FrAPPE: A Framework for the Analysis of Power Consumption in Peer-to-Peer Environments

by
Benjamin John King

Submitted in Partial Fulfillment of the Requirements
for the Degree of
Master
in the
Computing and Information Systems
Program

YOUNGSTOWN STATE UNIVERSITY
May, 2013

FrAPPE: A Framework for the Analysis of Power Consumption in Peer-to-Peer Environments

Benjamin John King

Signature:

Benjamin John King, Student Date

Approvals:

Graciela C. Perera, Thesis Advisor Date

Edmund B. Ickert, Committee Member Date

John R. Sullins, Committee Member Date

Bryan W. DePoy, Interim Dean of School of Graduate Studies & Research Date

Abstract

One of the most popular Internet applications is peer-to-peer (P2P) networking. P2P networking is an application in which multiple computers communicate over a network to share resources, such as files. The effectiveness of P2P networks is enhanced when multiple computers host a similar resource; this is called redundancy. Because its architecture requires computers to remain powered on for file sharing and redundancy, much power is consumed to operate these networks. One way in which researchers are working to solve this problem is by developing heuristics for reducing the amount power consumed by computers in P2P networks. While heuristics are best tested on real networks, doing so can be both challenging and expensive. This paper describes a solution in FrAPPE, a program in which heuristics may be tested in a simulated networking environment. In addition, FrAPPE compiles statistical information regarding the power consumption and computational expense of heuristic trials.

As a simulated networking environment, FrAPPE also doubles as a teaching tool for computer networking education. A number of different software tools exist that are often used to teach computer networking concepts in academic environments. No programs known to the author, however, illustrate the costs incurred by computer networking hardware. FrAPPE offers a simple environment that demonstrates basic networking concepts, and, through its statistical output, helps students understand the aforementioned costs.

CONTENTS

I. Introduction

Professionals in both industry and academia have become interested in reducing the carbon footprint and energy consumption of existing technologies. One area in particular that has gained attention is green computing. Studies show that at least 10% of all power consumed by technology in information and communications comes from computing machinery. One highly popular, up-and-coming application in computing is P2P networking, among the best known uses of which is file sharing. Millions of desktop computers are engaged in P2P networking activities each day. For P2P activities to be fulfilled in an effective manner, these computers must remain powered on 24 hours per day, 365 days per year. With so many computers involved in an application that requires the continuous consumption of power, and with this application gaining popularity, this particular application is one that is bound to increase in its overall use of power. Thus, it is important to devise mechanisms that will analyze heuristics to reduce the power consumption in P2P applications. Many approaches are being taken in effort to reduce power consumption in P2P networks. One specific approach is the exercise of heuristics in which power consumption is reduced by putting to sleep any computers on the network that need not be powered on.

To ensure the effectiveness of such heuristics, it is best that they be put into practice and tested on a live network. The most reliable test that can be performed is to test on a real network. There are, however, a number of drawbacks to testing on a real network. One problem is that testing on a real network can be costly, as one would need to either purchase or gain access all computer and networking hardware needed. Another problem is that configuring the network would be a cumbersome and time-consuming task, as each individual computer on the network would need configured individually. Depending on the nature of the experiment, additional time may also be needed in re-configuring each individual computer between trials. Finally, to achieve the desired topology, one may need to engage in a complicated arrangement to configure computers across a number of different networks, which may potentially span a large geographic region.

In lieu of testing on a real network, one can test using simulation. A number of software packages have been developed in which one can simulate networks with greater ease than their physical counterparts. As a result, trials can be conducted in greater swiftness and efficiency. While existing simulations provide increased efficiency, we have not yet found simulations with the capability to measure power consumption in P2P applications. A new mechanism, FrAPPE, has been developed to provide the capability lacking in other simulations to measure power consumption P2P file sharing applications (i.e. P2P applications).

FrAPPE offers two contributions. The first is a simulation to allow the testing of P2P application heuristics to reduce network power consumption; that is, the power consumption of all computers connected to the network. The second contribution is an interactive teaching tool for networking students to learn about the dynamics of P2P applications, as well as their power consumption. A number of teaching methodologies exist to help students better understand networking through the use of technology and interactive learning. While these methodologies have proven successful in their own right, they also have various shortcomings. In using programs such as Wireshark, students unfamiliar with networking concepts require a fair amount of education prior to the exercise to understand how to interpret trial results. In addition, none of the known methodologies provide a learning experience into the power consumption of P2P applications. With its approach to network topology and capabilities of power consumption measurement, FrAPPE compensates for the aforementioned shortcomings of other network education methodologies that we have encountered.

In the next section, other network simulation tools and network education methodologies will be discussed in further detail. In Section 3, details of the design of FrAPPE are explained. In Section 4, details of the implementation of FrAPPE are explained. In Section 5, experimental trials are conducted demonstrating the effectiveness of FrAPPE. Section 6 reviews in summary FrAPPE as a solution for analyzing power consumption heuristics, as well as a teaching tool for computer networking education. Section 7 discusses future plans in the development of FrAPPE.

II. Related Work

A. Programs for Power Consumption Heuristic Analysis

A number of different programs have been developed for testing applications of networking technology through emulation and simulation. While FrAPPE is not unique in its approach to simulating a networked computing environment, no programs known to the authors offer the ability to measure the amount of power consumed by a network. The following subsections will examine some of these solutions and explain how FrAPPE differs in its approach.

1) Neko: Neko is a software program that simulates distributed algorithms over a simple communication platform. Neko also offers the ability to test the same algorithms over a real network. Similar to FrAPPE, Neko is implemented in Java, allowing a high level of portability and extensibility. Also similar to FrAPPE is Neko's highly object-oriented architecture, which allows greater ease in extensibility. Unlike FrAPPE, however, Neko does not include the ability to measure the amount of power consumed throughout the execution of an algorithm. For this to be possible, one would need to develop a custom implementation to incorporate into Neko's existing code base. FrAPPE includes the ability to measure power consumption, eliminating the need to extend the existing code base for this particular purpose.

2) NetBed: NetBed is a hybrid solution that incorporates elements of emulation, simulation, and physical hardware. It intends to create an environment in which experiments of anyl scale can be conducted from a single system, all while overcoming the limitations presented by the three aforementioned areas.

While NetBed is similar to FrAPPE in its use of simulation software, it does not include methods for measuring power consumption within the network. For this to be possible, one would need to modify the code to NetBed's simulation software. In addition, so that power consumption is measured universally within NetBed, specially-designed and potentially complicated modifications would need made for its emulation and hardware elements.

Because the system is hosted in a single, remote location, numerous restrictions are imposed on those who wish to conduct experiments. Firstly, due to access limitations, end-users are inhibited in their ability to customize the system for their experiments. Should they wish to modify the system to suit any special needs, such as implementing methods for measuring power consumption, it would need done with both the permission and intervention of NetBed's system administrators. In contrast, FrAPPE can be modified without intervention from a third-party.

Also, before conducting experiments, end-users must establish connectivity with the NetBed system over the Internet. This condition creates a requirement for end-users to conduct experiments using only computers that possess an Internet connection. In addition, issues that would affect Internet access, such as network outages, would similarly affect access to NetBed. In contrast, FrAPPE runs locally on a computer, and does not require an Internet connection for experiments to be conducted.

3) OPNET: OPNET is a commercially-distributed network simulation tool, and can be used on any computer without need of an Internet connection. Simulations are defined by scripts written in Proto-C, an object-oriented language. Because of the latter property, the language gives scripts much power in extending the capabilities of OPNET. Thus, despite its lack of methods for measuring and analyzing power consumption, the program's flexible scripting language allows for this capability to be programmed into simulations.

Because of its proprietary development model, one drawback in using OPNET is that its extensibility, as far as end-users are concerned, ends with the simulation definition. The program source itself is not available for redistribution, and thus, cannot be modified. With FrAPPE, one can freely modify the program source, and redistribute the modified program to any computer that supports Java.

Another drawback in using OPNET is its cost of purchase. OPNET is a robust, popular and well-maintained program. It contains numerous features, simulates many different elements of computer networking, and offers a diverse array of analysis methods. For classrooms and small research groups seeking a simple program to distribute to students at minimal expense, OPNET may be too complicated and too costly. An open-source program freely available for distribution, FrAPPE better suits the aforementioned needs.

4) OMNeT++: OMNeT++ is a program that offers discrete event simulation within a modeled networking environment, and can be run on any Windows, Mac or Linux computer. OMNeT++ allows users to define many different properties of a simulated networking environment for experiments. In addition, the program is open-source, allowing users to freely edit the source code to tailor the program to their unique needs.

OMNeT++ is developed in C++, which gives the program an object-oriented architecture, and allows it to be extended beyond its core implementation. One drawback to OMNeT++, however, is that its C++ foundation makes the process of porting modified versions of the program to other operating systems cumbersome. For an operating system to run a modified version of OMNeT++, a compiled executable compatible with its platform is required. Such a requirement adds complexity to the maintenance of special changes.

In addition, OMNeT++ does not include the ability to measure power consumption in its simulations. This feature would require special implementation, subjecting it to the aforementioned disadvantages in portability and maintainability.

5) OverSim: OverSim is a network simulation program that runs locally on a computer without the need of an Internet connection. The program is implemented on top of the previously-mentioned OMNeT++ architecture, and includes the ability to run simulations over a modeled P2P network. Because of its C++ foundation, OverSim is subject to the same drawbacks as OMNeT++. Specifically, OverSim's C++ implementation makes porting modified versions of the program to other operating system a cumbersome task. In addition, similar to its ancestor, OverSim does not provide the ability for measuring power consumption in its simulations.

B. Educational Approaches

FrAPPE's simplicity, ease of portability, and compatibility with all Java-compatible computers make it an appropriate educational tool. Other tools and methods have been developed for computer networking education, but no other solutions known to the author offer an educational perspective of the power consumption habits of computing machinery.

1) Virtualization Technology and NetWire: Because real networks can be costly and complicated to setup and maintain, teaching networking education concepts using real networks can be a challenge. In some cases, it may even be impractical. To avoid such maintenance issues, teachers have been known to turn to simulation, emulation and virtualization software, similar to those programs and systems previously described in this paper.

One particular tool used in academic environments is NetWire, a program that employs network emulation to demonstrate computer networking concepts. This program is designed for configuration on separate machines, making use of their physical hardware in its simulations. While NetWire and other emulation and virtualization solutions illustrate how computer networks operate through the use of physical hardware, such solutions carry limitations. One specific limitation is their requirement to be installed on multiple, physically connected computers. For students, this is likely to restrict the learning experience to the classroom. In contrast, students can deploy FrAPPE on their own computers, where they may use it at their leisure.

In addition, no emulation, virtualization or simulation solutions known to the author offer insight into the power consumption habits of computing machinery. FrAPPE provides the ability to measure power consumption over a simulated network, and offers a reporting mechanism to allow students to observe power consumption over a span of time.

2) Problem-Based Learning: Problem-Based Learning (PBL) is an approach to computer networking education developed by the University of Salford. Designed to educate students of both the technical and business aspects of networking, FrAPPE would complement the existing methods of PBL.

Firstly, as a tool for technical education, FrAPPE would demonstrate how computers communicate with one another over a network. They would be able to observe how messages are constructed, propagated and responded to. Secondly, as a tool for business education, students can associate the power consumption rates of computing machinery with any costs, monetary and otherwise, that would be incurred. By using FrAPPE to teach the two aforementioned principles, students can better understand the real-world implications to running a computer network.

III. Design

A. General Architecture

FrAPPE is a program that simulates a network of interconnected computers, known as Nodes, all while measuring the amount of power consumed by each Node throughout execution. The program is comprised of a number of basic, high-level elements that make up a typical network.

The core object that drives simulations is the Network. This object acts as the infrastructure through which individual Nodes can communicate with one another. Nodes act as standalone, singular entities, and communicate with one another over the Network to fulfill the operations of one or more applications, such as P2P networking. To communicate, Nodes pass to one another a series of data-filled packets, or Messages. These Messages contain information used by other Nodes to carry out actions specific to a common effort. In FrAPPE, this common effort is a power consumption heuristic, where Nodes will power on or off based on their own respective states, as well as the states of their neighboring Nodes in the Network.

B. Application Startup

Upon startup, FrAPPE expects to receive a user-created configuration file. It is through this file in which the topology of the simulated Network is defined. The details of the Network and its Nodes defined by the file include:

- A unique ID number by which it is referenced by other Nodes
- The unique ID numbers of each resource to which it is connected
- The "Up Timeout", or the number of execution cycles to pass before the Node, while in an Up (powered on) state, evaluates whether or not to power itself off
- The "Down Timeout", or the number of execution cycles to pass before the Node, while in a Down (powered off) state, evaluates whether or not to power itself on
- The rate in kilowatt-hours at which it consumes power

Once the configuration file is obtained and parsed, FrAPPE will construct the simulated Network as defined. From this point, the Network life cycle will be initiated.

C. Network

Similar to an actual network, the Network in FrAPPE acts as the infrastructure through which Nodes interact with one another. Once created, the Network engages in its "life cycle", an infinite loop that runs in the background and enables Node activity. In a single cycle, Nodes send through the Network any Messages that are enqueued for delivery.

The Network plays a significant role in fulfilling execution of the active heuristic. By providing the core mechanism through which Nodes send and receive information, Nodes can continually process this information, thus continuing the flow of the heuristic.

Given the cyclic, methodical execution of its lifecycle, the Network, in conjunction with its Nodes and their Messages, can be considered a universal state machine of the simulated system.

D. Node

A Node is an individual member of a Network. It is representative of a singular unit of computing machinery, such as a server or personal computer. In the scope of heuristic execution, the work performed by Nodes is akin to that of a software application, such as a file sharing program.

In FrAPPE, Nodes serve a fundamental role in executing a heuristic. Upon construction of the Network, all information unique to a Node is to be self-contained in its own memory. Thus, Nodes are aware of their neighbors in the Network to which they are connected. Nodes are also aware of the resources possessed by their neighbors. Finally, Nodes possess two Message queues, one for those that are incoming, the other for those that are outgoing.

In a single cycle, the Node performs a number of actions. First, it receives and processes all Messages in its inbound Message queue. For each Message in the inbound queue, the Node will extract all its data. Based on the contents of the Message, the Node will perform a specific, appropriate action.

The action taken on incoming Messages is determined by the user. At its base level, a Node is an abstraction whose functions and properties are core to all simulations. The user must define and direct the actions taken for each "work cycle". Later in this section, implementation examples will be described in detail.

Another part of a Node's core functionality is to record a variety of information pertaining to the events that take place during each cycle. This information includes statistics, such as the amount of power consumed, the number of computations performed, and the number of Messages sent. Also included are any changes in the state of both the Nodes and the Network. This information is essential to evaluating the overall performance of a heuristic.

E. Message

A Message is a packet of information shared over the Network between Nodes. The information housed by a Message is commonly used to fulfill an operation for a specific application, such as file sharing. When a Message is sent to initialize an operation with another Node, it is known as a "request". Messages sent to respond to a Node's request, it is known as a "response". Typically, Nodes will anticipate responses after having sent a complementary request. It will, in addition, perform specific actions based on the contents of the Message.

A single Message contains the following fields of information:

- FromID

- ToID
- Direction
- Command
- Data

In FrAPPE, the Network aims to successfully transfer Messages to their appropriate destinations. To accomplish this goal, the Network reads the ToID number contained in the Message's destination address field. Using this number, the Network can deliver the Message to the intended Node. Once delivered, a Message will sit in the recipient Node's incoming queue until the Node has the chance to process it.

The FromID value specifies the address from which the Message originated. This is necessary for the recipient Node to know, as it may either perform an action based on the knowledge of where the Message came from, or more commonly, it must know where it may send a Message if the recipient Node chooses to respond.

The Direction value specifies whether a Message is a request to be sent to a Node, or a response to a previous request to be sent to the originator of the request.

The Command value specifies the high-level instruction to be acted upon by the receiving Node.

The Data value contains all data to be received by the destination Node. Typically, the Command value will dictate how the Data is to be interpreted and acted upon.

F. Extension Example: Timeout-Based Heuristic

To illustrate how end-user implementations are used by FrAPPE in its execution cycles, three examples are included in this paper. The first example is known as the Timeout-Based Heuristic (TBH). A simple implementation example, each Node execting TBH checks for whether its state is Idle (inactive), Up (powered on), or Down (powered off). If the Node's state is Idle and Up, an internal cycle counter is incremented. If the Up Timeout limit defined by the configuration file is reached, the Node's cycle counter is reset to 0, and its state is changed to Idle and Down. If the Node's state is Idle and

Down, the cycle counter is incremented each cycle. Once the Down Timeout defined by the configuration file is reached, the Node's cycle counter is reset to 0, and its state is changed to Idle and Up.

G. Extension Example: Order-Based Heuristic Framework

To illustrate a more complex example, a heuristic framework will be demonstrated in which Nodes decide whether or not to sleep based on the availability of the resources of their neighboring Nodes. In the Order-Based Heuristic (OBH), the evaluation is performed on the collective resources of all neighboring Nodes in a particular order. The two subsections that follow describe specific ordering schemes in further detail; this section covers the idea of an OBH framework, and its design in FrAPPE.

When in an Up and Idle state, the cycle counter of a Node will increment after each work cycle. Upon reaching its Up Timeout, a Node will send a Message to each of its neighbors, requesting a list of all of their resources. The Node will then wait until all resource lists are received, even if the wait spans multiple work cycles.

Once all resource lists have been received, the Node will compile the union of all resource IDs in a specific order. The order in which the resources IDs are arranged depends on the policy implementation. Examples of order policy implementation will follow in the next two subsections. Once the resource IDs have been compiled, they will be iterated through, and the counts of each resource will be collected. Iteration will continue until the compiled resource list has been examined in full. Iteration will also end if all of the Node's resources are found to be accounted for by all neighboring Nodes.

If a Node discovers that all of its neighbors collectively possess all of the same resources, it then deems its presence on the Network as unnecessary for the time being. At this point, the Node will power itself off. Otherwise, if at least one of its resources is found to be unique among its neighbors, then it will remain in an Up state.

If a Node powers itself off following analysis of its neighbors, then it will remain Down until its respective timeout value expires. At that point, the Node will power itself back on.

1) Largest-to-Smallest Order Policy: As an extension of the OBH framework, a Node implementing the Largest-to-Smallest Policy structures a list of resource IDs in descending order by the number of neighbors that possess the resource. The ultimate effect is that, as Nodes iterate through the ordered list, they will power off sooner if it finds that its resources are considered common among its neighbors. If a Node powers off sooner, it uses fewer computation cycles to come to this conclusion, thus reducing power consumption prior to entering an energy-efficient state. Otherwise, if it is found that at least one of the Node's resources are unique, it will remain in an Up state.

A savings in power consumption is achieved in the minimization of computation cycles used to check for redundant resources when a small number of them are unique. The reason is that Nodes are expected to sooner discover that their presence on the Network is not needed. Otherwise, if many of the resources on the Network are unique, a greater amount of power will be consumed in total, as all Nodes perform a greater number of computation cycles to determine whether or not they should remain in an Up state.

2) Random Order Policy: Another extension of the OBH framework, a Node implementing the Random Order Policy structures a list of resource IDs in random order. The effect is that power consumption used in determining whether or not the Node should shut off will be consistent across resource lists of all sizes.

IV. Implementation

A. Java Programming Language

FrAPPE is implemented using the Java programming language. The reason for this choice was two-fold. First, the general design of FrAPPE employs concepts of object-orientation. Each piece of the program is broken down into specific entities relative to a real-life system. Java is well-known for being a heavily object-oriented programming language. Thus, the program was considered an appropriate language to complement FrAPPE's design.

The second reason for choosing Java as FrAPPE's langauge of implementation is portability. Being implemented in Java, FrAPPE can be compiled once, and used on any computer that has the Java Runtime Environment installed. Examples of popular, modern operating systems for which the Java Runtime Environment exist include Windows, Linux and Mac OS. Because Java is supported on all of the most popular, contemporary operating systems, FrAPPE can be ported with ease. In addition, it makes for easy sharing of FrAPPE extensions, regardless of the target platform. For example, one could compile FrAPPE with custom extensions in Mac OS, and immediately share it with individuals using Windows-based PCs.

B. Program Architecture

1) Application Design and Lifecycle: To complement the object-oriented design of FrAPPE, individual Java objects were implemented for each, distinct design object. These distinct design objects include Network, NetworkNode and Message. The Network and its lifecycle are spawned by ImpApp, a central object that controls the application lifecycle. The topology of the Network is constructed by the ConfigFileParser class, which is also spawned by ImpApp.

The Network lifecycle operates as an infinite loop within its class, where each Node is invoked to receive all incoming Messages, perform its work, and then send any outgoing Messages generated in the process.

The NetworkNode class acts as a container of Node information, and possesses a number of methods to handle generic actions. At its core, the NetworkNode handles the sending and receiving of Messages. Responsibility is delegated to extension classes in three abstract methods: the “Work” method (doWork), the “Request Message Handler” method (handleRequestMessage), and the “Sleepiness” method (isSleepy). The doWork method is performed based on a combination of its current state and all Messages received from other Nodes. The handleRequestMessage method is performed for each incoming request Message. Finally, the isSleepy method is called to determine whether or not the Node’s presence in the Network is necessary, which ultimately leads to the decision of whether or not the Node should power itself off.

The Message class acts as a container for all information to be shared between two Nodes. Numeric containers are reserved for all direction and command items, including FromID, ToID, Command, and Direction. A generic object container is reserved for any data to be used in conjunction with Command. Methods exist exclusively for the specification and retrieval of the aforementioned information.

2) Use of Configuration File: As previously mentioned, the topology of the simulated Network is determined by a user-defined configuration file. In FrAPPE, the configuration file is to be defined in XML format, and follows a specific structure. At the top level, the algorithm (heuristic) to be performed is defined. This tells FrAPPE which NetworkNode implementation to use when the program starts. Afterward, the user must define the properties for each, individual Node. The properties to be defined for each Node include all properties previously specified in the Design section of this paper.

3) Program Startup and Input Arguments: At this time, FrAPPE operates as a command-line application. The inputs required are as follows:

- The file path of an XML-formatted configuration file that defines the details of the simulated Network
- The name of the heuristic to be run during execution
- The number of cycles to be run during execution

As the program runs, FrAPPE will produce output to three log files, all of which will be described in detail in the next section.

4) Logging: For FrAPPE to be fully effective, output must be produced to describe important details of the simulation, especially as it pertains to the state and activities of each Node. To address this need, FrAPPE generates three separate logs for user analysis: a Data Log, a Statistics Log, and an Execution Log.

The Statistics Log offers information regarding power consumption and Node performance over each work cycle in the execution. Specifically, the Statistics Log reports the number of computations performed, Messages sent over the Network, and power consumed in kilowatt-hours for each Node on the Network. In addition, running totals of these numbers are maintained both per Node and over the entire Network. The Statistics Log is compiled as a set of line items in comma-separated format, and written to a text file. This file is generated as such to allow for easy incorporation into programs designed to perform calculations on comma-separated values. One example of such a program is Microsoft Excel.

The Statistics Log offers similar information to the Data Log, but presents the information in a more readable format. Unlike the Data Log, the Statistics Log is not designed for incorporation into other programs. Instead, this log is intended to provide statistical snapshots of all Nodes during each individual cycle.

The Execution Log offers information regarding execution of the simulation, state transitions of all Nodes, and Message activity within the Network. Similar to its more statistical counterparts, this log is written to a text file, but is compiled as a series of line-item entries.

Combined, the supplied logs provide comprehensive information regarding the state of the Network throughout each execution cycle. With this raw data, one may derive an abundance of information to determine the effectiveness of a heuristic. For example, one may calculate the average total amount of power consumption by compiling a sum of kilowatt-hours consumed per cycle, and then dividing the sum by the total number of cycles. Another example might be analyzing the cost incurred in traffic by performing calculations against the number of Messages sent within the Network over a number of cycles.

The manner in which the data is to be interpreted is left to the user. Furthermore, users are invited to expand the existing logging functionality as desired to report any additional information of use to their experiments to suit any special computational needs.

5) Extensibility: An especially important feature of FrAPPE is its high level of extensibility. Using the Java programming language, one can extend the program to fit any special needs in their computer networking simulation. At the core of FrAPPE's extensibility is the NetworkNode class, an abstract container that represents a single unit of networked computing machinery. This class cannot operate on its own in a simulation; thus, one is required to provide an implementation of this class to define the different aspects of its behavior in certain situations. Details of the abstract properties in NetworkNode can be found in the Application Design and Lifecycle subsection.

In addition to NetworkNode, any class in the program can also be extended to support additional capabilities. One such example is the implementation of a specific networking protocol, such as TCP. In this case, one would modify the NetworkNode class to construct Messages that conform to the protocol's specification, and then modify the Message class to accept and organize data to fit the aforementioned model. Another practical example of extension is to modify Logging to include specialized statistics and information, such as running totals and details of additional Node data.

The task of extending FrAPPE is made easier by the highly object-oriented structure, meticulous organization, and simple design of the program. The role of each class and member is made clear through a comprehensive naming convention combined with an architecture that is modeled after that of a real network.

C. Examples of Extensibility

The inner workings of the TimeoutNode are simple. When invoked, it will increment a counter respective to its current state, whether it is Up or Down. When this counter has reached its prescribed maximum limit, it will reverse its current power state, from on to off, or vice versa.

OrderNode is implemented as an abstract class. It is designed to iterate through the resources of neighboring Nodes, with the specific order left for extending classes to implement. When invoked, the OrderNode increments the counter respective to its current state, whether it is Up or Down. When the active counter has reached its prescribed maximum limit, the Node will generate and send Messages to all neighboring Nodes. The Messages each contain a command that instructs the Node to respond with a list of all its resources. At this point, the Node will change state from Up and Idle to Up and Waiting. During this time, the Node will wait to collect each list of resources, while continuing to respond to any incoming request Messages.

Once all resource lists are collected, the Node will iterate through the complete list of resources based on the order defined by its implementation. It will then determine whether or not the Node should go to sleep. In checking for sleepiness, the Node will elect to change state to Down and Idle if it finds that its resources are not unique to the Network. Otherwise, if at least one of the Node's resources are found to be unique, the Node will change its state back to Up and Idle. While in a Down and Idle state, the Node will power back on as soon as its Down Timeout counter has been incremented to the maximum limit defined in the supplied configuration file.

An extension of OrderNode, OrderRandomNode fulfills the requirement prescribed by OrderNode to order the collection of all neighboring Node resource lists. This particular extension arranges the list in random order. Another extension of OrderNode, OrderLtoSNode fulfills the requirement prescribed by OrderNode to order the collection of all neighboring Node resource lists. This particular extension arranges the list in order of greatest to least count in terms of number of occurrences within the union.

V. Evaluation

With the different points of data offered by FrAPPE, heuristics can be analyzed and compared to other heuristics from a number of different perspectives. In this section, three experimental trials will be covered to illustrate the types of conclusions that can be reached with FrAPPE's data output.

In the following experiments, the three sample heuristics described earlier are evaluated. To reiterate, these heuristics are the Timeout-Based Heuristic (TBH), the Order-Based Heuristic of Largest-to-Smallest Order Policy (LSOP), and the Order-Based Heuristic of Random Order Policy (ROP). In each experiment, the topology used consists 16 Nodes, four of which acting as interconnected "Super Nodes", the other 12 divided in four groups of three, with one group connected to each Super Node. Figure 1 offers a visual depiction of the topology.

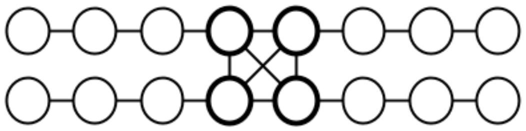
Fig. 1: An illustration of the topology used for each experiment. Super Nodes are highlighted with a bold outline.

Another fixed parameter shared across each experiment is the Down Timeout value. For all three experiments, each Node, if Down, will remain as such for one cycle before attempting to return to an Up state. Finally, each experimental trial covers an execution interval of 50 work cycles.

A. Experiment 1: Comparing Performance of Multiple Heuristics

When evaluating heuristics, it may be desired to understand how one heuristic compares in performance to another. Through FrAPPE, heuristic performance can be evaluated by the total number of Messages sent over the Network, the total number of computations performed by all Nodes in the Network, and the total amount of power consumed by all Nodes over all execution cycles in the trial.

In this experiment, all three heuristics are tested with the same parameters to understand their performance in comparison to one another. In addition to the fixed parameters specified earlier in this section, each Node exercises an Up Timeout interval of onecycle, and a file distribution in which half of the files shared across the Network are unique. The data generated by this experiment can be found in Table I.

Heuristic	Comps	Msgs	Power total (kWh)
Timeout	0	0	59999
Order (L to S)	1799	1799	119999
Order (Random)	1799	1799	119999

TABLE I: Computations, Messages and total power consumption (kWh) per heuristic.

As depicted by the generated data, TBH outperforms both LSOP and ROP in all three areas. It must be understood however, that this shows only the efficiency of TBH in terms of power consumption and computational expense. Because this heuristic disregards the need for file redundancy, it cannot be assumed by this data alone that the TBH is optimal for public use. This data does, however, put into perspective the cost of ensuring file availability across an actual network.

B. Experiment 2: Comparing Heuristic Performance Under Differing Environmental Conditions

When analyzing a single heuristic, it is helpful to understand how it performs under different environmental conditions. In this experiment, ROP is tested against four different file distribution schemes: no unique files, one unique file, half of all files unique, and all files unique. The data generated by this experiment can be seen in Table II.

File dist	Comps	Msgs	Power total (kWh)
No unique files	899	1271	89399
One unique file	979	1327	94499
Half unique files	1799	1799	119999
All unique files	1799	1799	119999

TABLE II: Computations, Messages and total power consumption (kWh) per file distribution.

In observing the generated data, it can be seen that ROP exhibits equal performance when operating among a half-unique file distribution scheme as when all files in the Network are unique. This particular heuristic performs best when fewer files in the Network are unique. To illustrate this point, note that when no unique files exist in the Network, power consumption is approximately 30% less than that of when at least half of the files are unique.

In studying a heuristic, sufficient data can be collected by FrAPPE to illustrate the conditions under which it is most efficient, as well as most inefficient. By understanding the strengths and weaknesses of a heuristic, this information can be used towards its improvement, or otherwise, it can be used as justification for seeking a heuristic better suited for the environment in study.

C. Experiment 3: Comparing Heuristic Performance Under Differing Operational Conditions

In developing a heuristic, tests under certain environmental conditions are aided by variables in the heuristic that, when tweaked, affect performance. In this experiment, ROP is tested with a file distribution in which half of the files in the Network are unique. The focus of this experiment is to understand how the heuristic performs under four different Up Timeout values. The data generated by this experiment can be seen in Table III.

Up T.O.	Comps	Msgs	Power total (kWh)
1	1799	1799	119999
2	1151	1205	119999
4	719	719	119999
8	305	305	119999

TABLE III: Computations, Messages and total power consumption (kWh) per Up Timeout value.

The data shows the total amount of power consumed by the Nodes to be constant across all four trials. The number of computations and Messages sent, however, decrease as the Up Timeout interval increases. While this would make sense given that Nodes are Down for longer, rendering them incapable of sending Messages or performing computations, the fact that power consumption remains constant indicates that savings is achieved in the long term by reducing the impact on the Network infrastructure that facilitates the sending of Messages. This data can be interpreted to indicate that the best use of this heuristic would balance a higher Up Timeout interval with a high degree of file availability.

VI. Conclusions and Future Work

A. Conclusions

This paper has demonstrated FrAPPE as an application with a two-fold purpose. Its primary purpose is to analyze the performance of power consumption heuristics designed for use in P2P networks. This solution has been shown as unique compared to other network simulation tools in that it contains the capability to measure power consumption.

The second purpose of FrAPPE is to help students learn about basic networking concepts, as well as understand the costs incurred by modern hardware on both small and large scales.

This paper covered details of both the design and implementation of FrAPPE, illustrating its object-oriented approach, and its intent to follow a real-life model in terms of organization and categorization. It was also demonstrated of how FrAPPE's Java-based platform makes for a high degree of portability, and its design allows for easy and robust extensibility.

Finally, this paper covered FrAPPE in action, as the evaluation section described examples of its proposed core points. The evaluation confirmed FrAPPE's effectiveness in analyzing power consumption in P2P networking heuristics, as well as suitability for use in academia to teach concepts in computer networking and power consumption.

B. Future Work

Though FrAPPE offers an immediate solution for analyzing power consumption heuristics in P2P networks, there are a number of things that will be done to better enhance its capability and usability. At this time, FrAPPE is operates by command line only. Though this does not affect the correctness of the data produced, adding enhanced usability through an intuitive user interface will help in swifter construction of network topology and analysis of execution data.

In its current form, FrAPPE offers the ability to define a small number of properties for each Node and the topology itself. While the current set of properties is sufficient for basic research into power consumption heuristics, research at a deeper level can only be conducted through extended implementation by the user. By expanding the simulated network and offering more, customizable properties, research into power consumption heuristics can be conducted at a deeper level, while requiring less work on the part of the user.

On the subject of expanding the capability, one category in particular that is taken for granted is the possibility of network anomalies. In its current state, FrAPPE assumes flawless network conditions, which is not likely in a real network, especially one that spans the Internet. In providing the ability to define specific anomalies in the simulated network, and the degree to which they occur, users may gain a more accurate idea of how a heuristic will perform in a real network. This is significant, as it is entirely possible for anomalies in a network to interrupt the flow of execution for a heuristic.

References

[1] M. Webb *et al.*, “Smart 2020: Enabling the low carbon economy in the information age,” *The Climate Group. London*, vol. 1, no. 1, pp. 1–1, 2008.

[2] G. Anastasi, I. Giannetti, and A. Passarella, “A bittorrent proxy for green internet file sharing: Design and experimental evaluation,” *Computer Communications*, vol. 33, no. 7, pp. 794–802, 2010.

[3] W. Bircher and L. John, “Complete system power estimation: A trickle-down approach based on performance events,” in *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, pp. 158–168, IEEE, 2007.

[4] D. Rossi, E. Sottile, and P. Veglia, “Black-box analysis of internet p2p applications,” *Peer-to-Peer Networking and Applications*, vol. 4, no. 2, pp. 146–164, 2011.

[5] J. Reich, M. Goraczko, A. Kansal, and J. Padhye, “Sleepless in seattle no longer,” in *USENIX ATC*, 2010.

[6] S. Rivoire, P. Ranganathan, and C. Kozyrakis, “A comparison of high-level full-system power models,” in *Proceedings of the 2008 conference on Power aware computing and systems*, pp. 3–3, USENIX Association, 2008.

[7] P. Urban, X. Defago, and A. Schiper, “Neko: a single environment to simulate and prototype distributed algorithms,” in *Information Networking, 2001. Proceedings. 15th International Conference on*, pp. 503 –511, 2001.

[8] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, “An integrated experimental environment for distributed systems and networks,” *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 255–270, Dec. 2002.

[9] X. Chang, “Network simulations with opnet,” in *Simulation Conference Proceedings, 1999 Winter*, vol. 1, pp. 307 –314 vol.1, 1999.

[10] I. Baumgart, B. Heep, and S. Krause, “Oversim: A flexible overlay network simulation framework,” in *IEEE Global Internet Symposium, 2007*, pp. 79 –84, may 2007.

[11] A. Varga and R. Hornig, “An overview of the omnet++ simulation environment,” in *Proceedings of the 1st international conference on Simulation tools and echniques for communications, networks and systems & workshops*, Simutools ’08, (ICST, Brussels, Belgium, Belgium), pp. 60:1–60:10, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[12] G. Perera, K. Christensen, M. Goncalves, and M. Kontitsis, “Studying heuristics to reduce power consumption in peer-to-peer servers.” 2007.

[13] E. Carniani and R. Davoli, “The netwire emulator: a tool for teaching and understanding networks,” *SIGCSE Bull.*, vol. 33, pp. 153–156, June 2001.

[14] N. Linge and D. Parsons, “Problem-based learning as an effective tool for teaching computer network design,” *Education, IEEE Transactions on*, vol. 49, pp. 5 – 10, feb. 2006.

[15] C. Burger and K. Rothermel, “A framework to support teaching in distributed systems,” *J. Educ. Resour. Comput.*, vol. 1, Mar. 2001.

[16] D. Dobrilovi and B. Odadi, “Virtualization technology as a tool for teaching computer networks,” *IJHSS*, vol. 1, no. 2, 2006.

APPENDIX A
USAGE DOCUMENTATION

This appendix offers instructions for running FrAPPE, and describes expectations of the inputs required and output generated.

A. Running FrAPPE

1) Prerequisites: FrAPPE is packaged as Java Archive File, or JAR. For a computer to run FrAPPE, the Oracle Java Runtime Environment (JRE), version 1.6 or newer, must be installed. At the time this paper was written, the latest version of Oracle JRE was available for download at http://java.com/en/download/index.jsp. Because Java is designed for multi-platform use, the same FrAPPE JAR binary file can be executed on any operating system environment where the Oracle JRE is installed.

2) Program Execution: FrAPPE is designed for execution in command line environments. In Linux and Mac OS, a command line environment can be found in the Terminal (or Console) application, where Windows offers a command line environment in CMD.EXE. Before running FrAPPE, it is recommended that the JRE "bin" directory be included in the command line environment path. For instructions on how to add the aforementioned directory to the path, consult the documentation for the respective command line environment or operating system.

To execute FrAPPE, first navigate to directory in which the FrAPPE JAR is located. Next, the JAR must be invoked through the Java executable binary located in the JRE "bin" directory. The form of the command must be as depicted in Figure 2.

To understand the command described above, it is helpful to examine its individual components. The first component, "java", is a reference to the Java executable binary, which invokes the Java Virtual Machine (JVM). It is through this program that FrAPPE

```
java -jar FrAPPE.jar <configuration file path> <name of
    heuristic> <number of cycles>
```

Fig. 2: The base form of the command used to execute FrAPPE.

```
java -jar FrAPPE.jar test.xml Timeout 25
```

Fig. 3: A sample call to FrAPPE.jar.

is executed. The second component, “-jar”, is an argument to the Java executable binary. This argument indicates that the JVM is to execute a JAR file. The third component, “FrAPPE.jar”, is a reference to the FrAPPE JAR file that resides in the current directory.

The next three components of the command are arguments to FrAPPE. It must be noted that the angle brackets are omitted in practice, and exist only for purposes of distinction. The first FrAPPE argument, “<configuration file path>”, is the fully-qualified path to the configuration file in which the target simulated network is defined. The second FrAPPE argument, “<name of heuristic>”, is the name of the heuristic to be run during execution. The third FrAPPE argument, “<number of cycles>”, is the number of cycles that are to be run by the simulated network during execution. Figure 3 depicts a sample call to FrAPPE.jar, employing the network defined in the file “test.xml”, specifying “Timeout” as the heuristic, and setting the number of cycle to be run at 25.

B. Input Configuration File

As previously mentioned, the simulated network used in FrAPPE executions is defined in an XML-formatted configuration file. The specification of the FrAPPE network configuration file can be best summarized as an XML schema, as depicted in Figure 4.

To illustrate how this schema is applied in defining a configuration file for FrAPPE, a sample configuration file is offered in Figure 5. This sample configuration file defines a four-node network in which each node is connected to one another, and all nodes possess one common resource (9).

C. Output Log Files

As FrAPPE executes, program output is written to three log files. These files reside inside a folder titled “data”, which is created in the same directory as that of the FrAPPE JAR. All three logs follow a common naming convention, which is depicted in Figure 6.

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="FrAPPE">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Imp">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Nodes">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="Node">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name="ID" type="␣xs:string" />
                          <xs:element name="Neighbors" type="xs:string" />
                          <xs:element name="Resources" type="xs:string" />
                          <xs:element name="UpTimeout" type="xs:string" />
                          <xs:element name="DownTimeout" type="xs:string" />
                          <xs:element name="PowerRatekWh" type="xs:string" />
                        <xs:sequence>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Fig. 4: The XML schema of the FrAPPE network configuration file.

For each log in an execution, its filename includes the name of the heuristic, a UNIX-style timestamp in number of milliseconds since January 1, 1970, and the log type, whether it be Execution, Statistics or Data. In Figure 7, a sample log file name is depicted for a Data Log running the Random Order Policy (OrderRandom) heuristic starting at millisecond 1367330951826.

1) Execution Log: The Execution Log includes information about the network and each individual node over all execution cycles. Types of activity recorded to this log include state changes and the sending and receiving of messages. Figure 8 is an excerpt from a sample Execution Log for a trial of the Random Order Policy heuristic.

2) Statistics Log: The Statistics Log includes a comma-separated data values for each node over all execution cycles. Types of activity per node per cycle recorded to this log include its current state, rate of power consumption, number of computations, and number of messages sent. Figure 9 is an excerpt of a sample Statistics Log for a trial of

```
<FrAPPE>
  <Imp>
    <Nodes>
      <Node>
        <ID>1</ID>
        <Neighbors>2,3,4</Neighbors>
        <Resources>1,2,9</Resources>
        <UpTimeout>1</UpTimeout>
        <DownTimeout>1</DownTimeout>
        <PowerRatekWh>150</PowerRatekWh>
      </Node>
      <Node>
        <ID>2</ID>
        <Neighbors>1,3,4</Neighbors>
        <Resources>3,4,9</Resources>
        <UpTimeout>2</UpTimeout>
        <DownTimeout>1</DownTimeout>
        <PowerRatekWh>150</PowerRatekWh>
      </Node>
      <Node>
      <ID>3</ID>
        <Neighbors>1,2,4</Neighbors>
        <Resources>5,6,9</Resources>
        <UpTimeout>1</UpTimeout>
        <DownTimeout>2</DownTimeout>
        <PowerRatekWh>150</PowerRatekWh>
      </Node>
      <Node>
        <ID>4</ID>
        <Neighbors>1,2,3</Neighbors>
        <Resources>7,8,9</Resources>
        <UpTimeout>2</UpTimeout>
        <DownTimeout>2</DownTimeout>
        <PowerRatekWh>150</PowerRatekWh>
      </Node>
    </Nodes>
  </Imp>
</FrAPPE>
```

Fig. 5: A sample FrAPPE network configuration file.

```
test_a-<heuristic>_<timestamp>_log_<logtype>
```

Fig. 6: The common naming convention for log files generated by FrAPPE.

the Random Order Policy heuristic.

3) Data Log: The Data Log includes a comma-separated data totals over all execution cycles. This log is similar to the Statistics Log, except the Data Log compiles totals of all computations, messages sent, and power consumed per cycle for the entire network. Figure 10 is an excerpt of a sample Data Log for a trial of the Random Order Policy

```
test_a-OrderRandom_1367330951826_log_data
```

Fig. 7: A sample FrAPPE log file name.

```
Work cycle started
Cycle count: 0
Cycle timestamp: 1367499822434


Starting work for node: 1
--------------------------------
Current state: UP_IDLE (0)
Current time count: 1
Timeout reached
Requesting list of node: 2
Requesting list of node: 3
Requesting list of node: 4
New state: UP_WAITING_FOR_LISTS (1)
Sending outgoing messages
Sending message to node: 2
Sending message to node: 3
Sending message to node: 4
Finished work for node: 1


Starting work for node: 2
--------------------------------
Current state: UP_IDLE (0)
Received list request from node: 1
Current time count: 1
Sending outgoing messages
Sending message to node: 1
Finished work for node: 2


Starting work for node: 3
--------------------------------
Current state: UP_IDLE (0)
Received list request from node: 1
Current time count: 1
Timeout reached
Requesting list of node: 1
Requesting list of node: 2
Requesting list of node: 4
New state: UP_WAITING_FOR_LISTS (1)
Sending outgoing messages
Sending message to node: 1
Sending message to node: 1
Sending message to node: 2
Sending message to node: 4
Finished work for node: 3


Starting work for node: 4
--------------------------------
Current state: UP_IDLE (0)
Received list request from node: 1
Received list request from node: 3
Current time count: 1
Sending outgoing messages
Sending message to node: 3
Sending message to node: 1
Finished work for node: 4


Work cycle complete. Waiting to restart.
```

Fig. 8: An excerpt from a sample Execution Log.

```
Cycle timestamp: 1367499822457
Cycle count: 5
--------------------------------
id,state,power_rate_kwh,computations,computations_total,msg_sent,msg_sent_total
1,0,150,9,27,3,16
2,0,150,9,18,2,14
3,0,150,9,27,1,16
4,0,150,9,18,0,14
--------------------------------
N/A,N/A,600,36,252,6,214
```

Fig. 9: An excerpt from a sample Statistics Log.

```
power_rate_kwh,computations,msg_sent
600,0,10
600,18,10
600,18,14
600,18,2
600,0,18
600,36,6
600,0,10
600,18,10
600,18,14
600,18,2
```

Fig. 10: An excerpt from a sample Data Log.

heuristic.

Appendix B
Extension Documenation

This appendix offers documentation for implementing a heuristic in FrAPPE. Also included is the source code to the three sample heuristics (TBH, LSOP, and ROP) described earlier in this paper.

A. *Extension Specification*

There are two parts to a heuristic implementation in FrAPPE. First, an extension of NetworkNode must be developed to perform the work of the heuristic. Second, the program must be revised to recognize the heuristic name as an input parameter, so that the extension may be instantiated.

1) NetworkNode Abstract Class: In each NetworkNode extension, three methods must be overridden:

- protected abstract void doWork(Iterator<Message> inMsgIter)
- protected abstract void handleRequestMessage(Message inMsg)
- protected abstract boolean isSleepy()

The first method, “doWork”, includes as a parameter an Iterator containing all inbound messages belonging to the node. It is expected that the implementation of this method will handle parsing and interpreting these messages, as their contents are likely to affect the logical decisions made during the work cycle.

The second method, “handleRequestMessage”, includes as a parameter a Message object representing a single inbound message. At this time, FrAPPE does not call this method from within the NetworkNode abstract class. Instead, it is recommended for this method to be called from within the “doWork” method of the same class. The purpose of the method is to encourage a single method in which inbound message parsing operations may be handled.

The third method, “isSleepy”, is similar to “handleRequestMessage” in that it is not called by the NetworkNode class. Instead, it is recommended that this method also be called from within the “doWork” method of the same class. The purpose of this method is to encourage a single method in which node conditions may be examined to determine whether or not a node should power itself off.

2) Connecting Extensions to FrAPPE: Once the three abstract methods described in the previous section have been implemented, one final step remains before a heuristic is ready for execution in FrAPPE. Upon executing the program, one of the first actions to be taken is to parse the configuration file so that the simulated network may be constructed. It is at this stage where the NetworkNode implementation of a heuristic is instantiated; specifically, this activity takes place inside the “parseNodeElement” method of the class “ConfigFileParser”. Figure 11 is a code sample illustrating how TBH, LSOP and ROP may be instantiated inside the aforementioned method.

```
if(_algorithm.equals(ALG_TIMEOUT)){
  newNode = new TimeoutNode(id, resources, neighbors, upTimeout, downTimeout, powerRatekWh
      , _log);
}
else if(_algorithm.equals(ALG_ORD_L_TO_S)){
  newNode = new OrderLtoSNode(id, resources, neighbors, upTimeout, downTimeout,
      powerRatekWh, _log);
}
else if(_algorithm.equals(ALG_ORD_RANDOM)){
  newNode = new OrderRandomNode(id, resources, neighbors, upTimeout, downTimeout,
      powerRatekWh, _log);
}
```

Fig. 11: Sample code illustrating how NetworkNode implementations of select heuristics may be instantiated.

In the above example, the Java String objects representing the heuristic names are created as global variables to be referenced later. For example, the String “Timeout” is represented by the variable “ALG_TIMEOUT”. It can be noted that each constructor shares a similar set of arguments, all of which are fulfilled by variables local to the ConfigFileParser class.

Naturally, it is encouraged for users of the program to extend the application of a heuristic beyond what is described in this paper. For instance, those users seeking improvements to the logging output received during execution may benefit from extending the existing logging classes to meet any otherwise unfulfilled needs.

B. Example: Timeout-Based Heuristic Source Code

The following listing is the source code to the Timeout-Based Heuristic:

```
package com.bk.frappe.imp.node.timeout;

import java.util.Iterator;

import com.bk.frappe.imp.app.Log;
import com.bk.frappe.imp.network.Message;
import com.bk.frappe.imp.network.NetworkNode;

public class TimeoutNode extends NetworkNode{
  public TimeoutNode(int id, int[] resources, int[] neighbors, int upTimeout, int
      downTimeout, int powerRatekWh, Log log){
    super(id, resources, neighbors, upTimeout, downTimeout, powerRatekWh, log);
  }

  @Override
  protected void doWork(Iterator<Message> inMsgIter){
    if(_state == STATE_UP_IDLE){
      _log.writeToExecLog("Current␣state:␣UP_IDLE␣(" + _state + ")");

      while(inMsgIter.hasNext()){
        handleRequestMessage(inMsgIter.next());
      }

      _timeCounter = (_upTimeout > 0 ? _timeCounter + 1 : 0);

      _log.writeToExecLog("Current␣time␣count:" + _timeCounter);

      if(_timeCounter == _upTimeout){
        _state = STATE_DOWN_IDLE;
        _timeCounter = 0;

        _log.writeToExecLog("Timeout␣reached");
        _log.writeToExecLog("New␣state:␣DOWN_IDLE␣(" + _state + ")");
      }
    }
    else if(_state == STATE_DOWN_IDLE){
      _log.writeToExecLog("Current␣state:␣DOWN_IDLE␣(" + _state + ")");

      while(inMsgIter.hasNext()){
        handleRequestMessage(inMsgIter.next());
      }

      _timeCounter = (_downTimeout > 0 ? _timeCounter + 1 : 0);

      _log.writeToExecLog("Current␣time␣count:␣" + _timeCounter);

      if(_timeCounter == _downTimeout){
        _state = STATE_UP_IDLE;
        _timeCounter = 0;

        _log.writeToExecLog("Timeout␣reached");
        _log.writeToExecLog("New␣state:␣UP_IDLE␣(" + _state + ")");
      }
    }
  }

  @Override
  protected void handleRequestMessage(Message inMsg){
    return;
  }

  @Override
  protected boolean isSleepy(){
    return true;
  }
}
```

C. Example: Order-Based Heuristic Source Code

The following listing is the source code to the Order-Based Heuristic. Note that this class, similar to its parent, is abstract. Further extension is required, which will be demonstrated by the Largest-to-Smallest and Random Order policies in the upcoming sections.

```
package com.bk.frappe.imp.node.order;

import java.util.Iterator;

import com.bk.frappe.imp.app.Log;
import com.bk.frappe.imp.network.Message;
import com.bk.frappe.imp.network.NetworkNode;
import com.bk.frappe.imp.network.ResourceMap;
```

```
public abstract class OrderNode extends NetworkNode{
  protected static final byte STATE_UP_WAITING_FOR_LISTS = 1;

  public OrderNode(int id, int[] resources, int[] neighbors, int upTimeout, int
      downTimeout, int powerRatekWh, Log log){
    super(id, resources, neighbors, upTimeout, downTimeout, powerRatekWh, log);
  }

  protected abstract int[] getOrder();

  @Override
  protected void doWork(Iterator<Message> inMsgIter){
    if(_state == STATE_UP_IDLE){
      _log.writeToExecLog("Current␣state:␣UP_IDLE␣(" + _state + ")");

      while(inMsgIter.hasNext()){
        handleRequestMessage(inMsgIter.next());
      }

      _timeCounter = (_upTimeout > 0 ? _timeCounter + 1 : 0);

      _log.writeToExecLog("Current␣time␣count:␣" + _timeCounter);

      if(_timeCounter == _upTimeout){
        _log.writeToExecLog("Timeout␣reached");

        Iterator<Integer> keyIter = _nodeMap.keySet().iterator();

        while(keyIter.hasNext()){
          Message outMsg = new Message(_id, keyIter.next(), Message.DIR_REQ, Message.
              CMD_LST, null);
          _outMessages.add(outMsg);

          _messagesSent = _messagesSent + 1;

          _log.writeToExecLog("Requesting␣list␣of␣node:␣" + outMsg.getToID());
        }

        _state = STATE_UP_WAITING_FOR_LISTS;

        _log.writeToExecLog("New␣state:␣UP_WAITING_FOR_LISTS␣(" + _state + ")");
      }
```

```
}
else if(_state == STATE_UP_WAITING_FOR_LISTS){
  _log.writeToExecLog("Current␣state:␣UP_WAITING_FOR_LISTS␣(" + _state + ")");

  while(inMsgIter.hasNext()){
    Message inMsg = inMsgIter.next();
    int direction = inMsg.getDirection();

    if(direction == Message.DIR_REQ){
      handleRequestMessage(inMsg);
    }
    else if(direction == Message.DIR_RES){
      if(inMsg.getCommand() == Message.CMD_LST){
        _nodeMap.put(inMsg.getFromID(), (int[])inMsg.getData());
        _log.writeToExecLog("Received␣list␣from␣node:␣" + inMsg.getFromID());
      }
    }
  }

  if(_nodeMap.hasAllResourceLists()){
    _timeCounter = 0;

    _log.writeToExecLog("Checking␣for␣sleepiness");

    if(isSleepy()){
      _state = STATE_DOWN_IDLE;

      _log.writeToExecLog("Node␣is␣sleepy:␣will␣power␣off␣for␣" + _downTimeout + "␣
          cycles");
      _log.writeToExecLog("New␣state:␣DOWN_IDLE␣(" + _state + ")");
    }
    else{
      _state = STATE_UP_IDLE;

      _log.writeToExecLog("Node␣is␣not␣sleepy:␣will␣remain␣powered␣on");
      _log.writeToExecLog("New␣state:␣UP_IDLE␣(" + _state + ")");
    }
  }
}
else if(_state == STATE_DOWN_IDLE){
  _log.writeToExecLog("Current␣state:␣DOWN_IDLE␣(" + _state + ")");

  while(inMsgIter.hasNext()){
      handleRequestMessage(inMsgIter.next());
    }

    _timeCounter = (_downTimeout > 0 ? _timeCounter + 1 : 0);

    _log.writeToExecLog("Current␣time␣count:␣" + _timeCounter);

    if(_timeCounter == _downTimeout){
      _state = STATE_UP_IDLE;
      _nodeMap.clearResourceLists();
      _timeCounter = 0;

      _log.writeToExecLog("Timeout␣reached");
      _log.writeToExecLog("New␣state:␣UP_IDLE␣(" + _state + ")");
    }
  }
}

@Override
protected void handleRequestMessage(Message inMsg){
  int command = inMsg.getCommand();
  int direction = inMsg.getDirection();

  if(direction == Message.DIR_REQ){
    if(command == Message.CMD_LST){
      _log.writeToExecLog("Received␣list␣request␣from␣node:␣" + inMsg.getFromID());

      int[] resources = null;

      if((_state & MASK_STATE_DOWN) == MASK_STATE_DOWN){
        resources = new int[0];
      }
      else{
        resources = _resources;

        _messagesSent = _messagesSent + 1;
      }

      Message outMsg = new Message(_id, inMsg.getFromID(), Message.DIR_RES, Message.
          CMD_LST, resources);
      _outMessages.push(outMsg);
    }
  }
  }

  @Override
  protected boolean isSleepy(){
    boolean retVal = false;

    ResourceMap localList = new ResourceMap(_resources);
    int[] order = getOrder();
    printOrder(order);

    for(int i = 0; i < order.length; i++){
      int[] neighborList = _nodeMap.get(order[i]);

      for(int j = 0; j < neighborList.length; j++){
        localList.increment(neighborList[j]);

        _computations = _computations + 1;
      }

      if(!localList.hasUniqueResources()){
        retVal = true;
        break;
      }
    }

    return retVal;
  }

  private void printOrder(int[] order){
    StringBuilder builder = new StringBuilder();
    builder.append("Node␣order:␣");

    for(int i = 0; i < order.length; i++){
      builder.append(order[i] +  "␣");
    }

    _log.writeToExecLog(builder.toString());
  }
}
```

1) Largest-to-Smallest Policy Source Code: The following listing is the source code to the Largest-to-Smallest Order Policy. It is implemented as an extension of the Order-

Based Heuristic.

```
package com.bk.frappe.imp.node.order.ltos;

import java.util.Iterator;

import com.bk.frappe.imp.app.Log;
import com.bk.frappe.imp.node.order.OrderNode;

public class OrderLtoSNode extends OrderNode{
  public OrderLtoSNode(int id, int[] resources, int[] neighbors, int upTimeout, int
      downTimeout, int powerRatekWh, Log log){
    super(id, resources, neighbors, upTimeout, downTimeout, powerRatekWh, log);
  }

  protected int[] getOrder(){
    int counter = 0;
    int position = 0;

    int[] order = new int[_nodeMap.size()];
    Iterator<Integer> iterator = _nodeMap.keySet().iterator();

    while(iterator.hasNext()){
      int resCnt = _nodeMap.get(iterator.next()).length;
      counter = (resCnt > counter ? resCnt : counter);
    }

    while(counter >= 0){
      iterator = _nodeMap.keySet().iterator();

      while(iterator.hasNext()){
        int id = iterator.next();

        if(counter == _nodeMap.get(id).length){
          order[position] = id;
          position = position + 1;
        }
      }

      counter = counter - 1;
    }

    return order;
  }
}
```

2) Random Policy Source Code: The following subsections list the source code to the Random Order Policy. The policy is implemented as an extension of the Order-Based Heuristic, and possesses two supporting classes to help fulfill randomization operations.

a) OrderRandomNode Source Code: The following listing is the source code to the OrderRandomNode class. This class is the central point of the Random Order Policy, and is a direct descendant of the Order-Based Heuristic abstract class.

```
package com.bk.frappe.imp.node.order.random;

import java.util.Iterator;

import com.bk.frappe.imp.app.Log;
import com.bk.frappe.imp.node.order.OrderNode;

public class OrderRandomNode extends OrderNode{
  private IntArrayShuffler  _shuffler = null;

  public OrderRandomNode(int id, int[] resources, int[] neighbors, int upTimeout, int
      downTimeout, int powerRatekWh, Log log){
    super(id, resources, neighbors, upTimeout, downTimeout, powerRatekWh, log);
    _shuffler = new IntArrayShuffler();
  }

  protected int[] getOrder(){
    Iterator<Integer> iterator = _nodeMap.keySet().iterator();
    int[] retVal = new int[_nodeMap.size()];

    for(int i = 0; iterator.hasNext(); i++){
      retVal[i] = iterator.next();
    }

    _shuffler.shuffle(retVal);

    return retVal;
  }
}
```

b) IntArrayShuffler Source Code: The following listing is the source code to the IntArrayShuffler class. The purpose of this class is to shuffle into random order a provided list of resource ID numbers.

```
package com.bk.frappe.imp.node.order.random;

public class IntArrayShuffler{
    private static final int MAX_SHUFFLE_TIMES = 1000;
    private RandomNumberGenerator _ranGen = null;

    public IntArrayShuffler(){
        _ranGen = new RandomNumberGenerator(1);
    }

    public void shuffle(int[] array){
        for(int i = 0; i < MAX_SHUFFLE_TIMES; i++){
            int arraySize = array.length;

            for(int j = 0; j < arraySize; j++){
                int from_index = j;
                int to_index = (int)(Math.ceil(_ranGen.getRandomNumber() * arraySize) -
                    1.0);

                int temp = array[from_index];
                array[from_index] = array[to_index];
                array[to_index] = temp;
            }
        }
    }
}
```

c) RandomNumberGenerator Source Code: The following listing is the source code to the RandomNumberGenerator class. The purpose of this class is to supply a random number, which is used by IntArrayShuffler in its random shuffling operation.

```
package com.bk.frappe.imp.node.order.random;

public final class RandomNumberGenerator{
  private static final long A = 16807;
  private static final long M = 2147483647;
  private static final long Q = 127773;
  private static final long R = 2836;

  private long _val = 0;

  public RandomNumberGenerator(int seed){
    setSeed(seed);
  }

  public void setSeed(int seed){
    _val = seed;
  }

  public double getRandomNumber(){
    long x_div_q = _val / Q;
    long x_mod_q = _val % Q;
    long x_new = (A * x_mod_q) - (R * x_div_q);

    if(x_new > 0){
      _val = x_new;
    }
    else{
      _val = x_new + M;
    }

    return((double) _val / M);
  }
}
```