kernel and the static API do not truly reflect their actual sizes on an embedded device where they can be customized. We plan to port the micro kernel and static API to a real device in near future.

### 3.1.2 Remote Temperature Sensing Device

In this experiment, we simulate an application running on a device with extremely limited resources, and analyze the minimal set of runtime services required for such applications.

We consider an embedded device that represents an intelligent thermometer. This device represents a base deployment scenario. The thermometer implements the JUCE micro-kernel. It also provides a native API that includes three methods: `network_read` to read from the network, `network_write` to write to the network, and `temperature_read` to read the temperature. The remote thermometer executes a program that continuously reads the temperature and sends the average value to the gateway device every 1000 cycles. Also the device sends an alarm if the temperature goes above a threshold.

The size of the code generated by the R-JIT compiler is 204 bytes. Since the application uses only primitive data types and only invokes static native methods, it does not require any runtime support beside what is already provided by the micro-kernel. Given that the application data consists of three local variables and a function argument, the entire application consumes a total of about 220 bytes.

Note that the micro kernel only supports primitive data types and static methods. The programmer, thus, has access to a non object-oriented subset of Java. This limits the kind of applications that can be easily written and targeted for such devices. However, the programming model may be sufficient for devices that support extremely limited amount of resources. Note that the JUCE approach represents a step ahead from the traditional approaches, which typically involves implementing these applications in assembly language. If the embedded device application is part of a larger distributed application, it is easier for the developer to deal with the same

programming language instead of having separate modules written in different languages.

### 3.1.3 Controlling Device Grids

In the second experiment, we outline the behavior of the dynamic runtime system and the usability of the available runtime services. The experiments show how each service affects the cost of application deployment.

The experiment simulates a grid of devices that monitor the integrity of a building structure. The devices are distributed over the surface of the structure. There are three kinds of devices: temperature sensors, vibration sensors and monitors. The temperature and vibration sensors monitor the current sensor readings and send periodic updates to the gateway. They also exchange the readings between them. Each such device notifies the closest *monitor* device when abnormal conditions are detected. Monitors primarily sense the behavior of the overall surface and react when the combination of temperatures and vibration levels crosses a certain acceptable level. The devices are equipped with low power communication antennas that allow them to communicate with their direct neighbors. For the sake of simplicity, the devices are placed in one-dimensional space (See Figure 4.).

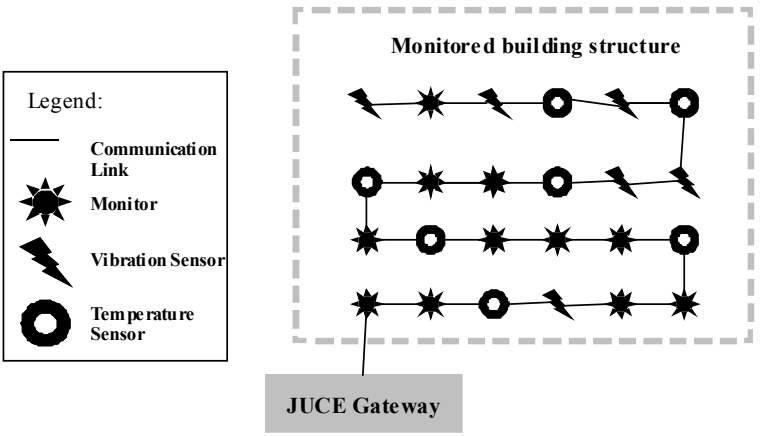


**Fig. 4.** Sensor Grid

The experiment involves using the gateway to control the grid of devices. The devices in the grid have limited resources. In addition, they provide a limited set of primitives that can be used to build applications. There is no predefined distribution of the devices. Thus, a device does not have any local information about distribution of the rest of devices. To overcome this problem, the device broadcasts a request identifying the type of the device it wants to interact with. The request is propagated until it reaches the closest requested device. The available device handles the request and sends information back to the requesting device. The devices, thus, implement a tunneling protocol to communicate with each other and with the gateway.

Each device runs a JUCE micro-kernel. A JUCE gateway communicates with the nearest device on the grid. Below we describe the behavior of the application:

- **Application propagation**: The application is first propagated from the gateway to the nearest device. The devices then use the tunneling protocol to migrate the application to all of the devices in the grid.
- **Message routing**: Each node continuously reads incoming messages and routes them according to an internal protocol..
- **Application adaptation**: The application uses polymorphism for implementing both generic and device specific behavior. The application uses a device API to detect the type of the device on top of which it is currently running. It then creates a device handler, which implements device specific behavior of the application.

In our simulation, we ran the application until each device has retrieved the code it requires. We then measured the size of the code executed on each device. Table 1 shows the code sizes for the devices. The total size of the code on each device is around 5KB, which is one order of magnitude less than a comparable application on a complete Java system. The devices customize their application code environment by downloading code specific to their needs. JUCE achieves this small footprint by loading only required methods of the application and runtime system. Note that JUCE uses methods as a basis for name resolution unlike the standard JVM approach where it is done at the class level. We note that the size of D-RTS is comparable to the size of the application.can be viewed as an empirical indication that the infrastructure scales down according to the needs of specific applications.

**Table 1.** Application code size

| Application code | Temperature Sensor | Vibration Sensor | Monitor |
|---|---|---|---|
| D-RTS | 1.7 KB | | |
| Device independent | 2 Kbytes | | |
| Device Specific | 1.4 KB | 1 KB | 0.2 KB |
| Total | 5.1 KB | 4.7 KB | 3.9 KB |

In Table 2 we show the resource requirements of the components of the dynamic runtime system.

**Table 2.** Sizes of components of the Dynamic Runtime System

| Function | Description | Memory Usage |
|---|---|---|
| rt_newarray | Provides support for primitive data type array allocation | 133 |
| rt_callstatic | Enables applications to make static method invocation. It intercepts programmer's calls, retrieves method from remote host and redirect the call to it. | 263 |
| rt_new | Handles object creation and constructor invocation | 312 |
| rt_callspecial | Invokes a static method referred by a symbolic reference | 204 |

| | | |
|---|---|---|
| rt_getfield | Resolves symbolic reference to an object's field | 132 |
| rt_virtual | Resolves symbolic reference to a virtual method and retrieves method from virtual method table reference | 332 |
| rt_virtual_quick | Replaces rt_virtual once a symbolic reference to a method is resolved. (if method is available locally, it can complete without invoking the JUCE gateway.) | 280 |

### 3.1.4 Dynamic Distribution Micro-Benchmark

In this experiment, we develop an application that performs optimally when the embedded device on which it executes has enough resources. The performance of the application degrades gracefully as less resources become available.

The application is a micro-benchmark designed to perform frequent invocations to the embedded device API. The application, thus, performs optimally if it is executed entirely on the embedded device. Due to memory constraints, the embedded device may not host the entire application. In these cases, the application has the capability to distribute itself between the gateway and the embedded device. Splitting the application, however, increases the network traffic, thereby decreasing the application's performance. By allowing the application to be reconfigured at runtime, the application can still improve its performance, as more memory becomes available and still function correctly under severe memory constraints.

The experiment analyzes the tradeoff between memory availability and application performances. For this purpose, we run the application with different memory availability and measure the traffic between the gateway and the device. To make the experiments more relevant for code migration effect analysis, we separate the memory allocated for code and data. Thus, the experiment can control directly the memory allocated for application code and eliminate the side effects produced by memory allocated for application data. The memory range used in the experiments spans between the minimum amount of memory required on the embedded device for the application to run and the entire application footprint.

Another parameter varied throughout the experiments is the problem size. Given the structure of the tested application, the number of times each method is invoked is proportional to the input size. This parameter directly affects the performance of the application. If each invocation is performed as RPC, the method invocation generates network traffic and slows the application down significantly. Thus, by changing the input size and memory, we can observe the tradeoff between resource availability and application performance. The results of the experiment are shown in Figure 5.

The graph in Figure 5 shows that beyond the memory size of 45KB, the network traffic is not affected by the problem size. This is because the entire application is executed on the embedded device. The rest of the graph show how the network traffic increases as the memory available on the embedded device reduces. The experiment highlights the ability of the JUCE infrastructure to manage dynamically the tradeoff between performance and resources availability. The general problem of dynamic application distribution is more complex and it goes beyond the scope of this paper.

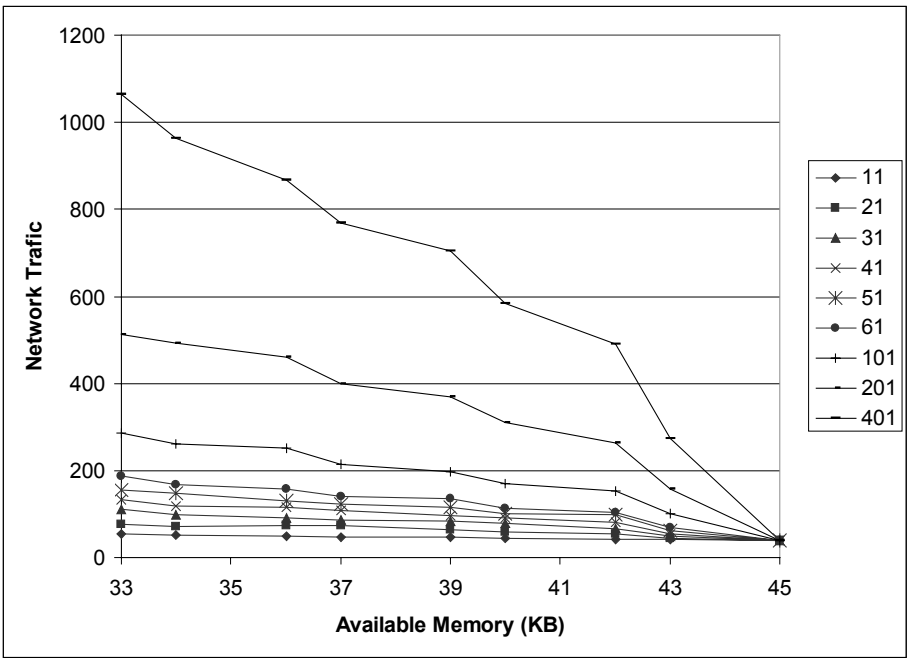


**Fig. 5.** Tradeoff between Memory and Application Performance

### 3.2 Security

The current architecture has several potential vulnerabilities. It assumes that devices can trust the gateways for enforcing all Java-specific security and safety constraints on application code. Further it assumes that the connection between the gateway and the device is secure. These assumptions follow the overall JUCE design principle of pushing many resource-intensive concerns into powerful nodes.

There are cases (especially for mobile devices) where a device can be placed in an environment for which the above assumptions do not hold. For instance, a PDA can be enabled to collaborate with devices in its proximity. In this environment, untrusted systems may be able to download malicious code to the remote device. A possible solution is to ensure that remote devices accept code that is authenticated by a trusted JUCE gateway. In this case, the PDA would have to find an appropriate JUCE gateway and request it to authenticate the downloaded code. Thus, the ability to enforce security depends on the ability to build a trust model effectively and cheaply between gateways and remote devices. We are currently examining ways in which this can be achieved.

### 3.3 Autonomy

In the proposed JUCE architecture, components of the application and runtime system are built incrementally and dynamically. This suggests that remote devices and gateways are closely coupled. For instance, as an application runs and comes across

new symbols, the device resident runtime system must interact with the gateway to resolve names and download new services. This dependence on the remote gateway, however, tends to reduce as an application reaches a point where all symbolic references have been resolved and required services have been downloaded. The devices, thus, tend to become autonomous over time.

In certain cases, this level of dependence may not be sufficient. In these cases, the JUCE gateway can force the embedded device to upload a set of methods and resolve a set of symbols before starting the application. The components of this set can be determined as a result of application flow analysis or semi-automatically when the set returned by the flow analysis is not bounded or too large.

# 4 Related Work

There has been significant research on developing ways in which ubiquitous devices can be used to assist in everyday-life activities (such as office interaction [1, 2], note-taking and home activities [3, 10, 11, 12]).

In this section, we focus primarily on middleware techniques for integrating ubiquitous devices in an overall distributed system framework.

We can classify the different middleware solutions according to how they address and manage interoperability among traditional and ubiquitous devices. The different solutions, thus, approach the interoperability problem at different levels and from different perspectives: At the **application level,** the main focus is placed on providing high level application programming models (such as mobile agents) that can deal in a convenient way with new types of devices. Solutions that provide application level interoperability typically assume the existence of an underlying middleware (for instance, a JVM) that allows the applications to be executed in a platform independent manner. At the **middleware level**, on the other hand, solutions deal with one or more of code, data, and protocol interoperability. The notion of protocol interoperability can cover the entire range of communication protocols from physical layer up to application layer. Note that the protocol interoperability is different from code and data interoperability in that while the first two deal with mediation between computational nodes, the protocol interoperability deals with mediation among communication channels.

Next we describe the different solutions.

## 4.1 Java Micro Edition

Java 2 Platform Micro Edition (J2ME) provides a compact Java middleware for embedded devices. J2ME features a *configuration layer* that allows the developer to adapt the system to the resources constraints that are specific to a given class of devices. Also, it provides a *profile layer* that allows the developer to provide access to the functionality specific to each type of device. In the J2ME approach both *configuration* and *profile* layers are defined at device's design time and cannot be changed dynamically according to particular application needs. J2ME provides two versions representing different levels of features/resource requirements trade-off. The

smaller JVM version – the KVM – still requires around 100KB for the base case scenario. The JUCE approach adds the possibility of scaling the system down for much smaller applications (on the order of 1KB). Also, while J2ME provides two distinct JVM choices, JUCE provides a continuous spectrum between a basic Java VM and a fully featured JVM.

### 4.2 Ninja Project

The Ninja project supports both application and protocol interoperability. It has developed techniques for integrating devices with limited computation capabilities into the Ninja Distributed System [5]. The devices, called motes, have limited computation and communication power. The devices are controlled by a micro operating system (TinyOS), and can be reprogrammed by downloading code into the devices. The Ninja system integrates these devices into the overall Ninja distributed system through an *Active Proxy* mechanism. An Active Proxy is aware of the protocols supported by each ubiquitous device and is responsible for bridging the gap between the protocols used in the rest of the system and the device. Active Proxy provides an extensible mechanism for adding different kinds of devices with differing protocols, thereby insulating the rest of the system from the differences in the devices. The Active Proxies, thus, implement an important design principle for developing ubiquitous computing systems: Instead of having all nodes support a fully featured system protocol, the responsibility can be placed in a small subset that then work as a gateway to the rest of the systems. The JUCE architecture applies the same principle in the area of code mediation.

The ideas proposed in the Ninja Project are similar to our approach. For instance, the TinyOS has functionalities similar to the one provided by the JUCE micro-kernel. Our work, however, adds the possibility of programming such a system in a high level language while still exploiting the advantages of native code execution. Also, there is a well-defined distinction between motes and regular Ninja nodes in the Ninja architecture. The JUCE architecture, on the other hand, opens up the possibility of utilizing any intermediate architecture scaling between extremely simple nodes up to fully featured ones.

### 4.3 PATH Project

The PATH project [27] addresses the problem of data level interoperability in the ubiquitous computing environments. The project is motivated by the fact that the proliferation of the new generation of computing devices would be associated with an explosion in the variety of data representation standards. Classic mediation solutions based on generally accepted standards or on conversion layers aware of a fixed set of formats fail to provide the flexibility and scalability required to deal with this continuously increasing number of data representations. PATH project proposes a compositional approach: the architecture consists of simple conversion modules capable of performing the conversion between two given data types. Every time two devices are required to exchange data, PATH infrastructure generates a chain of conversion modules that perform data conversion between the types supported by the two devices. Thus the PATH infrastructure can be extended dynamically to support a

potentially unlimited number of data formats. From conceptual perspective, the JUCE project complements PATH project in that while PATH provides data interoperability, JUCE provides code interoperability.

### 4.4 Teco

The focus of the Teco research group at University of Karlsruhe is on improving the protocol interoperability between ubiquitous computing devices and traditional distributed systems. The research group has developed an open architecture for integrating a wide range of communication protocols [7] such as Inferno [16], JetSend [17], Jini [18], Salutation [19], Tspaces [20], and UpnP [21].The primary focus in this project is to provide an abstraction over several communication protocols.

The code mobility features of the JUCE infrastructure also provide protocol-level interoperability. For instance, such an infrastructure can be enabled to dynamically add support for new protocols by downloading a new protocol handler. Devices can, thus, support any protocol without having to store the corresponding protocol code locally.

### 4.5 F-Desktop and other Examples of Application Solutions for Ubiquitous Computing Infrastructure

F-Desktop [29] framework is an example of mobile agent technology applied in the area of ubiquitous computing environments. The project focuses on the fact that in a ubiquitous computing environment, the application components are bound to the user and are therefore mobile. The research primarily focuses on issues such as detecting user's location or providing quality of services as application migrates to different types of devices. The system assumes the existence of a platform independent middleware.

There has been other efforts [24,25,26] in building software infrastructure for integrating mobile devices within a distributed framework. The primary focus in most of these approaches has been on building distributed programming models. All of these approaches assume that the devices can run Java programs directly. Our focus, on the other hand, is more on providing services that enable one to run any Java program on any device.

## 5 Summary

We have presented a Java middleware solution that can scale to a wide range of device sizes. The middleware provides services for running Java programs on remote ubiquitous devices. It achieves this by first compiling an application using a Just-In-Time compiler on a general-purpose gateway. In order to support runtime services required by the compiled code, the gateway also builds a runtime system customized specifically to the needs of the applications. The gateway then sends the compiled native code to a remote device. The remote device executes the applications and downloads the components of the runtime system dynamically. The proposed solution

is appealing for several reasons: it improves the performance of remote devices, and requires little resources on these devices. In addition, application programmers can continue to use Java's high-level abstractions and tools for developing applications for these devices.

Our current implementation focuses on the core features of the infrastructure. It does not provide full support for all Java applications. For instance, the current implementation does not support exception handling and multithreading. We are extending our implementation to provide a comprehensive set of functions. We are also looking at deploying the JUCE system within a real ubiquitous infrastructure in order to fully analyze its performance, scalability, and security characteristics.

## Acknowledgement

The authors would like to thank the reviewers for their comments. This work is supported by NSF grant no. CCR-0082677.

## References

1. Mark Weiser. Some Computer Science Issues in Ubiquitous Computing. *Communication of the ACM*, July 1993.
2. Roy Want, Bill Schilit, Norman Adams, Rich Gold, David Goldberg, Karin Petersen, John Ellis, Mark Weiser. An Overview of the ParcTab Ubiquitous Computing Experiment. *IEEE Personal Communications*, December 1995, Vol. 2 No.6, pp28-43.
3. Gregory D. Abowd and Elizabeth D. Mynatt. Charting Past, Present and Future Research in Ubiquitous Computing. *ACM Transactions on Computer-Human Interaction, Special issue on HCI in the new Millennium,* 7(1):29-58, March.
4. Steven D. Gribble, Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R. H. Katz, Z. M. Mao, S. Ross, and B. Zhao. The Ninja Architecture for Robust Internet-Scale Systems and Services.
5. Kari Kangas and Juha Röning. Using Mobile Code Interfaces to Control Ubiquitous Embedded Systems. *USENIX Workshop on Embedded Systems,* March 29-31, 1999.
6. Markus Lauff, Hans Werner Gellersen. Adaptation in a Ubiquitous Computing Management Architecture. *ACM Symposium on Applied Computing*, Como, Italy, March 2000.
7. Earl Barr, Raju Pandey, Michael Haungs. MAGE: A distributed Programming Model. In Proceedings of the International Conference of Distributed Computing Systems 2001.
8. Open JIT, www.openjit.org
9. G. D. Abowd. Classroom 2000: An experiment with the instrumentation of a living educational environment. *Pervasive Computing* Vol 38, No. 4.
10. Daniel Salber, Anind K. Dey, Rob J. Orr and Gregory D. Abowd. Designing for Ubiquitous Computing: A Case Study in Context Sensing. *GVU Technical*

*Report GIT-GVU-99-29. Submitted to the 1999 Conference on Human Factors in Computing Systems (*CHI '99*)*, July 1999.
11. Essa I., Ubiquitous Sensing for Smart and Aware Environments: Technologies towards the building of an Aware Home. *Position Paper for the DARPA/NSF/NIST Workshop on Smart Environments,* July 1999.
12. A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transaction on Computer Systems,* February 1984. 2(1).
13. James Stamos and Davis Gifford. Remote evaluation. In *ACM Transactions on Programming Languages and Systems,* October 1990. 12(4).
14. D. Chess, C. Harrison, and A. Kershenbaum. Mobile agents: Are they a good idea? *Technical report, IBM T.J. Watson Research Center,* 1995.
15. S. Dorward, R. Pike, D. L. Presotto, D. Ritchie, H. Trickey, and P. Winterbottom. The Inferno operating system. *Bell Labs Technical Journal,* 2(1):5-18, Winter 1997.
16. HP. JetSend Communications Technology Protocol Specification Version 1.5. *HP, White Paper,* 1999.
17. J. Waldo. Jini Architecture Overview. *Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303*, 1998. http://www.javasoft.com/products/jini/index.html
18. Salutation Consortium. Salutation Architecture: Overview, *White Paper,* 1998.
19. P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. TSpaces. *IBM Systems Journal*, 37(3):454-474, 1998. 6.
20. Christensson, Bengt and Larsson, Olof, "*Universal Plug and Play Connects Smart Devices*," WinHEC 99, 1999. http://www.axis.com/products/documentation/**UPnP**.doc
21. Sun Microsystems, Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices, *White Paper*, May 19, 2000.
22. Sheng Liang, Gilad Bracha, Dynamic Class Loading in the Java Virtual Machine, *OOPSLA '98,* 1998.
23. E. Kovacs, K. Rohrle, and M. Reich, Integrating Mobile Agents into Mobile Middleware, *Proc. Mobile Agents Int'l Workshop,* Springer-Verlag, Berlin, 1998, pp 124-135.
24. D. Kotz et al. "Agents TCL: Targeting the Needs of Mobile Computers, *Internet Computing,* July/Aug 1997, pp. 58-67.
25. S. Lipperts and A. Park, An agent-Based Middleware: A Solution for Terminal and User Mobility, *Computer Networks,* Sept 1999, pp 2053-2062.
26. Emre Kiciman, Armando Fox: Using Dynamic Mediation to Integrate COTS Entities in a Ubiquitous Computing Environment, *The Second International Symposium on Handheld and Ubiquitous Computing (huc2k)* Bristol, 25-27 September, 2000.
27. Barry Brumitt, Brian Meyers, John Krumm, Amanda Kern and Steven Shafer: EasyLiving: Technologies for Intelligent Environments, *The Second International Symposium on Handheld and Ubiquitous Computing (huc2k)* Bristol, 25-27 September, 2000
28. Kazunori Takashio, Gakuya Soeda, and Hideyuki Tokuda: A Mobile Agent Framework for Follow-Me Applications in Ubiquitous Computing Environment,

*The 21st International Conference on Distributed Computing Systems (ICDCS-21)* April 16-19, 2001. Phoenix (Mesa), Arizona, USA
29. MPI http://www.mpi-forum.org
30. David Curtis: Java, RMI and CORBA, White paper, *Object Management Group*
31. Calvin Austin and Monica Pawlan, JNI Technology, Advanced Programming for the Java™ Platform Chapter 5.