An Energy Consumption Framework for Distributed Java-Based Systems

Chiyoung Seo[1] Sam Malek[2] Nenad Medvidovic[1]

[1] Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781 U.S.A.
{cseo, neno}@usc.edu

[2] Department of Computer Science
George Mason University
Fairfax, VA 22030-4444 U.S.A.
smalek@gmu.edu

ABSTRACT

In this paper we define and evaluate a framework for estimating the energy consumption of Java-based software systems. Our primary objective in devising the framework is to enable an engineer to make informed decisions when adapting a system's architecture, such that the energy consumption on hardware devices with a finite battery life is reduced, and the lifetime of the system's key software services increases. Our framework explicitly takes a component-based perspective, which renders it well suited for a large class of today's distributed, embedded, and pervasive applications. The framework allows the engineer to estimate the software system's energy consumption at system construction-time and refine it at runtime. In a large number of distributed application scenarios, the framework showed very good precision on the whole, giving results that were within 5% (and often less) of the actually measured power losses incurred by executing the software. Our work to date has also highlighted a number of possible enhancements.

1. INTRODUCTION

Modern software systems are predominantly distributed, dynamic, and mobile. They increasingly execute on heterogeneous platforms, many of which are characterized by limited resources. One of the key resources, especially in long-lived systems, is battery power. Unlike the traditional desktop platforms, which have uninterrupted, reliable power sources, a newly emerging class of computing platforms have finite battery lives. For example, a space exploration system may comprise satellites, probes, rovers, gateways, sensors, and so on. Many of these are "single use" devices that are not rechargeable. In such a setting, minimizing the system's power consumption, and thus increasing its lifetime, becomes an important quality-of-service concern.

Consider the scenario depicted in Figure 1, in which seven software components are deployed on four battery-powered hardware hosts, and are communicating over the network. Without concerning ourselves with any other details of this application, we can ask a number of questions about its energy consumption. For example, does the location of a given component (e.g., c_4) impact its energy consumption rate? Would redeploying a component (e.g., c_4) from one host (e.g., H_4) to another (e.g., H_2) change the system's, or a given system service's, life span? Can we compare the likely energy consumption profiles of two or more candidate deployments? What is the best deployment for the system with respect to energy consumption?

The simple observation guiding our research is that if we could estimate the energy costs of a given software system in terms of its constituent software components ahead of its actual deployment, or at least early on during its execution, we would be able to answer the above questions. In turn, this would allow us to take appropriate actions to prolong the system's life span: unloading unnecessary or expendable software components, redeploying highly CPU-intensive components to more capacious hosts, collocating frequently communicating components, and so on.

To this end, in this paper we present a framework that estimates the power consumption of a distributed Java-based software system at the level of its components, both prior to and during runtime. We chose Java because of its intended use in network-based applications (including sensor networks [4]), its popularity, and very importantly, its reliance on a virtual machine, which justifies some simplifying assumptions possibly not afforded by other mainstream languages. We have evaluated our framework for precision on a large number of distributed Java applications, by comparing its estimates against actual electrical current measurements. Our results suggest that the framework is always able to estimate the power consumed by a distributed Java system to within 5% of the actual consumption.

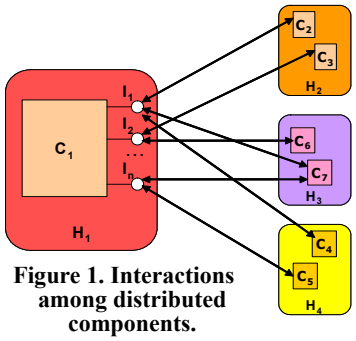

Figure 1. Interactions among distributed components.

One novel aspect of our estimation framework is its component-based development perspective, which renders it well suited for distributed, embedded, and pervasive applications. To facilitate component-level energy cost estimates, we suggest a computational energy cost model for a software component. We integrate this model with the component's communication cost model, which is based on the experimental results from previous studies. This integrated model results in highly accurate estimates of a component's overall energy cost. Furthermore, unlike most previous power estimation tools for embedded applications, we explicitly consider and model the energy overhead of a host's OS and an application's runtime platform (e.g., JVM) incurred in facilitating and managing the execution of software components. This further enhances the accuracy of our framework in estimating a distributed software system's energy consumption. Another contribution of this work is its ability to adjust energy consumption estimates at runtime efficiently and automatically, based on monitoring the changes in a small number of easily tracked system parameters

(e.g., size of data exchanged over the network, inputs to a component's interfaces, invocation frequency of each interface, etc.).

In the remainder of this paper we first present the related research in the energy estimation and measurement areas (Section 2). We then introduce our energy estimation framework (Section 3) and explain how it is applied to component-based Java systems. We round out the paper with a discussion of some current and planned applications of this research (Section 4).

2. RELATED WORK

Several studies have profiled the energy consumption of Java Virtual Machine (JVM) implementations. Farkas et al. [2] have measured the energy consumption of the Itsy Pocket Computer and the JVM running on it. A JVM generally has five stages during its life cycle [7]: *start*, *initialize*, *load main class, interpreter loop*, and *exit*. Farkas et al. have discussed different JVMs' design trade-offs in each stage and measured their energy consumption. The energy consumed at the *interpreter loop* stage corresponds to the actual energy required to execute a Java application, while the energy consumed by the other stages is constant [7]. Lafond et al. [7] have measured the energy consumption of each stage, and showed that the energy required for memory accesses usually accounts for 70% of the total energy consumed. However, none of these studies suggest a model that can be used for estimating the energy consumption of a distributed Java-based system.

There have been several tools that estimate the energy consumption of embedded operating systems (OSs) or applications. Tan et al. [10] have investigated the energy behaviors of two widely used embedded OSs, μC/OS [6] and Linux, and suggested their quantitative macro-models, which can be used as OS energy estimators. Sinha et al. [9] have suggested a web-based tool, *JouleTrack*, for estimating the energy cost of an embedded software running on StrongARM SA-1100 and Hitachi SH-4 microprocessors. While they certainly informed our work, we were unable to use these energy estimation tools directly in our targeted distributed Java domain because none of them provide generic energy consumption models, but instead have focused on individual applications running on specific OSs and platforms.

Previous research has also proposed solutions for the energy accounting problem by computing the overall energy consumption of a system in terms of individual processes within an operating system [13]. Although these approaches have some similarities with our estimation framework, we cannot adapt their results directly because they model the system's energy consumption at the level of processes and a process does not always correspond to a single software component (e.g., in Java, there are usually multiple software components running in a single JVM process).

Several studies [3,12] have measured the energy consumption of wireless network interfaces on handheld devices that use UDP for communication. They have shown that the energy usage by a device due to exchanging data over the network is directly linear to the size of data. We use these experimental results as a basis for defining a component's communication energy cost.

3. ENERGY COST FRAMEWORK

We model a distributed software system's energy consumption at the level of its components. A *component* is a unit of computation and state. In a Java-based application, a component may comprise a single class or a cluster of related classes. The energy cost of a software component consists of its *computational* and *communication* energy costs. The computational cost is mainly due to CPU processing, memory access, I/O operations, and so forth, while the communication cost is mainly due to the data exchanged over the network. In addition to these two, there is an additional energy cost incurred by an OS and an application's runtime platform (e.g., JVM) in the process of managing the execution of user-level applications. We refer to this cost as *infrastructure energy overhead*. In this section, we present our approach to modeling each of these three energy cost factors. We conclude the section by discussing how our framework can be applied to Java systems.

3.1. Computational Cost

In order to preserve a software component's abstraction boundaries, we determine its computational cost at the level of its public interfaces. A component's interface corresponds to a service it provides to other components.[1] While there are many ways of implementing an interface and binding it to its caller (e.g., RMI, event exchange), in the most prevalent case an interface corresponds to a method. In Section 3.2 we discuss other forms of interface implementation and binding (e.g., data serialization over sockets).

As an example, Figure 1 highlights a component c_1 on host H_1, c_1's provided interfaces, and the invocation of those interfaces by remote components. Given the energy consumption *iCompEC* resulting from invoking an interface I_i, and the total number b_i of invocations for the interface I_i, we can calculate the overall energy consumption of a component c_1 with n interfaces (expressed in *Joule or J*) as follows:

$$cCompEC(c_1) = \sum_{i=1}^{n} \sum_{j=1}^{b_i} iCompEC(I_i, j) \qquad \textbf{\textit{Eq. 1}}$$

In this equation, $iCompEC(I_i,j)$, the computational energy cost due to the j_{th} invocation of I_i, may depend on the input parameter values of I_i and differ for each invocation.

In Java, the effect of invoking an interface can be expressed in terms of the execution of JVM's 256 Java bytecode types, and its native methods. Bytecodes are platform-independent codes interpreted by JVM's interpreter, while native methods are library functions (e.g., `java.io.FileInputStream`'s `read()` method) provided by JVM. Native methods are usually implemented in C and compiled into dynamic link libraries, which are automatically installed with JVM. JVM also provides a mechanism for synchronizing threads via an internal implementation of a *monitor*. Therefore, we can model the energy consumption $iCompEC(I_i,j)$ of invoking an interface on a given JVM in terms of the energy costs of bytecodes, native methods, and *monitor* operations executed during the invocation. Unless two platforms have the same hardware configurations, JVMs, and OSs, the energy costs of each bytecode type, each native method, and a monitor operation will likely be different.

3.2. Communication Cost

Two components may reside in the same address space and thus communicate locally, or in different address spaces and communi-

1. We use the them "interface" in a broader sense than the language-level construct supported by Java. Our usage is consistent with component-based software engineering literature.

cate remotely. When components are part of the same JVM process but running in independent threads, the communication among the threads is generally achieved via native method calls (e.g., `java.lang.Object`'s `notify()` method). A component's reliance on native methods has already been accounted for in calculating its computational cost. When components run as separate JVM processes on the same host, Java sockets are usually used for their communication. Given that JVMs generally use native methods (e.g., `java.net.SocketInputStream`'s `read()`) for socket communication, this is also captured by a component's computational cost.

In remote communication, the transmission of messages via network interfaces consumes significant energy. Given the communication energy cost *iCommEC* due to invoking an interface I_i, and the total number b_i of invocations for that interface, we can calculate the overall communication energy consumption of a component c_1 with n interfaces (expressed in *Joule*) as follows:

$$cCommEC(c_1) = \sum_{i=1}^{n} \sum_{j=1}^{b_i} iCommEC(I_i, j) \qquad \textbf{Eq. 2}$$

In this equation, $iCommEC(I_i,j)$, the energy cost incurred by the j_{th} invocation of I_i, depends on the amount of data transmitted or received during the invocation and may be different for each invocation.

In our work, we focus on modeling the energy consumption due to the remote communication based on UDP. Since UDP is a much more light-weight networking protocol (e.g., it provides no congestion control, retransmission, and error recovery mechanisms) than TCP, it becomes prevalent more and more in embedded and resource-constrained computing domains [1,11]. Previous research [3,12] has shown that the energy consumption of wireless communication is directly proportional to the size of transmitted and received data. Based on these results, we model the communication energy consumption due to the j_{th} invocation of component c_1's interface I_i on host H_1 in terms of the size of transmitted and received data and the energy consumption of transmitting/receiving a unit of data on H_1. Note that once the platform-specific energy cost of sending/receiving a unit of data is profiled on each host by performing an offline measurement [8], the system parameters that need to be monitored on the host for estimating the communication energy cost are only the sizes of messages exchanged over the network.

3.3. Infrastructure Energy Consumption

Once the computational and communication costs of a component have been calculated based on its interfaces, its overall energy consumption is determined as follows:

$$overallEC(c) = cCompEC(c) + cCommEC(c) \qquad \textbf{Eq. 3}$$

However, in addition to the computational and communication energy costs, there are additional energy costs for executing a Java component incurred by JVM's garbage collection and implicit OS routines. During garbage collection, all threads except the Garbage Collection (GC) thread within the JVM process are suspended temporarily, and the GC thread takes over the execution control. We estimate the energy consumption resulting from garbage collection by determining the average energy consumption rate of the GC thread (*J/sec.*) and monitoring the total time that the thread is active (*sec.*).

Since a JVM runs as a separate user-level process in an OS, it is necessary to consider the energy overhead of OS routine calls for facilitating and managing the execution of JVM processes. There are two types of OS routines:

1. explicit OS routines (i.e., system calls), which are initiated by user-level applications (e.g., accessing files, or displaying text and images on the screen); and
2. implicit OS routines, which are initiated by the OS (e.g., context switching, paging, and process scheduling).

Java applications initiate explicit OS routine calls via JVM's native methods. Therefore, our computational model already accounts for the energy cost due to the invocation of explicit OS routines. However, we have not accounted for the energy overhead of executing implicit OS routines. Previous research has shown that process scheduling, context switching, and paging are the main consumers of energy due to implicit OS routine calls [10]. Therefore, we can estimate the overall infrastructure energy overhead *ifEC*(*p*) of a JVM process *p* in terms of the energy costs of the GC thread, process scheduling, context switching, and paging.

Since there is a singleton GC thread per JVM process, and implicit OS routines operate at the granularity of processes, we estimate the infrastructure energy overhead of a distributed software system in terms of its JVM processes. In turn, this helps us to estimate the system's energy consumption with higher accuracy. Unless two platforms have the same hardware configurations, JVMs, and OSs, the GC thread's energy consumption rate and the energy costs of process scheduling, context switching, and paging on one platform may not be the same as those on the other platform.

Once we have estimated the energy consumption of all the components, as well as the infrastructure energy overhead, we can estimate the system's overall energy consumption as follows:

$$systemEC = \sum_{i=1}^{cNum} overallEC(c_i) + \sum_{j=1}^{pNum} ifEC(p_j) \qquad \textbf{Eq. 4}$$

where *cNum* and *pNum* are, respectively, the numbers of components and JVM processes in the distributed software system.

3.4. Energy Consumption Estimation

In this section, we discuss how our framework can be used for estimating a distributed software system's energy consumption at the level of its components both during system construction-time and during runtime. Figure 2 shows the envisioned high-level process followed by a system engineer for estimating a distributed system's energy consumption using our framework.

In order to estimate a distributed system's energy cost at construction-time, we first need to characterize the computational energy cost of each component on its candidate hosts. To this end, we generate a large set of random inputs for a component's interface and invoke the interface with these inputs for calculating the energy consumption of the interface from our framework. If an interface's expected inputs are known at construction-time, we can use these inputs instead of a set of inputs generated randomly. To estimate the communication energy consumption of each interface, based on domain knowledge and types of input parameters and return values, we predict the average size of messages exchanged due to an interface's invocation. Using this data we can approximate the communication energy cost of interface invocation from

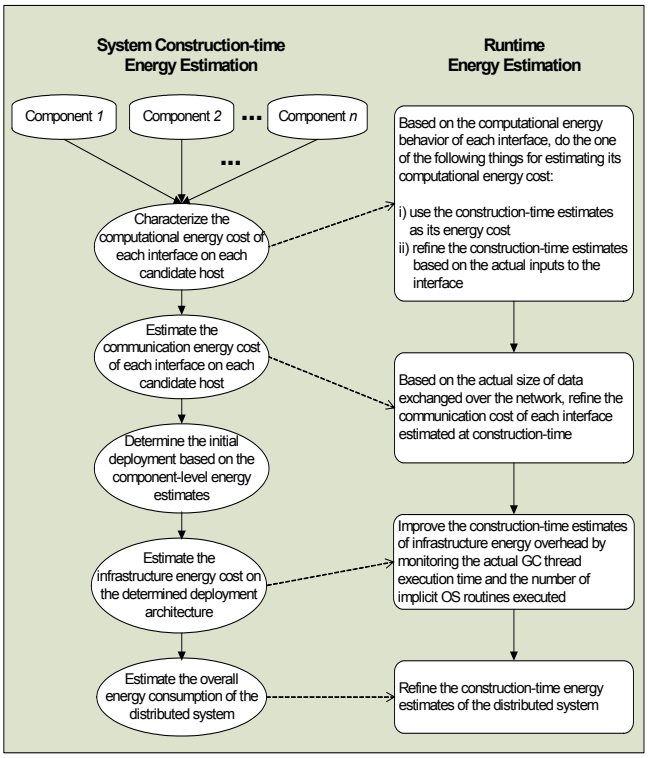

Figure 2. Energy consumption estimation steps at system construction-time and at runtime.

our communication energy model. Before estimating the entire distributed system's energy cost, we also need to determine the infrastructure's energy overhead, which depends on the deployment of the software (e.g., the number of components executing simultaneously on each host). Unless the deployment of the system's components on its hosts is fixed *a priori*, the component-level energy estimates can help us determine an initial deployment that satisfies the system's energy requirements (e.g., to avoid overloading an energy-constrained device). Once an initial deployment is determined, from our framework we can estimate the infrastructure's energy cost. We do so by executing all the components on their target hosts *simultaneously*, with the same sets of inputs that were used in characterizing the energy consumption of each individual component. Finally, we determine the distributed system's overall energy consumption based on the above energy estimates.

However, since the construction-time energy estimates are based on a system engineer's guesses or domain knowledge, they might be incorrect compared with the system's actual energy cost at runtime. Our framework can first refine the construction-time estimates of an interface's computational energy cost based on the actual inputs to the interface. For the communication cost of a component's interface, by monitoring the sizes of messages exchanged over the network, their effects on the interface's communication cost can be determined from our framework. In a similar way, our framework can improve the construction-time estimates of the infrastructure energy overhead based on the actual GC thread execution time and the number of implicit OS routines executed at runtime. Finally, based on these refined estimates, our framework can improve (possibly automatically) the construction-time energy estimates of a distributed system at runtime. Each dotted line in Figure 2 indicates the refinement step at runtime for the corresponding energy estimation step at system construction-time.

We have evaluated our energy estimation framework for a large number of distributed application scenarios by running them on top of Kaffe 1.1.5 JVM [5] on Compaq iPAQ PDAs. Our evaluation results suggest that the framework is always able to estimate the power consumed by a distributed Java system to within 5% of the actual consumption. For interested readers, refer to [8] that discusses our evaluation results in more detail.

4. CONCLUSION

In this paper we have presented a framework for estimating the energy consumption of Java-based software systems. Our primary objective in devising the framework has been to enable an engineer or a software agent to make informed decisions when adapting a system's architecture, such that the system's energy consumption is reduced and the lifetime of the system's critical services increases. Our framework explicitly takes a component-based perspective, which renders it well suited for a large class of today's distributed, embedded, and pervasive applications. The framework is applicable both during system construction-time and during runtime. In a large number of distributed application scenarios the framework has shown very good precision on the whole, giving results that have been within 5% (and often less) of the actually measured power losses incurred by executing the software. We consider the development and evaluation of the framework to be a critical first step in pursuing several avenues of further work, which has been identified as important in the areas of distributed, embedded, and pervasive systems. We have recently begun exploring, and successfully applying in an industrial setting, one such avenue for the framework.

5. REFERENCES

[1] W. Drytkiewicz, et al. pREST: a REST-based protocol for pervasive systems. In *Proc. of IEEE International Conference on Mobile Ad-hoc and Sensor Systems*, 2004.

[2] K. I. Farkas, et al. Quantifying the Energy Consumption of a Pocket Computer and a Java Virtual Machine. *ACM SIGMETRICS*, 2000.

[3] L. M. Feeney, et. al. Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment. In *Proceedings of IEEE INFOCOM*, 2001.

[4] JDDAC – Java Distributed Data Acquisition and Control. https://jddac.dev.java.net/, 2007.

[5] Kaffe 1.1.5. http://www.kaffe.org/, 2005.

[6] J. J. Labrosse. *MicroC/OS-II: The Real-Time Kernel.* CMP Books, 2002.

[7] S. Lafond, et al. An Energy Consumption Model for An Embedded Java Virtual Machine. *ARCS*, 2006.

[8] C. Seo, et al. An Energy Consumption Framework for Distributed Java-Based Software Systems. *Tech. Report, USC-CSE-2006-604*, 2006.

[9] A. Sinha, et al. JouleTrack - A Web Based Tool for Software Energy Profiling. In *Proceedings of DAC,* 2001.

[10] T. K. Tan, et al. Energy macromodeling of embedded operating systems. *ACM Trans. on Embedded Comp. Systems*, 2005.

[11] UPnP Device Architecture, http://www.upnp.org/, 2007.

[12] R. Xu, et al. Impact of Data Compression on Energy Consumption of Wireless-Networked Handheld Devices, *ICDCS*, 2003.

[13] H. Zeng, et al. ECOSystem: Managing Energy as a First Class Operating System Resource. *ACM ASPLOS*, 2002.