Motivating The Class Concept Runtime Environment in Java Language for Reliability Services and Ability Interface

{T.Gopinath, A.Satish Kumar, P.Rajesh Kumar}

Abstract:

The most java is one of the popular programming architectures because of its platform-independence. A monitoring infrastructure is a key component in each task aimed at evaluating the dependability of a starting be employed in mission and safety critical application, often with real time requirements. This infrastructure, named JVM on, collects data about both the state and the failures of the monitored Virtual Machine. The state of the JVM was defined according to the Java Virtual Machine specification. JVM on is constituted by three components: a monitoring agent which collects data from on it order JVM; a local monitor daemon that receives data from such an agent and updates the state of the JVM; a data collector, which stores events and state snapshots in a database. The impact on the performance of the JVM has been evaluated running the SPEC JVM98 benchmark suite. This paper describes research in the use of Java as J2SE, J2EE and J2ME virtual machine to develop cross platform computer vision applications. J2ME is powerful competitor of mobile operating system and drawn the attention of the leading manufacturers of the industry and became a hot spot of research. As it adopts the virtual machine Dalvik which is different from SUN Java and its Java application developing framework. The paper on takes the transplant of J2SE, J2EE application virtual machine's process with extension on J2ME applications process. The java experiments conducted to evaluate the compatibility, portability and efficiency of our library.

Key words: *Mobile mapping applications, J2ME MIDP,* Connected, Limited Device Configuration (CLDC), Connected Device Configuration (CDC), Mobile Information Device Profile (MIDP)**,** Kilo Virtual Machine (KVM), card Virtual Machine (CDC).

1. Introduction:

The Java™ programming language is a general-purpose, concurrent, class based, Object oriented language. It is designed to be simple enough that many programmer scan achieve fluency in the language. The Java programming language is related to C and C++ but is organized rather differently, with a number of aspects of and C++ omitted and a few ideas from other languages included. It is intended to be a production language, not a research language. The Java programming language is strongly typed. This specification clearly distinguishes between the *compile-time errors* that can and must be detected at compile time, and those that occur at run time. Compile time normally consists of translating programs into a machine-independent byte code representation. Run time activities include loading and linking of the classes needed to execute a program, optional machine code generation and dynamic optimization of the program and actual program execution. The Hot Spot VM is the garbage collector that comes with the Java Soft virtual machine. Its specification is delivered to all JVM providers (HP, IBM, BEA, etc.) and the standard implementation is deployed by Java Soft on Solaris, Windows and LINUX.

1.1 JVM Architecture in Java

In the Java virtual machine specification, the behaviour of a virtual machine instance is described in terms of subsystems, memory areas, data types, and instructions. These components describe an abstract inner architecture for the abstract Java virtual machine. The purpose of these components is not so much to dictate an inner architecture for implementations. It is more to provide a way to strictly define the external behavior of implementations. The specification defines the required behavior of any Java virtual machine implementation in terms of these abstract components and their interactions. Figure shows a block diagram of the Java virtual machine that includes the major subsystems and memory areas described in the specification. each Java virtual machine has a *class loader subsystem*: a mechanism for loading types (classes and interfaces) given fully qualified names. Each Java virtual machine also has an *execution engine*: a mechanism responsible for executing the instructions contained in the methods of loaded classes. A complete Java technology stack exists today to support embedded devices such as mobile phones. The stack is based on the Java 2 Platform, Micro Edition (J2ME™), and includes layers from the Java virtual machine to GUI support . These devices are characterized as small, battery-powered devices with limited, wireless connection to the In-

ternet. J2ME defines configurations and profiles, which, in combination with a Java virtual machine, make up the Java technology stack. A configuration of J2ME includes a Java virtual machine, as well as the Java programming language libraries that are required as the lowest common denominator of a range of embedded devices. A profile is a layer on top of the configuration that provides additional APIs for a specific class of devices. A particular combination of configuration and profile is appropriate only for specific Java virtual machines. J2ME fits in with the other editions of Java, J2SE and J2EE, as illustrated in FIGURE 1.1 Up to now, small, battery-powered devices are the domain of the KVM and the Mobile Information Device Profile (MIDP), also shown.

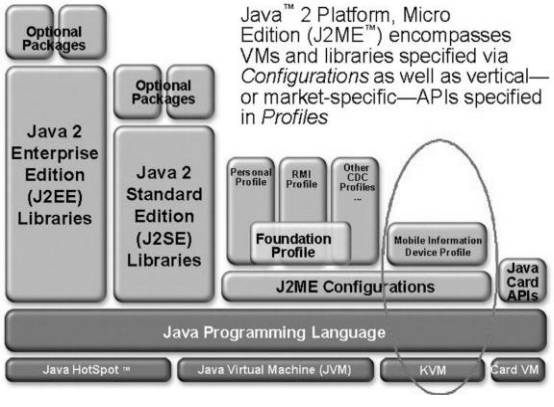

Figure 1.1**:** Java complete architecture

CLDC hotspot Implementation is now poised to take the place of the KVM as the high-performance Java virtual machine for the next generation of embedded devices. Java programming language

2. J2SE and J2EE JVM:

2.1 The internal architecture of the Java virtual machine

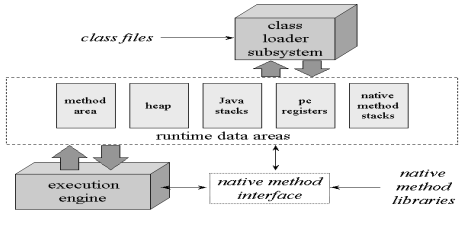

Figure: 2.1 The internal architecture of the Java virtual machine. In java we have Four Components (technologies) they are given bellow When a Java virtual machine runs a program, it needs memory to store many things, including byte codes and other information it extracts from loaded class files, objects the program instantiates, parameters to methods, return values, local variables, and intermediate results of computations. The Java virtual machine organizes the memory it needs to execute a program into several *runtime data areas*. Although the same runtime data areas exist in some form in every Java virtual machine implementation, their specification is quite abstract. Many decisions about the structural details of the runtime data areas are left to the designers of individual implementations. Different implementations of the virtual machine can have very different memory constraints. Some implementations may have a lot of memory in which to work, others may have very little. Some implementations may be able to take advantage of virtual memory, others may not. The abstract nature of the specification of the runtime data areas helps make it easier to implement the Java virtual machine on a wide variety of computers and devices. Some runtime data areas are shared among all of an application's threads and others are unique to individual threads. Each instance of the Java virtual machine has one *method area* and one *heap*. These areas are shared by all threads running inside the virtual machine. When the virtual machine loads a class file, it parses information about a type from the binary data contained in the class file. It places this type information into the method area. As the program runs, the virtual machine places all objects the program instantiates onto the heap. See Figure 2.1 for a graphical depiction of these memory areas.

2.2 Runtime data areas shared among all threads:

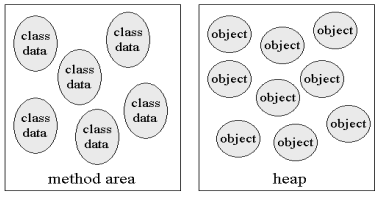

Figure: 2.2 Runtime data areas shared among all threads

As each new thread comes into existence, it gets its own *pc register* (program counter) and *Java stack*. If the thread is executing a Java method (not a native method), the value of the pc register indicates the next instruction to execute. A thread's Java stack stores the state of Java (not native) method invocations for the thread. The state of a Java method invocation includes its local variables, the parameters

with which it was invoked, its return value (if any), and intermediate calculations. The state of native method invocations is stored in an implementation-dependent way in *native method stacks*, as well as possibly in registers or other implementation-dependent memory areas.

The Java stack is composed of *stack frames* (or *frames*). A stack frame contains the state of one Java method invocation. When a thread invokes a method, the Java virtual machine pushes a new frame onto that thread's Java stack. When the method completes, the virtual machine pops and discards the frame for that method. The Java virtual machine has no registers to hold intermediate data values. The instruction set uses the Java stack for storage of intermediate data values. This approach was taken by Java's designers to keep the Java virtual machine's instruction set compact and to facilitate implementation on architectures with few or irregular general purpose registers. In addition, the stack-based architecture of the Java virtual machine's instruction set facilitates the code optimization work done by just-in-time and dynamic compilers that operate at run-time in some virtual machine implementations.

See Figure 2.3 for a graphical depiction of the memory areas the Java virtual machine creates for each thread. These areas are private to the owning thread. No thread can access the pc register or Java stack of another thread.

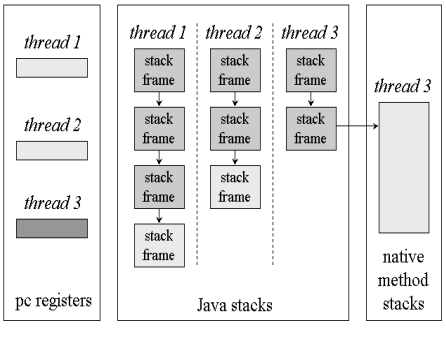

Figure 2.3 for a graphical depiction of the memory areas

3. J2ME JVM

J2ME stands for Java 2, Micro Edition. It is a stripped-down version of Java targeted at devices which have limited processing power and storage capabilities and intermittent or fairly low-bandwidth network connections. These include mobile phones, pagers, wireless devices and set-top boxes among other technology has evolved from a programming language designed to create machine-independent embedded systems into a robust, vendor-independent, machine-independent, server-side technology, enabling the corporate community to realize the full potential of web-centric applications. Java began with the release of the Java Development Kit (JDK). It was obvious from the start that Java was on a fast track to becoming a solution to the problems of many corporate systems. More interface and libraries were extended in the JDK as the corporate world demanded—and received—application programming interfaces (API) that addressed real-world situations. JDK API extensions fully integrated into the JDK with the release of the Java 2 Standard Edition (J2SE) of the JDK. J2SE contains all the APIs needed to build industrial strength Java applications. However, the corporate world felt J2SE lacked the strength required for developing enterprise-wide applications and for servicing the needs of developers of mobile and embedded systems .Again the corporate community pushed Sun Microsystems, Inc. to revise Java technology to address needs of an enterprise. Sun Microsystems, Inc. then launched the Java Community Program (JCP) that brought together corporate users, vendors, and technologists to develop a standard for enterprise Java API's Java 2 Platform Enterprise Edition, commonly referred to as Java 2 Enterprise Edition(J2EE), and the Java 2 Micro Edition (J2ME).

3.1 What's inside

J2ME application development divided into these five parts:

- Part I: J2ME Basics
- Part II: J2ME User Interface
- Part III: J2ME Data Management
- Part IV: J2ME Personal Information Manager Profile
- Part V: J2ME Networking and Web Services

Part I: J2ME Basics

The new web-centric corporation is changing the way in which it delivers highly efficient, enterprise-wide distributive systems to meet the round-the-clock instantaneous demand expected by thousands of concurrent users—anywhere at any time. The old way of building enterprise systems won't solve today's corporate IT requirements Technologists at Sun Microsystems, Inc. and the Java Community Program rewrote the way developers build large-scale, web-centric distributive systems by using Java 2 Enterprise Edition (J2EE), and Java 2 Micro Edition (J2ME). J2EE addresses complex server-side issues faced by programmers who develop these systems while J2ME addresses the need to create mobile and embedded components that makes these systems accessible from other than the desktop. Part I of this book introduces you to basic concepts used in J2ME technology and in Web services technology. These concepts focus on four areas of interest, beginning with an overview of J2ME that defines J2ME and illustrates J2ME's role in the evolutionary process of computer programming. The next area of interest examines the J2ME architecture. It is here where you roll up your sleeves and get your hands into the guts of J2ME to investigate how J2ME works within the Web services infrastructure. At first glance, you might feel overwhelmed by the power of J2ME. However, that feeling is short-lived because the third area of interest in Part I discuss J2ME best practices, showing you commonly used design principles used by J2ME programmers to build advanced J2ME Web centric distributive systems. Part I concludes with a look at J2ME design patterns used to solve common Programming problems that crop up during the development of a J2ME application.

Part II: J2ME User Interface

Nearly every J2ME application that you develop requires a way for a user to interact with it unless the application is an embedded closed system. An embedded closed system such as those that control an automobile's engine doesn't require input from the user but instead receives input from electro-mechanical devices. Abuser interface for a J2ME application is similar to yet different from a user interface that you find on a desktop application. They are similar in that both display options available to the user and then receive and process the option selected by the user.

However, a J2ME user interface is less sophisticated than those found on a desktop application because of the limited resources (i.e., screen size) that are available on a J2ME device (i.e., cellular phone). In Part II you'll learn database concepts of the J2ME user interface.

Part III: J2ME Data Management At the center of nearly every J2ME application is a repository of information that is accessed and manipulated by both service-side components, such as Web services, and client-side applications. A repository is a database management system that stores, retrieves, and maintains the integrity of information stored in its databases A J2ME application uses Java data objects, JDBC, and other technology that is necessary to interact with a database management system to provide information to the J2ME application. In Part III you'll learn database concepts in relation to Java data objects. You'll also explore the details of JDBC, which is used to connect to and interact with popular— and some not so popular—database management systems. And you'll also learn how to create and send requests for information and integrate the results of a request into your J2ME application.

Part IV: J2ME Personal Information Manager Profile Many corporations have practically made PDAs the *de facto* standard as a mobile communicator, especially since PDA and cell phone technologies have merged, causing a blur between PDAs and cell phones. That is, a PDA can be used as a cell phone and cell phones have incorporated PDA applications. Until recently, J2ME applications lacked the capability to interact with native PDA databases such as those used to store calendar, to-do list, and address information. The Java Community Process released a new Personal Information Manager (PIM) API, which is used to develop sophisticated J2ME applications. This enables J2ME applications to interact with the J2ME device's personal information database, which is used by the device's address book, notepad, and calendar applications. In Part IV of this book you'll explore this API and learn how to implement it in your J2ME application.

Part V: J2ME Networking and Web Services The glue that enables J2ME applications to interact with external applications, including server-side components, is networking capabilities. In Part V you'll learn how to implement routines that take advantage of a J2ME device's network features to open communications with other applications using a hard-wire or wireless network connection. You'll also learn how to utilize Web services to expand the horizon of your J2ME application. Web services are a web of services where services are software building blocks that are available on a network from which programmers can efficiently create large-scale distributive systems. You won't learn how to create Web services, but you will learn how to utilize them to increase the functionality of your J2ME application beyond the limited resources found on a J2ME de-

vice. In Part V, you'll also learn about Service Oriented Architecture Protocol SOAP), Universal Description, Discovery, and Web Services Description Language (WSDL), and how to implement them in your J2ME application.

3.2 J2ME Profiles

A profile consists of Java classes that enable implementation of features for either a particular small computing device or for a class of small computing devices. Small computing technology continues to evolve, and with that, there is an ongoing process of defining J2ME profiles. Seven profiles have been defined as of this writing. These are the Foundation Profile, Game Profile, Mobile Information Device Profile, PDA Profile, Personal Profile, Personal Basis Profile, and RMI Profile.

■ **The Foundation Profile** is used with the CDC configuration and is the core for nearly all other profiles used with the CDC configuration because the Foundation Profile contains core Java classes.
■ **The Game Profile** is also used with the CDC configuration and contains the necessary classes for developing game applications for any small computing device that uses the CDC configuration.
■ **The Mobile Information Device Profile** (MIDP) is used with the CLDC configuration and contains classes that provide local storage, a user interface, and networking capabilities to an application that runs on a mobile computing
device such as Palm OS devices. MIDP is used with wireless Java applications.
■ **The PDA Profile** (PDAP) is used with the CLDC configuration and contains classes that utilize sophisticated resources found on personal digital assistants. These features include better displays and larger memory than similar resources found on MIDP mobile devices (such as cell phones).
■ **The Personal Profile** is used with the CDC configuration and the Foundation Profile and contains classes to implement a complex user interface. The Foundation Profile provides core classes, and the Personal Profiles provide classes to implement a sophisticated user interface, which is a user interface that is capable of displaying
multiple windows at a time.
■ **The Personal Basis Profile** is similar to the Personal Profile in that it is used with the CDC configuration and the Foundation Profile. However, the Personal Basis Profile provides classes to implement a simple user interface, which is a user interface that is capable of displaying one window at a time.
■ **The RMI Profile** is used with the CDC configuration and the Foundation Profile to provide Remote Method Invocation classes to the core classes contained in the Foundation Profile.

4. J2ME Configuration?

A configuration defines the minimum Java technology that an application developer can expect on a broad range of implementing devices.

J2ME Connected, Limited Device Configuration (CLDC)

- specifies the Java environment for mobile phone, pager and wireless devices
- CLDC devices are usually wireless
- 160 - 512k of memory available for Java
- typically has limited power or battery operated
- network connectivity, often wireless, intermittent, low-bandwidth (9600bps or less)

J2ME Connected Device Configuration (CDC)

- Describes the Java environment for digital television set-top boxes, high end wireless devices and automotive telemetric systems.
- device is powered by a 32-bit processor
- 2MB or more of total memory available for Java
- network connectivity, often wireless, intermittent, low-bandwidth (9600bps or less)

These two configurations differ only in their respective memory and display capabilities.

J2ME Profile

A specification layer above the configuration which describes the Java configuration for a specific vertical market or device type.

J2ME Profiles

J2ME Mobile Information Device Profile (MIDP)

- this is the application environment for wireless devices based on the CLDC
- contains classes for user interface, storage and networking

J2ME Foundation Profile, Personal Basis, Personal and RMI profiles

- these are profiles for devices based on the CDC, which are not addressed in this tutorial

4.1 Virtual Machines

The CLDC and the CDC each require their own virtual machine because of their different memory and display capabilities. The CLDC virtual machine is far smaller than that required by the CDC and supports less features. The virtual machine for the CLDC is called the Kilo (Kubai) Virtual Machine (KVM) and the virtual machine for the CDC is called the CVM.

4.2 KVM

Everything will be connected to the Internet Customizable, Personal Services by downloading new services and applications from the internet currently, interactive games, banking and ticketing applications

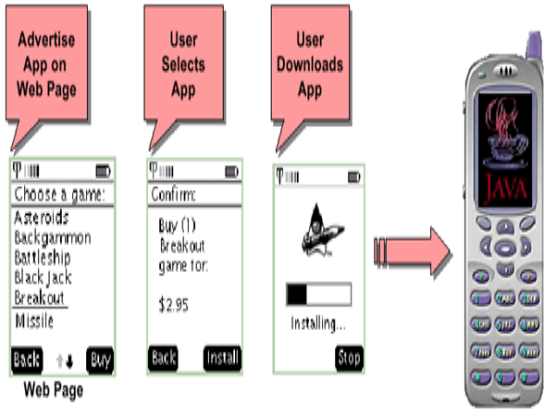

Figure : 4.2 Downloading customized services

Technology that enables this: Java2 Platform Micro Edition Configurations and Profiles
Three layers of software built upon the Host Operating System of the device:

4.2.1 JVM Layer

Configuration Layer
Profile Layer

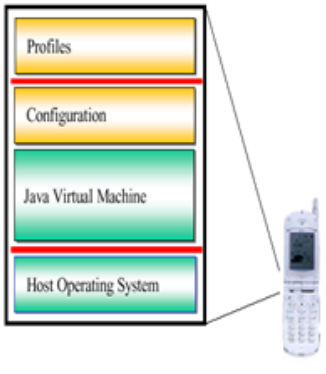

Figure : 4.2.1 J2ME software layer stack

J2ME keeps key features of Java Technology
Built-in consistency: run anywhere, any time, on any device
High-level OOP
Portability of code
Safe network delivery
Upward scalability with J2SE and J2EE 2 broad categories of products *Shared, fixed, connected information devices* (represented by CDC): Memory budget: 2-16 MB, most often using TCP/IP *Personal, mobile, connected information devices* (represented by CLDC): memory budget: about 128KB, low bandwidth, intermittent network connections, often not based on TCP/IP

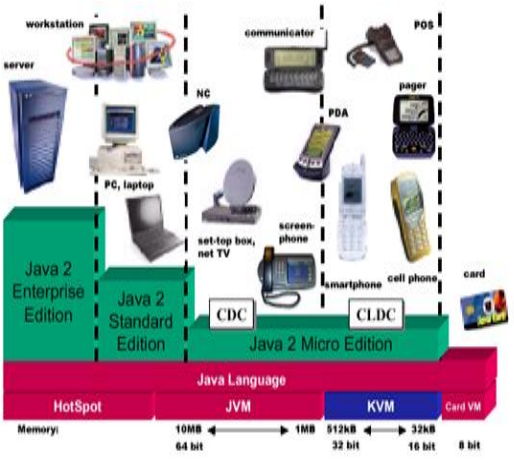

Figure : 4.2.2 Java 2 edition target editions

4.3 J2ME Building Blocks Configuration

Defines a minimum platform for a *horizontal* category or grouping of devices, each with similar requirements on a total memory budget and processing power.
Profile
Is layered on top of a configuration. It addresses the specific demands of a certain *vertical* device family. It guarantees interoperability within a family or domain White Paper from Sun.com

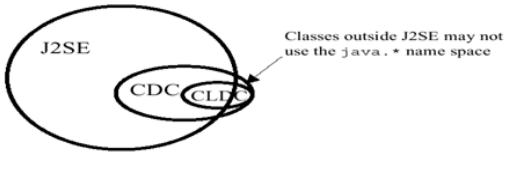

Figure: 4.3 relationship between J2ME configuration and J2SE
KVM is a compact, portable JVM intended for small, resource-constrained devices such as cell phones, pagers, etc.
The high-level design goal for KVM was to create the smallest possible *complete* JVM that would maintain all the central aspects of the Java programming language, and that would nevertheless run in a resource-constrained device with only a few tens or hundreds of KB of available memory (hence the name K, for Kilobytes

CLDC hotspot Implementation is a pure 32 bit virtual machine. This provides a large address space and

scalable architecture well-suited for mid- to high-end mobile phones. It is especially suited for the emerging 2.5G and next generation mobile phones, which typically have larger memory capacity.
Compact Object Layout CLDC HotSpot Implementation supports a compact object layout to reduce general memory consumption. A Java object has two parts. The first part is the object header, which provides reflective information and contains hash code and locking status. The second part is the object body, containing the object fields. Most other virtual machines use at least two words for the object header's however, since the average object size is small, object headers take up a big fraction of the total
object space. CLDC HotSpot Implementation introduces a new design, in which only one word is needed for the object header. In addition to reducing memory usage, object allocation becomes faster. Unified Resource Management A major benefit of CLDC HotSpot Implementation is unified resource management.
This means that all allocated data resides inside the object heap. Allocated data includes:

- Java level objects,
- Reflective objects, such as methods and classes,
- Compiler generated code, and
- Virtual machine internal data structures.

An important advantage of this unification is that the same garbage collector takes care of cleaning up all allocated resources, even compiled code. Almost t all other virtual machines have designated areas for user objects, reflective data, temporary data and generated code. Such a scheme results in memory fragmentation, multiple cleanup strategies and other complexities. CLDC HotSpot Implementation solves these issues by using the mark-sweep-compact garbage collector for everything. Another benefit of unified resource management is that compiled code can be removed dynamically to free up space for user-level objects. The CLDC HotSpot Implementation Garbage Collector A garbage collector automatically reclaims unused object memory and makes the freed memory available for new allocations. CLDC HotSpot Implementation uses an accurate generational mark-sweep-compact garbage collector, which results in:

- Fast object allocation
- Small garbage collection pauses
- No memory fragmentation

4.4 Execution Engine

In general, Java virtual machines with a compiler are an order of magnitude faster than those with only an interpreter. For that reason, CLDC HotSpot Implementation includes a dynamic compiler to provide fast byte code execution. A well-known problem with compiling byte codes into native instructions is that the generated code takes up four to eight times as much space as the original byte codes. Adaptive compilation alleviates this problem by only compiling methods that are recognized as "hotspots", i.e., the most frequently used parts of the application. The CLDC HotSpot Implementation dynamic compiler finds the hotspot by running a statistical profiler. To minimize the amount of compiled code, the CLDC HotSpot Implementation virtual machine includes an optimized interpreter used for infrequently executed methods. The CLDC HotSpot Implementation compiler is a simple one-pass compiler that utilizes the following basic optimizations: constant folding, constant propagation, loop peeling. The components of the CLDC HotSpot Implementation virtual machine are shown in FIGURE 4.4

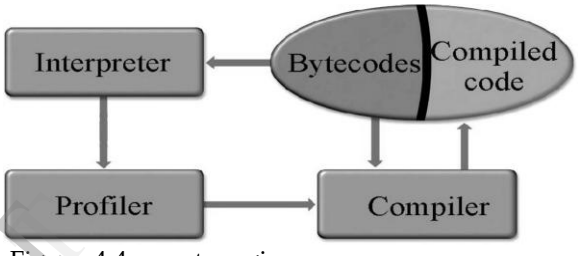

Figure :4.4 execute engine

Fast Thread Synchronization The Java programming language provides language-level thread synchronization, which makes it easy to express multi-threaded programs with fine-grained locking.
CLDC HotSpot Implementation uses a variant of the block structured locking mechanism developed for the HotSpot virtual machine. As a result, synchronization performance becomes so fast that it is no longer a performance bottleneck for Java programs.

5. CVM

A Java runtime environment is an implementation of Java technology for a specific target platform. It performs a middleware function with features common to a native application: it is installed, launched and run like a native application. But its real purpose is to launch, run and manage Java application software on the target platform.
The Connected Device Configuration (CDC) Java runtime environment is an implementation of Java technology for connected devices. These include mobile devices like PDAs and smartphones as well as attached devices like set-top boxes, printers and kiosks. CDC target devices can vary widely based on their features and purpose. FIGURE 1-1 describes some CDC target device categories and organizes them by their two most important characteristics:

purpose (fixed or general) and mobility (mobile or attached).

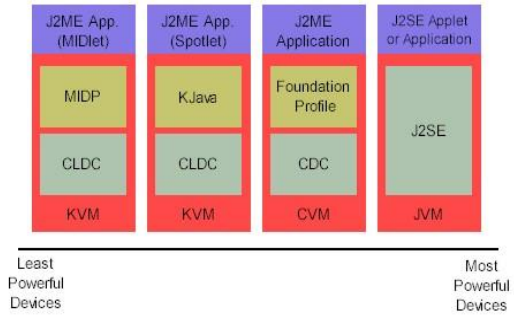

Figure 5: JVM architecture in J2ME

This runtime guide describes how to use the CDC Java runtime environment for different purposes including application development, runtime development and solution deployment.
This chapter briefly introduces the CDC Java runtime environment through the following:
■Goals
■Usage Contexts
■CDC Technology Implementations
■CDC Target Device Requirements
■Java ME Technology Standards
■Java ME API Choices
■CDC Application Features
■Application Management
■Developer Tools

The CDC Java runtime environment described in this runtime guide can operate in several different usage contexts:
■During *product development*, the CDC Java runtime environment has testing features that can help isolate problems while porting CDC technology to a new target platform. For example, the trace features provide details about code and method execution as well as garbage collection (GC) state.
■One of the final stages of product development is *TCK verification*. A TCK is a test suite that verifies the behavior of an implementation of Java technology. The TCK includes a test harness that runs a candidate Java runtime environment and launches a series of test Java applications. TCK verification is described in the TCK user guides listed in "Related Documentation" on page xvi.

■*Application development* for the CDC platform requires a target Java class library for compiling Java source code and a CDC Java runtime environment for testing and debugging. Chapter 8 provides more information about application development with the CDC Java runtime environment.

■When an application is complete and tested, it's ready for *deployment*. CDC provides a number of deployment mechanisms including preloading with Java Code Compact, managed application models like applets and xlets and network-based provisioning systems.

5.1 CDC Target Device Requirements

CDC is an adaptable technology that can support a range of connected target devices that exist today and in the future. The baseline system requirements of these connected devices are the following:
■Network connectivity
■32-bit RISC-based microprocessor
The memory requirements for a CDC Java runtime environment vary based on the native platform, the profile and optional packages and the application. See Section 4.4, "Memory Management" on page 4-7 for memory usage guidelines.
Other features of the CDC target device can include:
■a display for a graphical user interface (GUI)
■Unicode font support
■an open or proprietary native platform that provides operating system services

5.2 Running Applications

The CDC Java runtime environment includes CVM, the CDC application launcher, for loading and executing Java applications. This chapter describes basic use of the cvm command to launch different kinds of Java applications, as well as more advanced topics like memory management and dynamic compiler policies. cvm, the CDC application launcher is similar to java, the Java SE application launcher. Many of CVM's command-line options are borrowed from java. The basic method of launching a Java application is to specify the top-level application class containing the main() method on the cvm command-line. For example, cvm Hello World By default, cvm looks for the top-level application class in the current directory. As an alternative, the -cp and -class path command-line options can specify a list of locations where cvm can search for application classes. For example,
% cvm -cp /mylib: demo classes.jar Hello World
Here cvm searches for a top-level application class named Hello World, first in the directory /mylib and then in the archive file democlasses.jar. "Class Search Path Basics" on page 4-4 for more information about class search paths.
The -help option displays a brief description of the available command-line options. Appendix A provides a complete description of the command-line options available for cvm.

5.3 Running Managed Applications (*Personal Basis Profile and Personal Profile only*)

Managed application models allow developers to offload the tasks of deployment and resource management to a separate application manager. The CDC Java runtime environment includes sample application managers for two different application models:

■The *applet application model* was one of the first success stories of Java technology. An applet contains dynamic web content that a user can view and manipulate through an applet viewer, typically built into a web browser.

■The *xlet application model* is similar in purpose to the applet application model, but different in design. The main differences between an x let and an applet are that an xlet has a cleaner life cycle model and doesn't require an explicit dependency on AWT. These features make x lets more appropriate for embedded device scenarios like set-top boxes and PDAs.

5.3.1 Running an Applet (*Personal Profile only*)

The CDC Java runtime environment includes a simple applet launcher named sun. applet. Applet Viewer which displays each applet in a separate frame. Applet Viewer is a simplified version of the Java SE applet viewer utility (see http://java.sun.com/j2se/1.4.2/docs/tooldocs/solaris/appletviewer.html). The CDC version does not have a separate command-line utility and does not support all of the options available in the Java SE version, such as -debug, -encoding, and -J. The basic command syntax to launch an applet with Applet Viewer is: cvm sun. applet. AppletViewer *URL*

URL identifies an HTML file containing an APPLET, OBJECT or EMBED tag that identifies an applet. See http://java.sun.com/j2se/1.4.2/docs/tooldocs/appletviewertags.html for a description of the HTML tags supported by AppletViewer.

Here is a simple example of how to launch an applet based on the DemoApplet example in democlasses.jar:% cvm sun.applet.AppletViewer personal/DemoApplet.html

5.3.2 Running an Xlet (*Personal Basis Profile and Personal Profile only*)

The CDC Java runtime environment includes a simple xlet manager named com.sun.xlet.XletRunner. Xlets can be graphical, in which case the xlet manager displays each xlet in its own frame, or they can be non-graphical. The basic command syntax to launch XletRunner is:

% cvm com.sun.xlet.XletRunner { \ -name *xletName* \ (-path *xletPath* | -codebase *url-Path*) \ -args *arg1 arg2 arg3* ...} \ ...% cvm com.sun.xlet.XletRunner -filename *optionsFile*% cvm com.sun.xlet.XletRunner –usage

6. Concussion

With the help of JVM we can run any kind of applications in java runtime environment technologies and the dramatic increase of mobile communication devices the demand for applications run on these devices also increased where the consumers expects quality services.J2ME provides standard to meet this expectations. To keep pace with the demands for performance of the next generation of mobile phones and other wireless devices, Sun Microsystems saw the need for a new Java virtual machine technologist result is CLDC Hotspot Implementation, which achieves a performance gain of nearly an order of magnitude compared to the first generation of J2ME deployments. The CLDC Hotspot Implementation virtual machine was demonstrated on real devices at Java One 2002.This technology is available to device manufacturers under license from Sun Microsystems.

7.References:

[1] Minoi,J. L., Green P., Arab, S., 2002, Navigation Application with Mobile Telephony: Shortest-path, University Malaysia Sarawak, University of Manchester,
Map Asia 2002, http://www.gisdevelopment.net/ technology/lbs./techlbs008pf.htm

[2] Liu, J., 2002, Mobile Map: A Case Study in the Design &
Implementation of a Mobile Application, Computer Engineering Dept. of Carleton University, http://www.sce.
carleton.ca/wick/chameleon/mc/JaneThesis.pdf

[3] Revel, S., 2001, How Does Location Govern Task? The
Delivery of an Intelligent Tourist Aide over a Palm Device,
Univ. of Edinburgh, MSc Dissertation Plan, http://www.
geo.ed.ac.uk/~gisadmin/submission/msc0138/plan.html

[4] Virrantaus, K., Markkula, J., Garmash, A., Terziyan, V.,
Veijalainen, J., Katanosov, A., Tirri, H., 2001, Developing
GIS-supported Location Based Services, Helsinki University of Technology, University of Jyvaskyla, Finland,
Second International Conference on Web Information Systems Engineering (WISE'01), Volume 2, Japan, http://www.cs.jyu.fi/ai/papers/WGIS-01.pdf

[5] Megler, V., 2002, i-mode:From bandwidth problem into
Internetphenomenon, http://www-106.ibm.com/developerworks/library/wi-imode/index.html
[6] Keryer, P., Nara, T., 2001, i-mode: A Successful Lauch of
The Mobile Internet Market, Alcatel Telecommunications
Review – First Quarter 2001, http://atr.alcatel.de/hefte/01i_1/gb/pdf_gb/12keryergb.pdf
[7] Developnet, Why is J2ME MIDP superior to WAP, http://www.developnet.co.uk/wap.html
[8] Dru, M-A, Saada, S., 2001, Location-Based Mobile Services: The Essentials, Alcatel Telecommunications
Review – First Quarter 2001, http://atr.alcatel.de/hefte/01i_1/gb/pdf_gb/14drugb.pdf
[9] Goodman, D. J., 2000, The wireless Internet: Promises and Challenges, July 2000 IEEE Computer magazine.
[10] Knudsen, J., 2002, What's New in MIDP 2.0, http://wireless.java.sun.com/midp/articles/midp20/
[11] Laitinen, H., 2001, Sun's 2001 Worldwide Java Developer Conference, Development Tools for the J2ME,http://www.forum.nokia.com/main/1,35452,1_0_75,00.html.
[12] JBuilder MobileSet 3.01 Developer's Guide, http://info.borland.com/techpubs/jbuilder/jbuilder8/me/contents.html
[13] Capone, J. M., 2001, Java 2ME Bridging wireless Gap?,
http://www.onjava.com/lpt/a/1068
[14] Angelides, J., 2002, The right technology is vital for location based applications, Wireless Europe,

AUTHORS

First Author- T GOPINATH Pursing Master of Computer Application in JNTU, ANTHAPUR. In Siddhartha Institute of Engineering and Technology College, puttur, Andhra Pradesh, India. His field of interest is java Technologies .

Second Author –A SATISHKUMAR Pursing Master of Computer Application in JNTU, ANATHAPUR. In Siddhartha Institute of Engineering and Technology College, puttur, Andhra Pradesh, India. His field of interest is java Technologies .

Third Author –P RAJESH KUMAR Received the MCA POST GRADUATE In computer applications in JNTU, Hyderabad. He is currently working as an Assistant Professor , in Siddhartha institute of Engineering and Technology College, putter, Andhra Pradesh, India. His field of interest is java Technologies and data mining and data ware house