Improving Energy for Mobile Computers Through Process Migration Logging

Willard Thompson and Chuck Weddle
{wthompso, weddle}@cs.fsu.edu
Computer Science Department
Florida State University
April 20, 2004

Abstract: *In order to accelerate the efficiency of mobile computing, better solutions for conserving power for mobile computers need to be discovered. The battery power of mobile computers is barely tolerable and almost any amount of computation is difficult to sustain with such resources. In this research paper we explore finding an optimal logging block size for migrating processes on a mobile network in order to conserve power for a mobile computer. Bringing the process to the data ideally is more cost effective in terms of network communication and subsequently local computations than having the process remotely interact with a server in the case where a substantial amount of data is requested and used for computation.*

Keywords: mobile computing, power consumption, logging, process migration

1. Introduction

Mobile computing is one important feature of the ubiquitous computing paradigm that is becoming more pervasive in our lives. Mobile devices such as PDA's and cell phones are potentially becoming capable of providing applications to end users that are comparable to PC-based apps on a stationary network. Though, in order to bring mobile computing to fruition, efficient and robust services need to be provided for mobile users.

Mobile computing is not as reliable as PC-based computing within stationary networks. Current research literature has presented three significant and much needed research areas that are intrinsic to mobile computing [MS96, GF94]:

(1) *Resource consumption*, which includes power, size of memory, processor speed and disk capacity
(2) *Connectivity*, with respect to loosing signals from mobile users moving out of coverage range or via some form of interference
(3) *Security*, in terms of authentication and protecting data and transactions

Our focus is on proposing a software solution [JL98] to improving the power of mobile devices. [JL98] state that the challenge is to minimize energy consumption while not degrading performance.

We discuss the following within this paper: We end the introduction by further elaborating on our goals and state our hypothesis. Section 2 includes the related work. In section 3 we discuss the design and implementation of our framework called the Java Process Migration (JPM) system. In section 4 we discuss the results and measurements of the amount of power consumed for various experiments, and in section 5 we analyze those results. Finally in sections 6 and 7 we provide a conclusion, summary, and talk about future work.

1.1 Power Consumption

Because mobile computers use battery power, they are not able to sustain long periods of activity. Power is easily consumed with any significant computations, especially in conjunction with a windows-like graphical user interface. It is the computation issues in mobile computing that we concentrate our work.

In a client-server interaction, for instance a PDA communicating with a remote server, multiple requests are usually sent from a client to a server. The accumulation of network communications can be expensive, and hence can quickly drain the battery power of a client. However, migrating a process is one solution to reduce the communication between mobile clients and remote servers. Ideally, a client wishing to access the services of a remote server would package his intended process and send it to the server. While at the server the process can perform computations on

the server's data, and eventually return the pertinent results to the client. In this way, the client does not have to use its resources or power for any of the computations.

1.2 Research Goals

Since mechanisms for process migration currently exist, we aim to improve upon that concept. So, we hypothesize that the efficiency of migrating processes will be improved using block logging for processes. The idea of logging processes, in other words combining processes that need to be migrated, is to further minimize communication between mobile clients and servers. We build our own process migration environment using Java and test our solution on two Linux machines. We compare our results with process migration without logging.

2. Related Work

We have organized the related work into three main areas:

(1) Power Management
(2) Process Migration
(3) Logging

2.1 Power Management

Power management for mobile computers has traditionally been focused on hardware approaches [MO98]. Within the past decade, however, there has been a shift towards software solutions. In fact, [MO98] have proposed a load sharing solution for power conservation. Their idea is that jobs are transferred from a mobile client to a fixed server to decrease CPU utilization. They show that, via a simulation, the lifetime of a battery of a mobile computer increases by up to 20%, while using load sharing. Load sharing also speeds up the time to process jobs. [MO98] presents three algorithms to schedule jobs for load sharing. The goal of each of these algorithms is to determine if the cost to migrate a job becomes less expensive than the cost of executing a job locally.

In [RK98], a transport level protocol is designed and implemented for reducing power by suspending or completely shutting down communications for a period of time. During this suspended period the mobile device can do other relevant work, such as queuing data for future transmissions. Additionally, [JL98] discuss putting mobile devices into low power modes by decreasing their speed and functionality, temporarily. [RK98] have tested various communication suspension schemes, and show an 83% energy savings. In actuality, they measure their energy savings as 9% for laptops and about 40% for PDA's, while using their proposed protocol.

Another interesting facet of trying to reduce power consumption is the use of compression. Applying compression to bits before they are transmitted to a remote host can increase the energy of a mobile client [KB03]. In fact, [KB03] claim that transmitting a single bit, through the wireless medium, consumes a thousand times more energy than a single 32-bit calculation. Their premise is that as long as the energy for compressing data is less than the energy needed to send it then compression is ideal since it results in longer battery time for mobile computers [KB03]. Furthermore, they claim that there is an improvement of 51% in energy efficiency while using compression.

[SL01] claim that power management can be improved with a distributed power management design. They argue that event-driven operating systems are more efficient at conserving power than general operating systems. [SL01] compare two operating systems, albeit in the embedded environment, eCOS, a general purpose multi-tasking operating system, and TinyOS, which is event-driven. Their results show that TinyOS needs only half of the instruction memory and $^{1}/_{30}$ of the data memory. Ultimately, TinyOS demonstrates a factor of twelve improvement in power over eCOS. With respect to the distribution, [SL01] claim that integrating power management into the system level, architectural module level, circuit level, and the device level for better power efficiency. Adding global monitoring and module interfaces, where each architectural module has an interface that is responsible for its own local power management, will improve power efficiency.

Finally, [MA01] discuss improvements in energy usage through a modified dual-priority scheduling algorithm for real-time systems. They claim that energy can be saved by spreading process execution cycles up to the maximal time constraints allowed, by lengthening runtime tasks

from reducing processor speed. One of the main ideas of [MA01] is that when processes consume only 50% of their worst case execution time, their scheduling algorithm creates much less process idling time.

2.2 Process Migration

The process migration concept has been around for quite some time [DM00]. However, it is difficult to apply since, in general, it is not easy to provide the same resources that a given process needs to any node that it migrates to. In fact, some of the earlier work on process migration for operating systems was in [MP83], where no significant changes were needed to the operating system. No centralized algorithms were required, simple and common interfaces can be used independent of location, and the kernel did not have to change for sending and receiving messages [MP83]. Additionally, [AH89] discusses performance enhancement in distributed systems using process migration.

Since there is a lack of a generalized approach to process migration in operating systems, [TB02] have come up with a virtualized operating systems (vOS) approach by injecting functionality into running applications. [TB02] call this virtualization that decouples an application process from its physical environment. While sitting on top of Windows, vOS injects apps with virtualization software. The underpinning technology for virtualization is API interception. In essence, the indirection layer of library routines for API calls, allow the opportunity to capture, modify, and restore state information. The ultimate aim of vOS is to build neighborhoods of systems that cooperate together by running apps and sharing resources, while maintaining binary compatibility. Migration is permissible via the storing and usage of virtual handles by transparently re-mapping virtual handles to real handles during each process move. The authors conclude their work with a system performance analysis. They inject calc.exe with vIN DLL code, which virtualizes the target API. It took 526.2 ms to inject the process, and 1108.7 ms to perform the vIN setup and API interception. The process migration took 2,385.8 ms, which includes file write time. Finally, restoring the process took 2,172.0 ms.

Continuing on the virtual OS (*v*OS) scheme, [RN00] discusses injecting wrapper DLL's into apps at runtime for mobility. [RN00] focus on apps using sockets for migrating processes between machines to overcome the *application development barrier* gap that caused many distributed OS's to fail. The idea is to bridge this gap by enabling existing legacy apps to be executable on the newly distributed platform. [RN00] call this entire project *Computing Community* (CC) such that a group of systems is allowed to grow or shrink based on dynamic resource requirements through the transparent scheduling and migration of processes, without intrusive app re-design, re-coding, or re-compiling. The unobtrusive injection of *v*OS functionality is done by intercepting calls made from the app to the underlying runtime system and reinterpreting the call. For instance, in Windows NT this is done by wrapping Win32 API calls so that it can implement a variant, pass the call to the original API, or modify it to execute something different. Overall, the beauty of *v*OS enables process migration without changing the kernel, or app code for any platform, using interception mechanisms as described above.

[ME95] discuss process migration design issues for distributed operating systems in a loosely coupled environment with respect to system models, hardware platforms, migration methodologies, distributions of load, and transparency in networks. OS related considerations for when, where, how, and which process to migrate should be known. The most costly aspect of process migration is transferring the virtual memory. The most costly aspect of a process while in migration is the transferring of its state. [ME95] concludes that process migration overall is expensive and can be cumbersome to implement. Kernels that are smaller contain less information about processes, which implies minimal location dependencies of processes. [ME95] finally states that the ultimate decision of migrating a process is up to the distributed system itself.

More closely related to our experiment, [KR04] has proposed a message-passing interface that supports transparent Java process migration. Their solution is middleware that runs on top of the standard JVM that supports preemptive Java process migration and location-transparent communication services. Additionally, by using a Java Virtual Machine Debugger Interface, [KR04] claim that this solution is more portable than

existing solutions and less complex since no changes to the JVM are necessary.

Finally, another computing paradigm that has added to the process migration effort is mobile agents. Mobile agents are autonomous components of software that travel through a mobile network performing computations on and collecting data from various nodes. [DJ95] have investigated operating system support for mobile agents. Essentially, each agent is accompanied with a briefcase of folders containing a list of elements. These folders must be easy to transfer and hence cannot have elaborate index structures. Agents need to be able to read and write to local folders. Agents can make use of site-local folders to avoid excess baggage, when necessary. Also, agents cause other agents to execute via the meet command. Meet is analogous to a procedure call, where the briefcase is analogous to an argument list. Agents move from site to site by meeting with the local rexec agent.

2.3 Logging

Logging, in terms of clustering or grouping related entities together, is fairly new with respect to process migration. In fact, there is very few related work regarding logging processes. [AA00] and [SB04] are among the closest related work for logging migrating processes. [AA00] discusses forming cluster-heads for pockets of mobile nodes in an ad hoc network so that routing information can be minimally communicated. [SB04] talk about minimizing communication among mobile sensors within an ad hoc network. The idea is to cluster sensors so that they can communicate amongst each other and provide information hierarchically.

Given that not a lot of research literature exists for logging processes that migrate from node to node within a mobile network, we feel that we are providing a new mechanism for increasing the efficiency of process migration, by clustering processes to be migrated into a single transmission in order to further help conserve power on a client mobile device.

3. Experiment Constraints and Goals

The following are terms, acronyms, and definitions that will be used throughout the entire document.

- JPM is an acronym for Java Process Migration and is used to refer to this project.
- Java-PM is an acronym for Java-Process Migration and is used in the naming convention for the Java classes that make up the JPM solution.
- JPMP is an acronym for the JPM process.
- JPMD is an acronym for the JPM daemon.
- Java Virtual Machine will be referred to as the JVM.
- IPC is an acronym for Inter-process Communication.
- GUI is an acronym for Graphical User Interface.

There were some constraints that the project was operating within that directly affected what was done and how it was done. This project was to be completed in a short, finite time frame of 12 weeks. This was the amount of time available to determine what the project was going to be, design, implement, measure, and analyze. This time constraint led to several project imposed constraints that are identified below.

3.1 JPM System Well-Known Nodes

The JPM system has to have a set of nodes listed in a Java properties file that is read in upon system start up that defines the cluster of nodes that are running the JPM system. This is not realistic, the idea of knowing every other node that will be running the JPM system before the system is even started. This is a constraint put on the design of the system simply because how the nodes are found in the cluster that are running the JPM system is not important to the results that are trying to be obtained. Whether the nodes the processes are migrated to are taken from a predetermined list or properly sniffed out on the network does not impact the results this project is trying to find.

3.2 JPM Process Byte Size

The byte size or simply the size of the JPM process is important to know when logging processes together. Obtaining the actual byte size of a serialized Java object is not trivial and not supported by the Java libraries. Solutions do exist on how to determine this information but are tedious and CPU intensive. Again, how the byte

size is determined is not important to producing the results. The fact that a byte size is available that approximates the actual byte size of the processes running is what is important to be able to log processes. This is a constraint put on the design of the system. The Migrate getByteSize methods simple return an approximate size of the JPM processes.

3.3 JPM Process Round Trip Completion

The JPM processes that are migrated to a remote machine for execution are not ever returned to the machine that originally started them. This is certainly not realistic because the results of the executed process would be needed back on the machine that started the process. This is a constraint on the implementation of the system. Again, whether the results of the migrated process are returned to the original machine does not affect the measurements being taken in the experiments performed on the JPM system.

3.4 Java RMI

Java RMI was selected as the remote communication protocol to use between the JPM systems. Java RMI was selected because of its simplicity in implementation and due to the project time constraints, this simplicity was very attractive. Java RMI is expensive and has some impractical rules for migrating processes. Using Java RMI is a constraint imposed on the design and implementation of the system. In all likelihood, a communication protocol designed and built from scratch specifically for the purpose of migrating processes in logged process blocks would be more efficient.

3.5 JPM System Experiments

The experiments run on the JPM system were abbreviated due to time constraints. It would have been interesting to measure the JPM system in a larger cluster of nodes for a longer period of time than what was done. The experiments that were conducted were sufficient enough to show trends in the data. But more trends and trends of more interest might have been able to be discovered if more time was available.

4 Design

To show the potential power savings by migrating logged processes from one machine to another in a cluster of machines, we implemented a process logger and process migration system that this paper refers to as the JPM (Java Process Migration) system. There are two main components to the JPM system, one being the processes to be migrated and the second being the system that logs and migrates the processes from one machine to the next. This paper will refer to the processes to be migrated in the JPM system as JPM processes and will refer to the system that logs and migrates processes to other machines the JPM daemon.

The design discussion first covers the design details regarding the JPM processes and secondly covers the design of the JPM daemon.

4.1 JPM Processes

Any process that will potentially be logged, migrated, and executed on another machine in the JPM system will need to “plug” into the JPM process framework. By having the processes extend the JPM framework, a protocol of communication can be established that will allow for the process to be migrated and executed on another machine in the cluster of known machines.

A JPM process will register itself with the JPM daemon when it starts. The JPM Daemon will determine if the process should be migrated. For the purposes of our experiments, all processes that register with the JPM daemon are migrated. This is certainly an area that could be explored as to how to determine which processes are well suited for process logging and migration, which is noted as such in the Future Work section.

Once the process has registered, the JPM daemon will tell the process to prepare itself for process suspension and serialization. The application writer can implement any details specific to that process for preparation for suspension by simply implementing a certain method exposed by the JPM process framework. This is discussed in more detail in the Implementation section.

After the process is suspended, the JPM daemon will pass the serialized process to another machine for execution.

Figure 1 captures the essence of the communication between the process to be migrated and the JPM daemon.

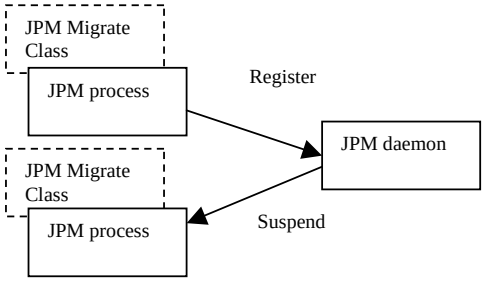

Figure 1*. Communication between process and JPM daemon.*

Figure 1 shows two processes interacting with the JPM daemon. The top process has just started and is registering itself with the JPM daemon so that it is known and can be considered for migration. The bottom process has already registered with the JPM daemon and will be migrated. The JPM daemon is telling this process to suspend itself to prepare for migration.

4.2 JPM Daemon

The JPM daemon is responsible for the management of the processes that are to be migrated to different nodes in the cluster for process execution so that energy improvements can be realized.

The JPM daemon handles the logging of process to be migrated for efficient transmission. The JPM daemon is the heart of the system, communicating with process locally and other JPM daemons within the cluster of known machines.

The JPM daemon is made up of several components that work together to log processes together and migrate these processes to other machines.

Figure 2 shows the logical design of the Java-PM daemon and the different components that make it what it is.

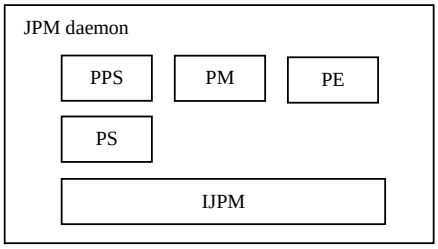

Figure 2. *Logical design of JPM daemon.*

```
JPM daemon Inner Components

PPS  - Pre-process Scheduler
PS   - Process Logger
PE   - Process Execution
PM   - Process Monitor
IJPM - Inter Java-PM daemon
       Communicator
```

Listing 1. *JPM daemon Inner Components.*

4.2.1 *Pre-Process Scheduler*

The Pre-Process Scheduler (PPS) component is responsible for handling the JPM process registration requests as well as determining if the process should even be migrated. This component will check the status of the process and examine its information. When the PPS component has a process to migrate, it suspends the process. In return the PPS receives the serialized process bytes from the now suspended process. The PPS is now finished working with the process and hands it off to the Process Logger.

4.2.2 *Process Logger*

The Process Logger (PS) component is responsible for grouping process to be migrated in the most efficient manner for optimal transmission. If two processes are being migrated to the same node in the cluster, the PS will send them together. This is the job of the PS component.

The PS produces process blocks that are a collection of processes to be transmitted to the same node. Only the cost of transferring the process block and not each individual process will be incurred. This results in less transmitted bytes and a more efficient solution. The results of our experiments show this effect. In fact, the size of the process block is adjustable real-time through the Process Monitor. This allows for experimentation

with the size of the process block to find optimal behavior.

4.2.3 *Process Execution*

The Process Execution (PE) component is responsible for resuming a migrated process on the local node. The PE component also handles completion of a migrated process once a process has come back from the migrated node.

4.2.4 *Inter-JPM Communicator*

The Inter-JPM Communicator (IJPM) is responsible for all communication with the other JPM daemons on other nodes in the cluster. The IJPM encapsulates the logic necessary for IPC communication and exposes convenient methods for each of the components to do what they need to do.

4.2.5 *Process Monitor*

The Process Monitor (PM) allows for control over the JPM daemon. The PM is a graphical user interface into the JPM daemon. The PM shows real-time information about the state of the JPM daemon. This information includes the number of process being migrated, the number of process blocks that have been built, etc. Please see the JPM daemon GUI guide in the Appendix that explains this important information about the PM.

The PM also allows for the user to actually stop and start certain components operation. This is part of the design to help run the experiments. Specifically, the Pre-Process Scheduler can be turn off so that none of the processes can register for migration with the JPM daemon and have to execute locally. Also, the Process Logger can be turned off so that the processes are migrated to other machines for execution but not in process blocks.

5 Implementation

After considering the design of the JPM system, we decided that Java would be the most appropriate language for implementation. Java Remote Method Invocation (RMI) allows for a straightforward implementation of a remote communication protocol. Also, Java provides a library of classes that handle most of the details of working with sockets. Lastly, Java is a portable language and the JPM system can be written once in Java and run on different platforms. All of these facts taken into consideration made Java a clear choice for implementing the JPM System. Hence the name Java Process Migration or JPM.

Following the design of the JPM system, the implementation can clearly be separated into two parts. The first part is the JPM process framework implementation that allows for Java processes to "plug" into the JPM system. The second part is the JPM daemon implementation. The JPM daemon implementation can further be broken down into the subcomponent implementations.

For a complete file listing of the source code that makes up the JPM system please view chart A in the appendix.

5.1 JPM Process Implementation

The JPM Daemon needs to be able to control the processes that it is going to migrate to another machine. This control is implemented through the Migrate class. Every Java process that wants to "plug" into the JPM system needs to extend the Migrate class.

The Migrate class implements the communication protocol between the process and the JPM daemon. The following methods that are implemented by the Migrate class are most important to the JPM process implementation and are worth discussing.

5.1.1 *protected void execute(Migrate mp)*

The execute method is called in the constructor of the Java process extending the Migrate class. This method opens a socket for communication with the JPM daemon through a well known JPM port. This method then registers the process with the JPM daemon and continually monitors the socket for future commands from the JPM daemon. If the process cannot register with the JPM daemon for any reason then the process executes locally.

5.1.2 *private void processCommand (JpmCommand cmd)*

The processCommand method takes a JpmCommand object as its only argument. When the monitoring of the socket receives a command from the JPM daemon the processCommand

method is called with the command received as the argument.

This method examines the JpmCommand for the command that was sent and makes the appropriate method call associated with that command.

To view the table of commands defined in JpmCommand, all of which Migrate needs to be able to handle, please view chart B in the appendix.

The following class is a trivial example of a JPM process yet shows the little implementation that is necessary for a JPM process to "plug" into the JPM system. This is achieved through the encapsulation of logic for registration, JPM daemon command processing, etc. in the Migrate class.

```
// Java Process Migration (JPM)
//
// Charles Weddle & Willard Thompson
//    @ Florida State University
//
// copywrite 2004, all rights reserved

import edu.fsu.jpm.jpmp.Migrate;

public class example extends Migrate {

  public static void main (String args[]) {

    example mp = new example();
    mp.setProcessName("Example");
    mp.execute(new example());
  }

  public void run() {

    int count = 0;

    try {

      while (++count < 10) {

        writeOutput("EXAMPLE: " + getJpmdName()
         + ": beep! beep!: " + count);
      sleep(1000);
      }
    }
    catch (InterruptedException exception) {

      exception.printStackTrace();
    }
  }

  protected void suspendProcess () {

  // prepare THIS process for migration, if
  // needed.
  }

  public int getByteSize() {

   return 3072; // 3k
  }
}
```

Notice that the example class extends the Migrate class. Also notice in the constructor two instances of the example class are instantiated. The first instantiated object is the controlling object, the one with which the JPM daemon will communicate. The second instantiated object is the one that is passed to the execute method and will be logged and eventually migrated to another machine for execution.

To view the table of the sequence of events during the registration of a JPM process with the JPM daemon see chart C in the appendix.

In the event that the JPM daemon does not deem the process as "migratable" and rejects its registration request, the JPM daemon would simply issue a start command to the JPM process and proceed to listen for other registration requests. The JPM process, upon receiving the start command, would know that it was not registered with the JPM daemon for migration and would execute itself locally.

5.2 JPM Daemon Implementation

The JPM daemon is really a collection of programs working together towards the goal of migrating processes. The main program, jpmd, is nothing more than a Java main method that starts all of the subcomponent threads appropriately. The interesting discussion pertaining to the JPM daemon lies within the implementation of the subcomponents.

Before discussing the implementation details of the subcomponents, it is worth mentioning how important JpmSharedMemory is to making the communication between all of the JPM daemon subcomponents work properly. JpmSharedMemory is a singleton object in the JPM daemon. The jpmd main method instantiates one JpmSharedMemory object that is referenced by all of the other jpmd subcomponents. The most important part of JpmSharedMemory are the collections it encapsulates and the interface it exposes to manipulate those collections. Below is a list of the different collection found in JpmSharedMemory and how each collection is used.

5.2.1 *Nodes Collection*

The nodes collection in shared memory contains all of the known nodes in the cluster of nodes that the processes might migrate to. The nodes collection is initially populated in the main method of jpmd from a Java properties flat file. This list of nodes in the properties file identifies the well known nodes in the cluster of nodes. For each node identified in this list, jpmd instantiates a new JpmNode object and pushes it on the nodes collections. Obviously, the nodes in the cluster running the JPM daemon would not necessarily be known at start up. This is an operating constraint identified in the Experiment Constraints section for good reason.

The subcomponent jpmd-ijpm-nodemgr uses this collection to actively identify potential nodes running JPM daemon and are in good status. This subcomponent pings these well known nodes every two seconds and checks their status.

The jpmd-ijpm subcomponent uses this collection to find a node that is active and available to migrate a process block to. Currently, this subcomponent simply selects the first available node that is active but it would be simple to update this to use a more sophisticated selection process.

5.2.2 *Pre-Processes Collection*

The pre-processes collection in shared memory contains all of the processes that have been registered by the jpmd-pps subcomponent. When the jpmd-pps receives a registration request and deems that process to be "migratable", this subcomponent instantiates a new JpmProcess object and pushes it on the pre-process collection.

The jpmd-ps subcomponent uses this list to know what processes are waiting to be logged into a process block for migration. The jpmd-ps will pop off the JpmProcess objects on this list when they become available. The jpmd-ps then logs that JpmProcess object into a JpmProcessBlock accordingly.

5.2.3 *Process Blocks Collection*

The process-blocks collection in shared memory contains all of the process blocks that are waiting to be migrated to another machine for execution. When the jpmd-ps subcomponent has a full JpmProcessBlock, the jpmd-ps pushes this JpmProcessBlock onto the process blocks collection.

The jpmd-ijpm subcomponent pops off JpmProcessBlocks in the process block collection when they are available. The jpmd-ijpm then migrates these JpmProcessBlocks to another machine.

5.2.4 *Migrated Process Block Collection*

The migrated process blocks collection in shared memory contains the process blocks that have been migrated to this machine. The RMI JpmdIjpmImpl pushes the migrated process blocks onto the migrated process block collection when it receives them from a remote JPM daemon.

The jpmd-pe subcomponent pops off the JpmProcessBlock objects on the migrated process blocks collection and executes each process contained in each process block.

5.3 JPM Daemon Subcomponents Implementation

5.3.1 *JPM Daemon Pre-Process Scheduler Subcomponent (jpmd-pps)*

The jpmd-pps subcomponent handles everything related to the JPM processes. This includes listening on a well know port via a socket for process registration requests, processing registration requests with the JPM system, determining if the process is "migratable", controlling the process through commands defined in JpmCommand, instantiating a new JpmProcess for every registered process and placing that JpmProcess on the pre-processes collection in shared memory, and finally obtaining the serialized process object and associating those object bytes with the appropriate JpmProcess object.

The processClient method implemented in the jpmd-pps subcomponent is most interesting. This method handles all of the requests received over the well known port from JPM processes wanting to register with the JPM system. This method first checks to see if the pre-process toggle is in a positive position before proceeding. If this toggle were in a negative position then the jpmd-pps would not even attempt to register any processes and instruct all process to run locally. This toggle was put in place to help run the experiments on the

JPM system. This is explained in more detail in the JPM Experiments section.

Next this method registers the process by instantiating a JpmProcess for it and setting the appropriate attributes. Next this method asks that the process suspend itself and send its serialized object bytes. A reference to the serialized object is set on the JpmProcess object for this process. Finally this method commands the process to exit gracefully and pushes the JpmProcess on the pre-process collection in shared memory.

5.3.2 *JPM Daemon Process Logger Subcomponent (jpmd-ps)*

The jpmd-ps subcomponent handles logging together the processes for migration. The most interesting method implemented by the jpmd-ps is the InflateList method.

The InflateList method continually examines the pre-process collection in shared memory. When a JpmProcess object is available, this method places this process in a JpmProcessBlock. It does so considering the size, in bytes, of the processes currently in the most recently instantiated JpmProcessBlock object.

The most important factor here is the maximum process block size. If the size of the current process is smaller than the size of the maximum process block size less the size of all the processes in the current process block, then the process is added to the current process block. If the size of the process is greater, then the current process block is closed and pushed on the process block collection in shared memory and a new JpmProcessBlock object is instantiated for which the process in waiting is added to.

The maximum process block size can be set dynamically through the JPM daemon GUI. This dynamic setting of the maximum process block size was put in place so that the appropriate experiments could be run on the JPM system.

5.3.3 *JPM Daemon Inter-JPM daemon Communicator Subcomponent (jpmd-ijpm)*

The jpmd-ijpm sub component handles the communication between the other JPM daemons running on the other nodes in the cluster. To help with the management of the other JPM daemons on the other nodes, this subcomponent has its own subcomponent, the jpmd-ijpm-nodemgr. The responsibilities associated with the jpmd-ijpm-nodemgr are discussed next.

The most interesting method implemented by the jpmd-ijpm is the thread run method. Within this method the jpmd-ijpm continually looks for JpmProcessBlocks on the process blocks collection in shared memory. When a process block is available, this method finds an available node and migrates this process block via RMI to a remote JPM daemon on another node.

5.3.4 *JPM Daemon Inter-JPM daemon Node Manager Subcomponent (jpmd-ijpm-nodemgr)*

The jpmd-ijpm-nodemgr subcomponent performs several administrative details so that the jpmd-ijpm can communicate via RMI with the other JPM daemons executing on remotes nodes. This subcomponent also keeps an active status of the other nodes running in the cluster by periodically checking the status of those other nodes.

5.3.5 *JPM Daemon Process Execution Subcomponent (jpmd-pe)*

The jpmd-pe subcomponent handles the execution of the migrated processes on the remote node. This subcomponent continually looks for migrated JpmProcessBlock objects on the migrated process blocks collection in shared memory. When a migrated process block is available, this subcomponent iterates through all of the processes in the process block and executes them on the remote node.

5.3.6 *JPM Daemon Process Monitor Subcomponent (jpmd-pm)*

The jpmd-pm implements the GUI for the JPM system. The Java Swing class libraries were used to implement the JPM daemon GUI. The GUI displays the current status of the JPM daemon. This includes the status of the nodes in the well known cluster of nodes. As well as the status of the pre-processes, process blocks, and migrated process blocks collections in shared memory.

For experimentation, the GUI has a toggle button for the pre-processor toggle and the process logger toggle. The maximum process block size can also be set through this GUI.

5.4 JPM Experiment

To be able to perform experiments on the JPM system and produce results to show that logging processes together before migration can save energy, we had to build a separate application to conduct the experiments. JPM Experiment is the application that was built to conduct the experiments. JPM Experiment consists of a Java program with a GUI front end that starts JPM processes. The JPM processes are started at a specific interval that can be dynamically set from the GUI. The number of JPM processes that are started can also be set from the GUI. The GUI displays the delay interval and sample size, the number of JPM processes to start, along with the count of each type of process that has been started.

6 Results and Measurements

Our results are organized into the three corresponding categories: Network Efficiency, Disk IO, and CPU Utilization. We ran a total of nine experiments for each of the three categories above. The first experiment consisted of not having the pre-processor on. All of the other eight experiments consisted of having the pre-processor on. The second experiment, however, had the process logger turned off and each subsequent experiment had the process logger turned on and incremented the process logger block size by two kilobytes intervals starting with 3K up to 15K. All intervals were arbitrarily chosen. The following charts are the averages of 300 runs for each experiment:

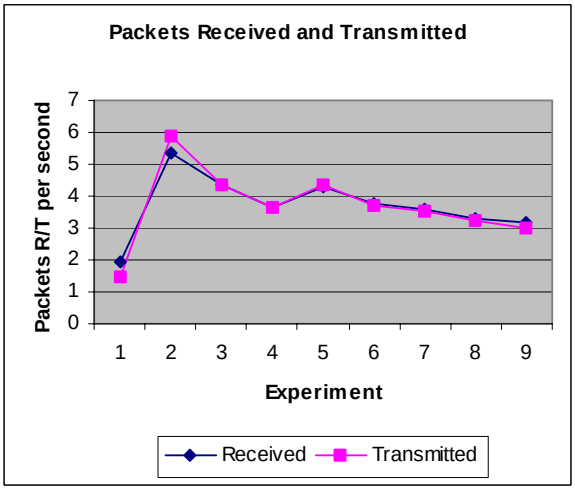

***Chart 1**. Packets received and transmitted.*

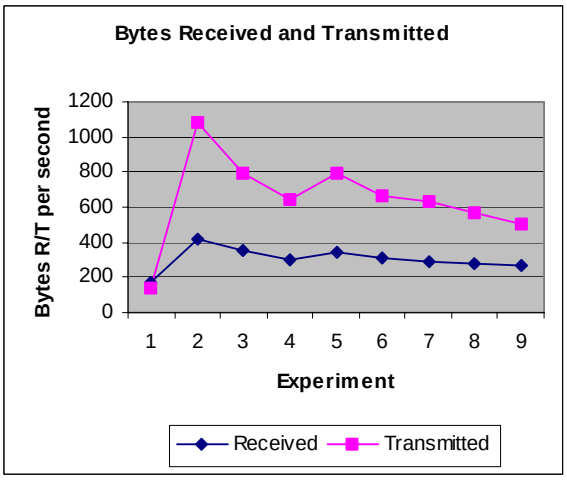

***Chart 2**. Bytes received and transmitted.*

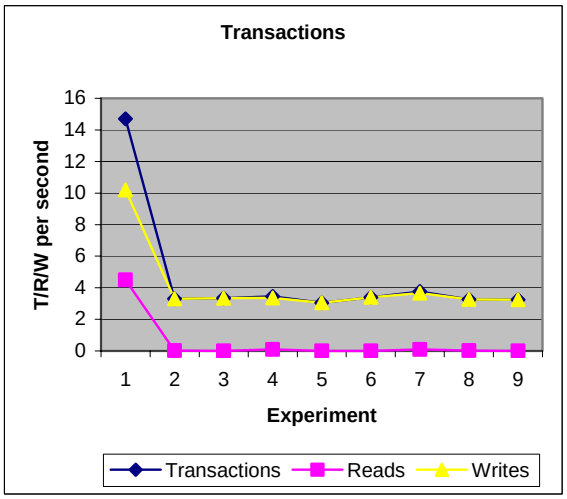

***Chart 3**. Disk reads/writes per second.*

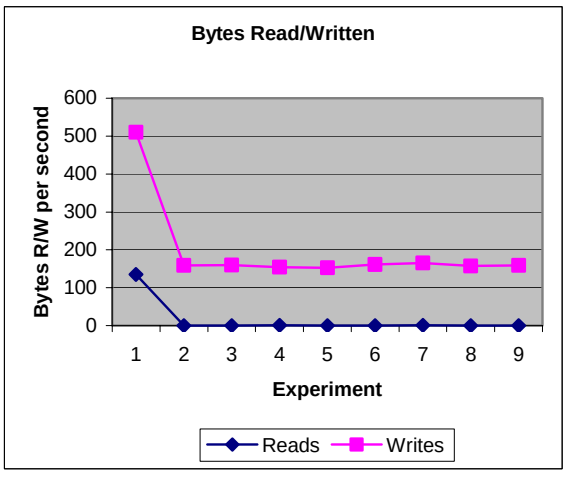

***Chart 4**. Bytes read/written to disk.*

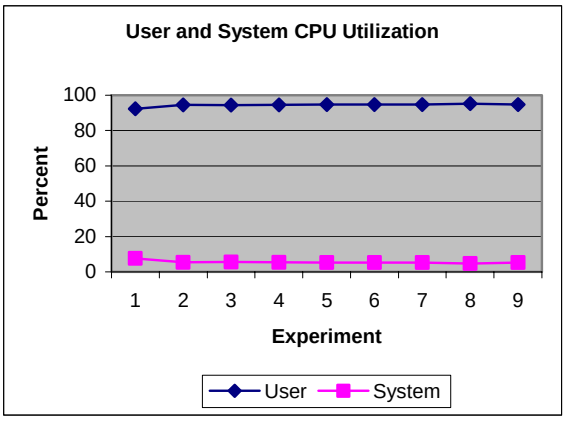

***Chart 5**. User and System CPU Utilization.*

Notice that charts 1 and 2 correspond to network efficiency. Charts 3 and 4 are for disk I/O, and chart 5 is for the CPU category.

The following is the criteria for each experiment along the horizontal axis of each chart:

1. Pre-processor off.
2. Pre-processor on; Process logger off.
3. Pre-processor on; Log block size 3K.
4. Pre-processor on; Log block size 5K.
5. Pre-processor on; Log block size 7K.
6. Pre-processor on; Log block size 9K.
7. Pre-processor on; Log block size 11K.
8. Pre-processor on; Log block size 13K.
9. Pre-processor on; Log block size 15K.

Each of the 300 runs consisted of a migrated process. Each migrated process was an instantiation of one of three possible processes:

(1) ProcOne (p1): Writes out a set of even numbers, where each number is delayed by a .5 second.
(2) ProcTwo (p2): Simulates interactions of a customer account, where each action is made every second.
(3) ProcThree (p3): Randomizes an array of elements and sorts them using selectionsort every second.

The following table contains the additional constraints for each experiment. It covers items such as the total number of process blocks, migrated processes and the division of the 300 processes.

Exp.	Blocks	Proc. Per blk.	PBS	Delay Interval	p1	p2	p3
1	0	0	5120	1000	0	0	0
2	300	1	5120	1000	84	87	119
3	169	1.78	3072	1000	102	101	97
4	124	2.42	5120	1000	111	91	98
5	183	1.64	7168	1000	103	94	103
6	130	2.31	9216	1000	98	106	96
7	109	2.75	11264	1000	84	114	102
8	98	3.06	13312	1000	118	87	95
9	76	3.95	15360	1000	104	93	103

Blocks: Total number of logged blocks.
PBS: Each process block size in KB.

***Table 1**. Additional experiment constraints.*

Finally, in order to flush out any extra system activity we began collecting each of our samples with a 30 second delay. In other words, for the first 30 seconds the JPM daemon was idle, and then after the 30 second mark we formally began migrating processes. However, the initial 30 seconds of idle time was included into our average measurements.

7 Analysis of Results

Overall, our results have shown improvement in network efficiency. However, unexpectedly, changes in Disk I/O and CPU utilization were barely noticeable.

In all of the charts, except for chart 5, there was a drastic change from experiment 1 to experiment 2, since respectfully the Pre-processor went from off to on. This of course performed as expected since no processes were migrated in experiment 1, which means that less network traffic was observed in charts 1 and 2 for experiment 1, and more disk I/O transactions were performed for experiment 1 as shown in charts 3 and 4.

For network activity, in experiments 2 through 9, it is shown that, except for experiment 5, there is a steady decrease in the number of packets and subsequently bytes transmitted per second. Experiment 5 could have been an anomaly. Regardless, this certainly has performed as expected since we gradually increased the block size for each experiment.

Disk I/O transactions have shown to be fairly steady for each of the eight experiments after the first experiment. Of course, writes require higher

throughput than reads, which explains why the line for writes is higher than the line for reads in charts 3 and 4. Overall, though, Disk I/O showed no significant signs of improvement.

The CPU utilization was perhaps the most disappointing measurement. As shown in chart 5, user and system level utilization was steady for all experiments. Since our process migration process was written in Java running on top of the JVM, our experiments were run at the user level. Perhaps not having complete control over the Linux machines might have interfered with any kind of isolation. For instance, there may have been other processes running on our machines that we were not aware of.

8 Conclusions

Considering the nature of our experiments it would be rather premature to arrive at any solid conclusions. The measurements for the network traffic were probably our most accurate measurements since we were able to have more control over the network medium. The measurements for Disk I/O may have indeed been accurate since there was a significant drop off in transactions and bytes read/written per second from the first experiment.

Therefore, we can conclude that, based on our experiments process migration at the user level will considerably improve mobile computing network efficiency.

8.1 What Was Learned

One very important lesson that was learned throughout the process of designing, implementing, and experimenting with the JPM system is to clearly define what it is that is to be measured and how to measure it. An elegant design and a well built software application is tremendous but is not worth much if it does not show the results that are desired.

Also, make sure it is clear how the measurement will be taken and how the data obtained from the measurements can show the desired results. Make sure that these details are well thought out before any designing or implementation begins. Make sure to allot enough time in the schedule to perform the experiments and analyze the data. Data analysis can be a time consuming event.

For additional insight, more of our work and project documents can be found at: http://ww2.cs.fsu.edu/~weddle/jpm/index.html.

9 Summary and Future Work

Due to time and manpower constraints some features of the JPM system were never fully implemented. These features that were left incomplete could certainly be implemented and experimented with in future work on process logging and process migration.

9.1 Process Caching

One potentially energy saving feature that the JPM system had intended to implement but did not was process caching. The idea is that processes that had been previously migrated to another machine could be cached. In the event that the same process or type of process were to be migrated to that machine again, the actual process object bytes would not have to be transferred. The cached process on the remote machine could simply re-execute on the remote machine. This would result in less bytes being transferred that could ultimately lead to less energy usage.

9.2 Optimal Process Scheduling for Process Migration

Another potentially energy saving feature that the JPM system did not implement but had intended to was a process scheduler that was optimized for processes to be migrated for execution. For example, this optimal process scheduler could consider processes that have been cached or similar process types as criteria for scheduling. This is a potentially interesting study of how best to schedule processes to be migrated. This study could answer the question what in fact are the criteria that are worthwhile to consider when scheduling processes to be migrated when process logging and process caching are in place.

Some more granular future work items include updating the current JPM system to actually examine a JPM process to determine if that process is a good candidate to be migrated or not. The type of criteria used to determine if one process is more “migratable” than another would have to be explored.

Another small feature to enhance the JPM system would be to actually rate the worthiness of the nodes in the cluster of nodes running the JPM system. Also, have the JPM system be able to sniff out the nodes in the network that are running the JPM system instead of having to supply a list of well-known and trusted nodes to the JPM system.

10 Acknowledgement

We would like to thank Louis Brooks, who is the systems administrator in the SAIT Laboratory, for letting us exclusively use a couple of Linux computers.

References

[AA00] A. Amis and R. Prakash, "Load-Balancing Clusters in Wireless Ad Hoc Networks", Proceedings of the 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET'00), 2000.

[AH89] A. Hac, "A Distributed Algorithm for Performance Improvement Through File Replication, File Migration, and Process Migration", IEEE Transactions on Software Engineering, Vol. 15, No. 11, 1989.

[DJ95] D. Johansen, et. al., "Operating System Support for Mobile Agents", Proceedings of the 5th Workshop Hot Topics in Operating Systems (HotOS), pp. 42 – 45, 1995.

[DM00] D. Milojicic, "Process Migration", ACM Computing Surveys, Vol. 32, No. 3, pp. 241 – 299, 2000.

[EP02] E. Pinheiro, et al., "Dynamic Cluster Reconfiguration for Power and Performance", Kluwer Academic Publishers, 2002.

[GF94] G. Forman, J. Zahorjan, "The Challenges of Mobile Computing", IEEE Computer Society Press, 1994.

[JH01] J. Hom, U. Kremer, "Energy Management of Virtual Memory on Diskless Devices", In Proceedings of the Workshop on Compilers and Operating Systems for Low Power, September 2001.

[JL98] J. Lorch, A. Smith, "Software Strategies for Portable Computer Energy Management", IEEE Personal Communications, 1998.

[JS98] J. Smith, "A Survey of Process Migration Mechanisms", ACM SIGOPS Operating Systems Review, 1998.

[KB03] K. Barr, K. Asanovic, "Energy Aware Lossless Data Compression", Proceedings of MobiSys 2003: The First International Conference on Mobile Systems, Applications, and Services, 2003.

[KR04] K. Ricky, et. al., "M-JavaMPI: A Java-MPI Binding with Process Migration Support", 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02), 2004.

[MA01] M. Angles Moncusi, et. al., "A Modified Dual-Priority Scheduling Algorithm for Hard Real-Time Systems to Improve Energy Savings", 10th International Conference on Parallel Architectures and Compilation Techniques (PACT '01), 2001.

[MB93] M. Bender, et. al., "Unix for Nomads: Making Unix Support Mobile Computing", USENIX Mobile and Location-Independent Computing Symposium, 1993.

[ME95] M. Eskicioglu, "Design Issues of Process Migration Facilities in Distributed Systems", Scheduling and Load Balancing in Parallel and Distributed Systems, IEEE Press, pp. 414 – 424, 1995.

[MF03] M. Franz, "A Fresh Look at Low-Power Mobile Computing", Kluwer Academic Publishers, Norwell, MA, USA, pp. 209 – 219, 2003.

[MN94] M. Nuttall, "A Brief Survey of Systems Providing Process or Object Migration Facilities", Operating Systems Review, Vol. 28, No. 4, pp. 64 - 80, 1994.

[MO98] M. Othman, S. Hailes, "Power Conservation Strategy for Mobile Computers Using Load Sharing", 5th ACM Conference on Computer and Communications Security, 1998.

[MP83] M. Powell, B. Miller, "Process Migration in DEMOS/MP", ACM Symposium on Operating Systems Principles, pp. 110 – 119, 1983.

[MS96] M. Satyanarayanan, "Fundamental Challenges in Mobile Computing", Symposium on Principles of Distributed Computing, pp. 1 – 7, 1996.

[NA01] N. AbouGhazaleh, et al., "Toward the Placement of Power Management Points in Real Time Applications", In Porceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'01), September 2001.

[RK98] R. Kravets and P. Krishnan, "Power Management Techniques for Mobile Communication", International Conference on Mobile Computing and Networking, 1998.

[RN00] R. Nasika, P. Dasgupta, "Transparent Migration of Distributed Communicating Processes", International Conference on Parallel and Distributed Computing Systems, 2000.

[SB04] S. Bandyopadhyay and E. Coyle, "Minimizing Communication Costs in Hierarchically-Clustered Networks of Wireless Sensors", The International Journal

of Computer and Telecommunications Networking archive, Vol. 44, Issue 1, pp. 1 - 16, 2004.

[SL01] S. Li, et. al., "Low Power Operating System for Heterogeneous Wireless Communication System", 10th International Conference on Parallel Architectures and Compilation Techniques (PACT'01), Barcelona, Spain, September 2001.

[TB02] T. Boyd and P. Dasgupta, "Process Migration: A Generalized Approach Using a Virtualized Operating System", Proceedings of the 22nd International Conference on Distributed Computing Systems ICDCS, 2002.

[TS03] T. Sato, I. Arita, "Constructive Timing Violation for Improving Energy Efficiency", Kluwer Academic Publishers, Norwell, MA, USA, pp. 137 – 153, 2003.