2007

The performance of Group Diffie-Hellman paradigms: a software framework and analysis

Kieran S. Hagzan

The Performance of Group Diffie-Hellman Paradigms: A Software Framework and Analysis

March 4, 2007

Kieran S. Hagzan
kieran@hagzan.homelinux.org
Department of Computer Science
Rochester Institute of Technology

Thesis Committee:

Advisor: Dr. Hans-Peter Bischof _____
Reader: Dr. Edith Hemaspaandra _____
Observer: Prof. Phil White _____

SUBMITTED TO THE
ROCHESTER INSTITUTE OF TECHNOLOGY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN COMPUTER SCIENCE

The Performance of Group Diffie-Hellman Paradigms: A Software Framework and Scalability Analysis

Abstract

A mobile computing environment typically involves groups of small, low-power devices interconnected through a mobile and dynamic network. Attempts to secure communication over these "ad-hoc" networks must be scalable to conserve the minimal resources of mobile devices as network sizes grow. In this project, the scalability of differing Group Diffie-Hellman security key generation implementations is examined. In theory, the implementation utilizing a data structure with the lowest theoretical run-time complexity for building the Diffie-Hellman group should prove the most scalable experimentally. A common modular framework was implemented to support generic Group Diffie-Hellman key agreement implementations abstracted from the underlying data structure and traversal mechanism. For comparison, linear, tree-based, and hypercubic Group Diffie-Hellman topologies were implemented and tested. Studies were conducted upon the results to compare the experimental scalability of each implementation to the other implementations as well as the theoretic predictions. The results indicate that the benefits of implementations with low theoretic-complexity are rarely experienced in smaller networks (less than 100 nodes,) and conversely implementations with high theoretic-complexities become unsuitable in larger networks (more than 100 nodes.) These experimental results match the theoretical predictions based on the mathematical properties of each implementation. Since mobile ad-hoc networks are typically small, less efficient, less complex implementations of Group Diffie-Hellman key agreement will suit most needs, however larger networks will require more efficient implementations.

The Performance of Group Diffie-Hellman Paradigms: A Software Framework and Scalability Analysis

Contents

This thesis is dedicated to my wife (who ensures that I remain sane,) and my parents (for without them, it would not exist.)

1 Introduction

1.1 Problem Description

Today small, mobile computing devices such as PDA's, laptops, tablet PC's, cellular phones, etc. are being utilized as much as their larger stationary counterparts. This has sparked the desire from end-users to experience the same functionality from these mobile devices without compromised security. Mobile devices are typically battery powered and designed with energy-efficient hardware (memory, processor, storage, peripherals, etc.) to conserve energy. Network media is typically wireless in these environments, communicating within a given physical broadcast range. These networks of devices are generally called "ad-hoc" networks due to their dynamic nature. This is clearly a new environment where scalability is necessary to preserve energy and compensate for the increased complexity of mobile network topology and security algorithms. As protocols used for maintaining mobile ad-hoc networks are mainly broadcast-based, the classical problems of point-to-point network computing evolve from two-party problems into arbitrary-party problems.

Data security has three main facets: confidentiality, integrity, and authenticity, typically accomplished through encryption, signatures, and hashing respectively. These algorithms typically utilize security keys as input, forcing security key generation algorithms to be addressed before the security algorithms themselves can proceed. In ad-hoc networks, a shared security key is required between all "trusted" devices in the network to secure broadcast transmissions. In addition, each device in an ad-hoc network is not required to be within the transmission range of every other device in the network (fore more info, see Obraczka et al. [2001]). In this scenario, security keys should change whenever the network topology changes, as well as on a regular basis to force forward and backward security. This high quantity of key agreement iterations combined with minimized computing resources further bolsters the requirement for scalability of implementation.

One classical solution to shared-key agreement is the Diffie-Hellman algorithm extended to groups. Diffie-Hellman itself is generically an algorithm between two parties who desire to establish a security key. The Diffie-Hellman algorithm can be extended using multiple iterations, where the parameters of each iteration are governed by an underlying data structure which maps the network onto an organized topology. The following sub-sections describe the notation used herein, the Diffie-Hellman algorithm and network topology variants implemented, and finally the expected theoretical results for each paradigm.

1.2 Background

1.2.1 Mathematical Notation

Note: Information in this section is summarized from the contents of Trappe and Washington [2002, pp63-103]

Groups, Rings, and Fields:

- Groups
  - A $binary\,operation$ on a set S is an operation that maps a pair $(a, b) \in S \times S$ to a value $a \in S$.
  - A group G is a 2-tuple, $(S, *)$ where S is a set and $*$ is a binary operation upon that set. The group must fulfill three specific properties:
    1. The group must be associative over the set S, i.e., $a * (b * c) = (a * b) * c$ for all $a, b, c \in S$.
    2. The group must define an identity element i such that $a * i = i * a = a$ for all $a \in S$.
    3. The group must define an inverse element a^{-1}such that $a*a^{-1} = a^{-1}*a = 1$ for all $a \in S$
  - If $a * b = b * a$ for all $a, b \in S$, a group G is considered a commutative or abelian group.
  - A group is considered finite if $|G|$ is finite. $|G|$ denotes the order of G.
  - A group G is denoted a $cyclic\,group$ if there exists a $g \in G$ such that for all $b \in G$ there exists an integer value i where $b = g^i$. Such a value g is denoted a $generator$ of the group G.
  - Every finite group of a prime order is cyclic.
- Rings
  - A ring is a 3-tuple $(S, +, \times)$ where:
    1. S is a set.
    2. $+$ and $\times$ are binary operations denoted "addition" and "multiplication" respectively.
    3. $(S, +)$ is a commutative group with additive identity 0
    4. The operation $\times$ is associative over the set S.
    5. There exists a multiplicative identity of 1 where $1 \times a = a \times 1 = a$ for all $a \in S$.
    6. The operation $\times$ is distributive over $+$, i.e. $a \times (b + c) = (a \times b) + (a \times c)$ for all $a, b, c \in S$.
  - A ring is commutative if $a \times b$ is commutative over S.
  - An element r of a ring R is denoted an $invertible\,element$ if there exists an $s \in R$ such that $r \times s = 1$, where s is denoted the $inverse$ of r.

- Fields
  - A field is a commutative ring R where all elements $\{i \in R, i \neq 0\}$ are invertible elements.
  - $\mathcal{F}_q$ denotes a $finite\, field\, of\, order\, q$, where $|\mathcal{F}| = q$ and q is finite.
  - The elements $\{r \in \mathcal{F}_q, r \neq 0\}$ form a group over multiplication denoted $\mathcal{F}_q^*$, the $multiplicative\, group\, of\, \mathcal{F}_q$.
  - A generator of the group $\mathcal{F}_q^*$ is known as a $primitive\, element$ of $\mathcal{F}_q$.

Exponentiation and the Logarithm

- Exponentiation in $\mathbb{R}$
  - Exponentiation defined over the real numbers "$\mathbb{R}$" is a function mapping elements of $\mathbb{R} \times \mathbb{R}$ to $\mathbb{R}$, denoted:
    $a = b^c$, $\{a, b, c \in \mathbb{R}\}$.
  - The *logarithm* function is defined over the real numbers as the inverse function of exponentiation, denoted:
    $\log_b(a) = c$, $\{a, b, c \in \mathbb{R}\}$,
    read "c is the logarithm root b of a."
- Exponentiation in $\mathbb{Z}$
  - Mathematical functions defined over the integers "$\mathbb{Z}$" belong to a class of functions known as $discrete\, functions$. Discrete functions often utilize the "congruence modulo" relation (or the remainder of integer division) over equality, a discrete function that maps elements of $\mathbb{Z}\times\mathbb{Z}$ to elements in $\mathbb{Z}$.
  - Congruence modulo a prime n generates a finite field of order $(n-1)$. The generator g of such a field is denoted a $primitve\, root$ modulo n.
  - Modular exponentiation is the discrete equivalent of exponentiation over the real numbers, denoted:
    $a \equiv b^c \,(mod\, n)$, $\{a, b, c, n \in \mathbb{Z}\}$
  - The discrete logarithm function is defined over the integers as the inverse function of modular exponentiation, denoted:
    $L_{D_b}(a) \equiv c \,(mod\, n)$, $\{a, b, c, n \in \mathbb{Z}\}$
    read "c is congruent to the $discrete\, logarithm$ root b of a, modulo n".
  - If the modulus n of a discrete logarithm is prime, then there exists at least one generator g of the cyclic group order $(n-1)$, and the discrete logarithm root g is therefore defined for all values $\{i \in \mathbb{N}, 1 \leq i \leq (n-1)\}$.

1.2.2 Two-Party Diffie-Hellman Extended To Groups

- *The Two-Party Diffie-Hellman Protocol*

In 1976, W. Diffie and M.E. Hellman published "New Directions in Cryptography," introducing their two-party key-agreement protocol. The protocol was intended to exchange a secret key over an insecure medium, and is based upon the difficulty of computing the discrete logarithm in a finite field. The exchange protocol proceeds as follows (from Trappe and Washington [2002, pp210-211]):

Precondition: Two Parties, "Alice" and "Bob", share public values p (a large prime,) and g (a primitive root modulo p.)	
Alice 1.) Alice generates a random private value x_a, and computes the public value: $p_{k_A} = g^{x_a} (mod\, p)$ 2.) Alice sends p_{k_A} and receives p_{k_B}. 3.) Alice computes shared key k: $k = p_{k_B}^{x_a} (mod\, p) = (g^{x_b})^{x_a} (mod\, p) = g^{x_a x_b} (mod\, p)$	*Bob* 1.) Bob generates a random private value x_b, and computes the public value: $p_{k_B} = g^{x_b} (mod\, p)$ 2.) Bob sends p_{k_B} and receives p_{k_A}. 3.) Bob computes shared key k: $k = p_{k_A}^{x_b} (mod\, p) = (g^{x_a})^{x_b} (mod\, p) = g^{x_a x_b} (mod\, p)$

Table 1

- *The Security of Diffie-Hellman*

In the Diffie-Hellman protocol, the shared key k is mathematically the result of exponentiation modulo p. If the values g, p, p_{k_A}, and p_{k_B} are public, an attacker would need to determine the value of $x_a x_b$ by computing the discrete logarithm root g modulo p of both $p_{k_A}$and p_{k_B}. Knowing only the public values of the protocol, this is considered mathematically infeasible, provided p is chosen as a "significantly large" prime and g is truly a primitive root modulo p. Considering the relation of congruence modulo n maps elements of the natural numbers to a finite field of order $(p-1)$, many exponents of g can produce the public keys p_{k_A} and p_{k_B}, since g is a generator of the cyclic group of order $(p-1)$. If p is sufficiently large, then an exhaustive search through all the $(p-1)$ possible exponents of g to determine the set S of possible values for x_a and x_b is computationally infeasible. Furthermore, all values of $x_a \in S$ and $x_b \in S$ would need to be exhausted to determine the values of $x_a x_b$.

In an environment where an eavesdropper "Eve" may be intercepting communication between Alice and Bob, Diffie-Hellman key agreement is vulnerable

to a "man-in-the-middle" attack. In this attack, Eve intercepts p_{k_A} from Alice and sends the public key $p_{k_{E_1}}$ to Bob. Eve then intercepts p_{k_B} from Bob and sends $p_{k_{E_2}}$ to Alice. At this point, Alice and Eve have established key k_1 and Eve and Bob key k_2. Eve can now intercept messages from Alice, modify and inspect the message contents, and transmit to Bob. Eve can also intercept messages from Bob and transmit to Alice maliciously. This attack is possible since no authentication of Alice or Bob is done during the protocol exchange. Solutions to this form of attack classically include digital signatures and certificates based upon public/private key-pairs obtained prior to a run of the Diffie-Hellman protocol. More recently, modified protocols certify Diffie-Hellman based public-keys before two parties exchange them.

- Extending Diffie-Hellman

The generic two-party Diffie-Hellman algorithm can be extended to groups as follows (from Steiner et al. [1996]) :

Precondition: Two Parties, Alice and Bob, have computed a Diffie-Hellman key $k_{i,j}$ and a third party wishes to secure group communication	
Alice / Bob	*Charlie*
1.) Alice and Bob use the key from the prior iteration ($k_{i,,j}$) for the generated private value x_{ab} and computes the public value: $p_{k_{Ab}} = g^{x_{ab}} (mod\, p)$	1.) Charlie generates a random private value x_c and computes the public value: $p_{k_c} = g^{x_c} (mod\, p)$
2.) Either Alice or Bob (or both) send $p_{k_{Ab}}$. Both receive p_{k_c}.	2.) Charlie sends p_{k_c} and receives $p_{k_{Ab}}$.
3.) Alice and Bob compute shared key k_{abc}: $k_{abc} = p_{k_c}^{x_{ab}} (mod\, p) = (g^{x_c})^{x_{ab}} (mod\, p) = g^{x_{ab}x_c} (mod\, p)$	3.) Charlie computes shared key k_{abc}: $k_{abc} = p_{k_{ab}}^{x_c} (mod\, p) = (g^{x_{ab}})^{x_c} (mod\, p) = g^{x_{ab}x_c} (mod\, p)$

Table 2

In general, to extend the Diffie-Hellman algorithm to a third party using the original two-party (Alice and Bob) algorithm (call the third party "Charlie"), Alice and Bob first perform the typical Diffie-Hellman exchange. The key value from a prior exchange denoted k_{ab} is then used by the Alice and Bob as the private contribution to a second Diffie-Hellman exchange with Charlie. Charlie broadcasts his public key p_{k_c} and thus both Alice and Bob are able to compute the second-round Diffie-Hellman key as can charlie after receiving public key $p_{k_{ab}}$. This process can be repeated as needed for a naive implementation of Group Diffie-Hellman without involving an underlying structure. To improve upon things, the nodes can also be abstracted to have a notion of structure such that key generation is done in an organized and predetermined fashion to take advantages of what parallelism is available.

In general, Diffie-Hellman key agreement is a function over $\mathbb{Z}_p^*$ that maps $\{[x_a, x_b] \in \mathbb{Z}_p^* \times \mathbb{Z}_p^*\}$ to $\{k \in \mathbb{Z}_p^*\}$ if p and g are constant (Trappe and Washington [2002, pp210-211]). To extend this notion to groups, the notion of a group itself must be clearly defined. In cryptography and data networking, groups and networks are typically synonymous in syntax and semantics, and in formal definition. In this work, a network or group is formally defined as a collection of two or more nodes arranged in a defined topology with a defined interconnection mechanism between the nodes. The Diffie-Hellman algorithm may be custom-tailored iteratively or recursively to traverse a given topology in a manner dependent upon the interconnection mechanism.

The following sections illustrate the linear, tree-based, and interconnection mechanisms for extending Diffie-Hellman to groups. These three connection mechanisms were chosen following research of the works Amir et al. [2001] and Amir et al. [2002]because the communication and computational cost for each is predictable based on the data structure of the implementation and as such the implementation should approximate our theoretical predictions. In each algorithm definition, we assume an underlying vector for the storage of nodes, with each node at a position p_i in the storage vector. That vector and the mathematical definition of the interconnection network are the sole means by which each topology is interpreted from a data vector into a visual structure. This interpretation, as opposed to an object-oriented view of a tree, hypercube, or array, supports a generic framework implementable in any programming language supporting arrays and broadcast networking.

1.2.3 Linear Group Diffie-Hellman

- *Topology and Interconnection Mechanism*

Consider a group of n nodes arranged in a vector or list at indices 0 through $n-1$, with an interconnection mechanism as follows:

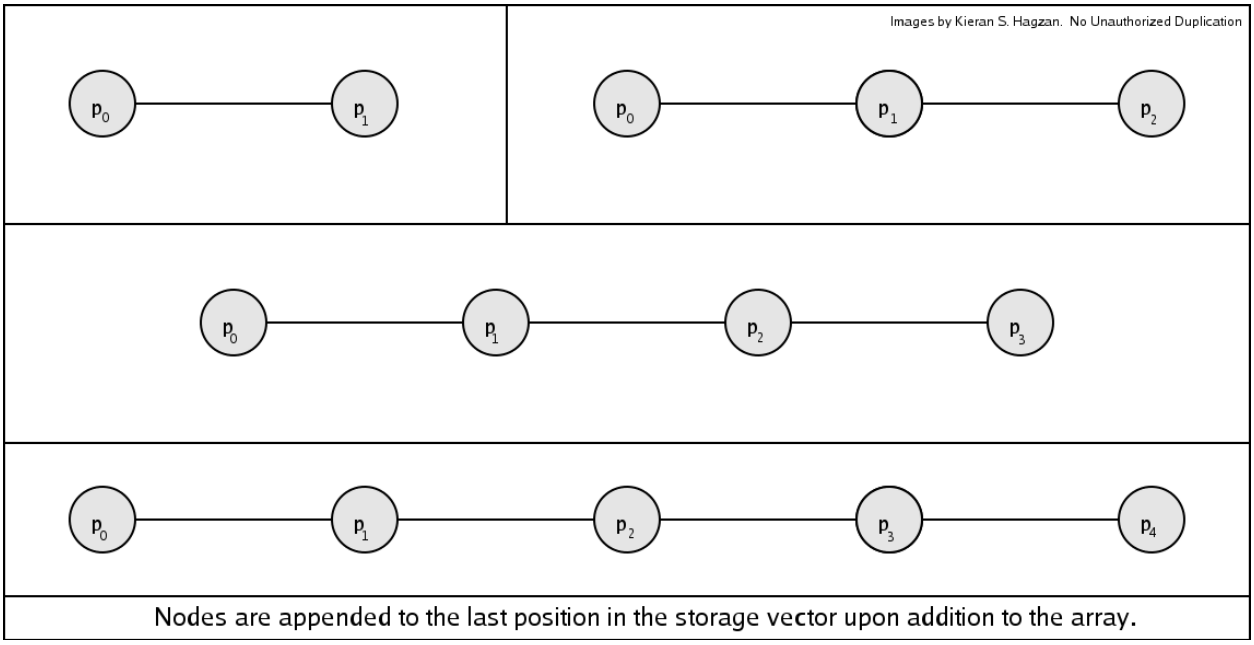


Figure 1

In the case of Linear Group Diffie-Hellman, we define the interconnection network as two connections for each node at position p_i , the "left" connection to the node at position p_{i-1} and the "right" connection to the node at p_{i+1} as shown in Figure 1. There exists no connection on the left of node p_0 nor to the right of node p_{n-1}.

- Topology Maintenance

1. Network Sponsorship

   The node at the highest index $n-1$ in the storage vector is designated the network sponsor. The sponsor is responsible for maintaining any alterations that occur to the topology.

2. Addition

   Upon addition to the array, nodes are inserted at the nth position in the storage vector, and network sponsorship and connectivity are updated accordingly.

3. Removal

   Upon removal from the array, a node in position p_i is deleted from the storage vector, the positions of all nodes with index greater than p_i are decremented, and network sponsorship and connectivity are updated accordingly.

4. Merging/Partitioning

   To merge two arrays a_0 and a_1, the storage vector representing a_1 is concatenated to the vector representing a_0, then network sponsorship and connectivity are updated accordingly. Partitioning a set of nodes s out of an array a occurs by performing a removal operation of each element in the set s on the array a.

- Linear Diffie-Hellman Propagation

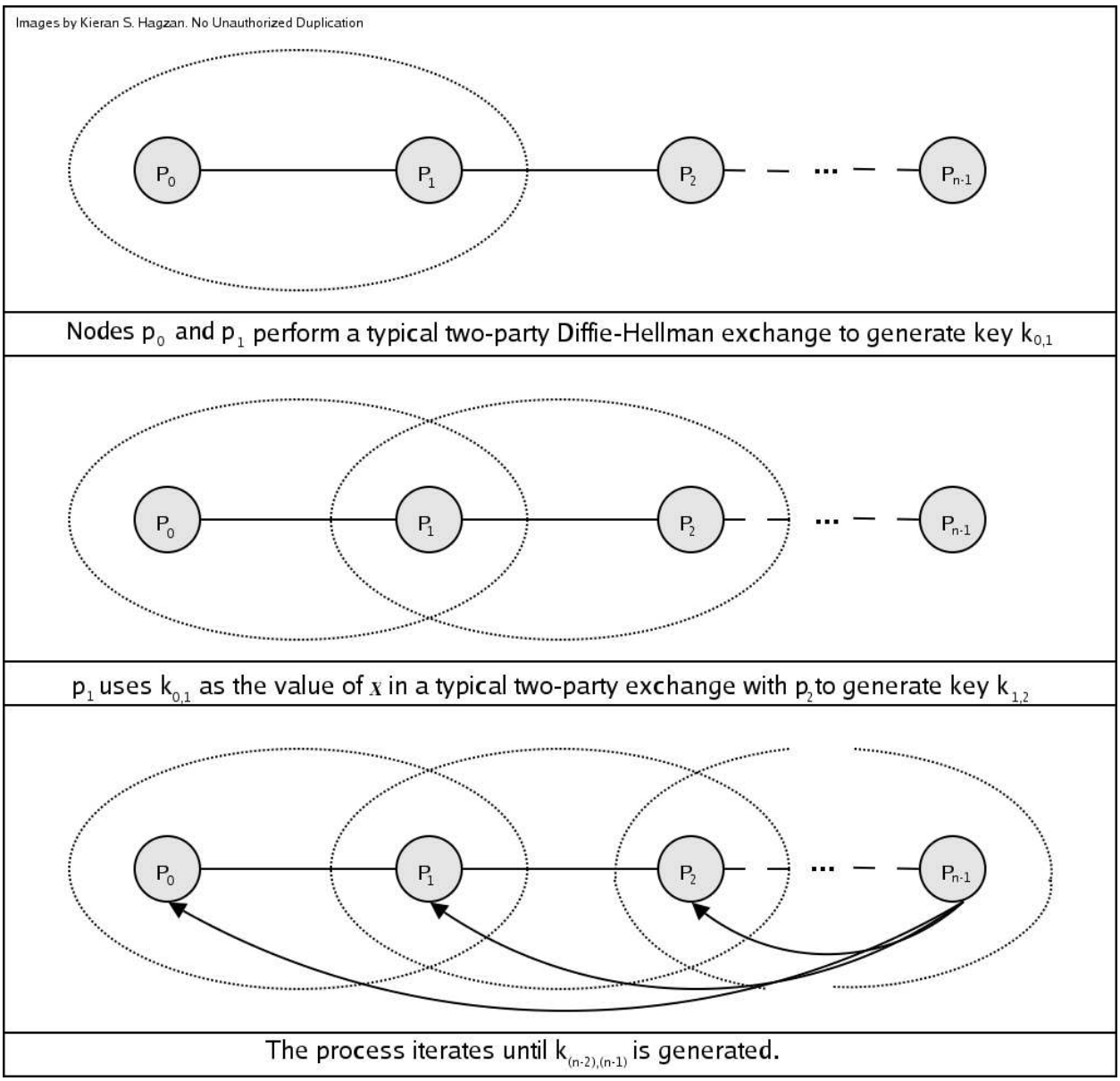


Figure 2

In the case of Linear Group Diffie-Hellman, the extending the typical two-party algorithm to groups amounts to applying the algorithm iteratively across the storage

vector of nodes from lowest index to highest based on the methods in section(1.2.2). The notion of Linear GDH was extruded from the CLIQUES protocol of Steiner et al. [1998]. Shown in Figure 2, initially the nodes at position p_0 and p_1 perform a two-party Diffie-Hellman exchange, using privately generated contributions of x_0 and x_1 to compute the public keys p_{k_0} and p_{k_1}, swapping them to computing key $k_{0,1}$. Node p_1 then uses the computed value of $k_{0,1}$ as the private contribution $x_{0,1}$ in an exchange with node p_2. p_2 uses privately generated contribution $x_2$to compute public key p_{k_2} and $p_1$generates a new "combined" public key $p_{k_{0,1}}$ from $x_{0,1}$ (which again, is the computed key $k_{0,1}$ from the last round). The process is repeated up to node p_{n-1}, i.e. node p_i contributes $k_{i-1,i}$ in a two party exchange with node p_{i+1} to generate combined public key $p_{k_{i-1,i}}$ (for all $i > 0$) and node p_{i+1} uses contribution x_{i+1} to generate public key $p_{k_{i+1}}$ until $i + 1 = n - 1$. When a key exchange occurs between the nodes at indices $n - 2$ and $n - 1$, the node at index $n - 1$ is aware of the "plain" public keys p_0 and p_{n-1} as well as "combined" public keys $p_{k_{i-1,i}}$ for $0 < i < n - 1$. The keys are then broadcast to the remaining nodes p_i for $0 < i < n - 1$, who must compute all keys $k_{i,i+1}$for all $i < n - 2$ (the rightmost node computes the group key in its first and only exchange). Mathematically the group key $k_{n-2,n-1}$is computed as:

$$k_{0,1} = (g^{x_0})^{x_1} (mod\, p) = g^{x_0 x_1} (mod\, p)$$
$$k_{i,i+1} = (g^{k_{i-1,i}})^{x_{i+1}} (mod\, p) = g^{(k_{i-1,i})(x_{i+1})} (mod\, p)$$
$$k_{n-2,n-1} = (g^{k_{n-3,n-2}})^{x_{n-1}} (mod\, p) = g^{(k_{n-3,n-2})(x_{n-1})} (mod\, p)$$
$$...$$

1.2.4 Tree-Based Group Diffie-Hellman

- Topology and Interconnection Mechanism

Consider a group of n nodes arranged in a list or array representing the leaves of a binary tree, with an interconnection network as follows:

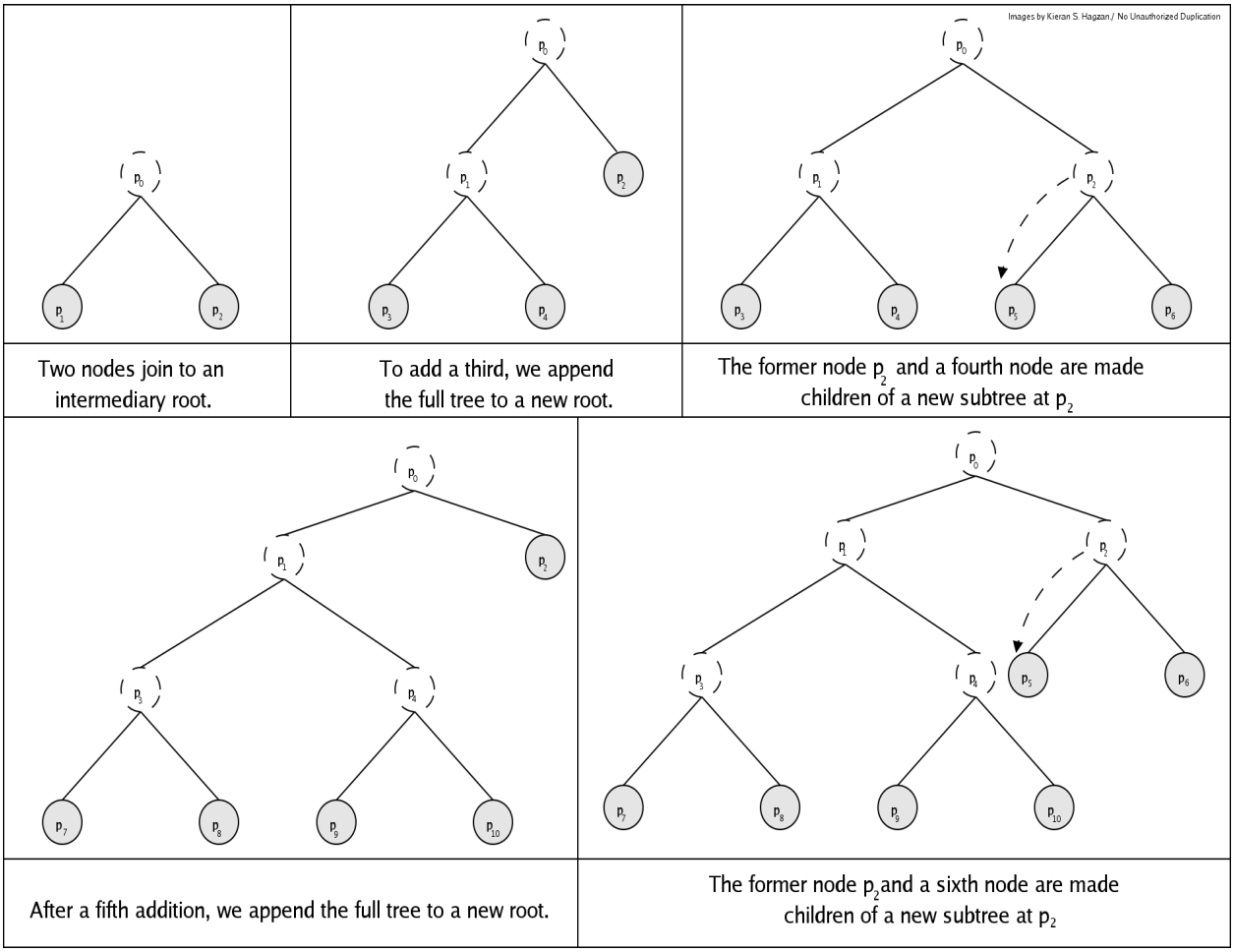


Figure 3

Tree-based Group Diffie-Hellman is implemented based upon the work of Kim et al. [2002]. In the case of TGDH, an interconnection network is defined as three connections for each node at position p_i in the storage vector. The "parent" connection to a node is at position $[(p_i - 1)/2]$, the "left child" connection of a node is at position $[2p_i + 1]$, and the "right child" of a node is at position $[2(p_i + 1)]$ in the storage vector. There exists no parent connection for the root node at position 0 in the storage vector, and no "offspring" connections to children of leaf nodes, by definition. A binary tree is considered "complete" if the number of leaf nodes in the tree is an even power of 2, and "incomplete" otherwise. The $leftmost\ shallow$ node of a tree is the leftmost leaf node whose depth is less than the total depth of the tree.

- Topology Maintenance

  The Tree-Based Group Diffie-Hellman protocol assumes that the binary tree is maintained such that all true computing "devices" are at the leaves of the binary tree. All non-leaf nodes are denoted $intermediary$ nodes, intended as "place-holders" during key agreement and key computation. In the above diagrams, nodes with dashed borders are intermediary nodes, filled nodes with solid borders are actual computing devices.

  1. Network Sponsorship

     The rightmost leaf-node in the topology is designated the network sponsor. The sponsor is responsible for maintaining any alterations that occur to the topology.

  2. Addition

     Addition to the tree occurs in a manner dependent on the tree's completion. If the tree is complete, then the existing tree is appended as the left subtree of a new intermediary root node, and the joining node is added as its right sibling. If the tree is complete, the leftmost-shallow node is appended as the left child of a new subtree rooted at the leftmost-shallow position, and the joining node is added as its sibling (see **Figure 3**.)

  3. Removal

     It is possible to decrease the depth of the tree and preserve left-to-right filling upon removal from a tree, dependent on the type (intermediary or actual) of the sibling of the node considered (p_i) for removal.

     If the sibling of p_i is a leaf node:

     If the sibling of the node to be deleted is a leaf node (node p_4 in the top-right illustration of Figure 3) then the sibling of p_i is stored in a temporary location and the subtree rooted at p_i's "uncle" (parent's sibling) is "promoted" to be p_i's parent. The original sibling of p_i is then returned from temporary storage via normal addition to the tree.

     If the sibling of p_i is an intermediary node:

     If the sibling of the node to be deleted is an intermediary node (node $p_2$in the bottom-left illustration of Figure 3) then the subtree rooted at p_i's sibling is promoted to be its parent.

1. Merging/Partitioning

   Merging two trees t_1 and t_2 occurs by "pruning" or collecting the leaf nodes of t_2 and performing a single addition for each to t_1. Partitioning a set of nodes from a tree occurs by performing a single removal operations for each node in the set to be partitioned from the tree.

- Tree-Based Diffie-Hellman Propagation

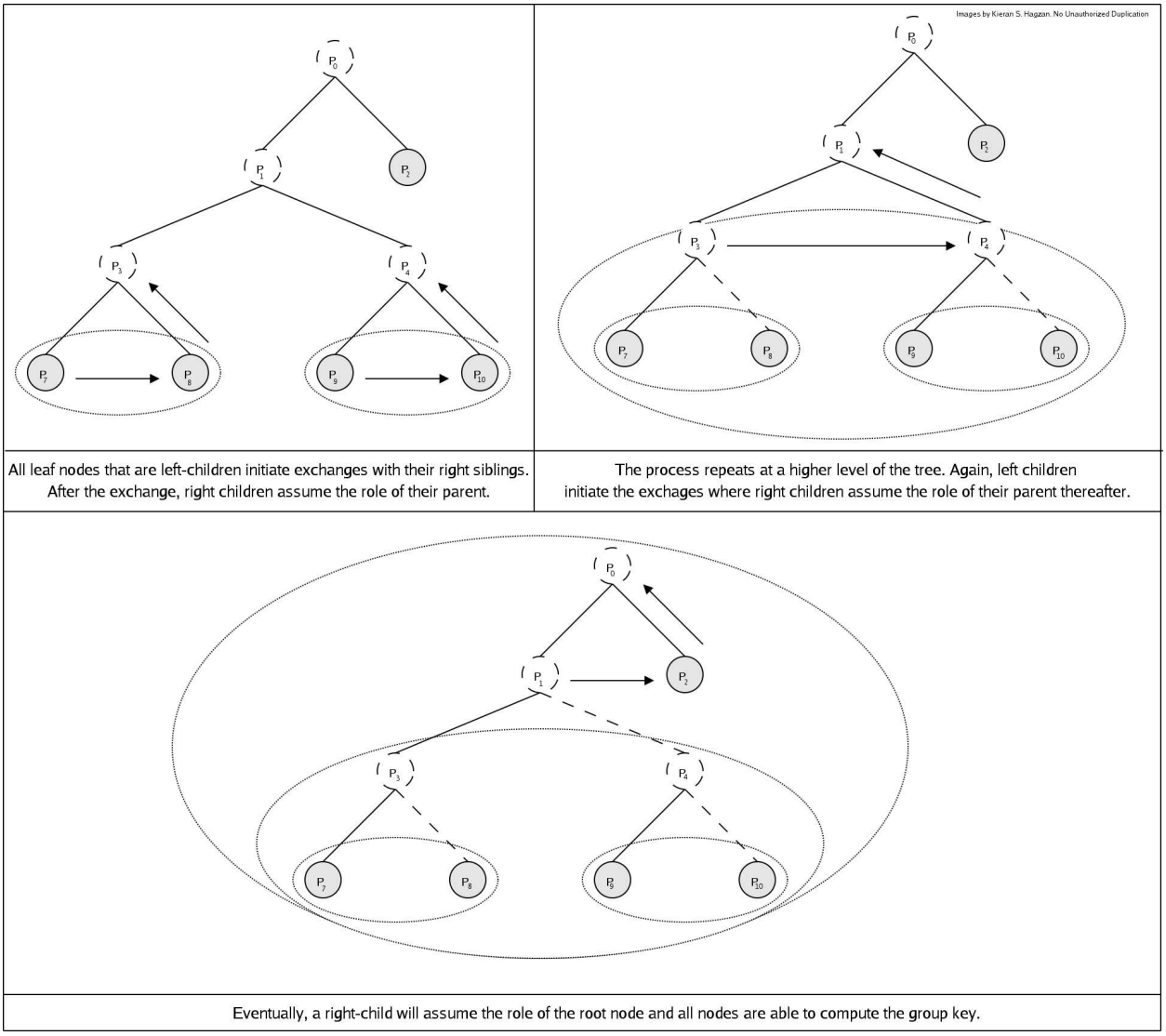


Figure 4

In the case of Tree-Based Group Diffie-Hellman, extending the typical two-party algorithm to groups requires applying the algorithm recursively in a depth-first, upward propagation through the topology. The general algorithm proceeds by computing keys for all non-leaf nodes in the topology, using the contributions $x_0 ... x_{n-1}$ propagated toward the root from the leaves. To initiate the exchange protocol, the sponsor (rightmost leaf node) broadcasts a message indicating the update request. The formal algorithm continues as follows:

1. All nodes p_i, repeat until the public keys for all nodes between p_i and the root node p_0 are known:

(a) All leaf nodes check the current topology to determine first if the node is the left or right child of its parent, and secondly if its sibling is a leaf or intermediary node.

(b) If the node represents a left child:

   i. It will initialize a two-party exchange with:

      A. Either the sibling directly if the sibling is a leaf

      B. Or the node "representing" the sibling (the rightmost leaf node of the subtree rooted at the sibling) if the sibling is an intermediary.

   and then listen for public key broadcasts. Otherwise,

(c) If the node represents a right child:

   i. The node will wait until either:

      A. Its sibling directly, or

      B. The node representing its sibling

   initializes an exchange with it. After a two-party key is computed and public-keys are broadcast, this right sibling then assumes the role (becomes "representative") of its parent.

In this algorithm, key generation begins in the leaf nodes and proceeds toward the root with Diffie-Hellman exchanges taking place between sibling nodes at the same depth. After an exchange, if a node is a left child of its parent, it simply waits for the broadcast of the public-key tree. Mathematically, a key between siblings $k_{l,r}$ is computed as follows for any right child r using private contribution x_r and sibling l who has public key p_{k_l}:

$k_{l,r} = (g^{x_r})^{p_{k_l}} (mod\, p) = g^{k_r p_{k_l}} (mod\, p)$

After the computation of $k_{l,r}$ if the parent of the siblings is not the root, the right sibling will then promote itself to be the parent, and use the value of $k_{l,r}$ as a private contribution to compute the "combined" public key $p_{k_{l,r}}$ and the left sibling waits for a broadcast of all public keys in the tree. If the parent is the root, the right node broadcasts the tree of all public keys (plain at the leaves and combined for the internal nodes), at which point all nodes can compute the keys for nodes between themselves and the root, eventually computing the group key at the root. In the top center illustration of Figure 3, the group key k_{p_0} at node p_0 is computed from an exchange between nodes p_1 and p_2 as follows:

1. The private contribution of node p_1 (denoted k_{p_1}) is computed by a Diffie-Hellman exchange between p_3 and p_4:
   For node p_3, the value of$k_{p_1} = (g^{x_3})^{p_{k_4}} (mod\, p) = g^{k_3 p_{k_4}} (mod\, p)$
   For node p_4, the value of $k_{p_1} = (g^{x_4})^{p_{k_3}} (mod\, p) = g^{k_4 p_{k_3}} (mod\, p)$

2. The node p_4 promotes itself to be the active node p_1, using k_{p_1} as private contribution x_1 to generate public key p_{k_1}.

3. An exchange is then performed between p_1 and sibling node p_2. p_2 uses public key p_{k_2} from its privately generated contribution x_2 as follows:
   For node p_1, the value of k_{p_0} is
   $k_{p_0} = (g^{x_1})^{p_{k_2}} (mod\, p) = g^{x_1 p_{k_2}} (mod\, p) = g^{(x_4 p_{k_3}) p_{k_2}} (mod\, p) = g^{(x_3 p_{k_4}) p_{k_2}} (mod\, p)$
   which follows from step 1 and for node p_2 the value of k_{p_2} is
   $k_{p_o} = (g^{x_2})^{p_{k_1}} (mod\, p) = g^{x_2 p_{k_1}} (mod\, p)$

Following the progression of steps to see the computation of key k_{p_0}, we can see that the key at the root of the tree is truly a contributory key comprised of contributions from each node in the tree.

1.2.5 Hypercubic Group Diffie-Hellman

- Topology and Interconnection Mechanism

Consider a group of n nodes arranged in a list or array representing a hypercube, with an interconnection network as follows:

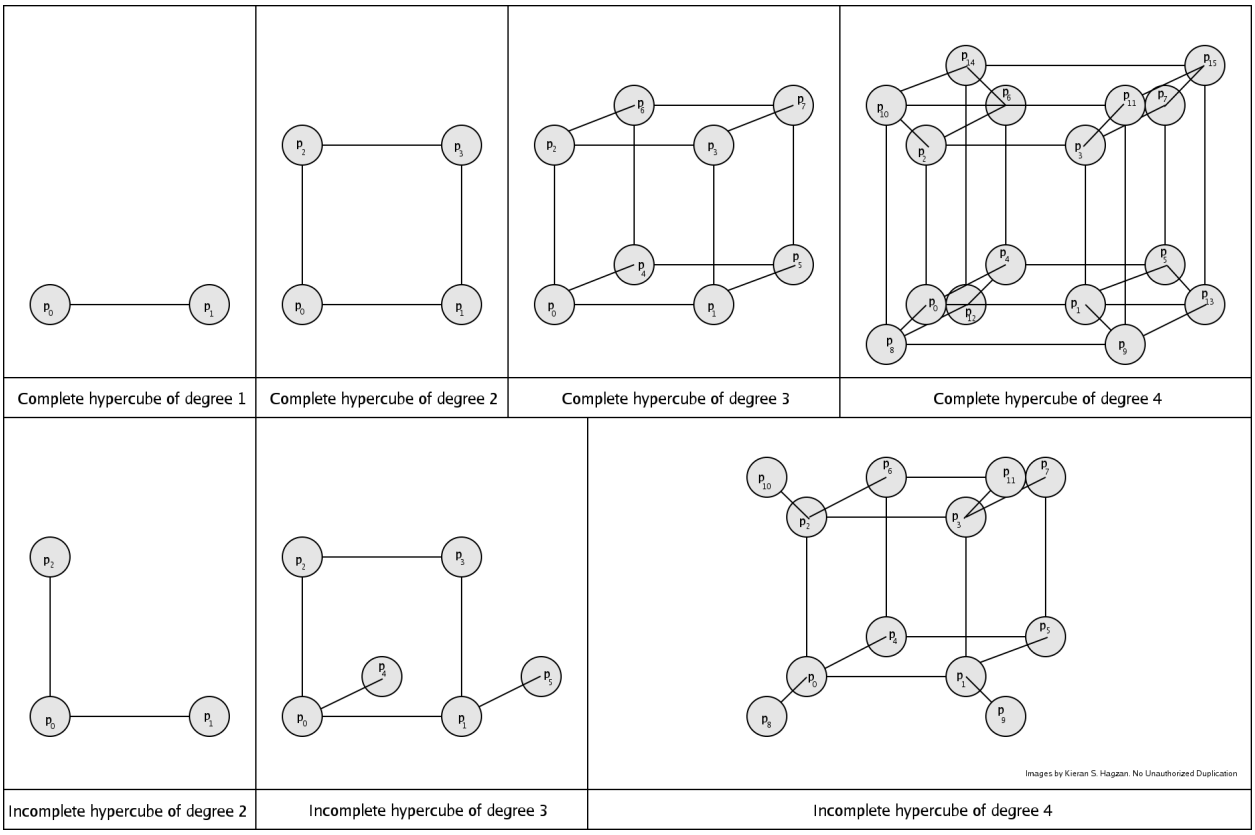


Figure 5

Hypercubic Group Diffie-Hellman is based upon the "Octopus Protocol" described by Becker and Wille [1998]. The notion of the octopus protocol is minimizing the total number of Diffie-Hellman exchanges in the overall group. The other advantage of the hypercube is that using our Gray Code mapping the structure fits into an array perfectly with no wasted space as in the tree-based model, allowing for a minimum data transfer when the group changes to re-key (McGrew and Sherman [1998]). In the case of Hypercubic Group Diffie-Hellman, an interconnection network is defined for each node based upon the degree, or dimension of the hypercube. The degree of the hypercube is defined as the number of bits required to represent the highest index($n-1$) of the storage vector holding the nodes of the hypercube. For instance, if the hypercube contains 8 nodes, $\{p_0 ... p_7\}$, the hypercube is of degree 3, since 3 bits are required in binary to encode the decimal index 7. If the hypercube contains 23 nodes, the hypercube is of degree 5 since 5 bits are required to represent the decimal integer 22 (10110 in binary). Connections to other nodes for any particular node p_i

are determined by "flipping" the the bits representing index i from most significant bit to least. Flipping bits in this order guarantees that any tenticular nodes perform their exchange with an inner node first to minimize synchronization issues later. For example:

> If the hypercube is of degree 4 containing up to 15 nodes, the node at index p_0 would represent its index as 0000. Therefore, node p_0 would have connections to nodes with indices at {1000, 0100, 0010, 0001} in binary, or $\{p_8, p_4, p_2, p_1\}$. Node p_1 would represent its index as 0001 and have connections with {1001, 0101, 0011, 0000} in binary, or $\{p_9, p_5, p_3, p_0\}$. If the degree of the hypercube were 5, node p_0 would have connections with $\{p_{16}, p_8, p_4, p_2, p_1\}$ and node p_1 would have connections with $\{p_{17}, p_9, p_5, p_3, p_0\}$, etc.

A hypercube is considered complete if the number of nodes contained is a even power of 2. When a hypercube is complete, the interconnection network is as described above. If the hypercube is incomplete (the number of nodes contained is not an exact power of 2), those nodes with an index whose most significant bit is set to 1 are considered *tenticular* nodes. Tenticular nodes posses only a single connection to the node whose index bitmask is that of the current node with the most significant bit flipped to 0. In the diagram above, p_8, p_9, p_{10}, p_{11} are tenticular nodes within the incomplete hypercube of degree 4 and size 12. Note that p_8 whose index is 1000 in binary has only a connection to 0000, or p_0, 1001 to 0001, 1010 to 0010, and 1100 to 0100 in this case. A hypercube node is $fully\,connected$ within its dimension if the node is either a member of a complete hypercube, or is not a member of the outermost dimension of an incomplete hypercube.

- Topology Maintenance

  1. Network Sponsorship

     The node at the last position $n - 1$ in the storage vector is designated the network sponsor. The sponsor is responsible for maintaining any alterations that occur to the topology.

  2. Addition

     Upon addition to the hypercube, nodes are inserted at a new $(n + 1)$st position in the storage vector, then network sponsorship and connectivity are updated accordingly.

  3. Removal

     Upon removal from the hypercube, a node in position p is deleted from the storage vector, the positions of all nodes with index greater than p is decremented, then network sponsorship and connectivity are updated accordingly.

  4. Merging/Partitioning

To merge two hypercubes h_1 and h_2, the storage vector representing h_2 is concatenated to the vector representing h_1, then network sponsorship and connectivity are updated accordingly. Partitioning a set of nodes from a hypercube occurs by performing a single removal operations for each node in the set to be partitioned from the hypercube.

- Hypercubic Diffie-Hellman Propagation

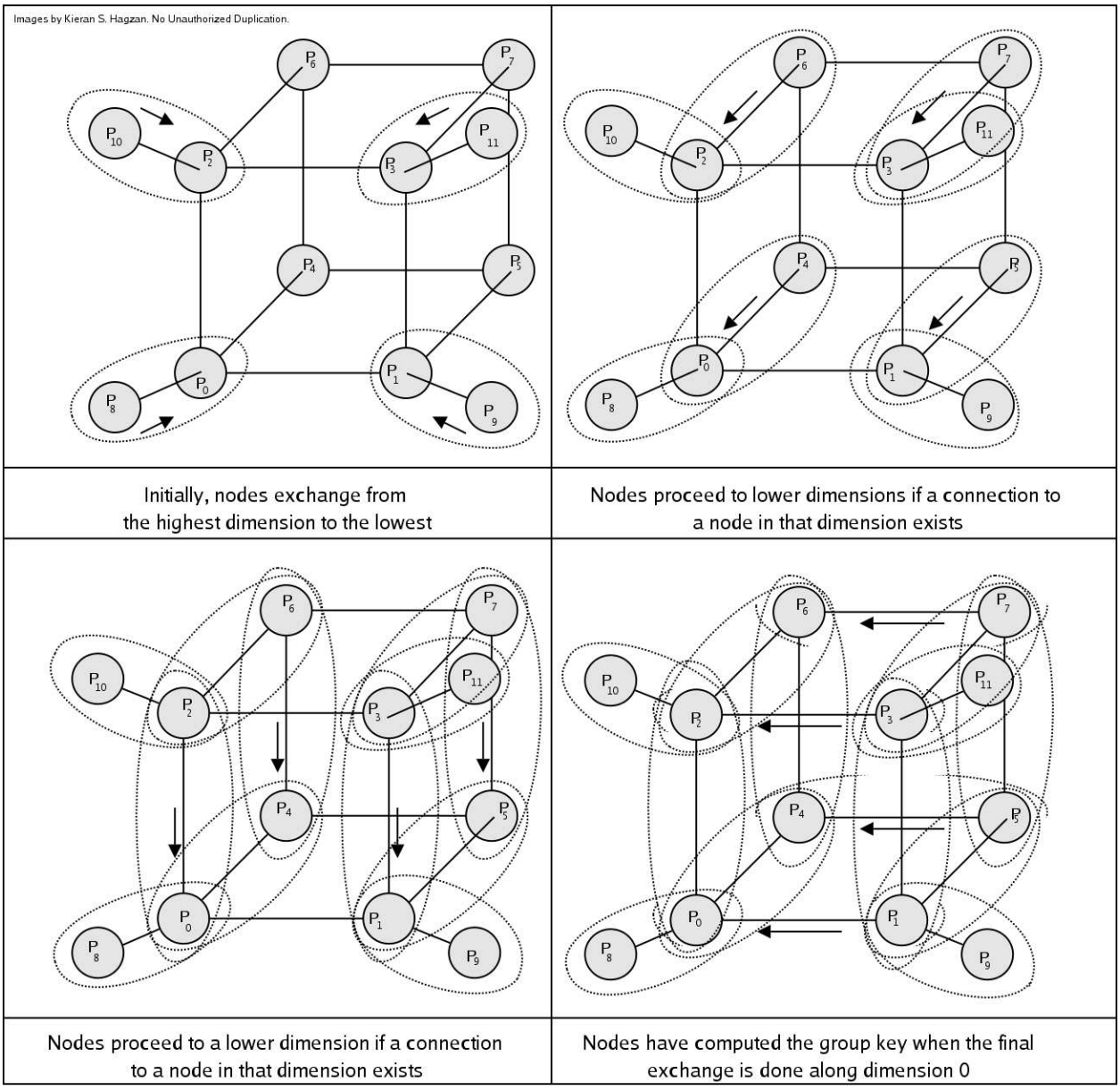


Figure 6

In the case of Hypercubic Group Diffie-Hellman, extending the typical two-party algorithm to groups requires for each node p_i, collapsing the hypercube based upon the index i from greatest dimension to dimension 1. A hypercube of dimension 1 is formally two nodes with a bi-directional connection between them, where the typical two-party algorithm is applied. For a node at position p_i in the storage vector of a hypercube of dimension d, the formal algorithm proceeds as follows:

$k = d$

While $k \geq 1$:

1. Determine the index of node $p_{i_k}^{\wedge}$, the result of flipping the value of bit k in the binary representation of the index i:

   (a) If the node $p_{i_k}^{\wedge}$ exists, perform an exchange with the node

   (b) If the node $p_{i_k}^{\wedge}$ does not exist, do nothing (this only happens when the outer dimension of the hypercube is incomplete.)

2. $k = k - 1$

When $k = 1$, then the sponsor node at array index 0 will broadcast the hypercube of public keys to all nodes such that any tenticular nodes in an incomplete outer dimension can generate the group key. In the bottom center illustration of Figure 3, the nodes p_4 and p_5 would compute the group key as follows:

1. The highest dimension nodes of the hypercube (in this example p_4 and p_5) will exchange with their counterpart nodes found by flipping the most significant bit of their index. In this example:
   p_4 has index 4 with binary representation 100 which will exchange with index 000 or node $p_0$to compute key $k_{4,0}$
   $k_{4,0} = (g^{x_4})^{p_{k_0}} (mod\, p) = g^{x_4 p_{k_0}} (mod\, p)$
   $p_3$has index 5 with binary representation 101 which will exchange with index 001 or node p_1 to compute key $k_{5,1}$
   $k_{5,1} = (g^{x_5})^{p_{k_1}} (mod\, p) = g^{x_5 p_{k_1}} (mod\, p)$

2. The outer dimension nodes p_4 and p_5 will now "freeze" and wait for the broadcast of public keys subsequently by nodes p_0 and p_1 respectively. The inner dimension nodes will continue on to collapse on the dimensions of the structure, in this example:
   p_2 with binary representation 010 will exchange with node p_0 with representation 000 to compute key $k_{2,0}$. In this exchange p_0 will use the value of $k_{4,0}$ computed in the outer dimension as its private contribution to compute public key $p_{k_{4,0}}$.
   $k_{2,0} = (g^{x_0})^{p_{k_2}} (mod\, p) = (g^{k_{4,0}})^{p_{k_2}} (mod\, p) = g^{(g^{x_4 p_{k_0}}) p_{k_2}} (mod\, p)$
   p_1 with binary representation 001 will exchange with node p_3 with binary representation 011 to compute $k_{3,1}$. In this exchange, p_1 will use the value of $k_{5,1}$ computed in the outer dimension as its private contribution to compute public key $p_{k_{5,1}}$.
   $k_{3,1} = (g^{x_1})^{p_{k_3}} (mod\, p) = (g^{k_{5,1}})^{p_{k_3}} (mod\, p) = g^{(g^{x_5 p_{k_1}}) p_{k_3}} (mod\, p)$

3. A final exchange is computed along the innermost dimension:
   $p_1$with representation 001 will flip its least significant bit to exchange with p_0 with binary representation 000 to compute the final key:
   $k_{1,0} = (g^{x_0})^{p_{k_1}} (mod\, p) = g^{(k_{2,0})(p_{k_1})} (mod\, p) = g^{(g^{(g^{(x_4)(p_{k_0})})} p_{k_2}) p_{k_1}}$
   p_3 with representation 011 will flip its least significant bit to exchange with p_2 with binary representation 010 to compute the final key:

$k_{3,2} = (g^{x_3})^{p_{k_2}} (mod\, p) = g^{(k_{3,1})(p_{k_2})} (mod\, p) = g^{(g^{(g^{(x_5)(p_{k_1})})p_{k_3})} p_{k_2}}$

4. Node p_0 will send the public keys used in the exchanges with first p_2 and secondly p_1 to node p_4. At the time of the exchange with node p_2, node p_0 used the value of $k_{4,0}$ to generate public key $p_{k_{4,0}}$, so p_4 can also generate $p_{k_{4,0}}$ and compute first $k_{2,0}$ as node p_0 did in step 2 and then repeat step 3 exactly to produce key $k_{0,1}$. Node p_1 will also send the public keys used in the exchanges with first p_3 and secondly p_0 to node p_5. At the time of the exchange with node p_3, p_1 used the value of $k_{5,1}$ to generate the public key $p_{k_{5,1}}$, so p_5 can now also generate $p_{k_{5,1}}$ and compute first $k_{3,1}$ as p_1 did in step 2 and then repeat step 3 exactly to produce key $k_{0,1}$.

1.2.6 Hypothesis

From Lee et al. [2002], heoretical bounds on the running times of each Group Diffie-Hellman protocol are based upon the number of two-party Diffie-Hellman computations performed by each node which in turn is based upon group size. Theoretical bounds on network data transfer for each algorithm are based upon the count and number of public-key broadcasts needed which is also based upon group size . This section presents the predicted theoretical results of running time and data transfer for the three paradigms. In each of the three cases, the data plotted is a representation of the complexity bound of the algorithm. When complexities are plotted, the notion of an additive and multiplicative constant are allowed to compensate for environmental factors. Therefore, the graphs that follow are meant to illustrate the rate of change in experimental timing data expected as group size grows, not necessarily meant to predict the experimental timing data for any given group size.

In general, when plotting a complexity of n for linear Group Diffie Hellman, this actually implies the general plot of $cn + d$ where c and d are constants, when plotting $2(n-1)$ for tree based Group Diffie-Hellman, this implies the general formula of $2c(n-1) + d$ where c and d are constants. The same follows for all graphs presented here for theoretical timing models. Finding such constants c and d for all of the experimental data such that a model can be mapped to predict running time for large groups using LGDH, TGDH, and HGDH is quite possible using numerical analysis methods such as LeGendre polynomials and Taylor Expansion, however such computation is beyond the scope of this work. The goal is to make a comparative analysis between the three protocols to examine if experimental data follows the theoretical trends mentioned in these hypotheses, and to make a conjecture about any deviation found in the experimental data.

1.2.6.1 Linear Group Diffie-Hellman (LGDH) During an instance of the LGDH protocol, each of the n nodes in the storage array could perform at most 2 exchanges, one with their left partner, and one with their right. The node at position p_0 has no left partner and the node at p_{n-1} has no right partner, so the worst-case scenario occurs for the node at position 0, who must perform n two-party computations.

Network data transfer consists of each node broadcasting at most 2 public keys, the public key used in the exchange with their right partner, and the public key used in the exchange with their left partner. The node at position p_0 has no left partner and the node at p_{n-1} has no right partner, so the total number of public keys broadcast to determine the key for a group of size n is $2(n-1)+c$ where c is a constant. LGDH is an algorithm where at most two nodes at any given time are "active," or currently performing a two-party exchange.

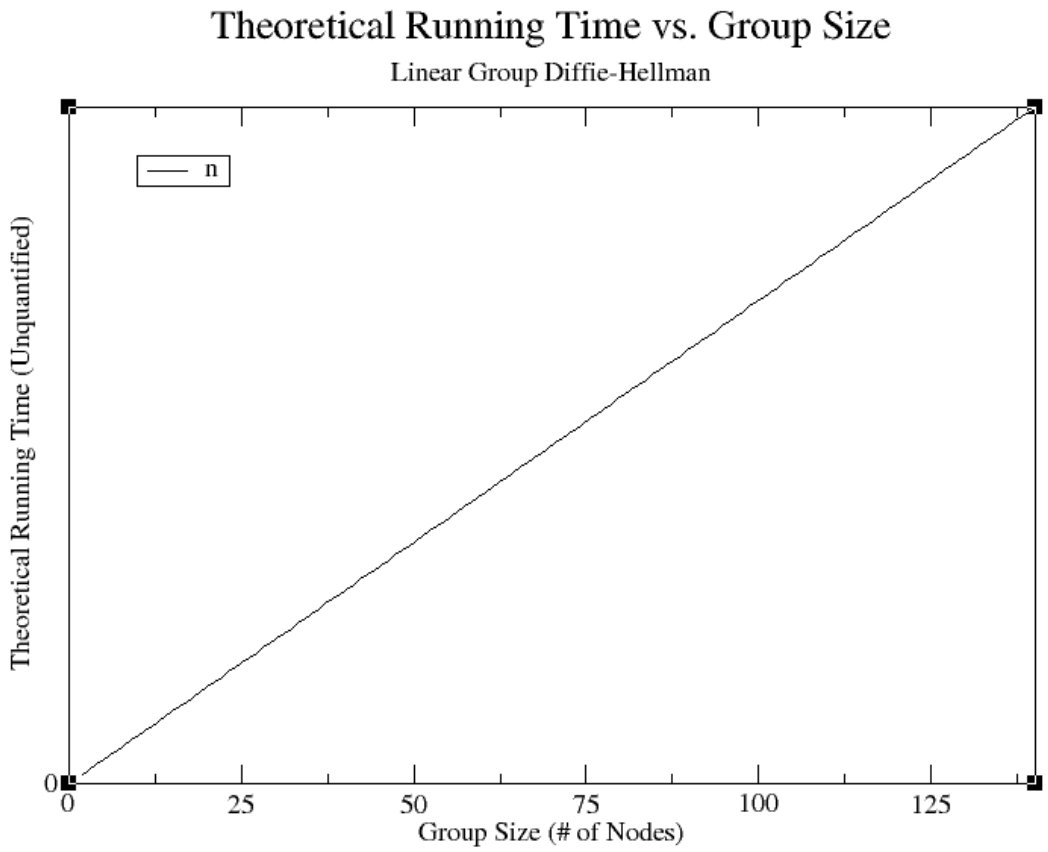
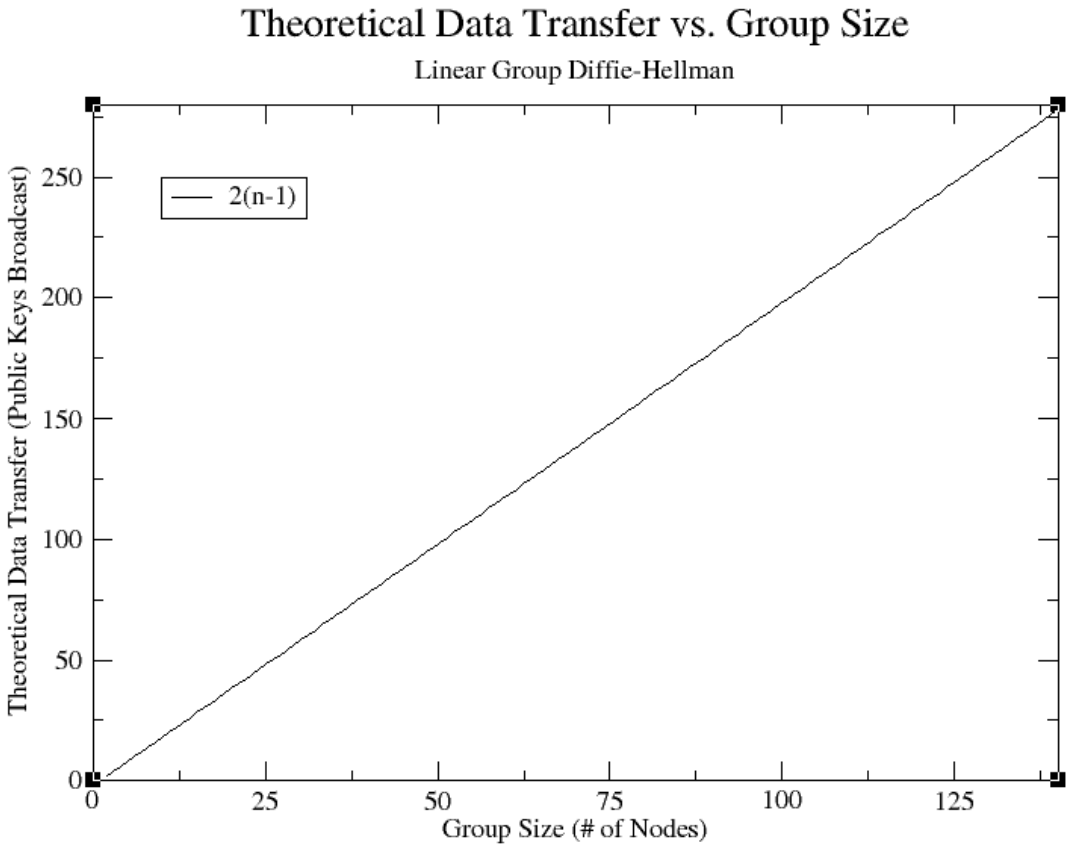


Figure 7

The Theoretical Running Time of LGDH



Theoretical Data Transfer Quantities for LGDH

1.2.6.2 Tree-Based Group Diffie-Hellman (TGDH) During an instance of the TGDH protocol, the number of exchanges is proportional to the number of exchanges per node multiplied by the node count. The number of exchanges per node is proportional to the height of the tree at hand. The height of the tree at hand is proportional to the mathematical ceiling of the logarithm base 2 of the number of leaves in the tree. For instance, a tree containing 7 nodes has a height of 3, therefore, each node could perform at most three exchanges before computing the root key. Since there are 7 nodes, a total of 21 exchanges are performed in total. In general, for a tree of size n, there are $ceiling(log_2(n))$ exchanges performed per node.

Network data transfer consists of two public-key broadcasts for all two-party exchanges. The number of two-party exchanges in total is equal to $n * ceiling(log_2(n))$, and the number of key-broadcasts is therefore $2n * ceiling(log_2(n))$. TGDH is an algorithm that is moderately parallelized; initially all leaf nodes are actively performing two-party exchanges at the same time, then leaves who are left children cease and those who are right children continue to compute, etc.

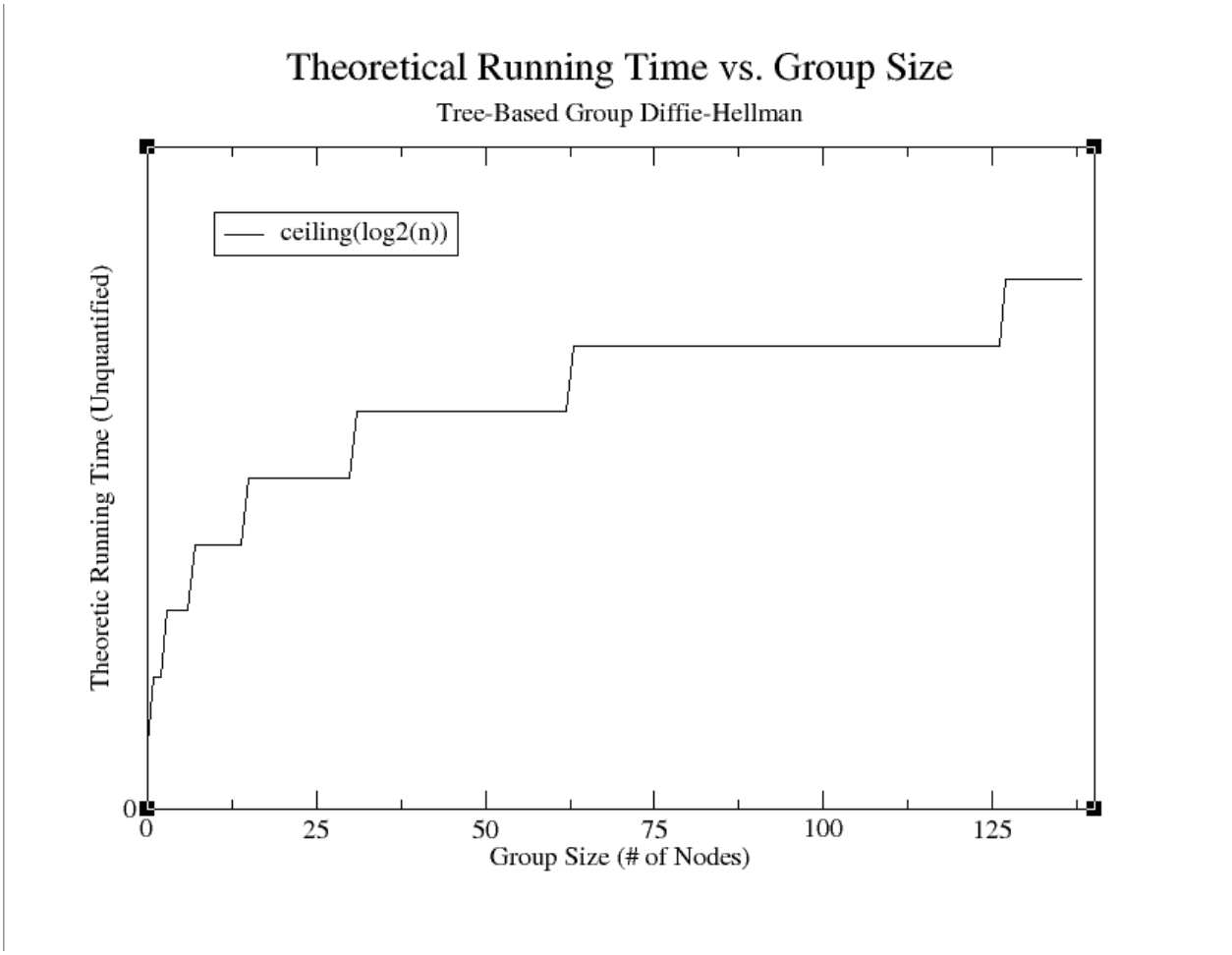
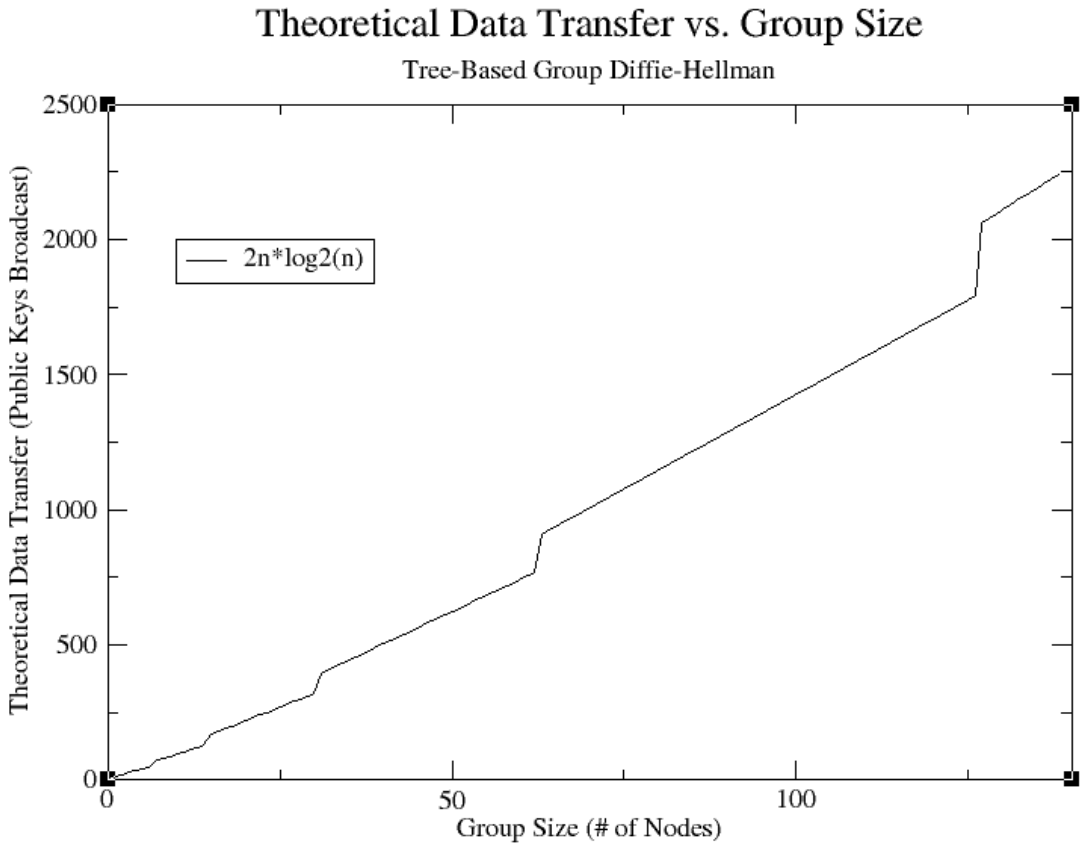


Figure 8

The Theoretical Running Time of TGDH



Theoretical Data Transfer Quantities for TGDH

1.2.6.3 Hypercubic Group Diffie-Hellman (HGDH) During an instance of the HGDH protocol, the number of exchanges is proportional to the number of exchanges per node. The number of exchanges per node is proportional to the dimension of the hypercube at hand. The dimension of the hypercube is proportional to the mathematical ceiling of the logarithm base 2 of the number of nodes. For instance, a cube containing 7 nodes is of dimension 3, therefore, each node could perform at most three exchanges before computing the root key. Since there are 7 nodes, a total of 21 exchanges are performed in total. In general, for a cube of size n, there are $ceiling(log_2(n))$ exchanges performed per node.

Network data transfer consists of two public-key broadcasts for each two-party exchange performed. Therefore, the number of public keys broadcast is $n*ceiling(log_2(n))$. HGDH is a fully parallelized algorithm, i.e., no node is inactive at any point during a key computation.

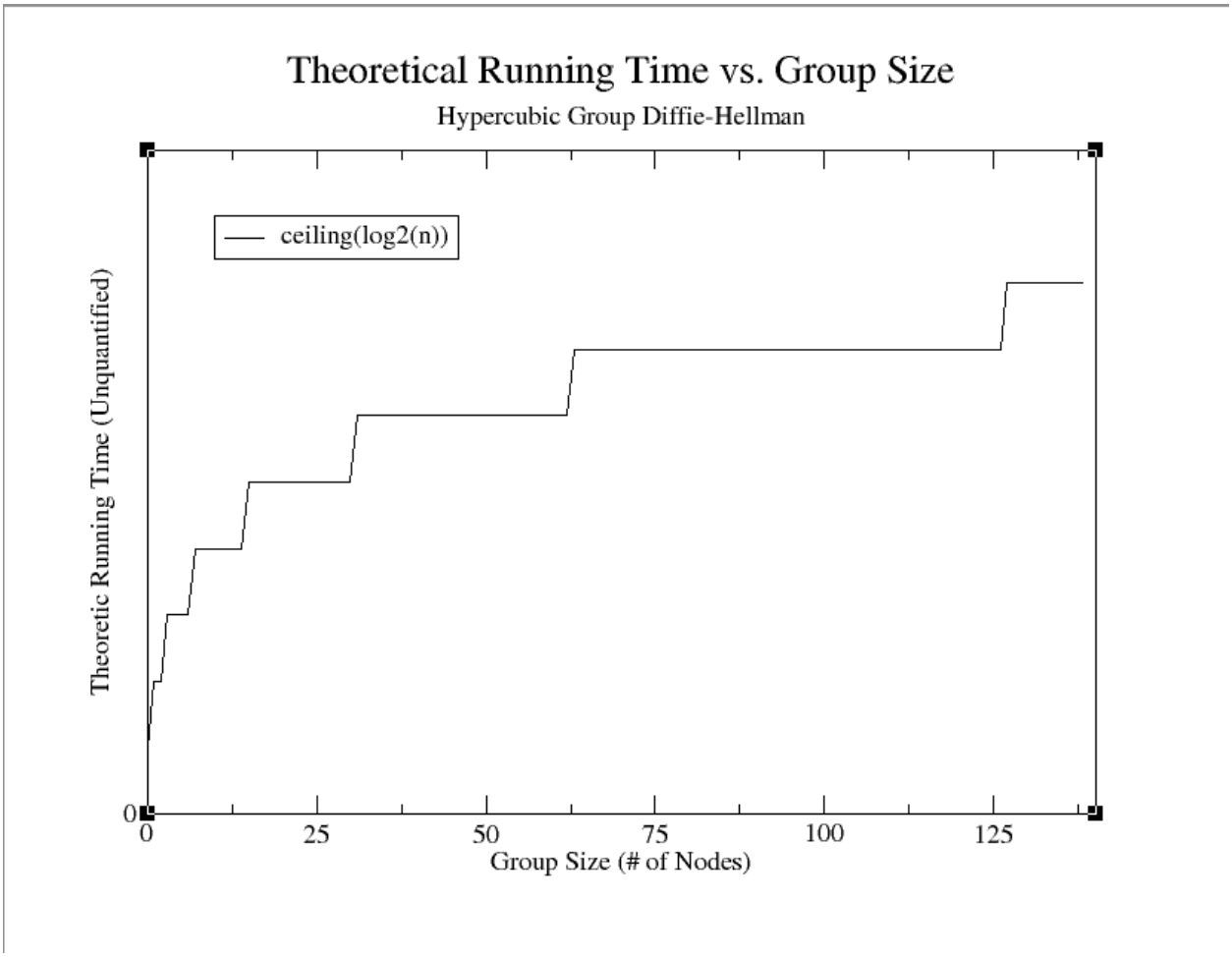
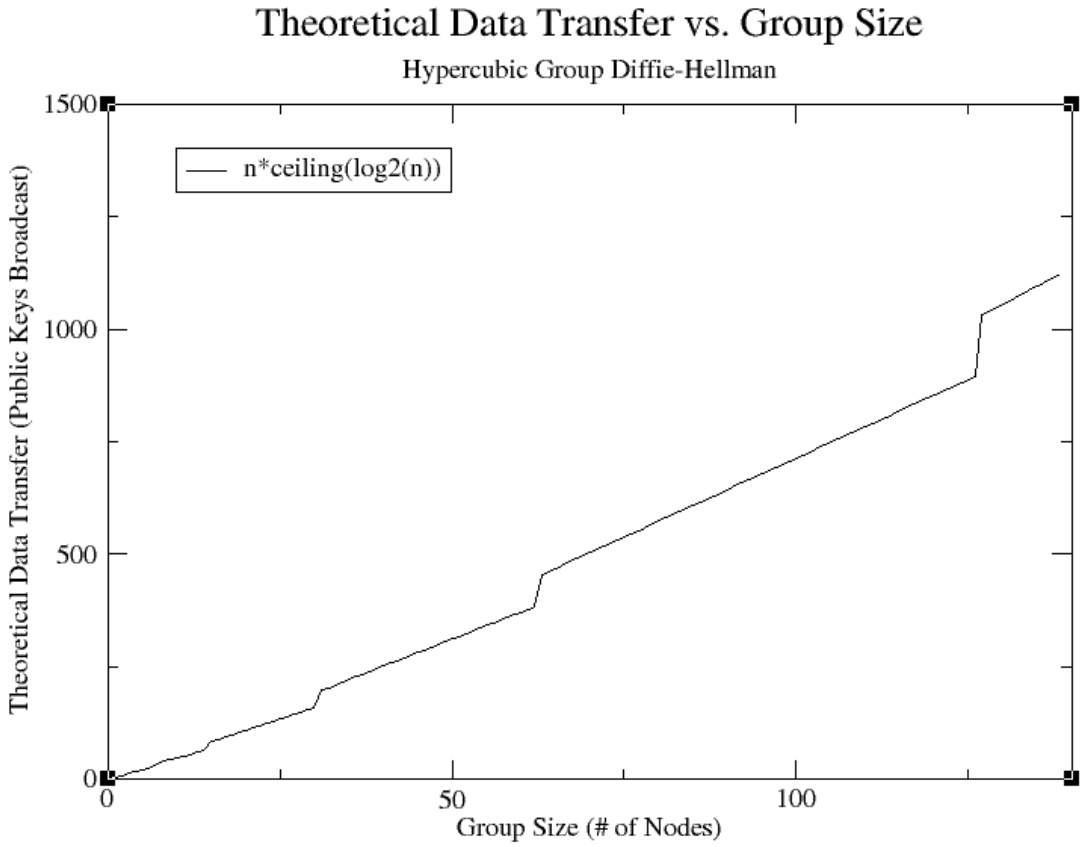


Figure 9

The Theoretical Running Time of HGDH



Theoretical Data Transfer Quantities for HGDH

4. Discussion of Hypothesis

Of the three group Diffie-Hellman algorithms presented, two algorithms (TGDH, HGDH) have logarithmic running times, and LGDH has a linear running-time. TGDH and HGDH are also parallelized algorithms, where as LGDH exhibits no parallelism whatsoever. Considering data-transfer quantities needed to generate a group key, all have quantities that are linearly proportional to the group size. Of these, LGDH experiences the fewest broadcast messages, HGDH the second least, and TGDH the greatest. Given these data, it is reasonable to assume that LGDH will perform the worst in large groups, HGDH the best, and TGDH somewhere in between. Environmental influences that will greatly effect key generation time are both network latency and the average processing ability of the devices at hand. In general, we can expect the theoretical results to be somewhat related to the number of Diffie-Hellman computations and network broadcasts in any given key exchange topology for a group of size n based on the works by Amir et al. [2003], Becker and Wille [1998], Amir et al. [2001], and Amir et al. [2002].

2 Methods

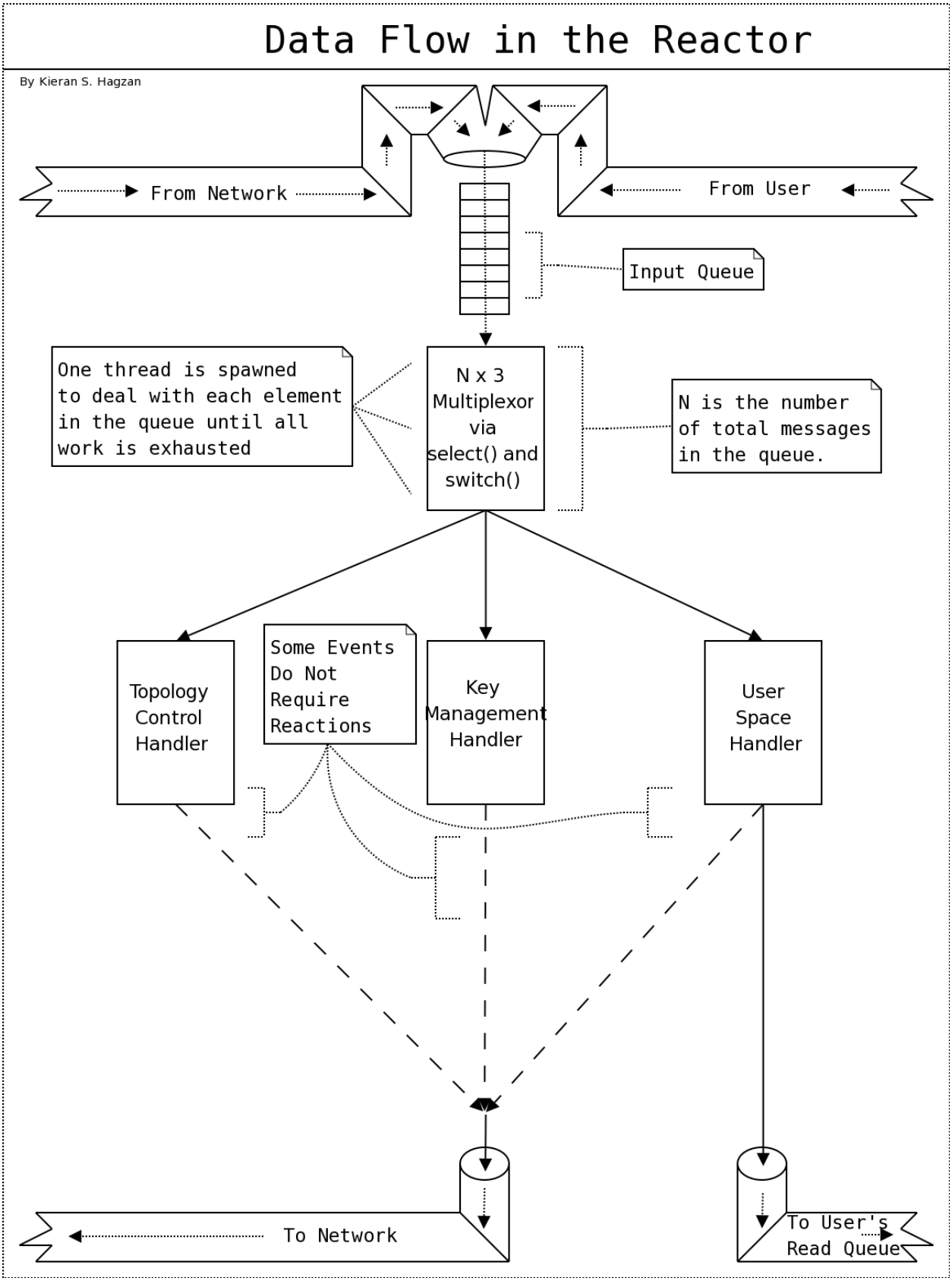
2.1 Software Design Model

2.1.1 Internal I/O Handling

To properly simulate a broadcast ad-hoc network environment within a wired network, network packets are addressed to the network's broadcast address. Typically, the broadcast address for a network is the network portion of the network IP address group masked with the network's *subnet mask* (combine the IP and subnet mask with the logical AND operation). Sending packets to this address allows for each node to send exactly one packet which is theoretically received by all other hosts listening to the broadcast address. Due to our broadcast addressing schema, UDP transport is used over TCP transport, introducing packet-loss and forcing the robustness of the underlying design. Efforts have been made by Adamson et al. [2000]to create reliable multicast in this same fashion.

Introduced in the Java programming language version 1.4 was the notion of the standard "select()" system call to perform I/O multiplexing, along with non-blocking I/O. Typically, this technique is used to reduce CPU overhead by polling the native network hardware to determine if there is any I/O to perform. If so, data is read to be processed, a response is generated and sent, finally the CPU will return back to the "select()" system call. If no I/O is ready, the CPU will either sit idle, or do some other work that is lying around. This is accomplished by a multithreaded, event-driven "reactor" pattern often used with large-scale and high-demand network programs. One main thread will hand each event-request off to a child thread, who will then do some work and die-off. Since the tasks performed by the child-threads are often long-lived, the benefits of multithreading outweigh the overhead of thread-creation and startup and is beneficial overall.

Network messages are split into three categories of handlers: topology control, key agreement, and user-space handlers. Each category is dependent on all others, so there is heavy state-synchronization that occurs between threads handling requests. The category of any particular message is encoded in the first byte of the message. The general design pattern is noted in the figure below.


Data Flow in the Reactor
By Kieran S. Hagzan
From Network
From User
Input Queue
One thread is spawned to deal with each element in the queue until all work is exhausted
N x 3 Multiplexor via select() and switch()
N is the number of total messages in the queue.
Topology Control Handler
Some Events Do Not Require Reactions
Key Management Handler
User Space Handler
To Network
To User's Read Queue

2.1.2 Topology Maintenance Abstraction

In order to standardize the topology management handler of the system, the operations on the topology were generalized into four distinct operations: join, leave, merge, and partition. The bulk operations of merge and partition are often expressed as multiple applications of join and leave. Depending on the topology however, the bulk operations can occasionally be accomplished by a single operation. Regardless of implementation, all Diffie-Hellman based key agreement strategies should be able to support these three operations, under the following preconditions and postconditions:

Join:
Precondition: One or more nodes exist in a group to join.
Postcondition: The joining node is added to the topology representative of the key agreement strategy at hand. The topology is distributed to all members, and the key-agreement layer is notified to update the key collaboratively, ensuring forward security.

Leave:
Precondition: One or more nodes desires to leave an existing group.
Postcondition: The leaving node is removed from the topology representative of the key agreement strategy at hand. The topology is distributed to all members, and the key-agreement layer is notified to update the key collaboratively, ensuring backward-security.

Merge:
Precondition: Two or more groups of nodes exist that wish to merge.
Postcondition: The merging groups are combined into one larger group. The merged topology is distributed to all members of both groups, and the key-agreement layer is notified to update the key collaboratively, ensuring forward-security.

Partition:
Precondition: A group of nodes wishes to leave the group at once.
Postcondition: The leaving group is removed from the topology representative of the key agreement strategy at hand. The topology is distributed to all members, and the key-agreement layer is notified to update the key collaboratively, ensuring backward-security.

Another assumption of this model is that any topology can be represented and stored as an array or list. The tree for TGDH is no more than clever iteration through a list, and the hypercube for HGDH is little more than a clever way to index the vertices of a cube (as a Gray Code mapping vertices onto an array named after Frank Gray). Implementing these structures with an array or list backing the underlying storage allows for polymorphism in the choice of structure, as any structure that can be mapped to a list will work in this setting.

2.1.3 Linear Group Diffie-Hellman Optimizations

Each exchange in the two-party GDH algorithm requires two broadcast messages, one sent by a distinct party denoted *A* and received by another distinct party denoted *B*, and a second message sent by party *B* and received by party *A*, In the LGDH protocol, the manner in which this bi-directional communication occurs is not specified. In this implementation key propagation occurs from left-to-right exclusively beginning with nodes p_0 and p_1 and repeating until nodes p_{n-1} and nodes p_{n-2} compute the final key. When the key reaches node p_{n-1} then all public keys used in the key computation have been collected and the node at position$n-1$ sends a broadcast message with the public keys.

At this point the subkey $k_{i+1,i+2}$ for all nodes k_i is computed with the public keys from the broadcast message and this process is repeated for $k_{i+2,i+3}$ all the way to $k_{n-2,n-1}$. Forcing a directional key propagation eliminates half of the synchronous communication (backward propagation) that occurs within the LGDH protocol, improving performance within small groups dramatically. Also, the structure is a complete structure with no wasted space (every array element represents one actual device in the topology) and is therefore space-efficient. Finally, the protocol is a fully event-driven protocol, meaning is is a "push" protocol and no node needs to poll any other to determine if that node is ready to perform an exchange. When a leftmost node initiates an exchange with its right neighbor, that neighbor is guaranteed to be awaiting the request and should act immediately if that node is still alive. This means synchronization latency from parallelism is 0.

2.1.4 Tree-Based Group Diffie-Hellman

To represent the binary tree as an array, the simple rule is that the children of parent i are at index $2i+1$ and $2(i+1)$ for the left and right child respectively. A depth-first search technique is used to find and traverse nodes in the tree, since leaf-nodes are guaranteed to be mostly at the end of the tree's storage array. Backward propagation is again reduced by each right-child collecting the "blind-key" subtree of their left sibling, and sending it to their parent's right sibling if the parent is not the root. If the parent is the root, the rightmost child broadcasts the "blind-key" tree, and all nodes compute the shared key using the blind-keys on the path from themselves to the root. Since the structure is partially-parallelized , there exists a substantial amount of opportunity for synchronization latency, since right nodes that promote to their parents must poll their right sibling to ensure the tree is synchronized. Otherwise, the right sibling may not have computed the key from its leaves to its current index, and both the left and right subtrees would determine different shared keys. The protocol is also not space-efficient, since there could be as many as $2^{(n-1)}$ nodes in a tree with n leaves, each having an m-bit shared key associated with it. Neither the LGDH or HGDH protocols are this theoretically inefficient.

2.1.5 Hypercubic Group Diffie-Hellman

The hypercubic structure is a method often used to manage work-sharing and distribution among a set of processors. Since the formation of a contributory key is not much more than this, it is reasonable to apply the hypercubic notion to solve the Diffie-Hellman shared-key problem. If the nodes of a hypercube are arranged using a Gray Code mapping, then the cube maps perfectly onto an array structure. A Gray Code is a binary position mapping where no two successive elements differ in only one digit. Translated to our hypercube, in our notation two adjacent nodes in the hypercube differ in only one dimension (one bit of their underlying positional index in the storage vector). Furthermore, each node flipping the bits of their index from MSB to LSB maps the traversal of the structure in linear time.

The traversal of the hypercube structure c an be fully parallelized, such that during any given computation in the process at most half of the nodes could be waiting and fully-blocked for their current exchange partners to synchronize. In practice, HGDH could work well except for synchronization issues hampering the fully parallel nature of the hypercube. In practice one node could finish an exchange and move on to initiate with the partner in the next dimension and that partner has not finished exchanging from the previous dimension. This introduces a small amount of latency for each parallel key exchange needed to recompute the group key.

2.2 Result Testbed

2.2.1 Testbed Hardware Platform

The computing platform used for the testbed included 137 Sun SPARC-II machines running between 440 and 650 MHz, containing between 256 and 512 MB. of memory. The nodes are interconnected via a 100Mbps, fully-switched Ethernet which spans one subnet. The testbed is a public-domain computing platform, so results were taken over a 1 week period to get an average-load sample. The devices were assumed to be fault-tolerant, thus no checks are performed to ensure that the device has not experienced an error or has left the topology maliciously. Such data sets (when they did occur) were discarded. Each node shared a common-filesystem via NFS, therefore each node used a specific output file mutually-exclusive of all other nodes and stored on each machine's local filesystem to avoid synchronization and file-locking issues. When the test-runs are complete, the output files are placed into the NFS directory tree.

2.2.2 Result Collection

The results collected are in the form of transcripts for each node during each test run. The transcripts are a log of the changes in group size, and the time needed to compute the group key after the change. Each node went through different stages of joining and merging until all available nodes were joined a group and computed the group key, making each transcript a distinct data set. The transcripts were then merged into one final result set by averaging the group size and time pairs to determine average cost per node. The testbed was then repeated 100 times over the course of 1 week, and the final result sets of each test-run were averaged. This double-averaging technique was done to eliminate erroneous data points automatically by increasing the degree-of-freedom present for all data sets.

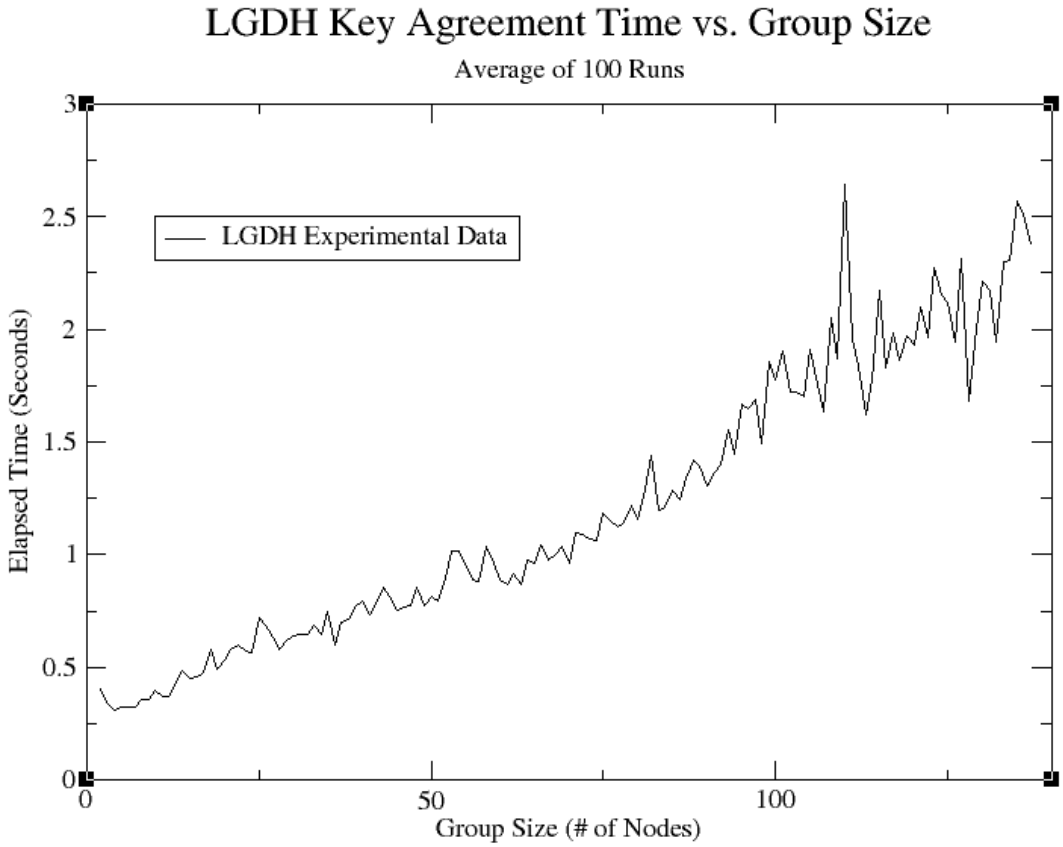
2.2.3 Result Normalization

Finally, the results were normalized via Taylor expansion about the final averaged data sets. A Taylor expansion is a tool of numerical analysis used to approximate the formula of a curve matching an arbitrary data set. The LeGendre polynomial could also have been used to determine the best representation of the data. Expansions were taken for degrees two up to eight, and the best fit expansion is chosen by visual inspection. The regression and expansion capabilities of the open-source graphing tool XMGrace was used to determine the final results and produce the graphs seen in the next section.

3 Results

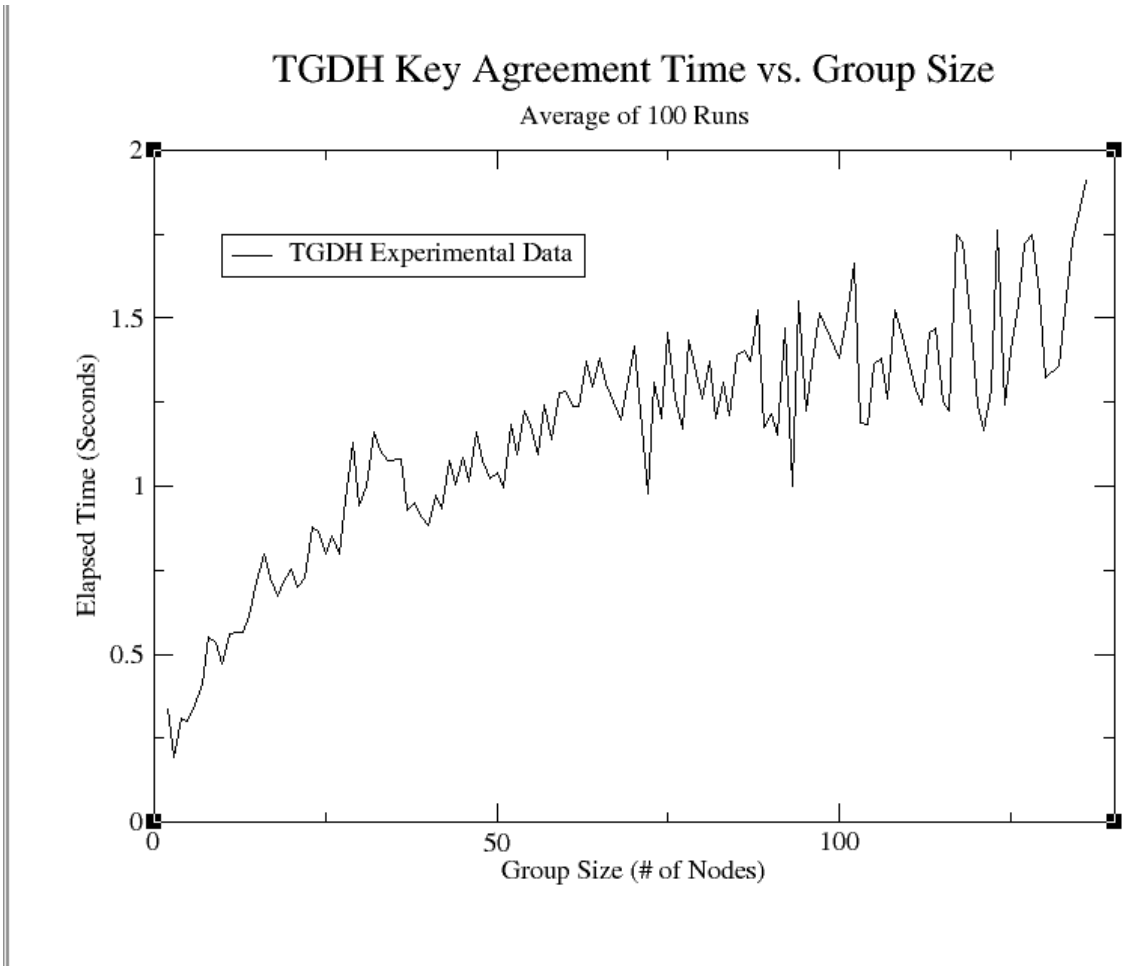
3.1 Raw Results

3.1.1 LGDH



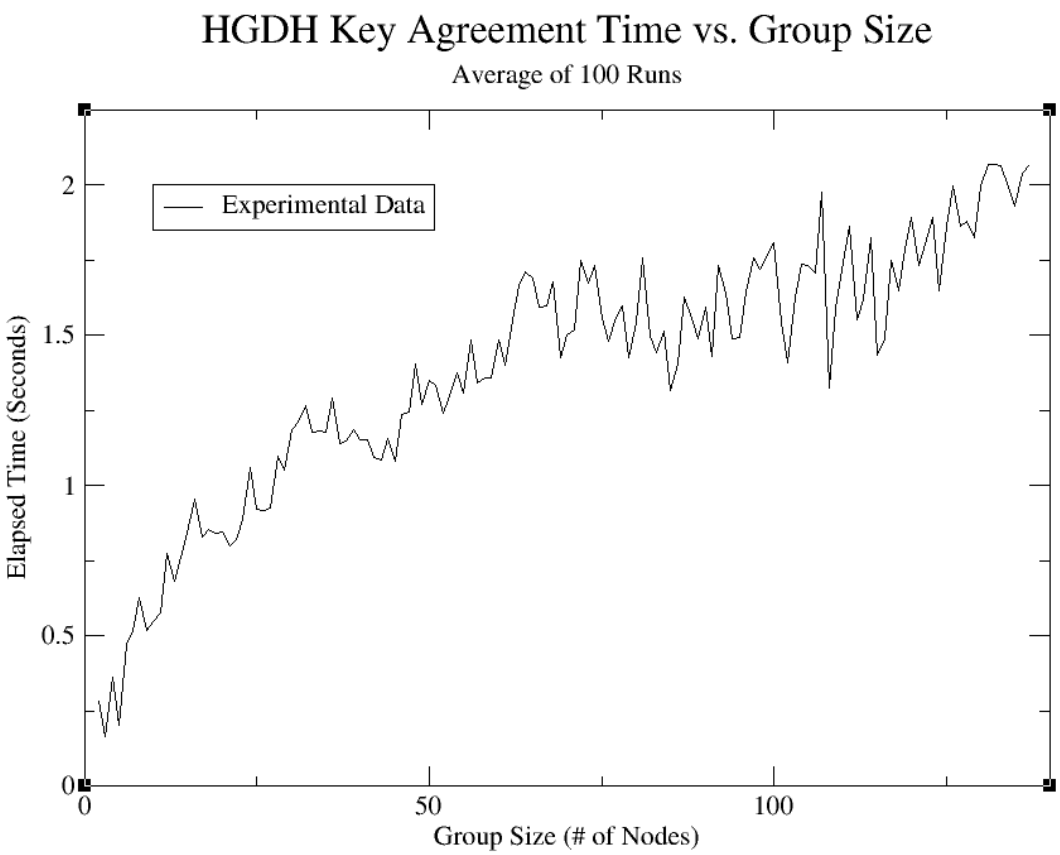
Raw Linear GDH Data

3.1.2 TGDH



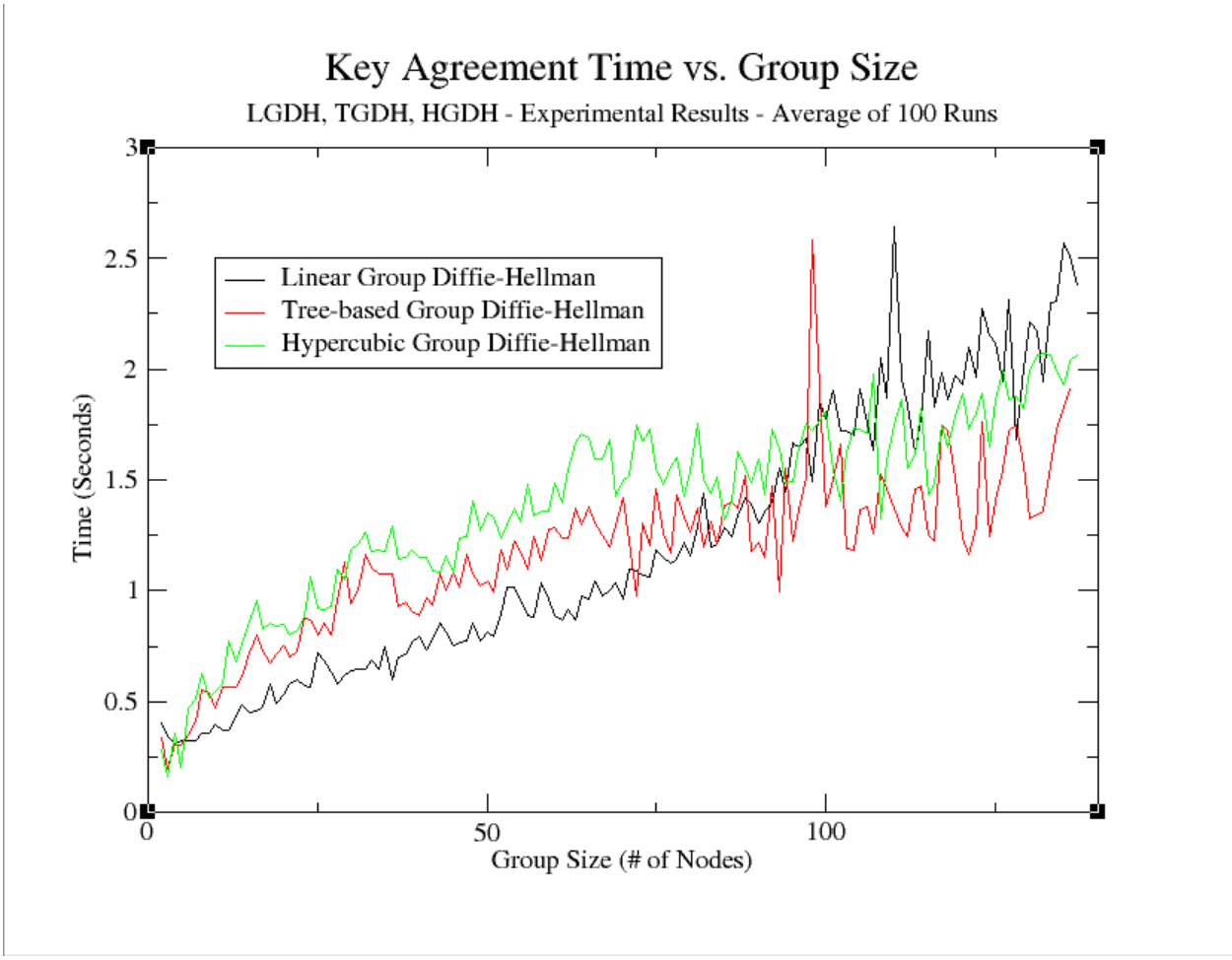
Raw Tree-Based Results

3.1.3 HGDH



Raw Hypercubic Results

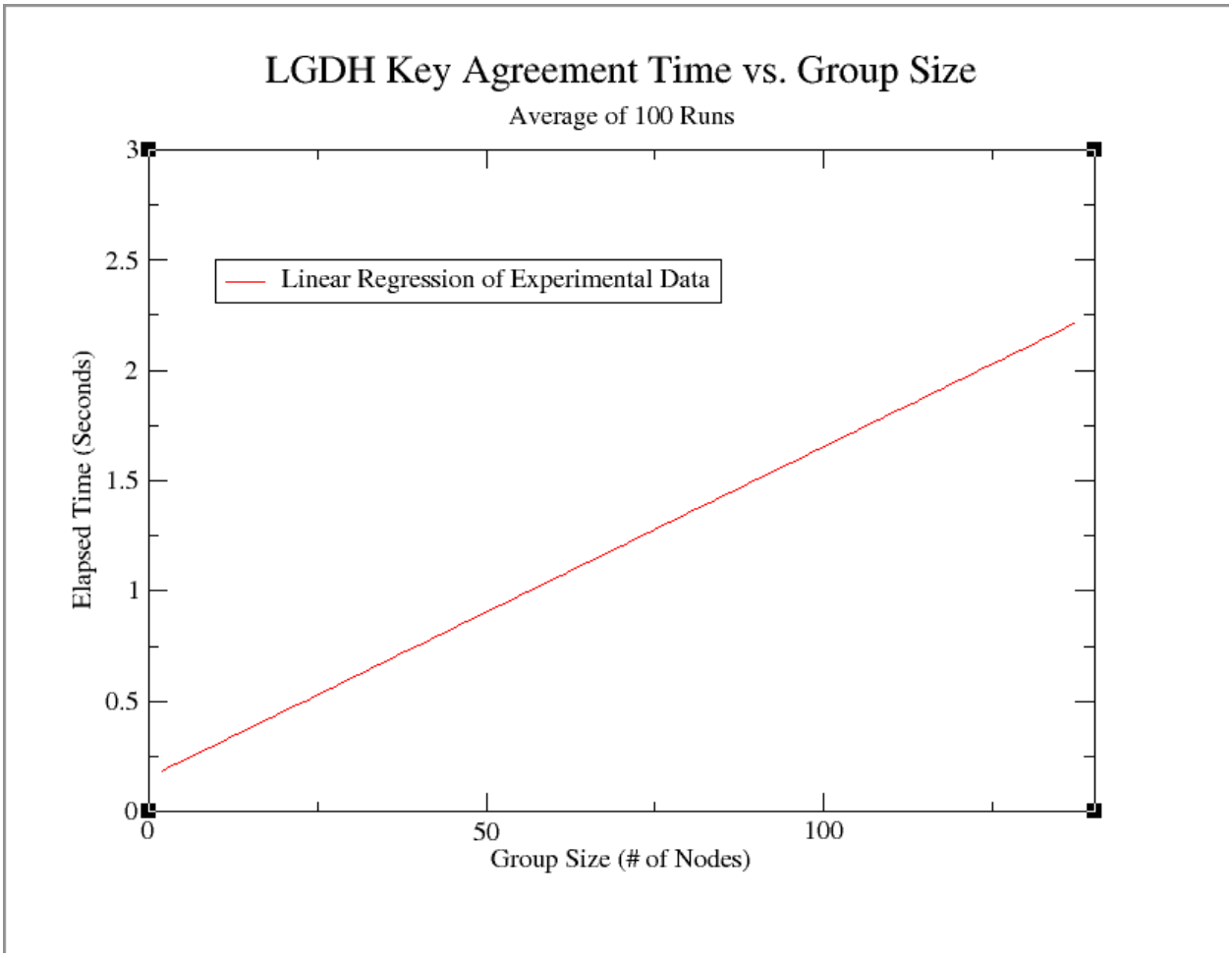
3.1.4 Raw Combined Results



Raw Combined Results

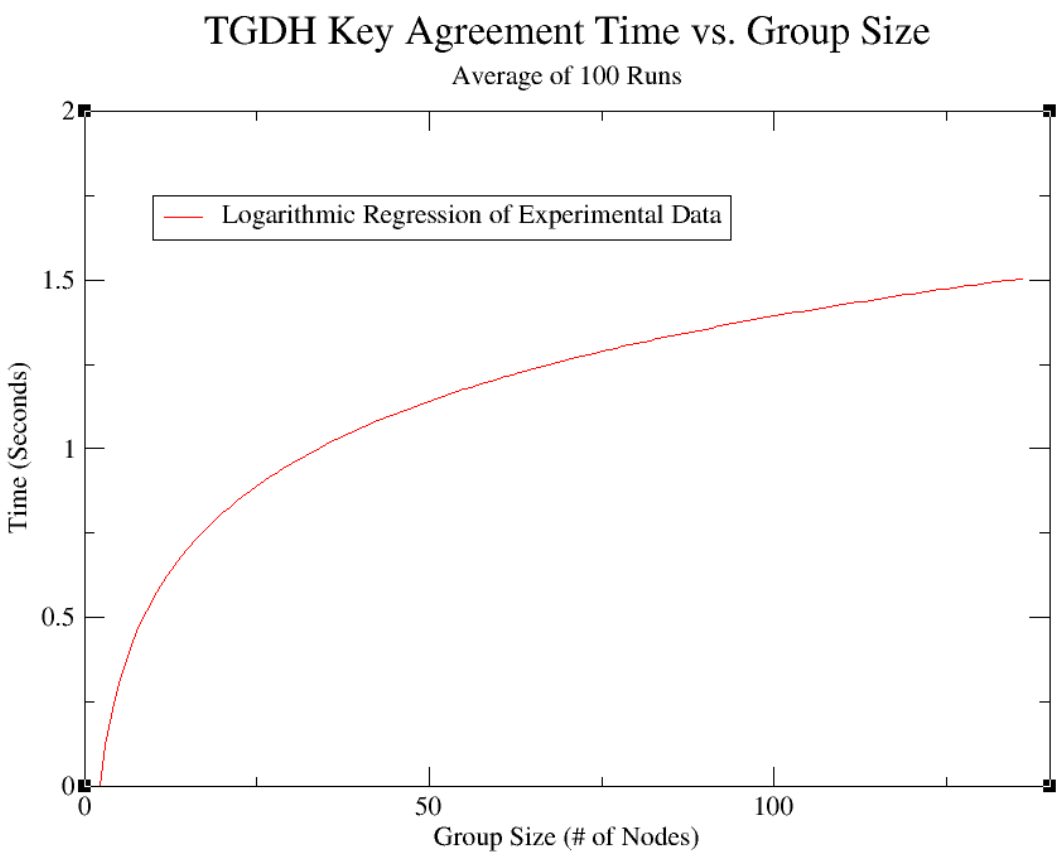
3.2 Normalized Results

3.2.1 LGDH



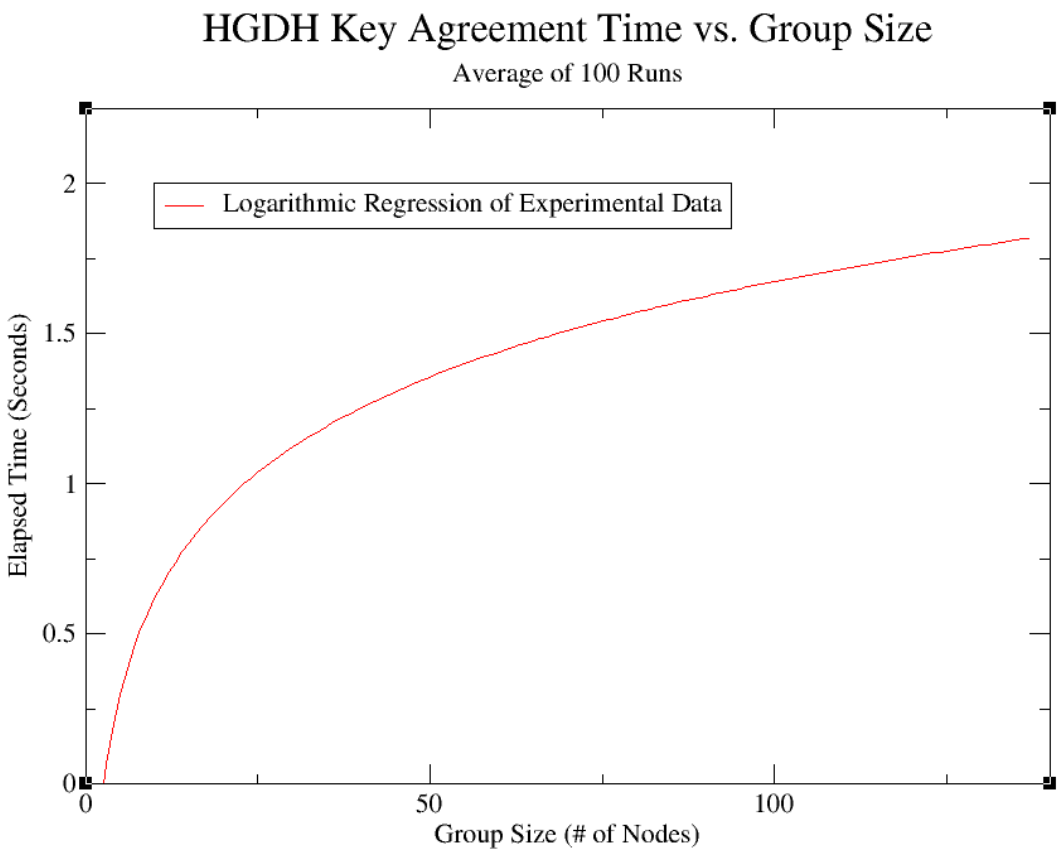
Normalized Linear GDH Results

3.2.2 TGDH



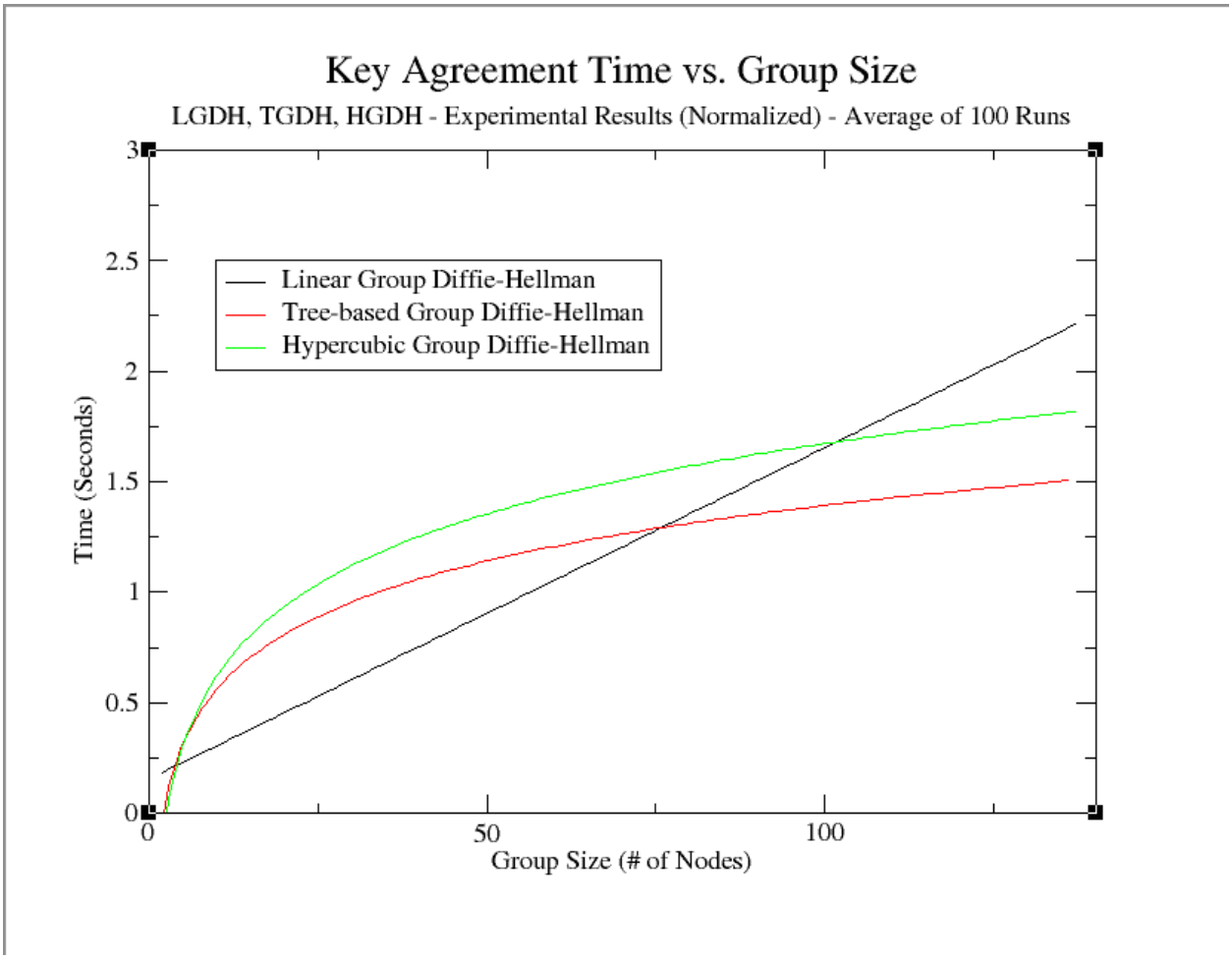
Normalized Tree-Based Results

3.2.3 HGDH



Normalized Hypercubic Results

3.2.4 Normalized Combined Results



Normalized Combined Results

4 Discussion

4.1 Variations Between Theory and Practice

When the theoretical performance of software is considered, there are many physical and environmental circumstances that are ignored in theory but in practice cause the performance of the solution to be less than optimal. In this particular case, the largest hindrance that can be considered a static quantity for each node that participated in the tests is the NFS overhead. As mentioned earlier, the use of machine-local files was incorporated to prevent this latency and the files were moved to NFS after the timing-studies were taken.

When the results are analyzed, it is apparent that LGDH performed remarkably better than expected, and HGDH performed remarkable worse than expected. The theoretical bounds on HGDH are computationally the best, however of the parallel algorithms, it performed worst. LGDH has a very poor theoretic computational bound, but performs far better than its parallel counterparts up to 75 nodes. Since each algorithm is optimized for its topology and traversal, a common I/O framework was utilized for these purposes, and the results normalized over many iterations taken during a 1 week span, these results can be considered a fair sample.

4.2 Causes of Variation

The real difference between the paradigms that causes the most latency is surely the synchronization overhead of parallelism. The non-parallel paradigm (LGDH) outperformed the parallel paradigms regardless of their ability to eliminate backward propagation up to a reasonable size closed-network. Of the parallel paradigms, the one that was "less parallel" (TGDH) outperformed the fully parallelized algorithm (HGDH), even though TGDH uses a far larger memory footprint and computation per node. Since there were no other measurable quantities of HGDH that could cause it to perform more poorly than TGDH, the conclusion left is parallel latency. This latency can also be used to explain the high-value at which the parallel algorithm benefits the situation, since at that point the computational complexity of LGDH has rendered it unusable in practice.

As mentioned, the memory footprint of TGDH and the "intermediary" nodes it utilizes may become prohibitive for huge ad-hoc networks, at which time HGDH is a feasible option. Regardless, LGDH is meant for ad-hoc networks of 100 devices or less, at which point a new solution is needed.

4.3 Improvements / Future Work

4.3.1 Key Sequencing

Currently, user-space messages are sent at a "best-effort" basis. Therefore, they may be encrypted with a security that, by the time the encryption is complete, is old and void. There are two distinct solutions, the first being to attach a key sequence

number to each key, and therefore each encrypted message, telling the nodes which key was used to encrypt each message. Security is not harmed since the sequence numbers and keys are completely unrelated, and having the sequence number would do nothing for an attacker. The second (and naive) solution would be to try all old prior keys and perform syntactic and semantic-analysis on the result, choosing the best match.

4.3.2 Failure Detection

As mentioned, this protocol (in its current state) does no form of failure detection. That is, if a node were to leave unexpectedly or maliciously, the protocol could break. It is designed to deal with lost packets, but identified nodes are expected to remain healthy for the duration of the tests. The solution here is some form of heartbeat mechanism that occurs in the same manner the protocol is traversed. When a node finds an exchange-partner non-existent, broadcast this fact to all other nodes, each check for their respective partner, and notify the "sponsor" accordingly, choosing a new sponsor if necessary. At least this way, the solution to the heartbeat scheme is guaranteed to follow the general complexity bound of the topology traversal and key-agreement paradigm.

4.3.3 Final End-User Interface

In essence, this software product is intended to mimic most of the functionality available in Java sockets proper, but with no user-accessible form of addressing. The user-space queuing is largely prototyped, but somewhat unimplemented. The queues and locking scheme needs to be synchronized for threading and son-of-data message encapsulation done to transform client-space messages into network-ready form. Also, any key-sequencing needs to be brought into the prototype. Other Diffie-Hellman variants could be added, hybrids, octopus, etc..

5 Appendices

5.1 Result Data

5.1.1 LGDH

Group Size	Key Agreement Time (Seconds)
2	0.40723045522018764
3	0.34250968456004405
4	0.3092227592267134
5	0.3263413407821234
6	0.3270269905533071
7	0.32522067957617856
8	0.3608326480263161
9	0.3613916773916764
10	0.39727102803738246
11	0.37408369408369435
12	0.3708730842911878
13	0.4356628029504741
14	0.48649070847851394
15	0.4547672514619889
16	0.45891850490196107
17	0.48103305322128764
18	0.5820204342273307
19	0.48977214377406914
20	0.5345782608695653
21	0.580240327380954
22	0.6025606060606067
23	0.5706601449275366
24	0.5660621468926555
25	0.7177409523809528
26	0.6885369230769207
27	0.6354879629629642
28	0.5807598214285723
29	0.6197308429118765
30	0.6376942857142865
31	0.6497195340501792
32	0.6472229729729739
33	0.6894715909090905
34	0.6490316742081447
35	0.7446600985221671
36	0.6031785714285723

Group Size	Key Agreement Time (Seconds)
37	0.7015202702702689
38	0.7154745916515424
39	0.7673273001508291
40	0.7943701923076912
41	0.733338282078473
42	0.786077996715928
43	0.8549651162790696
44	0.8175520527859237
45	0.753864102564103
46	0.7705660225442829
47	0.7762434292866087
48	0.8565748106060609
49	0.7716288265306105
50	0.8168199999999993
51	0.7931813725490192
52	0.8950631868131861
53	1.015793103448276
54	1.0134197530864213
55	0.9571431235431243
56	0.8914556451612897
57	0.8793170890188425
58	1.0394618226600993
59	0.9640423728813559
60	0.8899770114942528
61	0.8674650273224054
62	0.9150758661887705
63	0.867526077097505
64	0.9768168402777783
65	0.9596043076923079
66	1.040556818181819
67	0.9749534328358215
68	0.9996176470588236
69	1.0357506038647342
70	0.9650771428571445
72	1.090581275720166

LGDH (Continued)

Group Size	Key Agreement Time (Seconds)
73	1.0698862418106039
74	1.064197212837838
75	1.1822757575757565
76	1.1501273684210518
77	1.1236996047430823
78	1.1441995192307675
79	1.2176772151898716
80	1.1569484374999985
81	1.29274797786292
82	1.43955081300813
83	1.1959508032128516
84	1.2128286564625848
85	1.28389
86	1.2455799999999981
87	1.3406449553001278
88	1.42193899521531
89	1.393021535580528
90	1.30839312169312
91	1.3566505494505507
92	1.3976064311594214
93	1.5525178494623681
94	1.44853244680851
95	1.6711382775119628
96	1.6446921874999962
97	1.6861196772747646
98	1.4961938775510175
99	1.8552816627816644
100	1.773338846153844
101	1.9004873009040033
102	1.719240896358539
103	1.7228957928802566
104	1.7000725524475537
105	1.9086184873949599
106	1.7579456159822406
107	1.6360116822429922
108	2.053222993827158

Group Size	Key Agreement Time (Seconds)
109	1.8714352701325156
110	2.6396809090909104
111	1.959039745627977
112	1.8388794642857142
113	1.6187866273352995
114	1.7726303258145362
115	2.17086086956522
116	1.8271243842364537
117	1.986003418803417
118	1.8640310734463272
119	1.9709651860744293
120	1.930039999999998
121	2.0976847697756797
122	1.9662817622950828
123	2.2712800361336902
124	2.15082741935484
125	2.1175771428571446
126	1.9417248677248689
127	2.3120062992125976
128	1.684317187499999
129	2.0054819121447056
130	2.2133169230769227
131	2.1724599236641224
132	1.945685606060605
133	2.2963834586466167
134	2.3093253731343273
135	2.5658799999999986
136	2.5057242647058824
137	2.3802706019253215

5.1.2 TGDH

Group Size	Key Agreement Time (Seconds)
2	0.3393288174221315
3	0.19456511669339024
4	0.3090830970556164
5	0.302947270800212
6	0.3498852931752653
7	0.4118754221388373
8	0.5533706268221569
9	0.5361854808057775
10	0.47172351121423123
11	0.5630044733631558
12	0.5671471449487568
13	0.5662866015971627
14	0.6160350591112931
15	0.7178017241379314
16	0.7989032674118639
17	0.728303380782918
18	0.6728504716981102
19	0.7160832102412589
20	0.752962525667352
21	0.7021309699655357
22	0.7253411016949167
23	0.8786529411764705
24	0.866923406279732
25	0.8002461701003681
26	0.8531194029850748
27	0.8002145270270278
28	0.9585946097697923
29	1.128984757505773
30	0.940890318970341
31	1.0007456852791863
32	1.1614331328088852
33	1.1054333151878075
34	1.0746897983392636
36	1.0784567307692308
37	0.9263843098311826

Group Size	Key Agreement Time (Seconds)
38	0.9487675840978586
39	0.910510096575944
40	0.8856251808972505
41	0.9711081967213115
42	0.9328260283209703
43	1.0754116891457912
44	1.00420012911556
45	1.0852970085470093
46	1.0135115562403711
47	1.1616616314199384
48	1.0764867060561332
49	1.0228870858688315
50	1.0413378378378377
51	0.9943060708263064
52	1.1835985691573916
53	1.094862978723402
54	1.2227247706422038
55	1.180409793814433
56	1.0941158493248004
57	1.2431661129568097
58	1.137993991416312
59	1.278579310344825
60	1.282385534173852
61	1.237747008547008
62	1.2353914141414168
63	1.3696938356164385
64	1.296024271844659
65	1.381977595628416
66	1.3062031344183307
68	1.1964671532846756
70	1.4166080870917586
72	0.9779756944444442
73	1.308095703125001
74	1.2013392857142857
75	1.4577491694352154

TGDH (Continued)

Group Size	Key Agreement Time (Seconds)
76	1.2600213523131696
77	1.169613390928725
78	1.4338664122137454
79	1.334944055944058
80	1.2618336283185825
81	1.370316923076922
82	1.199572580645162
83	1.3108476027397262
84	1.2085167652859958
85	1.3874309133489442
86	1.402337962962966
87	1.3722058449809402
88	1.5232015065913398
89	1.1749632587859462
90	1.2161789667896663
91	1.1510899122807035
92	1.4709999999999994
93	0.9986541554959771
94	1.5524366197183086
95	1.2244631578947356
96	1.3688385416666697
97	1.514179028132992
98	2.5830134228187887
100	1.3815776397515613
101	1.5033061056105563
102	1.66192105263158
103	1.1885
104	1.1836555023923439
105	1.3628384798099769
106	1.3812065727699527
107	1.2581100917431194
108	1.5246451612903216
109	1.4608628048780483
111	1.285567567567568
112	1.243026785714284

Group Size	Key Agreement Time (Seconds)
113	1.4570110619469017
114	1.4702882096069874
115	1.251
116	1.222339055793991
117	1.7483697478991598
118	1.7242711864406781
119	1.5069201680672248
120	1.238208333333333
121	1.1644508196721313
122	1.2867950819672136
123	1.7603008130081297
124	1.2439435483870958
125	1.4038293333333332
126	1.5440312500000006
127	1.7226850393700783
128	1.7461322957198493
129	1.5690775193798443
130	1.3234086294416234
132	1.3592989949748735
134	1.7298272058823532
136	1.90771428571429

5.1.3 HGDH

Group Size	Key Agreement Time (Seconds)
2	0.28241318537859045
3	0.16426388115134605
4	0.3605236042692944
5	0.20211922639362961
6	0.46999611801242214
7	0.5158127373648852
8	0.6250615717821795
9	0.5191505167958665
10	0.5472925795052997
11	0.577521846370684
12	0.7729729080932788
13	0.6830138461538469
14	0.7628346833578799
15	0.8613180467091324
16	0.953971337579619
17	0.8301150708458563
18	0.8560633169934652
19	0.84255427631579
20	0.8476279999999989
21	0.8013449163449142
22	0.8207713358070522
23	0.8866375183194912
24	1.0612067307692297
25	0.9197626666666674
26	0.9166907484407476
27	0.926797078768912
28	1.097555194805193
29	1.0507347480106097
30	1.1847744791666666
31	1.21003444505194
32	1.2674879032258093
33	1.1785078201368517
34	1.1834582967515335
35	1.1762461904761912

Group Size	Key Agreement Time (Seconds)
36	1.2909710884353756
37	1.1407085320614752
38	1.1490126705653037
39	1.1843880341880342
40	1.1516774390243916
41	1.15390779298037
42	1.0923051378446116
43	1.0865651162790686
44	1.1550735294117627
45	1.0814035087719298
46	1.2383723702664786
47	1.2434536921151453
48	1.403363095238094
49	1.269581065759637
50	1.3498415384615383
51	1.3318553921568637
52	1.2396602564102561
53	1.2984581132075474
54	1.3753789173789168
55	1.3087207272727261
56	1.4827729591836734
57	1.3421122807017527
58	1.3585905172413784
59	1.3585819209039556
60	1.4861280701754407
61	1.399721967213113
62	1.5494442815249256
63	1.6646227824463118
64	1.710591517857143
65	1.6887222672064792
66	1.5952282828282816
67	1.59594328358209
68	1.679159663865544
69	1.42491304347826

HGDH (Continued)

Group Size	Key Agreement Time (Seconds)
70	1.5019010989010995
71	1.5193556338028154
72	1.7482881944444417
73	1.6742054794520524
74	1.7305094594594579
75	1.554417777777778
76	1.4805358851674626
77	1.5589831168831174
78	1.5991816239316232
79	1.4246592015579356
80	1.541034166666667
81	1.7560617283950601
82	1.4971940133037687
83	1.4411084337349387
84	1.5155627705627692
85	1.3172941176470585
86	1.405808139534883
87	1.627236781609195
88	1.5541803977272723
89	1.4874337078651696
90	1.594038888888886
91	1.4310500610500598
92	1.7307468944099387
93	1.6408620071684588
94	1.4879528875379933
95	1.4911431578947372
96	1.6453708333333326
97	1.7568188512518397
98	1.720155612244898
99	1.7690883838383833
100	1.806557499999999
101	1.5450544554455443
102	1.4065392156862757
103	1.6264514563106791

Group Size	Key Agreement Time (Seconds)
104	1.7363870192307698
105	1.730518095238095
106	1.7064834905660387
107	1.974740654205609
108	1.3250138888888887
109	1.59025871559633
110	1.7458681818181818
111	1.864258258258257
112	1.5508720238095224
113	1.6166039823008838
114	1.8226198830409357
115	1.4358376811594218
116	1.486968965517241
117	1.7492948717948722
118	1.6485367231638406
119	1.7885462184873897
120	1.8919041666666654
121	1.7315444214876012
122	1.8016946721311498
123	1.8900325203252033
124	1.6470935483870959
125	1.8608559999999983
126	1.9986095238095225
127	1.8609448818897614
128	1.8778339843750007
129	1.8224916201117285
130	1.9958846153846153
131	2.066882103477522
132	2.0679393939393926
133	2.062862990810359
134	1.9922926439232425
135	1.9311148148148147
136	2.0374488636363646
137	2.062498015110766

5.2 Source Code

5.2.1 v2/topo/tgdhTopology.java

```
package topo;

import utils.*;

import java.util.Vector;
import java.nio.ByteBuffer;
import java.util.StringTokenizer;

public class tgdhTopology implements Topology{

    /**
       A vector holding the tree itself.
    */
    private Vector NODES;

    /**
       A string-representation of the tree, for the printSideways method.
       DO NOT USE!!!
    */
    private String TREE_STR = "";


    public Topology newInstance(){
tgdhTopology t = new tgdhTopology();
t.NODES = new Vector();
return t;
    }

    public Topology newInstance(Vector v){
tgdhTopology t = new tgdhTopology();
t.NODES = v;
return t;
    }

    public Topology newInstance(byte[] bytes){
java.nio.ByteBuffer in = java.nio.ByteBuffer.allocate(bytes.length);
in.put(bytes);
in.position(0);
Vector v = new Vector();
while(in.position() < in.limit()){
    byte nul = in.get();
    if(nul != (byte)-1){

// Construct the node as it was destructed to accomodate
// The space saving hack....
Node n;
if(in.get() == (byte)-1){
    n = new Node();
    n.id(-1);
    int len = in.getInt();
    byte[] pk = new byte[len];
    in.get(pk);
    n.PUBLIC_KEY = new java.math.BigInteger(pk);
    v.add(n);
}
else{
    n = new Node();
    long id = in.getLong();
    n.id(id);
    int len = in.getInt();
    byte[] pk = new byte[len];
    in.get(pk);
    n.PUBLIC_KEY = new java.math.BigInteger(pk);
```

```
    v.add(n);
}

    }
    else v.add(null);
}
return newInstance(v);
    }


    /*
      ----------------------------------------------------------------------------------------------
    */

    public boolean add(Node o){
boolean retval = false;
synchronized(NODES){
    o.intermediary(false);

    // Prepare a new intermediary node, it is most likely needed...
    Node inter = new Node();
    inter.intermediary(true);

    int lowest = lowest(0);

    // If the tree array was full, we need to insert at the next available
    // array slot...
    if(lowest == Integer.MAX_VALUE)lowest = NODES.size();

    // Otherwise, if the lowest is 0, the tree is empty, so just insert...
    if(lowest == 0){
NODES.add(o);
    }

    // Else, determine if the depth of the tree is to be increased,
    // if so, insert at the root...we know this when the tree is fully
    // balanced, or the following equation holds...
    else if(// The index must be one the edge
    lowest == ((int)Math.pow(2, depth(lowest)))-1 &&
    //and greater than the current size
    lowest > NODES.size()-1
    ){
Vector temp = new Vector();
temp.add(0, inter);
addSubtree(NODES, temp, 0, 1);
addNode(o, temp, 2);
NODES = temp;
    }

    // If the depth is not to be increased, but we need to add another leaf,
    // then we need to promote and rearrange
    else{
// Create a new temporary tree...
Vector temp = (Vector)NODES.clone();

// If the point of insertion's parent is even indexed,
// Simply make this node a right-sibling of it and replace
// it's former location with an intermediate
if(parentOf(lowest)%2 == 0){
    Object parent = NODES.elementAt(parentOf(lowest));
    addNode(inter, temp, parentOf(lowest));
    addNode(parent, temp, lowest);
    addNode(o, temp, lowest+1);
}

// Otherwise we have to move the first "outermost" subtree
// to a new intermediary nodes left child, and
// insert this node at the intermediary's right sibling
```

```
else{
    if(lowest > NODES.size()-1){
// This is the point to insert at, see method trimPoint();
int tp = trimPoint(lowest);
int untrim = leftChildOf(tp);
int insertPoint = rightChildOf(tp);
killSubtree(temp, tp);
addNode(inter, temp, tp);
addSubtree(NODES, temp, tp, untrim);
addNode(o, temp, insertPoint);
    }
    else{
if(lowest < NODES.size() -1 && NODES.elementAt(lowest) == null){
    // Replace parent with new intermediary and make parent sibling of added node
    Node n = new Node();
    n.intermediary(true);
    Object parent = NODES.elementAt(parentOf(lowest));
    temp.set(parentOf(lowest), n);
    temp.set(lowest, o);
    temp.set(rightChildOf(parentOf(lowest)), parent);
}
    }
}
NODES = temp;
    }
}
return true;
    }

    /**
       Adds the specified node to the specified index in the specified vector.
       Room is made in the Vector if it is not big enough to hold the added
       element.
       @param o The object to add to this tree.
       @param v The vector holding the tree.
       @param index The index in the vector of the node to add to the tree.
    */
    private void addNode(Object o, Vector v, int index){
if(index >= v.size())v.setSize(index+1);
v.set(index, o);
    }

    /**
       Appends a subtree of the source tree to the destination tree at the given indices
       by calling itself recursively.
       @param src A vector containing the source subtree.
       @param dest A vector containing the destination subtree.
       @param srcIndex The index in the source vector of the root of the subtree to add.
       @param destIndex The index in the destination where the root of the added subtree
       should be appended.
    */
    private void addSubtree(Vector src, Vector dest, int srcIndex, int destIndex){
if(srcIndex >= src.size())return;
else{
    addNode(src.elementAt(srcIndex), dest, destIndex);
    addSubtree(src, dest, leftChildOf(srcIndex), leftChildOf(destIndex));
    addSubtree(src, dest, rightChildOf(srcIndex), rightChildOf(destIndex));
}
    }


    /*
      -----------------------------------------------------------------------------------------
    */

    public boolean merge(Topology t){
Vector v = t.getVector();
boolean retval = false;
```

```
synchronized(NODES){
    Vector temp = new Vector();
    temp = collect(v, temp, 0);
    for(int i = 0; i < temp.size(); i++){
add((Node)temp.elementAt(i));
    }
    retval=true;
}
return retval;
    }

    public Vector mergeKeyTrees(Vector left, Vector right, Vector dest){
dest.setSize(1);
dest.set(0, new Node());
addSubtree(left, dest, 0, 1);
addSubtree(right, dest, 0, 2);
return dest;
    }

    public boolean remove(Node n){
boolean retval = false;
return retval;
    }

    public boolean partition(Vector v){
boolean retval = false;
return retval;
    }

    public byte[] getBytes(){
synchronized (NODES){
    int length = 0;
    java.nio.ByteBuffer out = java.nio.ByteBuffer.allocate(EnvironmentVariables.getMTU());
    for(int i = 0; i < NODES.size(); i++){
Node n = (Node)NODES.elementAt(i);
if(n == null){
    out.put((byte)-1);
    length++;
}
else{
    out.put((byte)1);
    length++;
    out.put(n.getBytes());
    length+=(n.sizeInBytes());
}
    }
    byte[] retval = new byte[length];
    out.position(0);
    out.get(retval);
    return retval;
}
    }

    public Vector getVector(){
synchronized(NODES){
    return NODES;
}
    }

    public int size(){
int count = 0;
if(NODES != null){
    for(int i = 0; i < NODES.size(); i++){
Node n = ((Node)NODES.elementAt(i));
if (n != null && n.id() != -1l){
    count++;
}
    }
```

```
}
return count;
    }

    public Vector getGroupMembers(){
Vector nodes = new Vector();
Vector retval = new Vector();
if(NODES != null && NODES.size() > 0){
    collect(NODES, nodes, 0);
    for(int i = 0; i < nodes.size(); i++){
Node n = (Node)nodes.elementAt(i);
retval.add(new Long(n.id()));
    }
}
System.out.println("Returning "+retval.size()+" elements, size is "+size());
return retval;
    }

    /**
       Returns a copy of this tree as a human-readable String.
       @return A copy of this tree as a human-readable String.
    */
    public String toString(){
String retval = "";
retval += "Array View:\n-----------\n|";
for(int i = 0; i < NODES.size(); i++){
    Object o = NODES.elementAt(i);
    String id = ((o==null) ? "X" : ((Node)o).intermediary() ? " I " : ((Node)o).toString());
    retval += (id+"|");
}
retval += "\n\n"+"Tree View:\n----------\n"+(this.filter())+"\n";
return retval;
    }

    public Long sponsorID(){
return getSponsorID(NODES, 0);
    }

    public Long sponsorID(Vector v){
return getSponsorID(v, 0);
    }

    public Long getSponsorID(Vector v, int index){
int rightChild = rightChildOf(index);
int leftChild = leftChildOf(index);
if(rightChild < v.size() && v.elementAt(rightChild) != null){
    return getSponsorID(v, rightChild);
}
else if(leftChild < v.size() && v.elementAt(leftChild) != null){
    return getSponsorID(v, leftChild);
}
else{
    return new Long(((Node)v.elementAt(index)).id());
}
    }

    private Vector collect(Vector src, Vector dest, int index){
// Vector src = t.getVector();
if(index > src.size()-1)return dest;
if(src.elementAt(index) instanceof Node){
    Node n = (Node)src.elementAt(index);
    if(!n.intermediary()){
dest.add(n);
    }
}
dest = collect(src, dest, leftChildOf(index));
dest = collect(src, dest, rightChildOf(index));
return dest;
```

```
    }

    public Vector getSubtree(Vector dest, int index){
if(index >= NODES.size())return dest;
Node n = (Node)NODES.elementAt(index);
dest.add(n);
dest = getSubtree(dest, leftChildOf(index));
dest = getSubtree(dest, rightChildOf(index));
return dest;
    }

    /**
       Prints the tree in human-readable ascii format.
       @return A String representing a human-readable ascii-version of the current tree.
    */
    private String printTree(){
String retval = "";
retval += ("\n\nTree:\n-----\n");
/* Since printSideways(...) is recursive, we need this little global
   variablle hack...
*/
TREE_STR="";
printSideways(0,7,0);
return TREE_STR;
    }

    /**
       A recursive method to print the tree rotated 90* counter-clockwise.  One more pass
       through flip() and the tree is human-readable.  These two passes are proven through
       graphics theory to be the minimal number of passes needed to accomplish this task.
       @param node The node in the tree to begin with during traversal.
       @param space The space to use around a particular level's elements.
       @param inc The increment total of spaces to be added around a depth level's elements per
       iteration.
    */
    private void printSideways(int node, int inc, int space){
if(node < 0 || node >= NODES.size() || NODES.elementAt(node) == null){return;}
printSideways(rightChildOf(node), inc, space+inc);
for(int i= 0; i < space; i++)
    TREE_STR += "_";
Node n = (Node)NODES.elementAt(node);
if(n.intermediary())TREE_STR += "__/X\\__\n";
else{
    String s = n.toString();
    while(s.length() < 7){s = "0"+s;}
    s += "\n";
    TREE_STR+=s;
}
printSideways(leftChildOf(node), inc, space+inc);
    }


    /**
       Converts it's argument to a String[], of whom all entries
       are padded with whitespace to the same length. (I.E., make
       Java version of a a C-Style char[][] (character matrix) that
       is regular.
       @param tree A tree-representation from the print-tree method.
    */
    private String[] arrayize(String tree){
StringTokenizer toker = new StringTokenizer(tree, "\n", false);
String [] out = new String[toker.countTokens()];
int count = 0;
int maxLength = -1;
while(toker.hasMoreTokens()){
    String s = toker.nextToken();
    if(s.length() > maxLength)maxLength = s.length();
    out[count++] = s;
```

```
}

// Pad to the max length
for(int i = 0; i < out.length; i++){
    String s = out[i];
    while(s.length() < maxLength){
s +=" ";
    }
    out[i] = s;
}
return out;
    }

    /**
       Flips it's argument 90* clockwise. The argument must be an array'ized tree
       representation from arrayize() and printTree respectively, or the output
       will be goook.
       @param tree A String[] from the arrayize method.
    */
    private String flip(String[] tree){
String retval = "";
if(tree.length == 0)throw new IllegalArgumentException("Tree size must be greater than zero!");
for(int i = 0; i < tree[0].length(); i+=7){
    for(int j = tree.length-1; j >=0; j--){
String substring = tree[j].substring(i, i+7);
retval += substring;
    }
    retval += "\n";
}
return retval;
    }

    /**
       Returns a "graphically filtered" (ASCII) version of the tree (more readable).
       @return A "graphically filtered" (ASCII) version of the tree (more readable).
    */
    private String filter(){
String retval = "";
String[] a = this.arrayize(this.flip(this.arrayize(this.printTree())));
for(int i = 0; i < a.length-1; i++){
    String current = a[i];
    boolean underscore = false;
    for(int j = 0; j < current.length(); j+=7){
String chunk = current.substring(j, j+7);
String underchunk = a[i+1].substring(j, j+7);
// The following cases exhaust all possibilities of input passed to us
// This suffices for debugging purposes until a better algorithm is
// found...
if(!underscore && underchunk.equals("__/X\\__")){
    underscore = true;
    retval += "_______";
}
else if(chunk.equals("_______") && !underscore && underchunk.equals("_______")){
    retval += "       ";
}
else if(chunk.equals("_______") && underscore && underchunk.equals("_______")){
    retval += "_______";
}
else if(chunk.equals("__/X\\__")){
    underscore = true;
    retval += "__/X\\__";
}
else if (underscore && underchunk.equals("__/X\\__")){
    underscore = false;
    retval += "_______";
}
else if (chunk.equals("       ")){
    retval += "       ";
```

```
}
else if ( java.util.regex.Pattern.matches("[0-9]+", chunk) ){
    retval += chunk;
}
else if ( chunk.equals("_______") && java.util.regex.Pattern.matches("[0-9]+", underchunk)){
    retval += "_______";
}
else{
    // This should never happen here...
}
    }
    retval += "\n";
}
retval += a[a.length-1]+"\n";
return retval;
    }

    /**
       Returns the point at which a current computation involving promotion and rearrangemet
       to an odd-index valued node should trim the subtree to rearrange.
       @return The point at which a current computation involving promotion and rearrangemet
       to an odd-index valued node should trim the subtree to rearrange.
       @param index The index of the node to compute on.
    */
    private int trimPoint(int index){
int retval = index;
// Ahh a beautiful mathematical property fulfilled again....
// Keep iterating through "lineage" (parent()  method) until
// the index of that parent satisfies the following equation,
// representative of the fact that we at a node in the "rightmost"
// branch from the root.
while(retval != (int)(Math.pow(2, depth(retval)+1)-2)){
    retval = parentOf(retval);
}
return retval;
    }

    /**
       Returns the current depth of the tree.
       @return The current depth of the  tree.
    */
    private int depth(){
return depth(NODES.size());
    }

    /**
       Breadth-first leftmost search for this tree to determine optimal insertion point.
    */
    private int lowest(int index){
if(index >= NODES.size())return Integer.MAX_VALUE;
else if(NODES.elementAt(index) == null)return index;
else{
    int left = lowest(leftChildOf(index));
    int right = lowest(rightChildOf(index));
    if(right > left)return left;else return right;
}
    }

    /**
       Returns the current depth of the specified index in the vector holding the tree.
       @return The current depth of the specified index in the vector holding the tree.
       @param index The index of the node to compute on.
    */
    private int depth(int index){return (int)(Math.ceil((Math.log(index+2)/Math.log(2))-1));}

    /**
"Kills," or nullifies elements in the tree contained in the specified vector
beginning at the specified index recursively.
```

```
@param src The source vector holding the tree.
@param index The index in the vector of the root of the subtree to nullify.
    */
    private void killSubtree(Vector src, int index){
if(index >= src.size() || src.elementAt(index) == null)return;
else{
    addNode(null, src, index);
    killSubtree(src, leftChildOf(index));
    killSubtree(src, rightChildOf(index));
}
  }

    /**
       Returns the left child index of the specified index in the vector holding the tree.
       @return The left child index of the specified index in the vector holding the tree.
       @param index The index of the node to compute on.
    */
    public int leftChildOf(int index){return (2*index)+1;}


    /**
       Returns the right child index of the specified index in the vector holding the tree.
       @return The right child index of the specified index in the vector holding the tree.
       @param index The index of the node to compute on.
    */
    public int rightChildOf(int index){return 2*(index+1);}

    /**
       Returns the parent index of the specified index in the vector holding the tree.
       @return The parent index of the specified index in the vector holding the tree.
       @param index The index of the node to compute on.
    */
    public int parentOf(int index){
if(index <= 0)return -1;
return (index-1)/2;
    }

  public int indexOf(Long id){
    for(int i = NODES.size()-1; i >=0; i--){
      Node n = (Node)NODES.elementAt(i);
      if(n != null){
  if(new Long(n.id()).equals(id)) return i;
      }
    }
    System.out.println("[Tree] Meef???");
    System.exit(-1);
    return -1;
  }

}
```

5.2.2 v2/topo/lgdhTopology.java

```
package topo;

import utils.*;

import java.util.Vector;

public class lgdhTopology implements Topology{

    private Vector NODES;


    public Topology newInstance(){
lgdhTopology topo = new lgdhTopology();
```

```
topo.NODES = new Vector();
return topo;
    }

    public Topology newInstance(Vector v){
lgdhTopology topo = new lgdhTopology();
topo.NODES = v;
return topo;
    }

    public Topology newInstance(byte[] nodes){
lgdhTopology topo = new lgdhTopology();
topo.NODES = new Vector();
int count = 0;
java.nio.ByteBuffer bb = java.nio.ByteBuffer.allocate(nodes.length);
bb.put(nodes);
bb.position(0);
while (bb.position() < bb.limit()){
    int len = (int)bb.get();
    byte[] node = new byte[len];
    bb.get(node);
    Node n = new Node(node);
    topo.NODES.add(n);
}
return topo;
    }

    public boolean add(Node n){
synchronized(NODES){
    NODES.add(0, n);
    return true;
}
    }

    public boolean merge(Topology t){
synchronized(NODES){
    return NODES.addAll(t.getVector());
}
    }

    public boolean remove(Node n){
boolean retval = false;
return retval;
    }

    public boolean partition(Vector v){
boolean retval = false;
return retval;
    }

    public byte[] getBytes(){
synchronized(NODES){
    java.nio.ByteBuffer out = java.nio.ByteBuffer.allocate(EnvironmentVariables.getMTU());
    int count = 0;

    for(int i = 0; i < NODES.size(); i++){
byte[] a = ((Node)(NODES.elementAt(i))).getBytes();
out.put((byte)a.length);
out.put(a);
count += a.length+1;
    }

    byte[] retval = new byte[count];
    out.position(0);
    out.get(retval);
    return retval;
}
    }
```

```
    public Vector getVector(){
synchronized(NODES){
    return NODES;
}
    }

    public int size(){
synchronized(NODES){
    return NODES.size();
}
    }

    public Vector getGroupMembers(){
Vector retval = new Vector();
synchronized(NODES){
    for(int i = 0; i < NODES.size(); i++){
Node n = (Node)NODES.elementAt(i);
retval.add(new Long(n.id()));
    }
}
return retval;
    }

    public String toString(){
String retval = "";
synchronized(NODES){
    int a = NODES.size();
    if(a == 0){
retval = "Empty Set";
    }
    else{
for(int i = 0; i < a; i++){
    String id =  new Long(((Node)NODES.elementAt(i)).id()).toString();
    retval +="["+id+"]";
    if(i != (a-1)){
retval +=("\n");
    }
}
    }
}
return retval;
    }

    public Long sponsorID(){
return new Long(((Node)(NODES.elementAt((NODES.size()-1)))).id());
    }

    public Long sponsorID(Vector v){
return new Long(((Node)(v.elementAt((v.size()-1)))).id());
    }

    public int indexOf(Long id){
int index = -1;
for(int i = 0; i < NODES.size(); i++){
    Node n = (Node)(NODES.elementAt(i));
    if(new Long(n.id()).equals(id)){
index = i;
    }
}
return index;
    }

}
```

5.2.3 v2/topo/hgdhTopology.java

```
package topo;

import java.util.Vector;

public class hgdhTopology implements Topology{

    private Vector CUBE;

    public Topology newInstance(){
hgdhTopology h = new hgdhTopology();
h.CUBE = new Vector();
return h;
    }

    public Topology newInstance(Vector v){
hgdhTopology h = new hgdhTopology();
h.CUBE = v;
return h;
    }

    public Topology newInstance(byte[] nodes){
hgdhTopology h = new hgdhTopology();
h.CUBE = new Vector();
int count = 0;
java.nio.ByteBuffer bb = java.nio.ByteBuffer.allocate(nodes.length);
bb.put(nodes);
bb.position(0);
while (bb.position() < bb.limit()){
    int len = (char)bb.getChar();
    byte[] node = new byte[len];
    bb.get(node);
    Node n = new Node(node);
    h.CUBE.add(n);
}
return h;
    }

    public boolean add(Node n){
synchronized(CUBE){
    CUBE.add(n);
    return true;
}
    }

    public boolean merge(Topology others){
synchronized(CUBE){
    CUBE.addAll(others.getVector());
    return true;
}
    }

    public boolean remove(Node n){
boolean retval = false;
return retval;
    }

    public boolean partition(Vector v){
boolean retval = false;
return retval;
    }

    public byte[] getBytes(){
synchronized(CUBE){
    // A node must be far less than 64 bytes
    java.nio.ByteBuffer out = java.nio.ByteBuffer.allocate(40*CUBE.size());
```

```
    int count = 0;
    for(int i = 0; i < CUBE.size(); i++){
byte[] a = ((Node)(CUBE.elementAt(i))).getBytes();
out.putChar((char)a.length);
out.put(a);
count += a.length+2;
    }
    byte[] retval = new byte[count];
    out.position(0);
    out.get(retval);
    return retval;
}
    }

    public Vector getVector(){
synchronized(CUBE){
    return CUBE;
}
    }

    public int size(){
return CUBE.size();
    }

    public Vector getGroupMembers(){
Vector retval = new Vector();
synchronized(CUBE){
    for(int i = 0; i < CUBE.size(); i++){
Node n = (Node)CUBE.elementAt(i);
retval.add(new Long(n.id()));
    }
}
return retval;
    }

    public String toString(){
String retval = "";
synchronized(CUBE){
    int a = CUBE.size();
    if(a == 0){
retval = "---------\nEmpty Set\n---------";
    }
    else{
retval += "------------------------------\n";
for(int i = 0; i < a; i++){
    String id =  new Long(((Node)CUBE.elementAt(i)).id()).toString();
    retval += "[ " + pad(Integer.toString(i, 2)) + " | " + id + " ]" +"\n";
}
retval += "------------------------------\n";
    }
}
return retval;
    }

    private String pad(String s){
String retval = new String(s);
int maxlen = new java.math.BigInteger(Integer.toString(CUBE.size()-1)).bitLength();
int slen = s.length();
int its = maxlen-slen;
for(int i = 0; i < its; i++){
    retval ="0"+retval;
}
return retval;
    }

    public Long sponsorID(){
return new Long(((Node)(CUBE.elementAt((CUBE.size()-1)))).id());
    }
```

```
    public Long sponsorID(Vector v){
return new Long(((Node)(CUBE.elementAt((CUBE.size()-1)))).id());
    }

    public int indexOf(Long id){
int index = -1;
for(int i = 0; i < CUBE.size(); i++){
    Node n = (Node)(CUBE.elementAt(i));
    if(new Long(n.id()).equals(id)){
index = i;
    }
}
return index;
    }

}
```

5.2.4 v2/topo/Node.java

```
package topo;

import java.math.BigInteger;

public class Node{

  private long ID;
  private byte INTERMEDIARY;
  // Each node in the tree generates a public key through a diffie-hellman exchange.
  public BigInteger PUBLIC_KEY;

  public Node(){
    PUBLIC_KEY = new BigInteger("-1");
    ID = -1;
  }

  public Node(byte[] b){
    java.nio.ByteBuffer bb = java.nio.ByteBuffer.allocate(b.length);
    bb.put(b);
    bb.position(0);

    if(bb.get() == -1){
      ID = -1;
    }
    else{
      ID = bb.getLong();
    }

    int length = bb.getInt();
    byte[] shareval = new byte[length];
    bb.get(shareval);

    PUBLIC_KEY = new BigInteger(shareval);
  }

  public byte[] getBytes(){
    java.nio.ByteBuffer out = java.nio.ByteBuffer.allocate(sizeInBytes());

    // Hack to save space!!!
    if(ID == -1){
      out.put((byte)-1);
    }
    else{
      out.put((byte)1);
      out.putLong(ID);
    }
```

```
    byte[] a = PUBLIC_KEY.toByteArray();

    out.putInt(a.length);
    out.put(a);

    byte[] retval = new byte[sizeInBytes()];
    out.position(0);
    out.get(retval);

    return retval;
  }

  public long id(){
    return ID;
  }

  public void id(long id){
    ID = id;
  }

  public boolean intermediary(){
    return (ID == -1)?true:false;
  }

  public void intermediary(boolean value){
    INTERMEDIARY = ((value==true)?(byte)-1:(byte)1);
  }

  public String toString(){
    String s = new Long(ID).toString();
    return s.substring(((s.length()-7)>=0)?(s.length()-7):0, s.length());
  }

  // 1 byte intermediate flag
  // Possible 8 byte ID
  // 16/17 byte public key
  public int sizeInBytes(){
    int pkl = PUBLIC_KEY.toByteArray().length;
    return (ID == -1)?(pkl+5):(pkl+13);
  }

}
```

5.2.5 v2/topo/Topology.java

```
package topo;

import java.util.Vector;

public interface Topology{

    /**
       Adds a node to this <code>Topology</code>.
       @param n The node to add to this <code>Topology</code>.
       @return <code>true</code> - If the operation was a success, OR<br>
               <code>false</code> - If the operation failed.
     */
    public boolean add(Node n);

    /**
       Removes a node from this <code>Topology</code>.
       @param n The node to add to this <code>Topology</code>.
       @return <code>true</code> - If the operation was a success, OR<br>
               <code>false</code> - If the operation failed.
     */
```

```
    public boolean remove(Node n);

    /**
       Merges a given <code>Topology</code> into this <code>Topology</code>.
       @param t The topology to merge into this <code>Topology</code>.
       @return <code>true</code> - If the operation was a success, OR<br>
               <code>false</code> - If the operation failed.
     */
    public boolean merge(Topology t);

    /**
       Partitions a given <code>Vector</code> of <code>Node</code>s from this <code>Topology</code>.
       @param v A <code>Vector</code> of  <code>Node</code>s to partition from this <code>Topology</code>.
       @return <code>true</code> - If the operation was a success, OR<br>
               <code>false</code> - If the operation failed.
    */
    public boolean partition(Vector v);

    /**
       Returns the size (the number of <b>actual</b> devices) currently in this <code>Topology</code>.
       @return The number of actual devices in this <code>Topology</code>.
     */
    public int size();

    public Vector getGroupMembers();

    public byte[] getBytes();

    public Vector getVector();

    public Long sponsorID();

    public Long sponsorID(Vector v);

    public Topology newInstance();

    public Topology newInstance(Vector v);

    public Topology newInstance(byte[] bytes);

    public int indexOf(Long id);

}
```

5.2.6 v2/bin/Jumpstarter.java

```
package bin;

/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
 */

public class Jumpstarter{
  public static void main(String[] args){
boolean local;
if(args.length > 0 && args[0].indexOf("local") >= 0){
  local = true;
}
else local=false;
    try{
      java.nio.channels.DatagramChannel c = java.nio.channels.DatagramChannel.open();
      c.socket().setReuseAddress(true);
      c.socket().setBroadcast(true);
      c.socket().bind(new java.net.InetSocketAddress("0.0.0.0", 12333));
      java.nio.ByteBuffer out = java.nio.ByteBuffer.allocateDirect(18);
```

```
      out.putInt(0);
      out.putInt(0xffffffff);
      out.putChar(net.ProtocolConstants.JUMPSTART);

      out.position(0);
      java.net.InetSocketAddress outA;
      if(local){
  System.out.println("Yelling locally...");
  outA = new java.net.InetSocketAddress("localhost", 12333);
      }
      else{
    outA = new java.net.InetSocketAddress("255.255.255.255", 12333);
      }
      c.send(out, outA);
      c.close();
    }
    catch(Exception e){
      e.printStackTrace();
      System.exit(-1);
    }
  }
}
```

5.2.7 v2/utils/EnvironmentVariables.java

```
package utils;

public class EnvironmentVariables{

    /**
       Finds the debug level on the system.
    */
    public static int getDebugLevel(){
String DEBUG = System.getProperty("DEBUG");
if (DEBUG != null && java.util.regex.Pattern.matches("[0-9]+", DEBUG)){
    return Integer.parseInt(DEBUG);
}
else{
    return 0;
}
    }

    /**
       Finds the MTU variable on the system
    */
    public static int getMTU(){
String MTU = System.getProperty("MTU");
if (MTU != null && java.util.regex.Pattern.matches("[0-9]+", MTU)){
    if(getDebugLevel() > 2)
System.out.println("Setting network MTU to be "+MTU+"...");
    return Integer.parseInt(MTU);
}
else{
    return 65535;
}
    }

    /**
       Finds the PORT variable on the system
    */
    public static int getPort(){
String PORT = System.getProperty("PORT");
if (PORT != null && java.util.regex.Pattern.matches("[0-9]+", PORT)){
    if(getDebugLevel() > 2)
System.out.println("Using port "+PORT+"...");
    return Integer.parseInt(PORT);
```

```
}
else{
    return 12333;
}
    }

    /**
       Finds the INTERFACE variable on the system.
    */
    public static String getInterface(){
String INTERFACE = System.getProperty("INTERFACE");
if (INTERFACE != null && !INTERFACE.equals("")){
    if(getDebugLevel() > 2)
System.out.println("Trying interface "+INTERFACE+"...");
    return INTERFACE;
}
else{
    return null;
}
    }

    /**
       Finds the TOPOLOGY variable on the system.
    */
    public static String getTopology(){
String TOPOLOGY = System.getProperty("TOPOLOGY");
if (TOPOLOGY != null && !TOPOLOGY.equals("")){
    if(getDebugLevel() > 2)
System.out.println("\nTrying to use "+TOPOLOGY+"...");
    return TOPOLOGY;
}
else{
    return null;
}
    }

    /**
       Finds the JUMPSTART variable on the system.
    */
    public static boolean getJumpstart(){
String JUMPSTART = System.getProperty("JUMPSTART");
if (JUMPSTART != null && !JUMPSTART.equals("")){
    return true;
}
else{
    return false;
}
    }

    /**
       Finds the IPA variable on the system.
    */
    public static String getIPAddress(){
String IPA = System.getProperty("IPA");
if (IPA != null && !IPA.equals("")){
    // Check which representation we were given...
    if (java.util.regex.Pattern.matches("-{0,1}[0-9]+", IPA)){
if(getDebugLevel() > 2)
    System.out.println("Using IP address of "+IPA+"....");
return IPA;
    }
    else if (java.util.regex.Pattern.matches("((([0-9]{1,3})\\.){3})([0-9]{1,3})", IPA)){
if(getDebugLevel() > 2)
    System.out.println("Using IP address of "+IPA+"....");
return IPA;
    }
    else return null;
}
```

```
else{
    return null;
}
    }

}
```

5.2.8 v2/utils/AddressConversion.java

```
package utils;

import java.net.InetAddress;
import java.io.IOException;

public class AddressConversion{

    public static Long getBroadcastAddress(){
return new Long(0xffffffff00000000l);
    }

    public static void main(String [] args){
try{
    int ip = addrToInt(InetAddress.getByName("255.254.253.252"));
    System.out.println("255.254.253.252 -> "+ip);
    int vmid = 12345678;
    System.out.println("Using VMID of:"+vmid);
    Long id = createID(ip, vmid);
    System.out.println("Created ID:"+id);
    ip = getIP(id);
    System.out.println("Stripped out IP:"+ip);
    System.out.println("IP:"+intToAddr(ip));
    vmid = getVMID(id);
    System.out.println("Stripped out VMID:"+vmid);
}
catch(Exception e){
    e.printStackTrace();
    System.exit(-1);
}
    }

    /**
      Converts an <code>InetAddress</code> into a 32-bit integer.
      @return An integer representation of the given <code>InetAddress</code>.
      @param ia The <code>InetAddress</code> to convert.
    */
    public static int addrToInt(InetAddress ia){
if (ia == null) return -1;
java.nio.ByteBuffer buffer = java.nio.ByteBuffer.allocate(4);
buffer.put(ia.getAddress());
buffer.position(0);
return buffer.getInt();
    }

    /**
      Converts a 32-bit integer to an <code>InetAddress</code>.
      @param ina An integer representation of an <code>InetAddress</code>.
      @return The <code>InetAddress</code> representation of the integer.
    */
    public static InetAddress intToAddr(int ina) throws IOException{
String s = "";
// Don't forget the damn sign bit in java, hence the >>>
int part0 = (ina & 0xff000000) >>> 24;
int part1 = (ina & 0x00ff0000) >> 16;
int part2 = (ina & 0x0000ff00) >> 8 ;
int part3 = (ina & 0x000000ff);
s = (part0+"."+part1+"."+part2+"."+part3);
```

```
return InetAddress.getByName(s);
    }

    public static Long createID(int ip, int vmid){
java.nio.ByteBuffer bb = java.nio.ByteBuffer.allocate(8);
bb.putInt(ip);
bb.putInt(vmid);
byte[] a = new byte[8];
bb.position(0);
bb.get(a);
java.math.BigInteger bi = new java.math.BigInteger(a);
return new Long(bi.longValue());
    }

    public static int getIP(Long l){
long t = l.longValue();
t = t & 0xffffffff00000000l;
t = t >>> 32;
return (int)t;
    }

    public static int getVMID(Long l){
long t = l.longValue();
t = t & 0x00000000ffffffffl;
return (int)t;
    }
}
```

5.2.9 v2/utils/BlockingFIFOQueue.java

```
package utils;

import java.util.Vector;

public class BlockingFIFOQueue{

    private Vector ELEMENTS;

    public BlockingFIFOQueue(){
ELEMENTS = new Vector(0);
    }

    public boolean put(Object o){
synchronized(ELEMENTS){
    boolean b = ELEMENTS.add(o);
    ELEMENTS.notify();
    return b;
}
    }

    public Object take(){
synchronized(ELEMENTS){
    // If there is nothing, wait until there is something...
    if(ELEMENTS.size() == 0){
try{
    ELEMENTS.wait();
}
catch(InterruptedException e){
    e.printStackTrace();
}
    }
    // We have awoken, hence, there is something...
    return ELEMENTS.remove(0);
}
    }
```

```
}
```

5.2.10 v2/utils/FIFOQueue.java

```
package utils;

import java.util.Vector;

public class FIFOQueue{

    private Vector ELEMENTS;

    public FIFOQueue(){
ELEMENTS = new Vector(0);
    }

    public boolean put(Object o){
synchronized(ELEMENTS){
    boolean b = ELEMENTS.add(o);
    return b;
}
    }

    public Object take(){
synchronized(ELEMENTS){
    // If there is nothing, wait until there is something...
    if (ELEMENTS.size() == 0)return null;
    // We have awoken, hence, there is something...
    return ELEMENTS.remove(0);
}
    }

}
```

5.2.11 v2/ka/KeyAgreement.java

```
package ka;

import net.GroupManager;
import topo.*;
import java.math.BigInteger;
import java.util.Vector;


public interface KeyAgreement{

    public boolean updateKey(GroupManager manager, Topology topo, Long ID, BigInteger p, BigInteger g);

    public BigInteger getGroupKey();

    public BigInteger getPrivateKey();

    public Vector getConstants();

    // The source of the message, the public key to use in THIS exchange, and the key structure to pass on..
    public void diffieHellmanStage1(Long source, BigInteger puk, Vector v);

    public void diffieHellmanStage2(Long source, BigInteger puk, Vector v);

    public void diffieHellmanWait(Long source);

    public void finishKeyRefresh(Long source, Vector v);

}
```

5.2.12 v2/ka/hgdhKeyAgreement.java

```
package ka;

import utils.*;
import net.GroupManager;
import net.ProtocolConstants;

import topo.Node;
import topo.Topology;
import topo.hgdhTopology;

import java.nio.ByteBuffer;
import java.util.Random;
import java.util.Vector;
import java.math.BigInteger;

public class hgdhKeyAgreement implements KeyAgreement{

    // The keys used for encryption.....

    // The group key used for broadcast/multicast/unicast messages
    private BigInteger GROUP_KEY;

    // This particular node's original public and private keys
    private BigInteger PRIVATE_KEY;
    private BigInteger PUBLIC_KEY;

    private BigInteger CONTRIBUTION;

    private Vector KEY_VECTOR;

    private hgdhTopology TOPO;
    private BigInteger P;
    private BigInteger G;
    private Long ID;
    private GroupManager MANAGER;

    private int CURRENT_DIM;
    private int MAX_DIM;

    private int PARTNER_INDEX;
    private BigInteger MY_INDEX;

    private DiffieHellmanExchange CURRENT_EXCHANGE;

    private boolean COLLECT_PKS;
    private boolean IN_EXCHANGE;

    public boolean updateKey(GroupManager m, Topology topo, Long id, BigInteger p, BigInteger g){
if(! (topo instanceof hgdhTopology)){
    System.out.println("Topology not HGDH!!! Bailing!");
    System.exit(-1);
    return false;
}
else{
    TOPO = (hgdhTopology)topo;
    MANAGER=m;
    ID=id;
    P=p;
    G=g;
    CONTRIBUTION = new BigInteger(128, 100, new Random());
    GROUP_KEY = CONTRIBUTION;
    COLLECT_PKS = false;
    IN_EXCHANGE = false;

    // The maximum dimension is the minimum number of bits required to represent the highest
```

```
    // node index, i.e., Vector.size()-1;
    MAX_DIM = new BigInteger(new Integer(TOPO.getVector().size()-1).toString()).bitLength();
    CURRENT_DIM = MAX_DIM;

    MY_INDEX = new BigInteger(new Integer(TOPO.indexOf(ID)).toString());
    PARTNER_INDEX = MY_INDEX.flipBit(CURRENT_DIM-1).intValue();

    KEY_VECTOR = new Vector();

    if(PARTNER_INDEX >= 0 && PARTNER_INDEX < TOPO.getVector().size() && !perfectCube()){
COLLECT_PKS = true;
    }

    // I initiate this exchange if my bit is not set in the current dimension
    if(! MY_INDEX.testBit(CURRENT_DIM-1)){
initDH();
    }
    else{
// System.out.println(MY_INDEX+" awaiting stage1 from "+PARTNER_INDEX);
    }
    return true;
}
    }

    private boolean perfectCube(){
return (new BigInteger(new Integer(TOPO.getVector().size()).toString()).bitCount() == 1);
    }


    private void initDH(){
if(PARTNER_INDEX >= 0 && PARTNER_INDEX < TOPO.getVector().size()){
    try{
CURRENT_EXCHANGE = new DiffieHellmanExchange(P,G,GROUP_KEY);
byte[] apk = CURRENT_EXCHANGE.PK_ALICE.toByteArray();
ByteBuffer out = ByteBuffer.allocate(apk.length+8);
out.putChar(ProtocolConstants.DH_STAGE_1);
out.putChar((char)0);
out.putChar((char)apk.length);
out.put(apk);
out.putChar((char)0);
out.position(0);
byte[] outa = new byte[apk.length+8];
out.get(outa);
Node n = ((Node)TOPO.getVector().elementAt(PARTNER_INDEX));
MANAGER.SOCKET.rawWrite(outa, new Long(n.id()));
//System.out.println(MY_INDEX+" sent stage1 to "+PARTNER_INDEX);
// Hang around for the stage-2
    }
    catch(Exception e){
e.printStackTrace();
    }

}
else{
    // Drop down a level if possible....
    CURRENT_DIM--;
    if(CURRENT_DIM == 0){
finalizeDH();
    }
    else{
PARTNER_INDEX = MY_INDEX.flipBit(CURRENT_DIM-1).intValue();
if(! MY_INDEX.testBit(CURRENT_DIM-1)){
    //System.out.println(MY_INDEX+" sending stage1 to "+PARTNER_INDEX);
    initDH();
}
else{
    //System.out.println(MY_INDEX+" waiting on stage1 from "+PARTNER_INDEX);
}
```

```
    }
}
    }

    public void diffieHellmanStage2(Long source, BigInteger pk, Vector pks){
if(
   TOPO.indexOf(source) == PARTNER_INDEX
   ){
    // Finish off the exchange...
    CURRENT_EXCHANGE.setBobsPK(pk);
    GROUP_KEY = CURRENT_EXCHANGE.ENCRYPT_KEY;

    if(
       COLLECT_PKS &&
       CURRENT_DIM != MAX_DIM
       ){
KEY_VECTOR.add(pk);
    }

    // Proceed onward....
    CURRENT_DIM--;

    if(CURRENT_DIM == 0){
finalizeDH();
IN_EXCHANGE = false;
    }
    else{
PARTNER_INDEX = MY_INDEX.flipBit(CURRENT_DIM-1).intValue();
if(! MY_INDEX.testBit(CURRENT_DIM-1)){
    //System.out.println(MY_INDEX+" sending stage1 to "+PARTNER_INDEX);
    initDH();
}
else{
    //System.out.println(MY_INDEX+" waiting on stage1 from "+PARTNER_INDEX);
}
    }
}
else{
    //System.out.println(MY_INDEX+" REJECTING stage2 from index "+TOPO.indexOf(source)+" with ID:"+source);
}
    }

    public void diffieHellmanStage1(Long source, BigInteger pk, Vector pks){
if(
   TOPO.indexOf(source) != PARTNER_INDEX &&
   TOPO.indexOf(source) >= 0
   ){
    // Send a DH_WAIT
    //System.out.println(MY_INDEX+" forcing "+TOPO.indexOf(source)+" to wait.\nIn Exchange With:"+PARTNER_INDEX);
    ByteBuffer out = ByteBuffer.allocate(2);
    out.putChar(ProtocolConstants.DH_WAIT);
    out.position(0);
    byte[] outb = new byte[2];
    out.get(outb);
    try{
MANAGER.SOCKET.rawWrite(outb, source);
    }
    catch(Exception e){e.printStackTrace();}
    return;
}
else if(
TOPO.indexOf(source) == PARTNER_INDEX
){

    //System.out.println(MY_INDEX+" received stage1 from "+PARTNER_INDEX);

    CURRENT_EXCHANGE = new DiffieHellmanExchange(P,G,GROUP_KEY, pk);
    GROUP_KEY = CURRENT_EXCHANGE.ENCRYPT_KEY;
```

```
    if(
       COLLECT_PKS &&
       CURRENT_DIM != MAX_DIM
       ){
KEY_VECTOR.add(pk);
    }

    // Send a DH_STAGE_2
    try{
byte[] outa = CURRENT_EXCHANGE.PK_BOB.toByteArray();
ByteBuffer out = ByteBuffer.allocate(outa.length+6);
byte[] outb = new byte[outa.length+6];

out.putChar(ProtocolConstants.DH_STAGE_2);
out.putChar((char)outa.length);
out.put(outa);
out.putChar((char)0);
out.position(0);
out.get(outb);
MANAGER.SOCKET.rawWrite(outb, source);
//System.out.println(MY_INDEX+" sent stage2 to "+PARTNER_INDEX);
    }
    catch(Exception e){e.printStackTrace();}

    // Proceed onward....

    // If it is not a perfect cube, we will die here, since we
    // must have been a straggler in the outermost dimension...
    if(
       !perfectCube() &&
       MY_INDEX.testBit(MAX_DIM-1)
       ){
//System.out.println(MY_INDEX+" awaiting final vector from "+PARTNER_INDEX);
return;
    }

    CURRENT_DIM--;

    if(CURRENT_DIM == 0){
finalizeDH();
    }
    else{
PARTNER_INDEX = MY_INDEX.flipBit(CURRENT_DIM-1).intValue();
if(! MY_INDEX.testBit(CURRENT_DIM-1)){
    //System.out.println(MY_INDEX+" sending stage1 to "+PARTNER_INDEX);
    initDH();
}
else{
    //System.out.println(MY_INDEX+" waiting on stage1 from "+PARTNER_INDEX);
}
    }
}
else{
    //System.out.println(MY_INDEX+" REJECTING stage1 from index "+TOPO.indexOf(source)+" with ID:"+source);
    //System.out.println("Current Partner Is:"+PARTNER_INDEX);
    //System.out.println("Current Size Is:"+TOPO.getVector().size());
}
    }

    public void diffieHellmanWait(Long source){
if(TOPO.indexOf(source) == PARTNER_INDEX){
    try{
//System.out.println(MY_INDEX+" retrying exchange with "+PARTNER_INDEX);
Thread.currentThread().sleep(500);

byte[] outa = CURRENT_EXCHANGE.PK_ALICE.toByteArray();
ByteBuffer out = ByteBuffer.allocate(outa.length+8);
```

```
out.putChar(ProtocolConstants.DH_STAGE_1);
out.putChar((char)0);
out.putChar((char)outa.length);
out.put(outa);
out.putChar((char)0);

byte[] outb = new byte[outa.length+8];
out.position(0);
out.get(outb);
MANAGER.SOCKET.rawWrite(outb, source);
    }
    catch(Exception e){
e.printStackTrace();
    }
}
    }

    public void finishKeyRefresh(Long source, Vector v){
if(TOPO.indexOf(source) == PARTNER_INDEX){
    for(int i = 0; i < v.size(); i++){
BigInteger puk = (BigInteger)v.elementAt(i);
GROUP_KEY = puk.modPow(GROUP_KEY, P);
    }
    MANAGER.idleize();
}
    }

    private void finalizeDH(){
if(COLLECT_PKS){
    try{
BigInteger INITIAL_PARTNER = MY_INDEX.flipBit(MAX_DIM-1);
Long source = new Long(((Node)TOPO.getVector().elementAt(INITIAL_PARTNER.intValue())).id());

// count the bytes...
int count = 0;
for(int i = 0; i <  KEY_VECTOR.size(); i++){
    BigInteger bi = ((BigInteger)KEY_VECTOR.elementAt(i));
    byte[] b = bi.toByteArray();
    count += b.length+2;
}

ByteBuffer out = ByteBuffer.allocate(count+6);

out.putChar(ProtocolConstants.PK_STRUCT);
out.putChar((char)0);
out.putChar((char)KEY_VECTOR.size());
for(int i = 0; i < KEY_VECTOR.size(); i++){
    BigInteger bi = ((BigInteger)KEY_VECTOR.elementAt(i));
    byte[] b = bi.toByteArray();
    out.putChar((char)b.length);
    out.put(b);
}
byte[] outb = new byte[count+6];
out.position(0);
out.get(outb);
MANAGER.SOCKET.rawWrite(outb, source);
//System.out.println(MY_INDEX+" sent vector to "+INITIAL_PARTNER);
    }
    catch(Exception e){
e.printStackTrace();
    }
}
MANAGER.idleize();
    }

    public Vector getConstants(){
Random GENERATOR = new Random();
```

```
// Reset the public values, p, g, and our share...
BigInteger p = new BigInteger(128, 100, GENERATOR);
BigInteger g = new BigInteger(128, GENERATOR);
Vector out = new Vector();
out.add(p);
out.add(g);
return out;
    }

    public BigInteger getGroupKey(){
return GROUP_KEY;
    }

    public BigInteger getPrivateKey(){
return CONTRIBUTION;
    }

}
```

5.2.13 v2/ka/lgdhKeyAgreement.java

```
package ka;

import utils.*;
import net.GroupManager;
import net.ProtocolConstants;
import topo.Node;
import topo.Topology;
import topo.lgdhTopology;


import java.nio.ByteBuffer;
import java.math.BigInteger;
import java.util.Vector;
import java.util.Random;

public class lgdhKeyAgreement implements KeyAgreement{

    // The keys used for encryption.....

    // The group key used for broadcast/multicast/unicast message encryption
    private BigInteger GROUP_KEY;

    // This particular node's public and private keys
    // to be used for authentication
    // The personal public-key
    private BigInteger PUBLIC_KEY;
    // The private key corresponding to the above public key
    private BigInteger PRIVATE_KEY;

    // This node's personal contribution to the group key
    // and part of the public/private key pair....
    private BigInteger CONTRIBUTION;

    // Our current partner's index
    private int PARTNER_INDEX;
    private Vector KEY_VECTOR;


    private Long ID;
    private lgdhTopology TOPO;
    private BigInteger P;
    private BigInteger G;

    private GroupManager MANAGER;
```

```
    public boolean updateKey(GroupManager manager, Topology topo, Long id, BigInteger p, BigInteger g){
MANAGER=manager;
ID=id;
P=p;
G=g;
CONTRIBUTION = new BigInteger(128, 100, new Random());

boolean retval = false;

if(! (topo instanceof lgdhTopology)){
    System.out.println("Topology not LGDH!!! Bailing!");
    System.exit(-1);
    return retval;
}

else{
    retval = true;

    if(EnvironmentVariables.getDebugLevel() > 3){
System.out.println("Updating Encryption Key:");
System.out.println("Current P:"+P);
System.out.println("Current G:"+G);
System.out.println("Current Contribution:"+CONTRIBUTION);
    }

    TOPO = (lgdhTopology)topo;
    // Determine our index
    int index = TOPO.indexOf(ID);

    // Determine our current partner
    Node n;

    if(index == 0){
PARTNER_INDEX = 1;
initDHExchange(PARTNER_INDEX, CONTRIBUTION);
    }
    else{
PARTNER_INDEX = index-1;
    }
    return retval;
}
    }

    public void finishKeyRefresh(Long source, Vector v){
if(source.equals(new Long(((Node)TOPO.getVector().elementAt(TOPO.getVector().size()-1)).id()))){
    if(TOPO.indexOf(ID) == 0){
GROUP_KEY = CONTRIBUTION;
    }
    try{
for(int i = TOPO.indexOf(ID)+1; i < v.size(); i++){
    BigInteger puk = (BigInteger)v.elementAt(i);
    GROUP_KEY = puk.modPow(GROUP_KEY, P);
}
    }
    catch(Exception e){
e.printStackTrace();
    }
    MANAGER.idleize();
}
    }

    private void initDHExchange(int partnerIndex, BigInteger privateKey){

try{
    Node partner = (Node)(TOPO.getVector().elementAt(partnerIndex));

    int myIndex = TOPO.indexOf(ID);
    int groupSize = TOPO.getVector().size();
```

```
    DiffieHellmanExchange dhe;

    dhe = new DiffieHellmanExchange(P, G, privateKey);

    if(myIndex == 0 && partnerIndex == 1){
PUBLIC_KEY = dhe.PK_ALICE;
PRIVATE_KEY = dhe.ENCRYPT_KEY;

KEY_VECTOR = new Vector();
KEY_VECTOR.add(PUBLIC_KEY);
    }

    // Count the bytes...
    int count = 0;
    for(int i = 0; i < KEY_VECTOR.size(); i++){
BigInteger bi = ((BigInteger)KEY_VECTOR.elementAt(i));
byte[] b = bi.toByteArray();
count += b.length+2;
    }

    byte[] outa = dhe.PK_ALICE.toByteArray();
    ByteBuffer out = ByteBuffer.allocate(count+outa.length+8);

    out.putChar(ProtocolConstants.DH_STAGE_1);
    out.putChar((char)0);
    out.putChar((char)dhe.PK_ALICE.toByteArray().length);
    out.put(dhe.PK_ALICE.toByteArray());
    out.putChar((char)KEY_VECTOR.size());

    for(int i = 0; i < KEY_VECTOR.size(); i++){
BigInteger bi = ((BigInteger)KEY_VECTOR.elementAt(i));
byte[] b = bi.toByteArray();
out.putChar((char)b.length);
out.put(b);
    }

    byte[] outb = new byte[count+outa.length+8];
    out.position(0);
    out.get(outb);
    MANAGER.SOCKET.rawWrite(outb, new Long(partner.id()));

}
catch(Exception e){e.printStackTrace();}

    }

    public void diffieHellmanStage1(Long source, BigInteger puk, Vector v){
if(TOPO.indexOf(source) == PARTNER_INDEX){
    try{
DiffieHellmanExchange dhe = new DiffieHellmanExchange(P,G,CONTRIBUTION, puk);

PUBLIC_KEY = dhe.PK_BOB;
PRIVATE_KEY = dhe.ENCRYPT_KEY;

v.add(PUBLIC_KEY);
KEY_VECTOR = v;

GROUP_KEY = dhe.ENCRYPT_KEY;

if(TOPO.indexOf(ID) == TOPO.getVector().size()-1){

    // If we are the manager, we are done, broadcast the encryption key...

    // Count the bytes...
    int count = 0;
    for(int i = 0; i < KEY_VECTOR.size(); i++){
BigInteger bi = ((BigInteger)KEY_VECTOR.elementAt(i));
```

```
byte[] b = bi.toByteArray();
count += b.length+2;
    }

    ByteBuffer out = ByteBuffer.allocate(count+6);

    out.putChar(ProtocolConstants.PK_STRUCT);
    out.putChar((char)0);
    out.putChar((char)KEY_VECTOR.size());
    for(int i = 0; i < KEY_VECTOR.size(); i++){
BigInteger bi = ((BigInteger)KEY_VECTOR.elementAt(i));
byte[] b = bi.toByteArray();
out.putChar((char)b.length);
out.put(b);
    }

    byte[] outb = new byte[count+6];
    out.position(0);
    out.get(outb);
    MANAGER.SOCKET.rawWrite(outb, AddressConversion.getBroadcastAddress());
    MANAGER.idleize();
}
else{
    // Otherwise, keep propogating
    initDHExchange(TOPO.indexOf(ID)+1, GROUP_KEY);
}
    }
    catch(Exception e){}
}
    }

    public void diffieHellmanStage2(Long source, BigInteger puk, Vector v){
// Unused;
return;
    }

    public BigInteger getGroupKey(){
return GROUP_KEY;
    }

    public BigInteger getPrivateKey(){
return CONTRIBUTION;
    }

    public void diffieHellmanWait(Long source){
// Unused
    }

    public Vector getConstants(){
Random GENERATOR = new Random();
// Reset the public values, p, g, and our share...
BigInteger p = new BigInteger(128, 100, GENERATOR);
BigInteger g = new BigInteger(128, GENERATOR);
Vector out = new Vector();
out.add(p);
out.add(g);
return out;
    }

}
```

5.2.14 v2/ka/tgdhKeyAgreement.java

```
package ka;

import utils.*;
```

```
import net.GroupManager;
import net.ProtocolConstants;

import java.util.Vector;
import java.util.Random;
import java.nio.ByteBuffer;

import topo.Node;
import topo.Topology;
import topo.tgdhTopology;
import java.math.BigInteger;

public class tgdhKeyAgreement implements KeyAgreement{

    // The keys used for encryption.....

    // The group key used for broadcast/multicast/unicast messages
    private BigInteger GROUP_KEY;

    // This particular node's original public and private keys
    // These can be used for authentication
    private BigInteger PUBLIC_KEY;
    private BigInteger PRIVATE_KEY;

    private BigInteger CONTRIBUTION;

    private Vector KEY_VECTOR;

    private Long ID;
    private tgdhTopology TOPO;
    private BigInteger P;
    private BigInteger G;
    private GroupManager MANAGER;

    private int MY_INDEX;
    private int PARTNER_INDEX;
    private int KEY_INDEX;

    private boolean inExchange;
    private DiffieHellmanExchange CURRENT_EXCHANGE;

    public boolean updateKey(GroupManager m, Topology topo, Long id, BigInteger p, BigInteger g){
MANAGER = m;
ID=id;
P=p;
G=g;

CONTRIBUTION = new BigInteger(128, 100, new Random());
GROUP_KEY = CONTRIBUTION;

boolean retval = false;
inExchange = false;

if(! (topo instanceof tgdhTopology)){
    System.out.println("Topology not TGDH!!! Bailing!");
    System.exit(-1);
    return retval;
}
else{
    if(EnvironmentVariables.getDebugLevel() > 2){
System.out.println("Updating Encryption Key:");
    }

    TOPO = (tgdhTopology)topo;

    // First, we need to see if we are a left child, or a right child...
    MY_INDEX = TOPO.indexOf(ID);
    boolean leftChild = (MY_INDEX == TOPO.leftChildOf(TOPO.parentOf(MY_INDEX)));
```

```
    Vector temp = new Vector();
    KEY_VECTOR = new Vector();
    Node n = new Node();
    n.id(ID.longValue());

    KEY_INDEX = 0;
    KEY_VECTOR.add(n);

    if(leftChild){
TOPO.getSubtree(temp, TOPO.rightChildOf(TOPO.parentOf(MY_INDEX)));
PARTNER_INDEX = TOPO.indexOf(TOPO.sponsorID(temp));
inExchange = true;
initDH();
    }
    else{
TOPO.getSubtree(temp, TOPO.leftChildOf(TOPO.parentOf(MY_INDEX)));
PARTNER_INDEX = TOPO.indexOf(TOPO.sponsorID(temp));
inExchange = false;
    }

    retval = true;
    return retval;
}
    }

    private void initDH(){
try{
    Node partner = (Node)(TOPO.getVector().elementAt(PARTNER_INDEX));

    CURRENT_EXCHANGE = new DiffieHellmanExchange(P, G, GROUP_KEY);

    // put into this key vector, the public key of the node we represent...
    tgdhTopology keys = (tgdhTopology)TOPO.newInstance(KEY_VECTOR);
    ((Node)keys.getVector().elementAt(KEY_INDEX)).PUBLIC_KEY = CURRENT_EXCHANGE.PK_ALICE;
    KEY_VECTOR = keys.getVector();

    ByteBuffer out = ByteBuffer.allocate(EnvironmentVariables.getMTU());
    byte[] outa = CURRENT_EXCHANGE.PK_ALICE.toByteArray();

    out.putChar(ProtocolConstants.DH_STAGE_1);
    out.putChar((char)1);
    out.putChar((char)outa.length);
    out.put(outa);
    out.putChar((char)KEY_VECTOR.size());
    for(int i = 0; i < KEY_VECTOR.size(); i++){
Node n = (Node)KEY_VECTOR.elementAt(i);
if(n == null){
    out.putChar((char)0);
}
else{
    byte[] b = n.getBytes();
    out.putChar((char)b.length);
    out.put(b);
}
    }

    byte[] outb = new byte[out.position()];
    out.position(0);
    out.get(outb);
    MANAGER.SOCKET.rawWrite(outb, new Long(partner.id()));
    inExchange = false;
}
catch(Exception e){e.printStackTrace();}
    }

    public void diffieHellmanStage1(Long source, BigInteger pk, Vector pks){
if(!inExchange && TOPO.indexOf(source) != PARTNER_INDEX){
```

```
    ByteBuffer out = ByteBuffer.allocate(2);
    out.putChar(ProtocolConstants.DH_WAIT);
    out.position(0);
    byte[] outb = new byte[2];
    out.get(outb);
    try{
MANAGER.SOCKET.rawWrite(outb, source);
    }
    catch(Exception e){e.printStackTrace();}
}
else if (!inExchange && TOPO.indexOf(source) == PARTNER_INDEX){
    inExchange = true;
    DiffieHellmanExchange dhe = new DiffieHellmanExchange(P,G,GROUP_KEY, pk);
    GROUP_KEY = dhe.ENCRYPT_KEY;

    try{
// put into this key vector, the public key of the node we represent...
Node n = new Node();
n.id(((Node)TOPO.getVector().elementAt(MY_INDEX)).id());
n.PUBLIC_KEY = dhe.PK_BOB;
KEY_VECTOR.set(0, n);

// It would be nice if merge worked properly!
Vector v = new Vector();
KEY_VECTOR = TOPO.mergeKeyTrees(pks, KEY_VECTOR, v);

// Now, we must check the recursion...
if (TOPO.parentOf(MY_INDEX) == 0){
    // Broadcast the PK tree to the others....
    ByteBuffer out = ByteBuffer.allocate(EnvironmentVariables.getMTU());
    out.putChar(ProtocolConstants.PK_STRUCT);
    out.putChar((char)1);
    out.putChar((char)KEY_VECTOR.size());
    for(int i = 0; i < KEY_VECTOR.size(); i++){
Node no  = ((Node)KEY_VECTOR.elementAt(i));
if(no == null){
    out.putChar((char)0);
}
else{
    byte[] b = no.getBytes();
    out.putChar((char)b.length);
    out.put(b);
}
    }
    byte[] outb = new byte[out.position()];
    out.position(0);
    out.get(outb);
    MANAGER.SOCKET.rawWrite(outb, AddressConversion.getBroadcastAddress());
    MANAGER.idleize();
}
else{
    // Assume the role of our parent and proceed on...
    MY_INDEX = TOPO.parentOf(MY_INDEX);
    boolean leftChild = (MY_INDEX == TOPO.leftChildOf(TOPO.parentOf(MY_INDEX)));

    if(leftChild){
PARTNER_INDEX = TOPO.indexOf(TOPO.getSponsorID(TOPO.getVector(), TOPO.rightChildOf(TOPO.parentOf(MY_INDEX))));
inExchange = true;
initDH();
    }
    else{
PARTNER_INDEX = TOPO.indexOf(TOPO.getSponsorID(TOPO.getVector(), TOPO.leftChildOf(TOPO.parentOf(MY_INDEX))));
inExchange = false;
    }
}
inExchange = false;

    }
```

```
    catch(Exception e){e.printStackTrace();}
}
    }

    public void diffieHellmanStage2(Long source, BigInteger pk, Vector pks){
// Unused
    }

    public void finishKeyRefresh(Long source, Vector keys){

if(source.equals(TOPO.sponsorID())){
    tgdhTopology t = (tgdhTopology)TOPO.newInstance(keys);

    try{
while(MY_INDEX != 0){
    boolean leftChild = (MY_INDEX == TOPO.leftChildOf(TOPO.parentOf(MY_INDEX)));
    Node n;
    if(leftChild){
n = ((Node)t.getVector().elementAt(TOPO.rightChildOf(TOPO.parentOf(MY_INDEX))));
    }
    else{
n = ((Node)t.getVector().elementAt(TOPO.leftChildOf(TOPO.parentOf(MY_INDEX))));
    }

    BigInteger puk = n.PUBLIC_KEY;
    GROUP_KEY = puk.modPow(GROUP_KEY, P);
    MY_INDEX = TOPO.parentOf(MY_INDEX);
}
MANAGER.idleize();
    }
    catch(Exception e){e.printStackTrace();}
}
    }

    public Vector getConstants(){
Random GENERATOR = new Random();
// Reset the public values, p, g, and our share...
BigInteger p = new BigInteger(128, 100, GENERATOR);
BigInteger g = new BigInteger(128, GENERATOR);
Vector out = new Vector();
out.add(p);
out.add(g);
return out;
    }

    public BigInteger getGroupKey(){
return GROUP_KEY;
    }

    public BigInteger getPrivateKey(){
return CONTRIBUTION;
    }

    public void diffieHellmanWait(Long source){
if(TOPO.indexOf(source) == PARTNER_INDEX){
    try{
Thread.currentThread().sleep(500);

ByteBuffer out = ByteBuffer.allocate(EnvironmentVariables.getMTU());
out.putChar(ProtocolConstants.DH_STAGE_1);
out.putChar((char)1);
out.putChar((char)CURRENT_EXCHANGE.PK_ALICE.toByteArray().length);
out.put(CURRENT_EXCHANGE.PK_ALICE.toByteArray());

out.putChar((char)KEY_VECTOR.size());
for(int i = 0; i < KEY_VECTOR.size(); i++){
    Node n = (Node)KEY_VECTOR.elementAt(i);
    if(n == null){
```

```
out.putChar((char)0);
    }
    else{
byte[] b = n.getBytes();
out.putChar((char)b.length);
out.put(b);
    }
}
byte[] outb = new byte[out.position()];
out.position(0);
out.get(outb);
Node partner = ((Node)TOPO.getVector().elementAt(PARTNER_INDEX));
MANAGER.SOCKET.rawWrite(outb, new Long(partner.id()));
    }
    catch(Exception e){
e.printStackTrace();
    }
}
    }

}
```

5.2.15 v2/ka/DiffieHellmanExchange.java

```
package ka;

import utils.*;
import java.math.BigInteger;

public class DiffieHellmanExchange{

    private final BigInteger p;
    private final BigInteger g;
    private final BigInteger x;

    public BigInteger PK_ALICE;
    public BigInteger PK_BOB;
    public BigInteger ENCRYPT_KEY;

    public Long ALICE;
    public Long BOB;


    // Constructor for alice
    public DiffieHellmanExchange(
 BigInteger prime,
 BigInteger generator,
 BigInteger privateKey
 ){

// Generate a privateKey key....
x = privateKey;
g = generator;
p = prime;

// And public key....
PK_ALICE = generator.modPow(x,prime);
PK_BOB = null;
    }

    // Constructor for bob...
    public DiffieHellmanExchange(
 BigInteger prime,
 BigInteger generator,
 BigInteger privateKey,
 BigInteger public_key_alice
```

```
 ){

p = prime;
g = generator;
x = privateKey;

// And public key....
PK_ALICE=public_key_alice;
PK_BOB = g.modPow(x,p);
ENCRYPT_KEY = PK_ALICE.modPow(x, p);

if(EnvironmentVariables.getDebugLevel() > 3){
    System.out.println("\nInitial PrivateKey:"+x);
    System.out.println("Public Key (alice):"+PK_ALICE);
    System.out.println("Public Key (bob):"+PK_BOB);
    System.out.println("Encryption Key:"+ENCRYPT_KEY+"\n");
}
    }


    public void setBobsPK(BigInteger pk){
if(PK_BOB == null){
    PK_BOB = pk;
    ENCRYPT_KEY = PK_BOB.modPow(x, p);
    if(EnvironmentVariables.getDebugLevel() > 3){
System.out.println("\nInitial PrivateKey:"+x);
System.out.println("Public Key (alice):"+PK_ALICE);
System.out.println("Public Key (bob):"+PK_BOB);
System.out.println("Encryption Key:"+ENCRYPT_KEY+"\n");
    }
}
else if (EnvironmentVariables.getDebugLevel() > 3){
    System.out.println("Bob's Public Key Already Set!!!");
    System.out.println("Ignoring...");
}
    }
}
```

5.2.16 v2/results/Collector.java

```
import java.util.Hashtable;
import java.util.Vector;
import java.io.*;

public class Collector{

    public static void main(String[] args){
if(args.length != 1){
    System.out.println("Usage:\n\n\tjava\tCollector\t<Topology Name>");
    System.exit(-1);
}
else{
    File f = new File("./"+args[0]);
    if (! f.exists()){
System.out.println("Results for topology \""+args[0]+"\" do not seem to exist!");
System.exit(-1);
    }
    else{
Hashtable frequencies = new Hashtable();
String PREFIX="./"+args[0]+"/";

// For all test runs....
for(int testrun = 0; testrun < 500; testrun++){
    String DIRNAME = PREFIX+testrun+"/";

    // For all Machines in the run
```

```
    for (int machine = 1; machine < 200; machine++){
String FILENAME = DIRNAME+machine+".txt";
File resultfile = new File (FILENAME);
if (resultfile.exists()){
    try{
BufferedReader br =
    new BufferedReader(new InputStreamReader(new FileInputStream(resultfile)));

String line = br.readLine();
while(line != null){
    String[] vals = line.split(" ");

    Integer count = new Integer(vals[0]);
    Double time = new Double(vals[1]);

    if (frequencies.containsKey(count)){
Vector tots = (Vector)frequencies.get(count);
int num = ((Integer)tots.elementAt(0)).intValue();
double running = ((Double)tots.elementAt(1)).doubleValue();
Vector tot = new Vector();
tot.add(new Integer(num+1));
tot.add(new Double(running+time.doubleValue()));
frequencies.remove(count);
frequencies.put(count, tot);
    }
    else{
Vector tots = new Vector();
tots.add(new Integer(1));
tots.add(time);
frequencies.put(count, tots);
    }

    line = br.readLine();
}
    }
    catch(IOException e){e.printStackTrace();System.exit(-1);}
}
    }
}

for(int machines = 1; machines < 200; machines++){
    if(frequencies.containsKey(new Integer(machines))){
   Vector vals = (Vector)frequencies.get(new Integer(machines));
   int count = ((Integer)vals.elementAt(0)).intValue();
   double value = ((Double)vals.elementAt(1)).doubleValue();
   double result = (double)value / (double)count;
   System.out.println(machines+"\t"+result);
       }
}

    }
}
    }

}
```

5.2.17 v2/net/GroupSocket.java

```
package net;

import ka.*;
import topo.*;
import utils.*;

import java.nio.*;
import java.nio.channels.DatagramChannel;
```

```
import java.nio.channels.Selector;
import java.nio.channels.SelectionKey;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.io.IOException;
import java.util.Vector;
import java.math.BigInteger;

public class GroupSocket{

    private final int DEBUG;

    private final Long ID;
    private final int PORT;
    private final int MTU;

    private final Topology TOPOLOGY;
    private final KeyAgreement KA;
    private final GroupManager MANAGER;

    private final DatagramChannel CHANNEL;
    private final Selector SELECTOR;
    private final SelectionKey IOKEY;
    private boolean DONE;

    /*
      ------------------------------------------------------------------------------------------
      Constructor
      ------------------------------------------------------------------------------------------
     */

    public GroupSocket(Long id, int port, int mtu, Topology t, KeyAgreement k) throws IOException{
DEBUG=EnvironmentVariables.getDebugLevel();

ID = id;
PORT=port;
CHANNEL = DatagramChannel.open();

TOPOLOGY=t;
KA=k;
MTU=mtu;
configureIO();
SELECTOR = Selector.open();
IOKEY = CHANNEL.register(SELECTOR, SelectionKey.OP_READ);

// The following line is the glue that makes the components in this  whole shebang tie together
IOKEY.attach(new Reader(
IOKEY,
MTU,
(MANAGER=new GroupManager(
  id,
  this,
  TOPOLOGY,
  KA
  )
 ),
this
)
     );
DONE=false;
run();
    }

    /*
      ------------------------------------------------------------------------------------------
      Private Methods
      ------------------------------------------------------------------------------------------
    */
```

```
    private void run(){
// We need at least one thread under here....
// The main thread should be given back to
// user control at this point
Thread worker = new Thread(
   new Runnable(){
       public void run(){
   if(DEBUG > 2)
       try{
   System.out.println(
      "Network I/O Operational!\n[VM/IP:port] =>  ["+AddressConversion.getVMID(ID)
      +""+
      AddressConversion.intToAddr(AddressConversion.getIP(ID))+
      "]"
      );
       }
   catch(Exception e){
       e.printStackTrace();
   }
   while(!DONE){
       try{
   if(MANAGER.isIdle()){
       SELECTOR.select(500);
   }
   else{
       SELECTOR.select();
   }
       }
       catch(IOException e){
   if(DEBUG > 2){
       System.out.println("\nTrapped Ctrl+C!");
       System.out.println("Leaving the network gracefully...");
   }
   try{close();}catch(Exception f){}
   System.exit(-1);
   return;
       }
       java.util.Set readyKeys = SELECTOR.selectedKeys();
       java.util.Iterator it = readyKeys.iterator();
       int executed = 0;

       while(it.hasNext()){
   SelectionKey sk = (SelectionKey)it.next();

   if(sk != null){
       Object attach = sk.attachment();
       if(attach != null && attach instanceof Runnable){
   // Execute the reader in THIS thread
   // According to the NIO standard,
   // SelectionKeys are NOT thread safe.
   // It would be nice to have other threads
   // do this though.....
   ((Runnable)attach).run();
   executed++;
       }
   }
       }


       if(executed == 0){
   // If the manager is sitting idly....
   if(MANAGER.isIdle()){
       Vector message = MANAGER.getIdleMessage();
       try{
   if(message != null){
       Long dest = (Long)message.elementAt(0);
       ByteBuffer mess = (ByteBuffer)message.elementAt(1);
```

```
       byte[] out = new byte[mess.limit()];
       mess.get(out);
       rawWrite(out, dest);
   }
       }
       catch(Exception e){
   System.out.println("Couldn't write to "+ AddressConversion.getBroadcastAddress()+"!\n");
   System.out.println("I'm really busted...");
   e.printStackTrace();
   System.exit(-1);
       }

   }
       }

   }
       }});
worker.start();
    }


    private void configureIO() throws IOException{

// This allows for many instances of this
// class to run on the same machine...
CHANNEL.socket().setReuseAddress(true);

// This allows us to do real broadcast
CHANNEL.socket().setBroadcast(true);

// We do non-blocking I/O here...
CHANNEL.configureBlocking(false);

/* Bind to the broadcast address on the given port */
/* Yeah, we've been tricking the user all along */
/*
   We will span all the networks that this machine has
   interfaces to.....someday (1.5?) however, Sun might make
   the Socket.bind() operator take a NetworkInterface
   as a parameter, so the work is not for naught.....
*/
CHANNEL.socket().bind(new InetSocketAddress(PORT));
    }


    /*
      ----------------------------------------------------------------------------------------
      Public Methods
      ----------------------------------------------------------------------------------------
    */

    /*
      ------------------------------------------------------
      Typical write() methods in the Java standard
      ------------------------------------------------------
     */

    private void sockWrite(ByteBuffer out, InetSocketAddress outAddr) throws IOException{
String a1 = outAddr.getAddress().toString().trim();
String a2 = AddressConversion.intToAddr(AddressConversion.getIP(ID)).toString().trim();
if(a1.equals(a2)){
    if(DEBUG > 4)
System.out.println("Masking Local Host...");
    outAddr = new InetSocketAddress("255.255.255.255", PORT);
}
while(
      CHANNEL.send(
   out,
```

```
   outAddr
   )
      == 0
      )
    {}

if(DEBUG > 4)
    System.out.println("Write Successful!");
    }


    /**
       Writes a <code>byte[]</code> to a given destination.
       This method blocks until the message has been sent.
       @param b The byte array to send.
       @param destination The destination of this <code>byte[]</code> as specified by <code>utils.AddressConversi
    */
    public void write(byte[] b, Long destination) throws IOException{
ByteBuffer out = ByteBuffer.allocate(b.length+6);
out.putChar(ProtocolConstants.TOPLEVEL_MESSAGE);
out.putInt(b.length);
out.put(b);
out.position(0);
//
MANAGER.enqueue(destination, out);
    }

    /**
       Writes <i>len</i> bytes from the given <code>byte[]</code> to a given destination.
       This method blocks until the message has been sent.
       @param b The byte array to send.
       @param len The number of bytes from the given array to send.
       @param destination The destination of this <code>byte[]</code> as specified by <code>utils.AddressConversi
     */
    public void write(byte[] b, int len, Long destination) throws IOException{
if(len <= 0){
    throw new BufferUnderflowException();
}
else if(len > b.length){
    throw new BufferOverflowException();
}
ByteBuffer out = ByteBuffer.allocate(b.length+6);
out.putChar(ProtocolConstants.TOPLEVEL_MESSAGE);
out.putInt(b.length);
out.put(b, 0, len);
out.position(0);
//
MANAGER.enqueue(destination, out);
    }

    /**
       Writes <i>len</i> bytes from the given <code>byte[]</code> beginning at the specified <i>offset</i>
       to a given destination.
       This method blocks until the message has been sent.
       @param b The byte array to send.
       @param offset The offset in the given array where the copying of bytes should begin.
       @param len The number of btres to transfer starting at <i>offset</i>.
       @param destination The destination of this <code>byte[]</code> as specified by <code>utils.AddressConversi
     */
    public void write(byte[] b, int offset, int len, Long destination) throws IOException{
if(len <= 0){
    throw new BufferUnderflowException();
}
else if(len > b.length){
    throw new BufferOverflowException();
}
ByteBuffer out = ByteBuffer.allocate(len+6);
out.putChar(ProtocolConstants.TOPLEVEL_MESSAGE);
```

```
out.putInt(b.length);
out.put(b, offset, len);
out.position(0);
//
MANAGER.enqueue(destination, out);
    }

    public void rawWrite(byte[] b, Long destination) throws IOException{
InetAddress inaDEST = AddressConversion.intToAddr(AddressConversion.getIP(destination));
int vmidDEST = AddressConversion.getVMID(destination);
InetSocketAddress outAddr = new InetSocketAddress(inaDEST, PORT);
ByteBuffer out = ByteBuffer.allocate(b.length+12);
out.putInt(vmidDEST);
out.putInt(AddressConversion.getVMID(ID));
out.putInt(b.length);
out.put(b);
out.position(0);
sockWrite(out, outAddr);
    }

    public void rawWrite(byte[] b, int len, Long destination) throws IOException{
if(len <= 0){
    throw new BufferUnderflowException();
}
else if(len > b.length){
    throw new BufferOverflowException();
}
InetAddress inaDEST = AddressConversion.intToAddr(AddressConversion.getIP(destination));
int vmidDEST = AddressConversion.getVMID(destination);
InetSocketAddress outAddr = new InetSocketAddress(inaDEST, PORT);
ByteBuffer out = ByteBuffer.allocate(len+12);
out.putInt(vmidDEST);
out.putInt(AddressConversion.getVMID(ID));
out.putInt(b.length);
out.put(b, 0, len);
out.position(0);
sockWrite(out, outAddr);
    }

    public void rawWrite(byte[] b, int offset, int len, Long destination) throws IOException{
if(len <= 0){
    throw new BufferUnderflowException();
}
else if(len > b.length){
    throw new BufferOverflowException();
}
InetAddress inaDEST = AddressConversion.intToAddr(AddressConversion.getIP(destination));
int vmidDEST = AddressConversion.getVMID(destination);
InetSocketAddress outAddr = new InetSocketAddress(inaDEST, PORT);
ByteBuffer out = ByteBuffer.allocate(len+12);
out.putInt(vmidDEST);
out.putInt(AddressConversion.getVMID(ID));
out.putInt(b.length);
out.put(b, offset, len);
out.position(0);
sockWrite(out, outAddr);
    }

    /*
      -------------------------------------------------------
      Typical read() methods in the Java standard
      -------------------------------------------------------
     */

    // These will be some of the last things implemented...
    public Message read(){

Vector v = MANAGER.read();
```

```
Long sender = (Long)v.elementAt(0);
ByteBuffer message = (ByteBuffer)v.elementAt(1);

message.position(0);
byte[] contents = new byte[message.limit()];
message.get(contents);
return new Message(sender, contents);
    }
    /*
      ------------------------------------------------------
      Returns a vector of type Long containing all of the
      members currently in the group, according to the
      implementation of Topology.getGroupMembers()
      ------------------------------------------------------
     */

    // These will be some of the last things implemented...
    public Vector getGroupMembers(){
return MANAGER.getGroupMembers();
    }

    public BigInteger getGroupKey(){
return MANAGER.getGroupKey();
    }

    public BigInteger getPrivateKey(){
return MANAGER.getPrivateKey();
    }

    public void close() throws IOException{
DONE=true;
IOKEY.cancel();
CHANNEL.close();
SELECTOR.close();
    }

    /*
      ------------------------------------------------------------------------------------------
      ------------------------------------------------------------------------------------------
    */

}
```

5.2.18 v2/net/Identifier.java

```
/*
  This class property and copyright Kieran S. Hagzan 2003.
  No unauthorized duplication or exhibition.
 */

package net;

import utils.*;

import java.io.*;
import java.net.*;
import java.util.Vector;
import java.util.Enumeration;


/**
   <p>
  The purpose of this class is to identify a network node uniquely,
  both by network interface and virtual machine. The node is identified
  by a 64-bit <code>long</code> value, broken into two pieces.
```

```
  </p>
  <p>
  The first 32 (most significant) bits are the 32-bit IPv4 address
  which any node using this class for identification should bind to.
  The address is guaranteed to be checked and bindable if the constructor
  succeeds.
  </p>
  <p>
  The second 32 (least significant) bits are a uniformly distributed, 32-bit
  random value chosen by this <code>Identifier</code>. This is the value
  used to represent a particular instance of an identifier, and thus a VM.
  <b> There should only be one identifier constructed PER VIRTUAL MACHINE!</b>.
  </p>
*/
public class Identifier{

    /**
       A string representing the network card used by this <code>Identifier</code>.
     */
    private String netCardName;

    /**
       A 32-bit integer representing the IPv4 address used by this <code>Identifier</code>.
     */
    private int ip;

    /**
       A 32-bit random integer representing the Virtual Machine ID of this <code>Identifier</code>.
     */
    private int vmID;

    /**
       A 64-bit long representing the unique ID of any node using this <code>Identifier</code> properly.
     */
    private long id;

    /**
       An InetAddress to bind to.
     */
    private InetAddress addr;

    /** The internal representation of the DEBUG argument */
    private static int DEBUG_LEVEL;

    /* ------------------------------------------------------ */
    /*                   Constructors                         */
    /*                  and public methods                    */
    /* ------------------------------------------------------ */

    /**
       Tests for functionality of the <code>Identifier</code>.
     */
    public static void main(String[] args){

DEBUG_LEVEL = EnvironmentVariables.getDebugLevel();

if(DEBUG_LEVEL > 4){
    System.out.println("Checking constructors...");
}
try{

    if (DEBUG_LEVEL > 4)
System.out.println("NOARGS");
    Identifier id = new Identifier();
    if(DEBUG_LEVEL > 4)
System.out.println(id);

    if (DEBUG_LEVEL > 4)
```

```
System.out.println("\nINTERFACES");
    // There best be loopback...
    id = new Identifier("lo");
    if(DEBUG_LEVEL > 4)
System.out.println(id);

    try{
id = new Identifier("whoisthis");
System.out.println("Unexpected FAILURE!!!");
System.out.println("Bogus Interface Resolved!");
System.exit(-1);
    }
    catch(IOException e){
if (DEBUG_LEVEL > 1)
    System.out.println("Bogus Interface Test Passed....");
    }


    if (DEBUG_LEVEL > 4)
System.out.println("\nIP ADDRESSES");
    // Check the loopback
    id = new Identifier(2130706433);
    if(DEBUG_LEVEL > 4)
System.out.println(id);
    try{
// And a bogus ID
id = new Identifier(2130706431);
System.out.println("Unexpected FAILURE!!!");
System.out.println("Bogus Address Resolved!");
System.exit(-1);
    }
    catch(Exception e){
if (DEBUG_LEVEL > 1)
    System.out.println("Bogus IP Test Passed...");
    }

    if (DEBUG_LEVEL > 4)
System.out.println("\nIP's AND INTERFACES");
    // Check the loopback
    id = new Identifier("lo",2130706433);
    if(DEBUG_LEVEL > 4)
System.out.println(id);
    try{
// And a bogus ID
id = new Identifier("lo",123456789);
System.out.println("Unexpected FAILURE!!!");
System.out.println("Bogus IP/Interface Resolved!");
System.exit(-1);
    }
    catch(Exception e){
if (DEBUG_LEVEL > 1)
    System.out.println("Bogus Interface/IP Test Passed...");
    }

    System.out.println("\nAll tests passed. :-)");

}
catch (IOException e){
    if (DEBUG_LEVEL >= 1){
System.out.println("Tests failed!");
e.printStackTrace();
    }
    System.exit(-1);
}
    }

    /**
       Gets the system-wide debug level via <code>EnvironmentVariables.getDebugLevel()</code>.
```

```
     */
    private static void getDebug(){
DEBUG_LEVEL = EnvironmentVariables.getDebugLevel();
    }

    /**
       Attempts to identify this host by determining a valid network
       interface and IP address.
       An interface is valid if it exists and has a valid IPv4 address.
       (IPv6 addresses are not implemented yet.) Non-loopback interfaces
       are preferred over loopback interfaces. If more than one interface is
       found (the device is multi-homed,) the user is prompted for
       the interface to use. If only a loopback interface is
       available, this constructor falls back to use it.
     */
    public Identifier() throws IOException{
getDebug();
netCardName = null;
ip = -1;
determineID();
    }

    /**
       Attempts to identify this host using the given interface.  If the
       interface is not found, the user is informed and the VM will exit.
       @param networkInterface The network interface to bind to - typically "eth0", "hme0",
       or something like this.
    */
    public Identifier(String networkInterface) throws IOException{
getDebug();
netCardName = networkInterface;
ip = -1;
determineID();
    }

    /**
       Attempts to identify this host using the given IP address.
       If the address is not found bound to an interface on the local system,
       the user is informed and the VM will exit.
       @param ipa The ip address to bind to, in 32-bit - 2's compliment integer form.
    */
    public Identifier(int ipa) throws IOException{
getDebug();
netCardName = null;
ip = ipa;
determineID();
    }

    /**
       Attempts to identify this host using the given Inet4Address address.
       If the address is not found bound to an interface on the local system,
       the user is informed and the VM will exit.
       @param ina The 32-bit Inet4Address to bind to.
    */
    public Identifier(InetAddress ina) throws IOException{
getDebug();
ip = AddressConversion.addrToInt(ina);
netCardName = null;
determineID();
    }

    /**
       Attempts to identify this host using the given Inet4Address address
       bound on the specified interface.
       If the address is not found bound to the given interface on the local system,
       the user is informed and the VM will exit.
       @param ipa The 32-bit Inet4Address to bind to, in 2's compliment integer form.
       @param networkInterface A string representing the name of the network interface to use.
```

```
    */

    public Identifier(String networkInterface, int ipa) throws IOException{
getDebug();
netCardName = networkInterface;
ip = ipa;
determineID();
    }

    /**
       Attempts to identify this host using the given Inet4Address address.
       If the address is not found bound to an interface on the local system,
       the user is informed and the VM will exit.
       @param ipa The 32-bit Inet4Address to bind to.
       @param networkInterface A string representing the name of the network interface to use.
    */
    public Identifier(String networkInterface, InetAddress ipa) throws IOException{
getDebug();
netCardName = networkInterface;
ip = AddressConversion.addrToInt(ipa);
determineID();
    }

    /**
       Returns a <code>String</code> representation of this <code>Identifier</code>.
       @return <p><u>A String of the following form:</u><br><br>
                  <b>Network Address:</b>&lt;<i>IPv4 Address</i>&gt;<br>
                  <b>V.M. Address:</b>&lt;<i>Virtual Machine Address</i>&gt;<br>
                  <b>ID:</b>&lt;<i>[IPv4 Address | VM Address]</i>&gt;.
  </p>
     */
    public String toString(){
return ("Network Address:"+ip+"\nV.M. Address:"+vmID+"\nID:"+id+"\n");
    }

    /**
       @return the <code>InetAddress</code> of this interface.
     */
    public InetAddress getInetAddress(){
return addr;
    }

    /**
       @return the <code>int</code> represenation of this interfaces <code>InetAddress</code>.
     */
    public int getIP(){
return ip;
    }

    /**
       @return the <code>int</code> VM ID of this node.
     */
    public int getVMID(){
return vmID;
    }

    /**
       @return the <code>long</code> identifier of this node.
     */
    public long getID(){
return id;
    }

    /* ------------------------------------------------------ */
    /*                   Init. Methods                        */
    /* ------------------------------------------------------ */

    /**
```

```
       Determines the 64-bit ID for a node constructing this <code>Identifier</code>.
       First, a valid IPv4 address is obtained, and then the instance of the VM
       constructing this <code>Identifier</code> is given a random 32-bit ID.  The
       IP is shifted left 32-bits and added to the VMID to make a 64-bit <code>long</code>
       value.
     */
    private void determineID() throws IOException{
setIP();
setVMID();
// Do this with BigInteger......
java.nio.ByteBuffer bb = java.nio.ByteBuffer.allocate(8);
bb.putInt(ip);
bb.putInt(vmID);
byte[] a = new byte[8];
bb.position(0);
bb.get(a);
java.math.BigInteger bi = new java.math.BigInteger(a);
id = bi.longValue();
    }


    /**
       Attempts to determine a valid IPv4 address for this <code>Identifier</code>.
       <p>
       First, the devices of the local machine are opened and their names obtained.
       If this <code>Identifier</code> was constructed with a particular device name
       parameter, the existence of that device is verified.
       </p>
       <p>
       Secondly, the IPv4 addresses of any valid devices found are collected. If this
       <code>Identifier</code> was created with a particular device IPv4 address parameter,
       the availability of that address on a local interface (strictly the parameter if
       specified,) is verified.
     */
    private void setIP() throws IOException{
// Given the constructors, there are three possibilities

// The user was very specific in their wishes
if(netCardName != null && ip != -1){
    NetworkInterface nic = NetworkInterface.getByName(netCardName);
    if(nic != null){
// If we found the card, get the IPv4 address of it, a card cannot
// have more than one
if(hasIPv4Address(nic) && ip == AddressConversion.addrToInt(getIPv4Address(nic))){
    // The IP is set, return
    if(DEBUG_LEVEL > 4)
System.out.println("Address is Good!");
    addr = getIPv4Address(nic);
    return;
}
else throw new UnknownHostException("IP ("+AddressConversion.intToAddr(ip)+") not found on Network Interface ("+n
    }
    else throw new UnknownHostException("Network Interface ("+netCardName+") not found!");
}
// Either one or the other argument was specified,
// we have something to work with...
if(netCardName != null || ip != -1){
    if(netCardName != null){
// Check if this net card exists
NetworkInterface nic = NetworkInterface.getByName(netCardName);
if(nic != null){
    if (hasIPv4Address(nic)){
if(DEBUG_LEVEL > 4)
    System.out.println("Address is Good!");
ip = AddressConversion.addrToInt(getIPv4Address(nic));
addr = getIPv4Address(nic);
return;
    }
```

```
    else throw new UnknownHostException("IP ("+AddressConversion.intToAddr(ip)+") not found on Network Interface
}
else throw new UnknownHostException("Network Interface ("+netCardName+") not found!");
    }
    else{
// Check if this IP is on any network interfaces...
if(DEBUG_LEVEL > 4)
    System.out.println("\nChecking for validity of IPv4 address : ("+AddressConversion.intToAddr(ip)+")");

Enumeration cds = NetworkInterface.getNetworkInterfaces();
while(cds.hasMoreElements()){
    NetworkInterface card = (NetworkInterface)cds.nextElement();
    if(hasIPv4Address(card) && AddressConversion.addrToInt(getIPv4Address(card)) == ip){
// The IP is set, return
addr = getIPv4Address(card);
if(DEBUG_LEVEL > 4)
    System.out.println("Address is valid and bindable!");
return;
    }
}
throw new UnknownHostException("IP address ("+AddressConversion.intToAddr(ip)+") not found on any interface!");
    }
}
// We have nothing to work with, bummer
else{
    // Inspect the network cards and choose
    // a proper address
    if(DEBUG_LEVEL > 4)
System.out.println("\nChecking for a valid IPv4 address...");
    Enumeration nics = NetworkInterface.getNetworkInterfaces();
    Vector cards = new Vector();
    while(nics.hasMoreElements()){
NetworkInterface nic = (NetworkInterface)nics.nextElement();
if(hasIPv4Address(nic)){
    cards.add(nic);
}
    }
    chooseAddress(cards);
    if (DEBUG_LEVEL > 4)
System.out.println("Found valid and bindable address!");
    return;
}
    }

    /**
       Returns the <code>InetAddress</code> of a user-selected interface on this
       host.  This method is only called if this device is multi-homed, and an
       interface nor IP address was specified.
     */
    private InetAddress userPrompt(Vector nics) throws IOException{
System.out.println("\nHmm, seems like this device is multi-homed...");
System.out.println("I cannot span physical networks yet, that is the future...");

boolean valid = false;
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
int chosen = -1;

while(!valid){
    System.out.println("Please choose the interface to use from the list below:");
    for(int i = 0; i < nics.size(); i++){
System.out.println("("+i+") -> "+getIPv4Address((NetworkInterface)nics.elementAt(i)));
    }
    String line = "";
    System.out.print("[Interface?]:");
    line = br.readLine();
    if(java.util.regex.Pattern.matches("[0-9]+", line)){
chosen = Integer.parseInt(line);
if(chosen < nics.size() && chosen >= 0){
```

```
    valid = true;
    return getIPv4Address((NetworkInterface)nics.elementAt(chosen));
}
    }
}
// At here, we should have found a valid interface number
return null;
    }

    /**
       Selects the "best" IPv4 address to use for binding and identification
       on this host.  The "best" address is found by using first non-loopback
       interfaces, and if none are found, using the loopback interface.
     */
    private void chooseAddress(Vector nics) throws IOException{
if(nics.size() == 0){
    throw new UnknownHostException("No IPv4 Network Interfaces found (NOT EVEN LOOPBACK) !!!");
}
else{
    int nrNonLoopback = 0;
    int nrCards = nics.size();
    Vector nonLoopback = new Vector();
    for(int i = 0; i < nrCards; i++){
NetworkInterface nic = (NetworkInterface)nics.elementAt(i);
// Collect the non-loopback interfaces
if(!getIPv4Address(nic).isLoopbackAddress()){
    nonLoopback.add(nic);
    nrNonLoopback++;
}
    }
    if(nrNonLoopback == 0){
if(DEBUG_LEVEL > 4)
    System.out.println("Falling back to loopback!!");
for(int i = 0; i < nrCards; i++){
    NetworkInterface nic = (NetworkInterface)nics.elementAt(i);
    // Collect the non-loopback interfaces
    if(getIPv4Address(nic).isLoopbackAddress()){
ip = AddressConversion.addrToInt(getIPv4Address(nic));
addr = getIPv4Address(nic);
return;
    }
}
throw new UnknownHostException("No IPv4 Network Interfaces found (NOT EVEN LOOPBACK) !!!");
    }
    else if(nrNonLoopback > 1){
InetAddress chosen = userPrompt(nonLoopback);
ip = AddressConversion.addrToInt(chosen);
addr = getIPv4Address(NetworkInterface.getByInetAddress(chosen));
return;
    }
    else{
ip = AddressConversion.addrToInt(getIPv4Address((NetworkInterface)nics.elementAt(0)));
addr = getIPv4Address((NetworkInterface)nics.elementAt(0));
return;
    }
}
    }


    /**
       Sets the VM ID for this instance of an <code>Identifier</code> by randomly choosing 32-bits.
     */
    private void setVMID(){
java.math.BigInteger bi = new java.math.BigInteger(32, new java.util.Random());
vmID = bi.intValue();
    }
```

```
    /* ------------------------------------------------- */
    /*                   Helper Methods                  */
    /* ------------------------------------------------- */

    /**
      Determines if the  <code>NetworkInterface</code> specified has a valid
      IPv4 address bound to it.
      @return <code>true</code> - if there is a valid IPv4 address bound to the given interface<br>
              <code>false</code> - otherwise.
      @param ni The <code>NetworkInterface</code> to check
    */
    private boolean hasIPv4Address(NetworkInterface ni){
Enumeration addrs = ni.getInetAddresses();
while(addrs.hasMoreElements()){
    InetAddress ia = (InetAddress)addrs.nextElement();
    if (ia instanceof Inet4Address){
return true;
    }
}
return false;
    }

    /**
      Determines if the  <code>NetworkInterface</code> specified has a valid
      IPv6 address bound to it.
      @return <code>true</code> - if there is a valid IPv6 address bound to the given interface<br>
              <code>false</code> - otherwise.
      @param ni The <code>NetworkInterface</code> to check
    */
    private boolean hasIPv6Address(NetworkInterface ni){
Enumeration addrs = ni.getInetAddresses();
while(addrs.hasMoreElements()){
    InetAddress ia = (InetAddress)addrs.nextElement();
    if(ia instanceof Inet6Address){
return true;
    }
}
return false;
    }

    /**
      Returns the <code>InetAddress</code> representing the IPv4 addreess of the the  specified
      <code>NetworkInterface</code>.
      @return The valid <code>InetAddress</code> of the given <code>NetworkInterface</code> OR<br>
              <code>null</code> if one does not exist
      @param ni The <code>NetworkInterface</code> to resolve the IPv4 address of.
    */
    private InetAddress getIPv4Address(NetworkInterface ni){
Enumeration addrs = ni.getInetAddresses();
while(addrs.hasMoreElements()){
    InetAddress ia = (InetAddress)addrs.nextElement();
    if(ia instanceof Inet4Address){
return ia;
    }
}
return null;
    }

    /**
      Returns the <code>InetAddress</code> representing the IPv6 addreess of the the  specified
      <code>NetworkInterface</code>.
      @return The valid <code>InetAddress</code> of the given <code>NetworkInterface</code> OR<br>
              <code>null</code> if one does not exist
      @param ni The <code>NetworkInterface</code> to resolve the IPv6 address of.
    */
    private InetAddress getIPv6Address(NetworkInterface ni){
Enumeration addrs = ni.getInetAddresses();
while(addrs.hasMoreElements()){
```

```
    InetAddress ia = (InetAddress)addrs.nextElement();
    if(ia instanceof Inet6Address){
return ia;
    }
}
return null;
    }

}
```

5.2.19 v2/net/GroupManager.java

```
package net;


import topo.*;
import ka.*;
import utils.*;

import java.nio.ByteBuffer;
import java.util.Vector;
import java.math.BigInteger;

public class GroupManager{

    public Long ID;
    public Topology TOPOLOGY;

    private boolean JOINED;
    private boolean IDLE;
    private boolean MANAGER;

    private KeyAgreement KA_METHOD;

    private char NODE_STATUS;
    private char PROTO_STAGE;

    private Object LOCK;
    public GroupSocket SOCKET;
    private Decoder DECODER;

    private Long MANAGER_ID;

    private Long LAST_MESSAGE_SOURCE;
    private long LAST_MESSAGE_TIME;
    private ByteBuffer OUTBOUND;

    private long CURRENT_TIME;
    private long ELAPSED;

    private FIFOQueue IN_QUEUE;
    private BlockingFIFOQueue OUT_QUEUE;

    /*
      ----------------------------------------------------------------
      Constructor
      ----------------------------------------------------------------
     */

    public GroupManager(Long id, GroupSocket socket, Topology topo, KeyAgreement ka){

IN_QUEUE = new FIFOQueue();
OUT_QUEUE = new BlockingFIFOQueue();
ID = id;
SOCKET=socket;
```

```
TOPOLOGY = topo.newInstance();
Node n = new Node();
n.id(id.longValue());
TOPOLOGY.add(n);

KA_METHOD = ka;

LOCK = new Object();

NODE_STATUS = State.UNJOINED;

if(EnvironmentVariables.getJumpstart()){
    PROTO_STAGE = State.DEAD_BATTERY;
}
else{
    PROTO_STAGE = State.IDLE;
}

if(EnvironmentVariables.getDebugLevel() > 1){
    System.out.println(TOPOLOGY);
}

    }

    /*
      ----------------------------------------------------------------
      Private Methods
      ----------------------------------------------------------------
     */
    private boolean timedOut(){
long elapsed = System.currentTimeMillis()-LAST_MESSAGE_TIME;
if(elapsed > 2000 && PROTO_STAGE != State.REFRESHING_KEY)return true;
return false;
    }

    private void timestamp(){
LAST_MESSAGE_TIME = System.currentTimeMillis();
    }

    private void allocate(int size){
OUTBOUND = ByteBuffer.allocate(size);
OUTBOUND.position(0);
    }

    private void writeMessage(Long destination){
OUTBOUND.position(0);
byte[] out = new byte[OUTBOUND.limit()];
OUTBOUND.get(out);
try{
    SOCKET.rawWrite(out, destination);
}
catch(Exception e){
    System.out.println("Write error on "+SOCKET+" to "+destination+"...exiting...");
    System.out.println(e);
    System.exit(-1);
}
    }

    private void printDebugStats(String message, Long ID){
if(EnvironmentVariables.getDebugLevel() > 3){
    System.out.println("N.S.: 0x"+Integer.toString(NODE_STATUS, 16));
    System.out.println("P.S.: 0x"+Integer.toString(PROTO_STAGE, 16));
    System.out.println(message+" :"+ID);
}
    }

    /*
      ----------------------------------------------------------------
```

```
      Public Methods
      ---------------------------------------------------------------
     */

    /*
      Methods used to evaluate the nature of the group...
     */

    public boolean isIdle(){
synchronized(LOCK){
    boolean retval = false;
    if(PROTO_STAGE == State.IDLE)
retval = true;
    return retval;
}
    }

    public Vector getIdleMessage(){

Vector retval = null;
ByteBuffer temp;

final char currentStatus = NODE_STATUS;
switch (currentStatus){
case State.UNJOINED:
    if(EnvironmentVariables.getJumpstart()){
retval = null;
    }
    else{
temp = ByteBuffer.allocate(2);
temp.putChar(ProtocolConstants.JOIN_GROUP_REQUEST);
temp.position(0);
retval = new Vector();
retval.add(AddressConversion.getBroadcastAddress());
retval.add(temp);
    }
    return retval;

case State.ACTING_MANAGER:
    // Dequeue something if it is there...

    Object um = null;
    synchronized(IN_QUEUE){
um = IN_QUEUE.take();
    }
    if(um != null){
retval = (Vector)um;
    }
    else{
temp = ByteBuffer.allocate(2);
temp.putChar(ProtocolConstants.MERGE_GROUP_REQUEST);
temp.position(0);
retval = new Vector();
retval.add(AddressConversion.getBroadcastAddress());
retval.add(temp);
    }
    return retval;

case State.JOINED:
    // Dequeue something if it is there...
    um = null;
    synchronized(IN_QUEUE){
um = IN_QUEUE.take();
    }
    if(um != null){
retval = (Vector)um;
    }
    return retval;
```

```
}

return retval;
    }

    public void enqueue(Long destination, ByteBuffer contents){
synchronized(IN_QUEUE){
    Vector stuffs = new Vector();
    stuffs.add(destination);
    stuffs.add(contents);
    IN_QUEUE.put(stuffs);
}
    }

    public void toplevelMessage(Long source, ByteBuffer message){
Vector v = new Vector();
v.add(source);
v.add(message);
OUT_QUEUE.put(v);
    }

    public Vector read(){
return (Vector)OUT_QUEUE.take();
    }

    public Vector getGroupMembers(){
return TOPOLOGY.getGroupMembers();
    }

    public BigInteger getPrivateKey(){
return KA_METHOD.getPrivateKey();
    }

    public BigInteger getGroupKey(){
return KA_METHOD.getGroupKey();
    }

    public void jumpstart(){
synchronized(LOCK){
    if(
       (NODE_STATUS == State.UNJOINED) &&
       (PROTO_STAGE == State.DEAD_BATTERY) &&
       EnvironmentVariables.getJumpstart()
       ){

if(EnvironmentVariables.getDebugLevel() > 3)
    System.out.println("["+ID+"] Jumpstarting!");

System.setProperty("JUMPSTART", "");
PROTO_STAGE = State.IDLE;
    }
}
    }

    /*
      Methods used to decode incoming messages
     */

    public void reject(Long source){
synchronized(LOCK){
    if(
       (
(NODE_STATUS == State.UNJOINED && PROTO_STAGE == State.JOIN_CLIENT) ||
(NODE_STATUS == State.UNJOINED && PROTO_STAGE == State.JOIN_SERVER) ||
(NODE_STATUS == State.ACTING_MANAGER && PROTO_STAGE == State.MERGE_CLIENT)
) &&
       LAST_MESSAGE_SOURCE.equals(source)
```

```
       ){
printDebugStats("["+ID+"] Reject From: ", source);
PROTO_STAGE = State.IDLE;
timestamp();
return;
    }
    else if(timedOut()){
PROTO_STAGE = State.IDLE;
    }
}
    }

    public void joinGroupRequest(Long source){
synchronized(LOCK){
    if(
       (NODE_STATUS == State.UNJOINED || NODE_STATUS == State.ACTING_MANAGER) &&
       (PROTO_STAGE == State.IDLE)
       ){

printDebugStats("["+ID+"] Join Request From: ", source);

// Send join response..
LAST_MESSAGE_SOURCE = source;
allocate(2);
OUTBOUND.putChar(ProtocolConstants.JOIN_GROUP_RESPONSE);
writeMessage(source);
PROTO_STAGE = State.JOIN_SERVER;

timestamp();

    }
    else if(timedOut()){
PROTO_STAGE = State.IDLE;
if(NODE_STATUS == State.UNJOINED ||  NODE_STATUS == State.ACTING_MANAGER){
    joinGroupRequest(source);
}
    }
}
    }

    public void joinGroupResponse(Long source){
synchronized(LOCK){
    // Send join confirm..
    if(
       (NODE_STATUS == State.UNJOINED && PROTO_STAGE == State.IDLE)
       ){

printDebugStats("["+ID+"] Join Response From: ", source);

allocate(2);
OUTBOUND.putChar(ProtocolConstants.JOIN_GROUP_CONFIRM);
writeMessage(source);

PROTO_STAGE = State.JOIN_CLIENT;

MANAGER_ID = source;
LAST_MESSAGE_SOURCE = source;
timestamp();
return;
    }
    else if(timedOut()){
PROTO_STAGE = State.IDLE;
timestamp();
if(NODE_STATUS == State.UNJOINED ||  NODE_STATUS == State.ACTING_MANAGER){
    joinGroupResponse(source);
}
    }
    else{
```

```
printDebugStats("["+ID+"] REJECTING-> ", source);
allocate(2);
OUTBOUND.putChar(ProtocolConstants.REJECT);
writeMessage(source);
    }
}
    }

    public void joinGroupConfirm(Long source){
synchronized(LOCK){
    if(
       (NODE_STATUS == State.UNJOINED || NODE_STATUS == State.ACTING_MANAGER) &&
       (PROTO_STAGE == State.JOIN_SERVER) &&
       (source.equals(LAST_MESSAGE_SOURCE))
       ){

printDebugStats("["+ID+"] Join Confirm From: ", source);

// Add this guy to our group and proceed on
Node n = new Node();
n.id(source.longValue());
TOPOLOGY.add(n);
byte[] outb = TOPOLOGY.getBytes();
allocate(outb.length+2);

OUTBOUND.putChar(ProtocolConstants.TOPOLOGY);
OUTBOUND.put(outb);

writeMessage(AddressConversion.getBroadcastAddress());

if(EnvironmentVariables.getDebugLevel() > 1 )
    System.out.println(TOPOLOGY);

LAST_MESSAGE_SOURCE = source;
timestamp();

updateSponsor();

timestamp();
    }
    else if(timedOut()){
PROTO_STAGE = State.IDLE;
    }
}
    }

    public void mergeGroupRequest(Long source){
synchronized(LOCK){
    // Send a merge response...
    if(
       ( NODE_STATUS == State.ACTING_MANAGER) &&
       (PROTO_STAGE == State.IDLE)
       ){

printDebugStats("["+ID+"] Merge Request From: ", source);

// Send join response..
LAST_MESSAGE_SOURCE = source;
allocate(2);
OUTBOUND.putChar(ProtocolConstants.MERGE_GROUP_RESPONSE);
writeMessage(source);
PROTO_STAGE = State.MERGE_SERVER;
timestamp();

    }
    else if(timedOut()){
PROTO_STAGE = State.IDLE;
timestamp();
```

```
if(NODE_STATUS == State.ACTING_MANAGER){
    mergeGroupRequest(source);
}
    }
}
    }

    public void mergeGroupResponse(Long source){
synchronized(LOCK){
    // Send a confirmation with a copy of our topology...
    if(
       (NODE_STATUS == State.ACTING_MANAGER) &&
       (PROTO_STAGE == State.IDLE)
       ){

printDebugStats("["+ID+"] Merge Response From: ", source);

// Allocate the space for our topology
byte[] a = TOPOLOGY.getBytes();
allocate(a.length+2);
OUTBOUND.putChar(ProtocolConstants.MERGE_GROUP_CONFIRM);
OUTBOUND.put(a);
writeMessage(source);

PROTO_STAGE = State.MERGE_CLIENT;
// We were a manager, and will be no more, so doing this IS proper...
MANAGER_ID = source;
LAST_MESSAGE_SOURCE = source;

timestamp();
return;
    }
    else if(timedOut()){
PROTO_STAGE = State.IDLE;
if(NODE_STATUS == State.ACTING_MANAGER){
    mergeGroupResponse(source);
}
    }
    else{
printDebugStats("["+ID+"] REJECTING-> ", source);
allocate(2);
OUTBOUND.putChar(ProtocolConstants.REJECT);
writeMessage(source);
    }
}
    }

    public void mergeGroupConfirm(Long source, Topology t){
synchronized(LOCK){
    if(
       (NODE_STATUS == State.ACTING_MANAGER) &&
       (PROTO_STAGE == State.MERGE_SERVER) &&
       (source.equals(LAST_MESSAGE_SOURCE))
       ){

printDebugStats("["+ID+"] Merge Confirm From: ", source);

LAST_MESSAGE_SOURCE = source;

// Add this topology to ours and send the mamma
TOPOLOGY.merge(t);

byte[] outb = TOPOLOGY.getBytes();
allocate(outb.length+2);
OUTBOUND.putChar(ProtocolConstants.TOPOLOGY);
OUTBOUND.put(outb);

writeMessage(AddressConversion.getBroadcastAddress());
```

```
if(EnvironmentVariables.getDebugLevel() > 1 )
    System.out.println(TOPOLOGY);

LAST_MESSAGE_SOURCE = source;
timestamp();

updateSponsor();

timestamp();
    }
    else if(timedOut()){
PROTO_STAGE = State.IDLE;
    }
}
    }

    public void topologyUpdate(Long source, Topology t){
synchronized(LOCK){
    if(
       (
(
 NODE_STATUS == State.UNJOINED &&
 PROTO_STAGE == State.JOIN_CLIENT
 )||
(
 NODE_STATUS == State.JOINED &&
 PROTO_STAGE == State.IDLE
 )
) &&
       (source.equals(MANAGER_ID))
       ){

printDebugStats("["+ID+"] Topology Update: ", source);

TOPOLOGY = t;

if(EnvironmentVariables.getDebugLevel() > 3)
    System.out.println("Topology Update!");

if(EnvironmentVariables.getDebugLevel() > 1)
    System.out.println(TOPOLOGY);

LAST_MESSAGE_SOURCE = source;
timestamp();

updateSponsor();

timestamp();
    }
    else if(
    (
     NODE_STATUS == State.ACTING_MANAGER &&
     PROTO_STAGE == State.MERGE_CLIENT
     )&&
    source.equals(MANAGER_ID)
    ){

// We must send this on to our followers.....
TOPOLOGY = t;

byte[] outb = TOPOLOGY.getBytes();
allocate(outb.length+2);
OUTBOUND.putChar(ProtocolConstants.TOPOLOGY);
OUTBOUND.put(outb);
OUTBOUND.position(0);
writeMessage(AddressConversion.getBroadcastAddress());
```

```
if(EnvironmentVariables.getDebugLevel() > 3)
    System.out.println("Topology Update!");

if(EnvironmentVariables.getDebugLevel() > 1)
    System.out.println(TOPOLOGY);

LAST_MESSAGE_SOURCE = source;
timestamp();

updateSponsor();

timestamp();
    }
    else if(timedOut()){
PROTO_STAGE = State.IDLE;
    }
    else if (!source.equals(MANAGER_ID)){
/*
System.out.println("--------------------------------------");
System.out.println("          DEBUG                       ");
System.out.println("--------------------------------------");
System.out.println("Manager:"+MANAGER_ID);
System.out.println("Source:"+source);
System.out.println(TOPOLOGY);
System.out.println("--------------------------------------");
*/
    }
}
    }

    public void pkStruct(Long source, Vector v){
synchronized(LOCK){
    if(
       PROTO_STAGE == State.REFRESHING_KEY
       ){
KA_METHOD.finishKeyRefresh(source, v);
    }
    else if(timedOut()){
PROTO_STAGE = State.IDLE;
    }
}
    }

    public void idleize(){
synchronized(LOCK){
    if(PROTO_STAGE == State.REFRESHING_KEY){
ELAPSED = System.currentTimeMillis() - CURRENT_TIME;
if(EnvironmentVariables.getDebugLevel() > 0)
    System.out.println(TOPOLOGY.size()+" "+((double)ELAPSED/1000.0));
if(EnvironmentVariables.getDebugLevel() > 2)
    System.out.println("Encryption Key:"+KA_METHOD.getGroupKey());
if(MANAGER_ID.equals(ID)){
    try{
// a nasty little thing, but it may be the case
// that in this implementation, our group is still
// refreshing their key....we should wait for them
Thread.currentThread().sleep(1200);
    }
    catch(Exception e){}
}
PROTO_STAGE = State.IDLE;
    }
}
    }

    public void keyRefresh(Long source, BigInteger p, BigInteger g){
synchronized(LOCK){
    if (
```

```
(
 NODE_STATUS == State.JOINED &&
 PROTO_STAGE == State.IDLE
 )&&
source.equals(MANAGER_ID)
){
PROTO_STAGE = State.REFRESHING_KEY;
CURRENT_TIME = System.currentTimeMillis();
KA_METHOD.updateKey(this, TOPOLOGY, ID, p, g);
    }
    else if(timedOut()){
PROTO_STAGE = State.IDLE;
    }
    else if (!source.equals(MANAGER_ID)){
/*
System.out.println("--------------------------------------");
System.out.println("          DEBUG                       ");
System.out.println("--------------------------------------");
System.out.println("Manager:"+MANAGER_ID);
System.out.println("Source:"+source);
System.out.println(TOPOLOGY);
System.out.println("--------------------------------------");
*/
    }
}
    }


    public void diffieHellmanStage1(Long source, BigInteger pubk, Vector pks){
synchronized(LOCK){
    if(PROTO_STAGE == State.REFRESHING_KEY){
KA_METHOD.diffieHellmanStage1(source, pubk, pks);
    }
    else if(timedOut()){
PROTO_STAGE = State.IDLE;
    }
}
    }

    public void diffieHellmanWait(Long source){
synchronized(LOCK){
    if(PROTO_STAGE == State.REFRESHING_KEY){
KA_METHOD.diffieHellmanWait(source);
    }
    else if(timedOut()){
PROTO_STAGE = State.IDLE;
    }
}
    }

    public void diffieHellmanStage2(Long source, BigInteger pubk, Vector pks){
synchronized(LOCK){
    if(PROTO_STAGE == State.REFRESHING_KEY){
KA_METHOD.diffieHellmanStage2(source, pubk, pks);
    }
    else if(timedOut()){
PROTO_STAGE = State.IDLE;
    }
}
    }

    private void updateSponsor(){

// Sponsor Updates Occur Whenever the topology changes
// And also after the new topology has been acquired....
// Therefore, why not do key agreement now and save a network
// message???
```

```
// Perform the actual sponsor switch
MANAGER_ID = TOPOLOGY.sponsorID();

// If we are now the manager, then perform so...
if(MANAGER_ID.equals(ID)){


    if(EnvironmentVariables.getDebugLevel() > 3)
System.out.println("I am the MIGHTY SPONSOR...");

    NODE_STATUS = State.ACTING_MANAGER;

    // send the REFRESH_KEY message....
    // since I am manager, we'll use my prime and generator
    BigInteger p;
    BigInteger g;

    try{
Vector v = KA_METHOD.getConstants();
p = ((BigInteger)v.elementAt(0));
g = ((BigInteger)v.elementAt(1));
byte[] pa = p.toByteArray();
byte[] ga = g.toByteArray();
allocate(pa.length+ ga.length +6);
OUTBOUND.putChar(ProtocolConstants.KEY_REFRESH);
OUTBOUND.putChar((char)pa.length);
OUTBOUND.put(pa);
OUTBOUND.putChar((char)ga.length);
OUTBOUND.put(ga);

if(EnvironmentVariables.getDebugLevel() > 3)
    System.out.print("Writing key refresh...");
writeMessage(AddressConversion.getBroadcastAddress());
if(EnvironmentVariables.getDebugLevel() > 3)
    System.out.println("done!");

PROTO_STAGE = State.REFRESHING_KEY;

CURRENT_TIME = System.currentTimeMillis();
KA_METHOD.updateKey(this, TOPOLOGY, ID, p, g);

    }
    catch(Exception e){e.printStackTrace();}
}
else{
    if(EnvironmentVariables.getDebugLevel() > 3){
System.out.println("I am wormtongue!");
    }
    // wait for the key broadcast...
    // it should be immediate....
    // but you never know....
    NODE_STATUS = State.JOINED;
    PROTO_STAGE = State.IDLE;
}
    }

}
```

5.2.20 v2/net/NetworkDevice.java

```
package net;

import utils.*;
import topo.*;
import ka.*;
```

```
import java.io.IOException;
import java.net.*;

import java.math.BigInteger;
import java.util.Vector;

public class NetworkDevice{

    private int DEBUG;
    public int PORT;
    public int MTU;

    public String IPA;
    private int VMID;

    private String INTERFACE;
    private Identifier IDENT;
    private String TOPOLOGY;

    private Topology TOPO;
    private KeyAgreement KA;

    private GroupSocket GS;

    public static void main(String[] args){
try{
    new NetworkDevice();
}
catch (IOException e){
    e.printStackTrace();
}
    }

    /*
      ----------------------------------------------------------------------------------------
      Public Methods
      ----------------------------------------------------------------------------------------

     */

    // Let the identifier find the best interface
    // Use the default port and IPv4 address
    public NetworkDevice() throws IOException{
// Snag anything in the environment...
getEnvironment();

// Get a unique network identifier and check some hardware...
obtainID();

if (DEBUG > 2){
    System.out.println("\nNetwork Identity Obtained!");
    System.out.println("-------------------------");
    System.out.println("IP:"+IDENT.getInetAddress().toString().substring(1));
    System.out.println("VMID:"+IDENT.getVMID());
    System.out.println("Network ID:"+IDENT.getID());
    System.out.println("-------------------------\n");
}

// At this point, prelimenary network checks are a success
// Proceed....
obtainTopology();
obtainKeyAgreement();
startDevice();
    }

    /*
      ----------------------------------------------------------------------------------------
      Public Methods
```

```
      ---------------------------------------------------------------------------------------
    */

    public void close() throws IOException{
GS.close();
    }

    /*
      ------------------------------------------------------
      Typical write() methods in the Java standard
      ------------------------------------------------------
    */
    public void write(byte[] b, Long destination) throws IOException{
GS.write(b, destination);
    }

    public void write(byte[] b, int len, Long destination) throws IOException{
GS.write(b, len, destination);
    }

    public void write(byte[] b, int offset, int len, Long destination) throws IOException{
GS.write(b, offset, len, destination);
    }

    /*
      ------------------------------------------------------
      Typical read() methods in the Java standard
      ------------------------------------------------------
    */

    public Message read() throws IOException{
return GS.read();
    }

    /*
      ------------------------------------------------------
      Returns a vector of type Long containing all of the
      members currently in the group, according to the
      implementation of Topology.getGroupMembers()
      ------------------------------------------------------
    */

    public Vector getGroupMembers(){
return GS.getGroupMembers();
    }

    public BigInteger getGroupKey(){
return GS.getGroupKey();
    }

    public BigInteger getPrivateKey(){
return GS.getPrivateKey();
    }

    /*
      ---------------------------------------------------------------------------------------
      Private Methods
      ---------------------------------------------------------------------------------------

     */

    private void obtainKeyAgreement(){
if (DEBUG > 3){
    System.out.println("\nDynamically loading key agreement for "+TOPOLOGY);
}

try{
    Class topoKA = ClassLoader.getSystemClassLoader().loadClass("ka."+TOPOLOGY+"KeyAgreement");
```

```
    if(DEBUG > 3)
System.out.println("The name of the loaded class is: "+topoKA.getName());

    Object obj = topoKA.newInstance();
    if(obj instanceof KeyAgreement){
KA = (KeyAgreement)obj;
if (DEBUG > 2)
    System.out.println("Key Agreement successfully loaded!!!");
    }
    else{
System.out.println("Class ka."+TOPOLOGY+"KeyAgreement.class was successfully loaded, but it does not implement in
System.out.println("Please adhere to proper coding standards....");
System.exit(-1);
    }
}
catch(ClassNotFoundException e){
    System.out.println("Class ka."+TOPOLOGY+"KeyAgreement.class NOT FOUND!");
    System.out.println("Are you sure that "+TOPOLOGY+"KeyAgreement.class is in the \"ka\" directory of the base p
    System.exit(-1);
}
catch(InstantiationException e){
    System.out.println("Class ka."+TOPOLOGY+"KeyAgreement.class found, but could not be instantiated!");
    System.out.println("Are you sure that "+TOPOLOGY+"KeyAgreement.java compiles cleanly???");
    System.exit(-1);
}
catch(IllegalAccessException e){
    System.out.println("Class ka."+TOPOLOGY+"KeyAgreement.class found, but could not be instantiated!");
    System.out.println("Are you sure that "+TOPOLOGY+"KeyAgreement.java compiles cleanly???");
    System.exit(-1);
}
    }



    private void obtainTopology(){
if(TOPOLOGY == null){
    if (DEBUG > 2){
System.out.println("No Topology Specified!!!");
System.out.println("Defaulting to tree-based heirarchy!");
    }
    TOPOLOGY="tgdh";
}
if (DEBUG > 2){
    System.out.println("\nDynamically loading topology control for "+TOPOLOGY);
}

try{

    Class topo = ClassLoader.getSystemClassLoader().loadClass("topo."+TOPOLOGY+"Topology");
    if(DEBUG > 4)
System.out.println("The name of the loaded class is: "+topo.getName());

    Object obj = topo.newInstance();
    if(obj instanceof Topology){
TOPO = (Topology)obj;
if (DEBUG > 2)
    System.out.println("Topology successfully loaded!!!");
    }
    else{
System.out.println("Class topo."+TOPOLOGY+"Topology.class was successfully loaded, but it does not implement inte
System.out.println("Please adhere to proper coding standards....");
System.exit(-1);
    }
}
catch(ClassNotFoundException e){
    System.out.println("Class topo."+TOPOLOGY+"Topology.class NOT FOUND!");
    System.out.println("Are you sure that "+TOPOLOGY+"Topology.class is in the \"topo\" directory of the base pac
    System.exit(-1);
```

```
}
catch(InstantiationException e){
    System.out.println("Class topo."+TOPOLOGY+"Topology.class found, but could not be instantiated!");
    System.out.println("Are you sure that "+TOPOLOGY+"Topology.java compiles cleanly???");
    System.exit(-1);
}
catch(IllegalAccessException e){
    System.out.println("Class topo."+TOPOLOGY+"Topology.class found, but could not be instantiated!");
    System.out.println("Are you sure that "+TOPOLOGY+"Topology.java compiles cleanly???");
    System.exit(-1);
}
    }

    private void startDevice() throws IOException{
if (DEBUG > 3){
    System.out.println("\nStarting Network I/O....");
}
GS = new GroupSocket(new Long(IDENT.getID()), PORT, MTU, TOPO, KA);
    }

    private void obtainID() throws IOException{
// We were given an IP address
if(IPA != null && !IPA.equals("")){
    InetAddress ina = null;
    int ipa = -1;

    // We could have been given one of the two representations...

    // The integer representation...
    if (java.util.regex.Pattern.matches("-{0,1}[0-9]+", IPA)){
ipa = Integer.parseInt(IPA);
    }
    // The IP representation
    else if (java.util.regex.Pattern.matches("((([0-9]{1,3})\\.){3})([0-9]{1,3})", IPA)){
ina = Inet4Address.getByName(IPA);
    }

    // We were given an interface
    if(INTERFACE != null && !INTERFACE.equals("")){
if(ina != null){
    IDENT = new Identifier(INTERFACE, ina);
}
else if(ipa != -1){
    IDENT = new Identifier(INTERFACE, ipa);
}
else{
    IDENT = new Identifier(INTERFACE);
}
    }
    // We were not given an interface
    else{
if(ina != null){
    IDENT = new Identifier(ina);
}
else if(ipa != -1){
    IDENT = new Identifier(ipa);
}
else{
    IDENT = new Identifier();
}
    }
}
// We were not given an IP address
else{
    // We were given an interface
    if(INTERFACE != null && !INTERFACE.equals("")){
IDENT = new Identifier(INTERFACE);
    }
```

```
    // We were not given an interface
    else{
IDENT = new Identifier();
    }
}
    }

    private void getEnvironment(){
DEBUG = EnvironmentVariables.getDebugLevel();
PORT = EnvironmentVariables.getPort();
MTU = EnvironmentVariables.getMTU();
IPA = EnvironmentVariables.getIPAddress();
INTERFACE = EnvironmentVariables.getInterface();
TOPOLOGY = EnvironmentVariables.getTopology();
    }


}
```

5.2.21 v2/net/Reader.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
*/

package net;

import utils.*;

import java.util.Vector;
import java.nio.channels.SelectionKey;
import java.nio.ByteBuffer;
import java.net.InetSocketAddress;

public class Reader implements Runnable{

    private final SelectionKey KEY;

    private ByteBuffer BUFFER;
    private final int MTU;

    private boolean EXECUTED;

    // Where to decode incoming messages...
    private final GroupManager MANAGER;

    // Where to write outgoing messages
    private final GroupSocket SOCKET;

    private final int DEBUG;

    private long last;

    private final Decoder DECODER;

    public Reader(SelectionKey sk, int mtu, GroupManager manager, GroupSocket socket){
EXECUTED = false;
DEBUG=EnvironmentVariables.getDebugLevel();
SOCKET=socket;
MTU=mtu;
KEY=sk;
BUFFER= ByteBuffer.allocateDirect(MTU);
MANAGER = manager;
DECODER = new Decoder(MANAGER);
last = System.currentTimeMillis();
    }
```

```
    public void run(){
try{
    BUFFER.position(0);
    InetSocketAddress a = (InetSocketAddress)((java.nio.channels.DatagramChannel)(KEY.channel())).receive(BUFFER)

    // Here, each thread has the "key" until they have read something....
    if(a != null && BUFFER.position() != 0){
BUFFER.position(0);
int vmidDEST = BUFFER.getInt();
int vmidSOURCE = BUFFER.getInt();
int length = BUFFER.getInt();

byte[] message = new byte[length];
BUFFER.get(message);

int ipSOURCE = AddressConversion.addrToInt(a.getAddress());
Long idSource = AddressConversion.createID(ipSOURCE, vmidSOURCE);

// Here, we can send this work off to another thread, they
// will never see the SelectionKey

DECODER.decode(message, vmidDEST, idSource);
    }

    else if(MANAGER.isIdle() && (System.currentTimeMillis() - last) > 1000){
try{
    Vector message = MANAGER.getIdleMessage();
    if(message != null){
Long dest = (Long)message.elementAt(0);
ByteBuffer mess = (ByteBuffer)message.elementAt(1);
byte[] out = new byte[mess.limit()];
mess.get(out);
SOCKET.rawWrite(out, dest);
last = System.currentTimeMillis();
    }
}
catch(Exception e){
    System.out.println("Couldn't write to "+ AddressConversion.getBroadcastAddress()+"!\n");
    System.out.println("I'm really busted...");
    e.printStackTrace();
    System.exit(-1);
}
    }

    KEY.interestOps(KEY.OP_READ);
    KEY.attach(this);
}
catch(Exception e){ e.printStackTrace();}
    }

}
```

5.2.22 v2/net/Decoder.java

```
package net;

import utils.*;
import topo.Topology;
import topo.Node;

import java.nio.ByteBuffer;
import java.math.BigInteger;
import java.util.Vector;

public class Decoder{
```

```
    private final GroupManager MANAGER;
    private ByteBuffer BUFFER;
    private int DEST;
    private Long SOURCE;

    public Decoder(GroupManager manager){
MANAGER=manager;
    }

    public void decode(byte[] b, int dest, Long source){
BUFFER = ByteBuffer.allocate(b.length);
BUFFER.put(b);
BUFFER.position(0);
DEST=dest;
SOURCE=source;

// Don't act on our own messages...
if(
   // First, it was not sent by us,
   !SOURCE.equals(MANAGER.ID)
   ){
    if(
       // Make sure it was sent to either us or broadcast....
       (AddressConversion.getVMID(MANAGER.ID) == DEST || DEST == 0)
       ){
final char OPCODE = BUFFER.getChar();

switch(OPCODE){

case ProtocolConstants.TOPLEVEL_MESSAGE:
    int lengt = BUFFER.getInt();
    System.out.println("Message of length "+lengt+" received!");
    byte[] messa = new byte[lengt];
    BUFFER.get(messa);

    ByteBuffer mess = ByteBuffer.allocate(messa.length);

    mess.put(messa);
    mess.position(0);

    MANAGER.toplevelMessage(SOURCE, mess);
    break;

case ProtocolConstants.REJECT:
    if(EnvironmentVariables.getDebugLevel() > 4)
System.out.println("JOIN_REJECT");
    MANAGER.reject(SOURCE);
    break;

case ProtocolConstants.JOIN_GROUP_REQUEST:
    if(EnvironmentVariables.getDebugLevel() > 4)
System.out.println("JOIN_GROUP_REQUEST");
    MANAGER.joinGroupRequest(SOURCE);
    break;

case ProtocolConstants.JOIN_GROUP_RESPONSE:
    if(EnvironmentVariables.getDebugLevel() > 4)
System.out.println("JOIN_GROUP_RESPONSE");
    MANAGER.joinGroupResponse(SOURCE);
    break;

case ProtocolConstants.JOIN_GROUP_CONFIRM:
    if(EnvironmentVariables.getDebugLevel() > 4)
System.out.println("JOIN_GROUP_CONFIRM");
    MANAGER.joinGroupConfirm(SOURCE);
    break;
```

```
case ProtocolConstants.MERGE_GROUP_REQUEST:
    if(EnvironmentVariables.getDebugLevel() > 4)
System.out.println("MERGE_GROUP_REQUEST");
    MANAGER.mergeGroupRequest(SOURCE);
    break;

case ProtocolConstants.MERGE_GROUP_RESPONSE:
    if(EnvironmentVariables.getDebugLevel() > 4)
System.out.println("MERGE_GROUP_RESPONSE");
    MANAGER.mergeGroupResponse(SOURCE);
    break;

case ProtocolConstants.MERGE_GROUP_CONFIRM:
    if(EnvironmentVariables.getDebugLevel() > 4)
System.out.println("MERGE_GROUP_CONFIRM");

    byte[] out = new byte[BUFFER.limit() - BUFFER.position()];
    BUFFER.get(out);
    Topology top = MANAGER.TOPOLOGY.newInstance(out);
    MANAGER.mergeGroupConfirm(SOURCE, top);

    break;

case ProtocolConstants.JUMPSTART:
    MANAGER.jumpstart();
    break;

case ProtocolConstants.DH_STAGE_1:
    if(EnvironmentVariables.getDebugLevel() > 4)
System.out.println("DH_STAGE_1");

    boolean nodes = ((int)BUFFER.getChar()) == 1;
    int leng = (int)BUFFER.getChar();
    byte[] pubk = new byte[leng];
    BUFFER.get(pubk);
    BigInteger pub = new BigInteger(pubk);

    int count = (int)(BUFFER.getChar());
    Vector pks = new Vector();

    for(int i = 0; i < count; i++){
int le = BUFFER.getChar();
byte[] a = new byte[le];
BUFFER.get(a);
Object pk;
if(!nodes){
    BigInteger pk1 = new BigInteger(a);
    pk = pk1;
}
else{
    Node n = new Node(a);
    pk = n;
}
pks.add(pk);
    }
    MANAGER.diffieHellmanStage1(source, pub, pks);
    break;

case ProtocolConstants.DH_STAGE_2:
    if(EnvironmentVariables.getDebugLevel() > 4)
System.out.println("DH_STAGE_2");
    leng = (int)BUFFER.getChar();
    pubk = new byte[leng];
    BUFFER.get(pubk);
    pub = new BigInteger(pubk);

    count = (int)BUFFER.getChar();
    pks = new Vector();
```

```
    for(int i = 0; i < count; i++){
int le = BUFFER.getChar();
byte[] a = new byte[le];
BUFFER.get(a);
BigInteger pk = new BigInteger(a);
pks.add(pk);
    }
    MANAGER.diffieHellmanStage2(source, pub, pks);
    break;

case ProtocolConstants.DH_WAIT:
    if(EnvironmentVariables.getDebugLevel() > 4)
System.out.println("DH_WAIT");
    MANAGER.diffieHellmanWait(source);
    break;

case ProtocolConstants.PK_STRUCT:
    nodes = ((int)BUFFER.getChar()) == 1;
    if(EnvironmentVariables.getDebugLevel() > 4)
System.out.println("PK_STRUCT");
    count = (int)BUFFER.getChar();
    pks = new Vector();
    for(int i = 0; i < count; i++){
int le = BUFFER.getChar();
byte[] a = new byte[le];
BUFFER.get(a);
Object pk;
if(!nodes){
    BigInteger pk1 = new BigInteger(a);
    pk = pk1;
}
else{
    if(le == 0){
pk = null;
    }
    else{
Node n = new Node(a);
pk = n;
    }
}
pks.add(pk);
    }
    MANAGER.pkStruct(source, pks);
    break;

case ProtocolConstants.KEY_REFRESH:
    if(EnvironmentVariables.getDebugLevel() > 4)
System.out.println("KEY_REFRESH");
    int l = (int)BUFFER.getChar();
    byte[] in = new byte[l];
    BUFFER.get(in);
    BigInteger p = new BigInteger(in);

    l = (int)BUFFER.getChar();
    in = new byte[l];
    BUFFER.get(in);
    BigInteger g = new BigInteger(in);

    MANAGER.keyRefresh(SOURCE, p, g);
    break;

case ProtocolConstants.TOPOLOGY:
    if(EnvironmentVariables.getDebugLevel() > 4)
System.out.println("TOPOLOGY");

    byte[] outb = new byte[BUFFER.limit() - BUFFER.position()];
    BUFFER.get(outb);
```

```
    Topology t = MANAGER.TOPOLOGY.newInstance(outb);
    MANAGER.topologyUpdate(SOURCE, t);

    break;

}
    }
    else if(EnvironmentVariables.getDebugLevel() > 4)
System.out.println(
   "VMID Does Not Match!\n"+
   "ME:"+AddressConversion.getVMID(MANAGER.ID)+"\n"+
   "DEST:"+DEST
   );
}
else if(EnvironmentVariables.getDebugLevel() > 4)
    System.out.println("I Sent This!");
    }
}
```

5.2.23 v2/net/State.java

```
package net;

public class State{

    // The network NODE_STATUS states
    public static final char UNJOINED = 0x00;
    public static final char JOINED = 0x01;
    public static final char ACTING_MANAGER = 0x02;

    // The protocol stage states

    // Not in a network protocol stage
    public static final char IDLE = 0x10;
    public static final char DEAD_BATTERY = 0x11;

    // Somewhere in a join stage
    public static final char JOIN_SERVER = 0x20;
    public static final char JOIN_CLIENT = 0x21;
    public static final char JOIN_PENDING = 0x22;

    // Somewhere in a merge stage
    public static final char MERGE_SERVER = 0x30;
    public static final char MERGE_CLIENT = 0x31;
    public static final char MERGE_PENDING = 0x32;

    // Somewhere in a key-update stage
    public static final char REFRESHING_KEY = 0x40;

}
```

5.2.24 v2/net/ProtocolConstants.java

```
package net;

public class ProtocolConstants{

    public static final char JOIN_GROUP_REQUEST = 0x00;
    public static final char JOIN_GROUP_RESPONSE = 0x01;
    public static final char JOIN_GROUP_CONFIRM = 0x02;

    public static final char MERGE_GROUP_REQUEST = 0x10;
    public static final char MERGE_GROUP_RESPONSE = 0x11;
```

```
    public static final char MERGE_GROUP_CONFIRM = 0x12;

    public static final char KEY_REFRESH = 0x20;
    public static final char PK_STRUCT = 0x21;
    public static final char DH_STAGE_1= 0x22;
    public static final char DH_STAGE_2= 0x23;
    public static final char DH_WAIT= 0x24;

    public static final char TOPLEVEL_MESSAGE = 0xe0;

    public static final char REJECT = 0xfd;
    public static final char JUMPSTART = 0xfe;
    public static final char TOPOLOGY = 0xff;

}
```

5.2.25 v2/net/Message.java

```
package net;

public class Message{

    public Long SOURCE;
    public byte[] MESSAGE;

    public Message(Long source, byte[] message){
SOURCE=source;
MESSAGE=message;
    }

}
```

5.2.26 v2/demo/DemoMaster.java

```
package demo;

import net.NetworkDevice;
import utils.AddressConversion;

import java.util.Vector;
import java.nio.ByteBuffer;
import java.math.BigInteger;

public class DemoMaster{
    public static void main(String[] args){
if(args.length != 1){
    System.out.println("Usage:\n\tjava DemoMaster <groupSize>");
    System.exit(-1);
}
NetworkDevice device = null;
try{
    device = new NetworkDevice();
}
catch (Exception e){
    e.printStackTrace();
    System.exit(-1);
}
System.out.println("Device Created!");
Vector members = null;
try{
    while(
  (members = device.getGroupMembers()).size() != Integer.parseInt(args[0])
  ){
```

```
System.out.println("Size is too small ("+members.size()+"), waiting...");
Thread.currentThread().sleep(1000);
    }
}
catch(Exception e){
    e.printStackTrace();
}
System.out.println("Group of Proper Size!");

byte[] out = new String("Hello World!").getBytes();
BigInteger mess = new BigInteger(out)
; while(device.getGroupKey() == null){}
BigInteger result = mess.xor(device.getGroupKey());
System.out.println("Encrypting:"+new String(out));
System.out.println("Writing :"+result+" as "+result.toByteArray().length+" bytes...");

try{
    device.write(result.toByteArray(), AddressConversion.getBroadcastAddress());
    System.out.println("Written!");
    Thread.currentThread().sleep(1000);
    System.exit(0);
}
catch(Exception e){
    e.printStackTrace();
}
    }
}
```

5.2.27 v2/demo/DemoSlave.java

```
package demo;

import net.NetworkDevice;
import net.Message;

import java.util.Vector;
import java.math.BigInteger;

public class DemoSlave{
    public static void main(String[] args){
if(args.length != 1){
    System.out.println("Usage:\n\tjava DemoMaster <groupSize>");
    System.exit(-1);
}
NetworkDevice device = null;
try{
    device = new NetworkDevice();
}
catch (Exception e){
    e.printStackTrace();
    System.exit(-1);
}
System.out.println("Device Created!");

try{
    Vector members = null;
    while(
  (members = device.getGroupMembers()).size() != Integer.parseInt(args[0])
  ){
System.out.println("Size is too small ("+members.size()+"), waiting...");
Thread.currentThread().sleep(1000);
    }

    Message message = device.read();
    System.out.println("Message read!");
```

```
    BigInteger bi = new BigInteger(message.MESSAGE);
    System.out.println("Received:"+bi);

    while(device.getGroupKey() == null){}
    BigInteger original = bi.xor(device.getGroupKey());
    System.out.println("Unencrypted:"+new String(original.toByteArray()));
    System.exit(0);
}
catch(Exception e){
    e.printStackTrace();
}
    }
}
```

5.2.28 gens/generator.java

```
public class generator{
public static void main(String[] args){
double bottom = Math.log(2.0);
for(int i = 2; i <= 140; i++){
double top = Math.log((double)i);
int whole = (int)Math.ceil(top/bottom);
System.out.println(2*i*whole);
}
}
}
```

5.2.29 v1/HGDH/Hypercube.java

```
import java.util.Vector;
import java.io.*;

public class Hypercube{



  public static void main(String[] args){
    Hypercube STRUCT = new Hypercube();
    try{
      BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
      while(true){
System.out.print("\nHypercube >> ");
String command = br.readLine();
String[] operands = command.split("\\s");
if(operands.length == 0){
  operands = new String[1];
  operands[0] = "";
}
final String operator = operands[0].toLowerCase();
if(operator.indexOf("add") >= 0 ){
  if(
     operands.length < 2 ||
     !java.util.regex.Pattern.matches("[0-9]+", operands[1])
     ){
    System.out.println("Add -> Wrong Syntax:\n"+
       "Hypercube>> add <nrNodes>");
  }
  else{
    int nrToAdd = Integer.parseInt(operands[1]);
    for(int i = 0; i < nrToAdd; i++){
      long rand = (long)(Math.random()*1000000000.0);
      Node n = new Node();
      n.id(rand);
```

```
      STRUCT.add(n);
    }
  }
}
else if(operator.indexOf("rem") >= 0){
  if(
     operands.length < 2 ||
     !java.util.regex.Pattern.matches("[0-9]+", operands[1])
     ){
    System.out.println("rem -> Wrong Syntax:\n"+
       "Hypercube>> rem <nodeID>");
  }
  else{
    long rem = Long.parseLong(operands[1]);
    boolean gone = false;
    for(int i = 0; i < STRUCT.size(); i++){
      Node n = (Node)(STRUCT.getVector().elementAt(i));
      if(n.toString().indexOf(new Long(rem).toString()) >= 0){
STRUCT.remove(n);
gone = true;
      }
    }
    if(!gone){
      System.out.println("No resemblance of "+rem+" could be found!");
    }
  }
}
else if(operator.indexOf("merge") >= 0){
  if(
     operands.length < 2 ||
     !java.util.regex.Pattern.matches("[0-9]+", operands[1])
     ){
    System.out.println("Merge -> Wrong Syntax:\n"+
       "Hypercube>> merge <HypercubeSize>");
  }
  else{
    int nrToAdd = Integer.parseInt(operands[1]);
    Hypercube na = new Hypercube();
    for(int i = 0; i < nrToAdd; i++){
      long rand = (long)(Math.random()*1000000000.0);
      Node n = new Node();
      n.id(rand);
      na.add(n);
    }
    STRUCT.merge(na);
  }
}
else if(operator.indexOf("part") >= 0){

}
else if(operator.indexOf("quit") >= 0){
  System.out.println("Goodbye!");
  System.exit(0);
}
else{
  System.out.println("Unknown Operator! -> ["+operands[0]+"]");
}
System.out.println("Structure:\n"+STRUCT);
      }
    }
    catch (IOException e){
      e.printStackTrace();
      System.exit(0);
    }
  }
```

```
  private Vector CUBE;

  public Hypercube(){
    CUBE = new Vector();
  }

  public Hypercube(Vector v){
    CUBE = v;
  }

  public Hypercube(byte[] nodes){
    CUBE = new Vector();
    int count = 0;
    java.nio.ByteBuffer bb = java.nio.ByteBuffer.allocate(nodes.length);
    bb.put(nodes);
    bb.position(0);
    while (bb.position() < bb.limit()){
      int len = bb.getInt();
      byte[] node = new byte[len];
      bb.get(node);
      Node n = new Node(node);
      CUBE.add(n);
    }
  }

  public void add(Node n){
    synchronized(CUBE){
      CUBE.add(0, n);
    }
  }

  public void remove(Node n){
    synchronized(CUBE){
      CUBE.remove(n);
    }
  }

  public void merge(Hypercube others){
    synchronized(CUBE){
      CUBE.addAll(others.getVector());
    }
  }

  // Functions as Vector.removeAll();
  public void partition(Vector ousted){
    synchronized(CUBE){
      CUBE.removeAll(ousted);
    }
  }

  public int size(){
    return CUBE.size();
  }

  public Vector getVector(){
    synchronized(CUBE){
      return CUBE;
    }
  }

  public String toString(){
    String retval = "";
    synchronized(CUBE){
      int a = CUBE.size();
      if(a == 0){
retval = "---------\nEmpty Set\n---------";
      }
      else{
```

```
retval += "-----------------------------\n";
for(int i = 0; i < a; i++){
  String id =  new Long(((Node)CUBE.elementAt(i)).id()).toString();
  retval += "[ " + pad(Integer.toString(i, 2)) + " | " + id + " ]" +"\n";
}
retval += "-----------------------------\n";
      }
    }
    return retval;
  }

  private String pad(String s){
    String retval = new String(s);
    int maxlen = new java.math.BigInteger(Integer.toString(CUBE.size()-1)).bitLength();
    int slen = s.length();
    int its = maxlen-slen;
    for(int i = 0; i < its; i++){
      retval ="0"+retval;
    }
    return retval;
  }

  public long getSponsor(){
    return ((Node)(CUBE.elementAt((CUBE.size()-1)))).id();
  }

  public byte[] getBytes(){
    synchronized(CUBE){
      // A node must be far less than 64 bytes
      java.nio.ByteBuffer out = java.nio.ByteBuffer.allocate(64*CUBE.size());
      int count = 0;
      for(int i = 0; i < CUBE.size(); i++){
byte[] a = ((Node)(CUBE.elementAt(i))).getBytes();
out.putInt(a.length);
out.put(a);
count += a.length+4;
      }
      byte[] retval = new byte[count];
      out.position(0);
      out.get(retval);
      return retval;
    }
  }

  public int indexOf(long id){
    int index = -1;
    for(int i = 0; i < CUBE.size(); i++){
      Node n = (Node)(CUBE.elementAt(i));
      if(n.id() == id){
index = i;
      }
    }
    return index;
  }

  public int dimension(){
    return new java.math.BigInteger(new Integer(CUBE.size()-1).toString()).bitLength();
  }

}
```

5.2.30 v1/HGDH/Node.java

```
import java.math.BigInteger;

public class Node{
```

```
private long ID;
private byte INTERMEDIARY;
// Each node in the tree generates a public key through a diffie-hellman exchange.
public BigInteger PUBLIC_KEY;

public Node(){
  PUBLIC_KEY = new BigInteger("-1");
  ID = -1;
}

public Node(byte[] b){
  java.nio.ByteBuffer bb = java.nio.ByteBuffer.allocate(b.length);
  bb.put(b);
  bb.position(0);

  if(bb.get() == -1){
    ID = -1;
  }
  else{
    ID = bb.getLong();
  }

  int length = bb.getInt();
  byte[] shareval = new byte[length];
  bb.get(shareval);

  PUBLIC_KEY = new BigInteger(shareval);
}

public byte[] getBytes(){
  java.nio.ByteBuffer out = java.nio.ByteBuffer.allocate(sizeInBytes());

  // Hack to save space!!!
  if(ID == -1){
    out.put((byte)-1);
  }
  else{
    out.put((byte)1);
    out.putLong(ID);
  }

  byte[] a = PUBLIC_KEY.toByteArray();

  out.putInt(a.length);
  out.put(a);

  byte[] retval = new byte[sizeInBytes()];
  out.position(0);
  out.get(retval);

  return retval;
}

public long id(){
  return ID;
}

public void id(long id){
  ID = id;
}

public boolean intermediary(){
  return (ID == -1)?true:false;
}

public void intermediary(boolean value){
  INTERMEDIARY = ((value==true)?(byte)-1:(byte)1);
```

```
  }

  public String toString(){
    String s = new Long(ID).toString();
    return s.substring(((s.length()-7)>=0)?(s.length()-7):0, s.length());
  }

  // 1 byte intermediate flag
  // Possible 8 byte ID
  // 16/17 byte public key
  public int sizeInBytes(){
    int pkl = PUBLIC_KEY.toByteArray().length;
    return (ID == -1)?(pkl+5):(pkl+13);
  }

}
```

5.2.31 v1/HGDH/Decoder.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
 */

import java.nio.channels.SelectionKey;
import java.nio.ByteBuffer;

public class Decoder implements Runnable{

  private SelectionKey KEY;
  private ByteBuffer MESSAGE;
  private boolean EXECUTED;

  int length;
  byte[] tree;


  public Decoder(SelectionKey key, ByteBuffer mesg){
    KEY=key;
    EXECUTED = false;
    MESSAGE=mesg;
  }

  public void run(){
    synchronized(this){
      if(EXECUTED)return;
      else EXECUTED=true;
    }
    /* Decode a message */
    long src = MESSAGE.getLong();
    long dest = MESSAGE.getLong();
    if(src != Reactor.ID && (dest == Message_OPC.BROADCAST || dest == Reactor.ID )){
      final char type = MESSAGE.getChar();
      switch(type){

      case Message_OPC.JUMPSTART:
Reactor.PROCESSOR.jumpstart();
break;

/* ----------------- Join Messages ----------------- */
      case Message_OPC.JOIN_REQUEST:
Reactor.PROCESSOR.join_request(src);
break;

      case Message_OPC.JOIN_RESPONSE:
Reactor.PROCESSOR.join_response(src);
break;
```

```
      case Message_OPC.JOIN_CONFIRM:
byte[] rest = new byte[Reactor.NODE.sizeInBytes()];
MESSAGE.get(rest);
Node n = new Node(rest);
Reactor.PROCESSOR.join_confirm(n);
break;

      case Message_OPC.JOIN_REJECT:
Reactor.PROCESSOR.join_reject(src);
break;

/* ---------------- Merge Messages ----------------- */

      case Message_OPC.MERGE_REQUEST:
Reactor.PROCESSOR.merge_request(src);
break;

      case Message_OPC.KEY_UPDATE:
int plen = MESSAGE.getInt();
byte[] p = new byte[plen];
MESSAGE.get(p);
java.math.BigInteger prime = new java.math.BigInteger(p);

int glen = MESSAGE.getInt();
byte[] g = new byte[glen];
MESSAGE.get(g);
java.math.BigInteger generator = new java.math.BigInteger(g);

Reactor.PROCESSOR.keyUpdate(src, prime, generator);
break;

      case Message_OPC.MERGE_RESPONSE:
Reactor.PROCESSOR.merge_response(src);
break;

      case Message_OPC.MERGE_CONFIRM:
int length = MESSAGE.getInt();
byte[] array = new byte[length];
MESSAGE.get(array);
Hypercube his = new Hypercube(array);

Reactor.PROCESSOR.merge_confirm(src, his);
break;

      case Message_OPC.DH_WAIT:
Reactor.PROCESSOR.dh_wait(src);
break;

      case Message_OPC.DH_STAGE_1:

length = MESSAGE.getInt();
byte[] pa = new byte[length];
MESSAGE.get(pa);
prime = new java.math.BigInteger(pa);

length = MESSAGE.getInt();
byte [] ga = new byte[length];
MESSAGE.get(ga);
generator = new java.math.BigInteger(ga);

length = MESSAGE.getInt();
byte[] pka = new byte[length];
MESSAGE.get(pka);
java.math.BigInteger public_key = new java.math.BigInteger(pka);

Reactor.PROCESSOR.diffieHellmanStageOne(src, prime, generator, public_key);
break;
```

```
      case Message_OPC.DH_STAGE_2:
length = MESSAGE.getInt();
pa = new byte[length];
MESSAGE.get(pa);
java.math.BigInteger pkb = new java.math.BigInteger(pa);

Reactor.PROCESSOR.diffieHellmanStageTwo(src, pkb);
break;

      case Message_OPC.MERGE_REJECT:
Reactor.PROCESSOR.merge_reject(src);
break;

      case Message_OPC.SPONSOR_SWITCH:
long newS = MESSAGE.getLong();

int len = MESSAGE.getInt();
byte[] a4 = new byte[len];
MESSAGE.get(a4);

Reactor.PROCESSOR.sponsor_switch(src, newS, new Hypercube(a4));
break;


/* ------------------------------------------------ */

      case Message_OPC.H_COMMIT:
Reactor.PROCESSOR.h_commit(src);
break;

      case Message_OPC.H_STRUCTURE:
length = 0;
try{

  int l = MESSAGE.getInt();
  byte[] ab = new byte[l];
  MESSAGE.get(ab);
  Hypercube h = new Hypercube(ab);
  Reactor.PROCESSOR.array_update(src, h);

}
catch(Exception e){
  if (e instanceof java.nio.BufferUnderflowException){
    System.out.println("Needed:"+length+"\nAvailable:"+(MESSAGE.limit()-MESSAGE.position()));
    System.out.println("Network MTU Exceeded By Group Structure Transmission!\n"+
       "Fragmentation Not Implemented Yet!"
       );
    System.exit(-1);
  }
}
break;

      case Message_OPC.KEY_ARRAY:
length = 0;
try{
  length = MESSAGE.getInt();
  java.util.Vector vect = new java.util.Vector();
  for(int f = 0; f < length; f++){
    int leng = MESSAGE.getInt();
    byte[] arra = new byte[leng];
    MESSAGE.get(arra);
    java.math.BigInteger bb = new java.math.BigInteger(arra);
    vect.add(bb);
  }
  Reactor.PROCESSOR.key_array(src, vect);
}
catch(Exception e){
```

```
  if (e instanceof java.nio.BufferUnderflowException){
    System.out.println("Needed:"+length+"\nAvailable:"+(MESSAGE.limit()-MESSAGE.position()));
    System.out.println("Network MTU Exceeded By Group Structure Transmission!\n"+
       "Fragmentation Not Implemented Yet!"
       );
    System.exit(-1);
  }
}
break;
      }
    }

  }

}
```

5.2.32 v1/HGDH/Device.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
 */

/**
   The main executable class for Kieran S. Hagzan's master's thesis work at R.I.T. .
   This class parses command-line arguments, checks their validity, creates a new
   <code>Reactor</code>and displays usage information if necessary.
 */
public class Device{

  /**
     A newly created <code>Reactor</code> to be run.
   */
  private final Reactor REACTOR;

  /**
      Creates a new <code>Reactor</code> on the given port
      using the specified network MTU.
      @param port The port to bind the <code>Reactor</code> on, (1024 <= port <= 65535).
      @param mtu The Maximum Transmission Unit of the <code>Reactor</code> (1 <= mtu <= 65535).
   */
  private Device(int port, int mtu, boolean jumpstart, int maxNodes){
    REACTOR = new Reactor(port, mtu, jumpstart, maxNodes);
    REACTOR.setDebugLevel(2);
    REACTOR.run();
  }

  /**
     The main method invoked by the VM at program call.
     @param args The command-line arguments passed to this class.
   */
  public static void main(String[] args){
    boolean jumpstart;
    int mtu;
    int port;
    int max;

    // Bomb out if the user cannot type "--help"
    if (args.length > 4)exitWithUsage();
    // grab the outermost argument - the jumpstart mode if present
    if(args.length == 4){
      String s = args[2];

      if ((s.indexOf("true") < 0) && (s.indexOf("false") < 0)){
exitWithUsage();
      }
```

```
      if(s.indexOf("true") >= 0){
jumpstart = true;
      }
      else{
jumpstart = false;
      }
    }
    else{
      jumpstart = false;
    }

    // Grab the first argument - the port if present
    if(args.length == 4){
      if(!java.util.regex.Pattern.matches("[0-9]+", args[3])){
exitWithUsage();
      }
      max = Integer.parseInt(args[3]);
    }
    else{
      max = 100;
    }

    // Grab the secondmost argument - the MTU if present
    if(args.length >= 2){
      if(!java.util.regex.Pattern.matches("[0-9]+", args[1])){
exitWithUsage();
      }
      mtu = Integer.parseInt(args[1]);
      if (mtu < 33 || mtu > 65535){exitWithUsage();}
    }
    else{
      mtu = 512;
    }

    // Grab the first argument - the port if present
    if(args.length >= 1){
      if(!java.util.regex.Pattern.matches("[0-9]+", args[0])){
exitWithUsage();
      }
      port = Integer.parseInt(args[0]);
      if(port < 1024 || port > 65535){exitWithUsage();}
    }
    else{
      port = 12333;
    }

    // Let them know what they have done....
    //    System.out.println("Initializing on port "+port+" using MTU of "+mtu+
    //        (jumpstart?" with":" without")+" jumpstart mode...");

    new Device(port, mtu, jumpstart, max);
  }

  /**
     Displays proper usage of this class.
   */
  private static void exitWithUsage(){
    System.out.println("\nUsage:\n------\n"+
       "\t\"java Device --help\" will display this screen\n"+
       "\t\"java Device <port> <mtu> <jumpstart> \"\n\nWhere:\n------\n"+
       "<port> - is an optionally specified number between 1024 and 65535 (defaults to 12333)\n"+
       "<mtu>  - is an optionally specified number between 33 and 65535 (defaults to 512)\n"+
       "<jumpstart> - is one of \"true\" or \"false\" indicating the jumpstart status of the node\n"
       );
    System.exit(1);
  }
}
```

5.2.33 v1/HGDH/DiffieHellmanExchange.java

```
import java.math.*;
import java.util.Random;
import java.nio.ByteBuffer;

public class DiffieHellmanExchange{

  public BigInteger p;
  public BigInteger g;
  public BigInteger x;
  public BigInteger public_key_alice;
  public BigInteger public_key_bob;
  public BigInteger secret_key;

  public long ALICE;
  public long BOB;

  // Constructor for bob...
  public DiffieHellmanExchange(
       long alice,
       long bob,
       BigInteger prime,
       BigInteger generator,
       BigInteger share,
       BigInteger alices_public_key
       ){
    ALICE=alice;
    BOB=bob;
    p = prime;
    g = generator;
    public_key_alice = alices_public_key;

    Random rand = new Random();

    x = share;

    // And public key....
    public_key_alice = alices_public_key;
    public_key_bob = g.modPow(x,p);
    secret_key = alices_public_key.modPow(x, p);
  }



  // Constructor for alice
  public DiffieHellmanExchange(long alice,
       long bob,
       BigInteger prime,
       BigInteger generator,
       BigInteger share
       ){
    ALICE=alice;
    BOB=bob;

    Random rand = new Random();

    // Generate a private key....
    x = share;
    g = generator;
    p = prime;

    // And public key....
    public_key_alice = generator.modPow(x,prime);

    // Now, things are a little different than in TGDH
```

```
    byte[] po = prime.toByteArray();
    byte[] go = generator.toByteArray();

    ByteBuffer bb = ByteBuffer.allocate(Reactor.MTU);
    bb.putChar(Message_OPC.DH_STAGE_1);

    bb.putInt(po.length);
    bb.put(po);

    bb.putInt(go.length);
    bb.put(go);

    byte[] pko = public_key_alice.toByteArray();
    bb.putInt(pko.length);
    bb.put(pko);

    byte[] out = new byte[bb.position()];
    bb.position(0);
    bb.get(out);

    Reactor.write(out, bob);
  }



}
```

5.2.34 v1/HGDH/Discoverer.java

```
import java.nio.channels.SelectionKey;
import java.nio.ByteBuffer;

public class Discoverer implements Runnable{

  private final SelectionKey KEY;
  private boolean EXECUTED;

  public Discoverer(SelectionKey key){
    KEY=key;
    EXECUTED=false;
  }

  public void run(){
    synchronized(this){
      if(EXECUTED)return;
      EXECUTED=true;
    }
    ByteBuffer bb = ByteBuffer.allocate(2);
    bb.putChar(Message_OPC.JOIN_REQUEST);
    bb.position(0);
    byte[] a = new byte[bb.limit()];
    bb.get(a);
    Reactor.write(a, Message_OPC.BROADCAST);
    KEY.attach(Reactor.READER);
    KEY.interestOps(KEY.OP_READ);
  }

}
```

5.2.35 v1/HGDH/Jumpstarter.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
```

```
 */

public class Jumpstarter{
  public static void main(String[] args){
boolean local;
if(args.length > 0 && args[0].indexOf("local") >= 0){
  local = true;
}
else local=false;
    try{
      java.nio.channels.DatagramChannel c = java.nio.channels.DatagramChannel.open();
      c.socket().setReuseAddress(true);
      c.socket().setBroadcast(true);
      c.socket().bind(new java.net.InetSocketAddress("0.0.0.0", 12333));
      java.nio.ByteBuffer out = java.nio.ByteBuffer.allocateDirect(18);
      out.putLong(0xdeadbeefl);
      out.putLong(0xffffffffffffffffl);
      out.putChar(Message_OPC.JUMPSTART);
      for(int i = 0; i < 3; i++){
out.position(0);
java.net.InetSocketAddress outA;
if(local){
  System.out.println("Yelling locally...");
  outA = new java.net.InetSocketAddress("localhost", 12333);
}
else{
    outA = new java.net.InetSocketAddress("255.255.255.255", 12333);
}
      c.send(out, outA);
      }
      c.close();
    }
    catch(Exception e){
      e.printStackTrace();
      System.exit(-1);
    }
  }
}
```

5.2.36 v1/HGDH/Merger.java

```
import java.nio.channels.SelectionKey;
import java.nio.ByteBuffer;

public class Merger implements Runnable{

  private final SelectionKey KEY;
  private boolean EXECUTED;

  public Merger(SelectionKey key){
    KEY=key;
    EXECUTED=false;
  }

  public void run(){
    synchronized(this){
      if(EXECUTED)return;
      EXECUTED=true;
    }
    ByteBuffer bb = ByteBuffer.allocate(2);
    System.out.println("Requesting Merge...");
    bb.putChar(Message_OPC.MERGE_REQUEST);
    bb.position(0);
    byte[] a = new byte[bb.limit()];
    bb.get(a);
    Reactor.write(a, Message_OPC.BROADCAST);
```

```
    KEY.attach(new Reader(KEY));
    KEY.interestOps(KEY.OP_READ);
  }

}
```

5.2.37 v1/HGDH/Message_OPC.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
 */

public class Message_OPC{

  public static final long BROADCAST = 0xffffffffffffffl;

  public static final char JUMPSTART =0xffff;

  public static final char JOIN_REQUEST =0x0001;
  public static final char JOIN_RESPONSE =0x0002;
  public static final char JOIN_CONFIRM =0x0003;
  public static final char JOIN_REJECT =0x0004;


  public static final char H_COMMIT =0x0005;
  public static final char H_STRUCTURE = 0x0006;

  public static final char MERGE_REQUEST = 0x0007;
  public static final char MERGE_RESPONSE = 0x0008;
  public static final char MERGE_CONFIRM =0x0009;
  public static final char MERGE_REJECT =0x000a;

  public static final char SPONSOR_SWITCH =0x000b;

  public static final char DH_STAGE_1 = 0x000c;
  public static final char DH_STAGE_2 = 0x000d;
  public static final char KEY_UPDATE = 0x000e;
  public static final char DH_WAIT = 0x000f;
  public static final char KEY_ARRAY = 0x0010;

}
```

5.2.38 v1/HGDH/MessageProcessor.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
*/

import java.nio.channels.SelectionKey;
import java.nio.ByteBuffer;
import java.math.BigInteger;
import java.util.Vector;

public class MessageProcessor{

  /*
    Structural Stuff....
   */

  private Hypercube CUBE;
  private Hypercube TEMP_CUBE;
  private int CURRENT_DIMENSION;
  private Vector COLLECTABLES;
```

```
private final int MTU;
public boolean JUMPSTART;

private final Object STATE_LOCK;

private boolean IN_EXCHANGE;
private boolean TIMEOUT_MATTERS;

private long LAST_SOURCE;

private long SPONSOR_ID;
private long NEW_SPONSOR;

public char NODE_STATE;
public char PROTO_STATE;

private ByteBuffer BUFFER;

// Stuff for key agreement....
private int CURRENT_PARTNER;
private long CURRENT_PARTNER_ID;

private BigInteger MY_PRIVATE;
private BigInteger CURRENT_PRIVATE;
private BigInteger ENCRYPTION_KEY;

private BigInteger FIRST_PK;
private BigInteger INITIAL_PK;

// Diffie-Hellman Parameters...
private BigInteger CURRENT_P;
private BigInteger CURRENT_G;
private java.util.Random GENERATOR;

private DiffieHellmanExchange CURRENT_EXCHANGE;

// Stuff for gauging key agreement times
private long START_TIME;
private double ELAPSED;

long lastMeaningful = 0;
long TIMEOUT = 2000;

// --------------------------------

public MessageProcessor(int mtu, boolean jumpstart){
  MTU=mtu;

  INITIAL_PK = null;
  IN_EXCHANGE = false;

  GENERATOR = new java.util.Random();

  JUMPSTART=jumpstart;

  NODE_STATE = State.INIT_ONLY;
  STATE_LOCK = new Object();

  CUBE = new Hypercube();
  Node n = new Node();
  n.id(Reactor.ID);
  CUBE.add(n);
}

/* ------------------------------------------------------------------------------------- */
/* ------------------------------ Helper Functions ------------------------------------- */
/* ------------------------------------------------------------------------------------- */
```

```
 public char getProtoState(){
   return PROTO_STATE;
 }

 public char getNodeState(){
   return NODE_STATE;
 }

 public void setProtoState(char state){
   synchronized(STATE_LOCK){
     PROTO_STATE = state;
   }
 }

 public void setNodeState(char state){
   synchronized(STATE_LOCK){
     NODE_STATE = state;
   }
 }

 public void setStates(char nstate, char pstate){
   synchronized(STATE_LOCK){
     NODE_STATE = nstate;
     PROTO_STATE = pstate;
   }
 }

 private void allocate(int size){
   BUFFER = ByteBuffer.allocate(size);
 }

 public void writeMessage(ByteBuffer b, long dest){
   byte[] a = new byte[b.limit()];
   b.position(0);
   b.get(a);
   Reactor.write(a, dest);
 }

 public void printDebug(String message, long src){
   if(Reactor.DEBUG >= 3){
     System.out.println(
"\n"+message+" "+src+"\n"+
"Node State:0x"+Integer.toString(NODE_STATE, 16)+"\n"+
"Proto State:0x"+Integer.toString(PROTO_STATE, 16)+"\n"
);
     }
 }

 public boolean timedOut(){
   boolean retval = (System.currentTimeMillis()-lastMeaningful > TIMEOUT) && TIMEOUT_MATTERS;
   return retval;
 }

 /**
    Computes the currently known key from the current tree iteratively...
 */
 private void computeKey (Vector v){
   BigInteger roundKey = CURRENT_PRIVATE;
   for(int i = 1; i < v.size(); i++){
     roundKey = ((BigInteger)v.elementAt(i)).modPow(roundKey, CURRENT_P);
   }
   ENCRYPTION_KEY = roundKey;
   ELAPSED = System.currentTimeMillis() - START_TIME;
   System.out.println(CUBE.size()+"\t"+(ELAPSED / 1000));
   PROTO_STATE = State.CALM;
 }
```

```
  private boolean haveStraggler(){
    // We have a straggler if:
    // 1.) Our highest bit is not set (active in first ex.)
    BigInteger me = new BigInteger(new Integer(CUBE.indexOf(Reactor.ID)).toString());
    if(!me.testBit(CUBE.dimension()-1)){
      // 2.) The number of nodes is not an exact power of 2
      if(CUBE.size() != (int)(Math.pow(2.0, CUBE.dimension()))){
// 3.) And our initial partner exists...
int initialPartner = me.flipBit(CUBE.dimension()-1).intValue();
if(initialPartner < CUBE.size()){
  return true;
}
return false;
      }
      return false;
    }
    return false;
  }

  /* ------------------------------------------------------------------------------------ */
  /* ----------------------------- Message Reaction Functions ----------------------------- */
  /* ------------------------------------------------------------------------------------ */

  /* ************************************************************************************ */
  /*                            Functions to deal with key agreement                      */
  /* ************************************************************************************ */

  private boolean active(){
    BigInteger me = new BigInteger(new Integer(CUBE.indexOf(Reactor.ID)).toString());
    return !(me.testBit(CURRENT_DIMENSION-1));
  }

  /**
     Causes all "non-global-sponsor" nodes to initiate the "appropriate" action in response
     to a "KEY_UPDATE" request. i.e., if nodes have an even index in their tree positions,
     they must initiate a Diffie-Hellman key exchange with their partners.  Otherwise,
     they sit and wait for a request.
   */
  public void keyUpdate(long src, BigInteger p, BigInteger g){
    if(
       (src  == SPONSOR_ID) &&
       (NODE_STATE == State.JOINED_N_G_S) &&
       (PROTO_STATE == State.CALM)
       ){

      TIMEOUT_MATTERS = false;
      PROTO_STATE = State.UPDATING_KEY;

      START_TIME = System.currentTimeMillis();

      // Update my private...
      CURRENT_P = p;
      CURRENT_G = g;
      MY_PRIVATE = new BigInteger(128, 100, GENERATOR);
      CURRENT_PRIVATE = MY_PRIVATE;

      // Update the current dimension
      CURRENT_DIMENSION = CUBE.dimension();

      // Update my initial exchange partner
      updatePartner();

      // Start the melee....
      COLLECTABLES = new Vector();
      initDiffieHellman();
    }
    else if(timedOut()){
      TIMEOUT_MATTERS = false;
```

```
      lastMeaningful = System.currentTimeMillis();
      keyUpdate(src,p,g);
    }
  }

  /**
      Initialize a Diffie-Hellman key exchange at the current point in the tree,
      with the current key, prime, and personal share. If the node at which this
      exchange is to take place is an intermediary node, we will query it's "sponsor"
      node.
  */
  public void initDiffieHellman(){
    if(active()){
      if(CURRENT_PARTNER < CUBE.size()){
DiffieHellmanExchange dhe =  new DiffieHellmanExchange(Reactor.ID,
       CURRENT_PARTNER_ID,
       CURRENT_P,
       CURRENT_G,
       CURRENT_PRIVATE);
CURRENT_EXCHANGE = dhe;
// Hang around for stage 2
      }
      // If our current partner does not exist, we are a straggler!!!!
      else{
CURRENT_DIMENSION--;
if(CURRENT_DIMENSION > 0){
  updatePartner();
  initDiffieHellman();
}
      }
    }
    else if(
    CURRENT_DIMENSION == CUBE.dimension() &&
    CUBE.size() != (int)(Math.pow(2.0, CUBE.dimension()))
    ){
    }
  }

  /**
     Message handler for the request of a Diffie-Hellman key exchange.  From a cryptographic
     perspective, this indicates that this node is "Bob," or the non-initiating party...
   */
  public void diffieHellmanStageOne(long src, BigInteger p, BigInteger g, BigInteger pk){
    if(
       (src == CURRENT_PARTNER_ID) &&
       (PROTO_STATE == State.UPDATING_KEY) &&
       (NODE_STATE == State.JOINED_G_S || NODE_STATE == State.JOINED_N_G_S)
       ){

      // I am bob...
      DiffieHellmanExchange dhe = new DiffieHellmanExchange(
    src,
    Reactor.ID,
    CURRENT_P,
    CURRENT_G,
    CURRENT_PRIVATE,
    pk
    );

      CURRENT_EXCHANGE = dhe;
      CURRENT_PRIVATE = CURRENT_EXCHANGE.secret_key;
      COLLECTABLES.add(pk);

      // Perform the DH stage 2....
      byte[] a = dhe.public_key_bob.toByteArray();
      allocate(a.length+6);
      BUFFER.putChar(Message_OPC.DH_STAGE_2);
      BUFFER.putInt(a.length);
```

```
      BUFFER.put(a);
      writeMessage(BUFFER,src);

      //Proceed to the next dimension if there is one...
      CURRENT_DIMENSION--;
      if(CURRENT_DIMENSION > 0){
// IF WE ARE A STRAGGLER, DON'T DO THIS!!!!
// We can catch this here, because all stragglers
// are passive in their outer dimension, i.e.,
// they are "bob", and will get a stage 1
if(
   !(
     (CURRENT_DIMENSION == (CUBE.dimension()-1)) &&
     (CUBE.size() != (int)(Math.pow(2.0, CUBE.dimension()))) &&
     (new BigInteger(Integer.toString(CUBE.indexOf(Reactor.ID))).testBit(CUBE.dimension()-1))
     )
   ){
  updatePartner();
  initDiffieHellman();
}
      }
      // Otherwise, we're done here...
      else{
ENCRYPTION_KEY = CURRENT_PRIVATE;
ELAPSED = System.currentTimeMillis() - START_TIME;
System.out.println(CUBE.size()+"\t"+(ELAPSED / 1000));
if(haveStraggler()){
  sendVector(new BigInteger(new Integer(CUBE.indexOf(Reactor.ID)).toString()).flipBit(CUBE.dimension()-1).intValu
}
PROTO_STATE = State.CALM;
      }
    }
    else if(timedOut()){
      PROTO_STATE = State.CALM;
      TIMEOUT_MATTERS = false;
    }
    else{
      // Warn this guy to try again in a few???
      allocate(2);
      BUFFER.putChar(Message_OPC.DH_WAIT);
      writeMessage(BUFFER, src);
    }
  }

  public void dh_wait(long src){
    if(
       (src == CURRENT_PARTNER_ID)&&
       (PROTO_STATE == State.UPDATING_KEY) &&
       (NODE_STATE == State.JOINED_G_S || NODE_STATE == State.JOINED_N_G_S)
       )
      {
// This SHOULD work???
initDiffieHellman();
      }
  }

  // yep, stage 2 is back...
  public void diffieHellmanStageTwo(long src, BigInteger pk_bob){
    if(
       (src == CURRENT_PARTNER_ID) &&
       (NODE_STATE == State.JOINED_G_S || NODE_STATE == State.JOINED_N_G_S) &&
       (PROTO_STATE == State.UPDATING_KEY)
       )
      {
CURRENT_EXCHANGE.public_key_bob = pk_bob;
COLLECTABLES.add(CURRENT_EXCHANGE.public_key_bob);

CURRENT_EXCHANGE.secret_key = pk_bob.modPow(CURRENT_PRIVATE, CURRENT_P);
```

```
CURRENT_PRIVATE = CURRENT_EXCHANGE.secret_key;

//Proceed to the next dimension if there is one...
CURRENT_DIMENSION--;
if(CURRENT_DIMENSION > 0){
  updatePartner();
  initDiffieHellman();
}
// Otherwise, we're done here
else{
  // BUG HERE SOMEWHERE!!!
  ENCRYPTION_KEY = CURRENT_PRIVATE;
  ELAPSED = System.currentTimeMillis() - START_TIME;
  System.out.println(CUBE.size()+"\t"+(ELAPSED / 1000));
  if(haveStraggler()){
    sendVector(new BigInteger(
      new Integer(
  CUBE.indexOf(Reactor.ID)
  ).toString()
      ).flipBit(
CUBE.dimension()-1
).intValue()
       );
  }
  // Calm ourselves
  PROTO_STATE = State.CALM;
}

      }
    else if(timedOut()){
      PROTO_STATE = State.CALM;
      TIMEOUT_MATTERS = false;
    }
  }

  private void sendVector(int position){

    ByteBuffer temp  = java.nio.ByteBuffer.allocate(17*(COLLECTABLES.size()+4));

    temp.putInt(COLLECTABLES.size());
    int count = 4;

    for(int i = 0; i < COLLECTABLES.size(); i++){
      BigInteger bi = (BigInteger)(COLLECTABLES.elementAt(i));
      byte[] out = bi.toByteArray();
      count += 4;
      temp.putInt(out.length);
      count += (out.length);
      temp.put(out);
    }

    byte[] t = new byte[count];
    temp.position(0);
    temp.get(t);

    allocate(count+2);
    BUFFER.putChar(Message_OPC.KEY_ARRAY);
    BUFFER.put(t);
    writeMessage(BUFFER, ((Node)CUBE.getVector().elementAt(position)).id());
    // If we just sent this, we're done...
    PROTO_STATE = State.CALM;
  }

  private void updatePartner(){
    BigInteger me = new BigInteger(new Integer(CUBE.indexOf(Reactor.ID)).toString());
    CURRENT_PARTNER = me.flipBit(CURRENT_DIMENSION-1).intValue();
    if(CURRENT_PARTNER < CUBE.size()){
      CURRENT_PARTNER_ID = ((Node)(CUBE.getVector().elementAt(CURRENT_PARTNER))).id();
```

```
    }
  }

  /* ############################################################################################  */


  /* ********************************************************************************************* */
  /*                                    End key agreement                                          */
  /* ********************************************************************************************* */


  /* ********************************************************************************************* */
  /*                             Functions to delegate a global sponsor                            */
  /* ********************************************************************************************* */


  private void sponsorUpdate(){
    synchronized(STATE_LOCK){
      lastMeaningful = System.currentTimeMillis();

      // Get the sponsor
      long sponsor = CUBE.getSponsor();
      // Literally do the switch......
      SPONSOR_ID = sponsor;


      // Make a conceptual switch of necessary....
      if (NODE_STATE == State.JOINED_G_S && SPONSOR_ID != Reactor.ID){
NODE_STATE = State.JOINED_N_G_S;
PROTO_STATE = State.CALM;
      }
      else if(NODE_STATE == State.JOINED_N_G_S && SPONSOR_ID == Reactor.ID){
NODE_STATE = State.JOINED_G_S;
PROTO_STATE = State.CALM;
      }

      if (NODE_STATE == State.JOINED_G_S){

PROTO_STATE = State.UPDATING_KEY;
TIMEOUT_MATTERS = false;

// Reset the public values, p, g, and our share...
CURRENT_P = new BigInteger(128, 100, GENERATOR);
CURRENT_G = new BigInteger(CURRENT_P.bitLength(), GENERATOR);

// Broadcast the update request...
allocate(Reactor.MTU);
byte[] pa = CURRENT_P.toByteArray();
byte[] ga = CURRENT_G.toByteArray();
allocate(pa.length+ga.length+10);
BUFFER.putChar(Message_OPC.KEY_UPDATE);
BUFFER.putInt(pa.length);
BUFFER.put(pa);
BUFFER.putInt(ga.length);
BUFFER.put(ga);

TIMEOUT_MATTERS = false;

// Start the timer to include the write to
// compensate for the missing read...
writeMessage(BUFFER, Message_OPC.BROADCAST);
START_TIME = System.currentTimeMillis();

// Initially, the private key is our share
MY_PRIVATE = new BigInteger(128, 100, GENERATOR);
CURRENT_PRIVATE = MY_PRIVATE;

// Update our dimension
```

```
CURRENT_DIMENSION = CUBE.dimension();

// Update our partner....
updatePartner();

// Start the melee....
COLLECTABLES = new Vector();
initDiffieHellman();
      }
      else{
PROTO_STATE = State.CALM;
      }
    }
  }

  public void sponsor_switch(long src, long newS, Hypercube h){
    synchronized(STATE_LOCK){
      if(src == SPONSOR_ID && NODE_STATE == State.JOINED_N_G_S){
// This looks ugly????
TEMP_CUBE = h;
LAST_SOURCE = newS;
SPONSOR_ID = newS;
PROTO_STATE = State.JOIN_CLIENT_PENDING;
lastMeaningful = System.currentTimeMillis();
      }
    }
  }


  /* ********************************************************************************** */
  /*                                End global sponsor                                  */
  /* ********************************************************************************** */



  /* ********************************************************************************** */
  /*                                Protocol Function Handlers                          */
  /* ********************************************************************************** */


  public void join_probe(){
    synchronized(STATE_LOCK){
      allocate(2);
      BUFFER.putChar(Message_OPC.JOIN_REQUEST);
      writeMessage(BUFFER, Message_OPC.BROADCAST);
    }
  }

  public void merge_probe(){
    synchronized(STATE_LOCK){
      allocate(2);
      BUFFER.putChar(Message_OPC.MERGE_REQUEST);
      writeMessage(BUFFER, Message_OPC.BROADCAST);
    }
  }

 public void jumpstart(){
    synchronized(STATE_LOCK){
      if(NODE_STATE == State.INIT_ONLY){
NODE_STATE = State.JOINING;
PROTO_STATE = State.CALM;
join_probe();
Reactor.KEY.attach(Reactor.READER);
Reactor.KEY.interestOps(Reactor.KEY.OP_READ);
lastMeaningful = System.currentTimeMillis();
      }
    }
  }
```

```
  public void join_request(long src){
    synchronized(STATE_LOCK){
      if(
 (NODE_STATE == State.JOINING || NODE_STATE == State.JOINED_G_S) &&
 PROTO_STATE==State.CALM
 ){

TIMEOUT_MATTERS=true;
PROTO_STATE = State.JOIN_SERVER;
allocate(2);
BUFFER.putChar(Message_OPC.JOIN_RESPONSE);
writeMessage(BUFFER, src);
LAST_SOURCE = src;
lastMeaningful = System.currentTimeMillis();
      }
      else if(timedOut()){
PROTO_STATE = State.CALM;
join_request(src);
      }
    }
  }

  public void merge_request(long src){
    synchronized(STATE_LOCK){
      if(
 NODE_STATE == State.JOINED_G_S &&
 PROTO_STATE == State.CALM
 ){

TIMEOUT_MATTERS=true;
PROTO_STATE = State.MERGE_SERVER;
allocate(2);
BUFFER.putChar(Message_OPC.MERGE_RESPONSE);
writeMessage(BUFFER, src);
LAST_SOURCE = src;
lastMeaningful = System.currentTimeMillis();
      }
      else if(timedOut()){
PROTO_STATE = State.CALM;
merge_request(src);
      }
    }
  }

  public void join_response(long src){
    synchronized(STATE_LOCK){
      if(
 NODE_STATE == State.JOINING &&
 (PROTO_STATE == State.CALM)
 ){
TIMEOUT_MATTERS=true;
PROTO_STATE = State.JOIN_CLIENT;
SPONSOR_ID = src;

Node n = new Node();
n.id(Reactor.ID);
allocate(n.sizeInBytes()+2);

BUFFER.putChar(Message_OPC.JOIN_CONFIRM);
BUFFER.put(n.getBytes());

writeMessage(BUFFER, src);
LAST_SOURCE = src;
lastMeaningful = System.currentTimeMillis();
      }
      else{
if(timedOut()){
```

```
  PROTO_STATE = State.CALM;
  join_response(src);
}
else{
  // Tell em' to look elsewhere...
  allocate(2);
  BUFFER.putChar(Message_OPC.JOIN_REJECT);
  writeMessage(BUFFER, src);
}
      }
    }
  }

  public void merge_response(long src){
    synchronized(STATE_LOCK){
      if(
 NODE_STATE == State.JOINED_G_S &&
 PROTO_STATE == State.CALM
 ){
TIMEOUT_MATTERS=true;

PROTO_STATE = State.MERGE_CLIENT;
LAST_SOURCE = src;

byte[] a = CUBE.getBytes();
allocate(a.length + 6);
BUFFER.putChar(Message_OPC.MERGE_CONFIRM);
BUFFER.putInt(a.length);
BUFFER.put(a);

writeMessage(BUFFER, src);
lastMeaningful = System.currentTimeMillis();
      }
      else{
if(timedOut()){
  PROTO_STATE = State.CALM;
  merge_response(src);
}
else{
// Tell them to look elsewhere..
allocate(2);
BUFFER.putChar(Message_OPC.MERGE_REJECT);
writeMessage(BUFFER, src);
}
      }
    }
  }

  public void join_confirm(Node n){
    synchronized(STATE_LOCK){
      if(
 (NODE_STATE == State.JOINING || NODE_STATE == State.JOINED_G_S) &&
 PROTO_STATE == State.JOIN_SERVER
 ){
if(n.id() == LAST_SOURCE){

  TEMP_CUBE  = new Hypercube(CUBE.getBytes());
  TEMP_CUBE.add(n);
  byte[] a = TEMP_CUBE.getBytes();
  allocate(a.length + 6);
  BUFFER.putChar(Message_OPC.H_STRUCTURE);
  BUFFER.putInt(a.length);
  BUFFER.put(a);

  writeMessage(BUFFER, Message_OPC.BROADCAST);
  PROTO_STATE = State.JOIN_SERVER_PENDING;

  lastMeaningful = System.currentTimeMillis();
```

```
}
else if(timedOut()){
  PROTO_STATE = State.CALM;
}
      }
      else if(timedOut()){
PROTO_STATE = State.CALM;
      }
    }
  }

  public void merge_confirm(long src, Hypercube h){
    synchronized(STATE_LOCK){
      if(
 NODE_STATE == State.JOINED_G_S &&
 PROTO_STATE == State.MERGE_SERVER
 ){
if(src == LAST_SOURCE){

  // Grab the array transmitted to us and
  // add it to our own...

  // Grab the array transmitted to us and
  // add it to our own...
  TEMP_CUBE = CUBE;
  TEMP_CUBE.merge(h);
  byte[] a = TEMP_CUBE.getBytes();
  allocate(a.length + 6);
  BUFFER.putChar(Message_OPC.H_STRUCTURE);
  BUFFER.putInt(a.length);
  BUFFER.put(a);

  writeMessage(BUFFER, Message_OPC.BROADCAST);
  PROTO_STATE = State.MERGE_SERVER_PENDING;
  lastMeaningful = System.currentTimeMillis();
}
else if(timedOut()){
  PROTO_STATE = State.CALM;
}
      }
      else if(timedOut()){
PROTO_STATE = State.CALM;
      }
    }
  }

  public void join_reject(long src){
    synchronized(STATE_LOCK){
      if(src == LAST_SOURCE && NODE_STATE < 3){
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
lastMeaningful = System.currentTimeMillis();
      }
    }
  }

  public void merge_reject(long src){
    synchronized(STATE_LOCK){
      if(src == LAST_SOURCE && NODE_STATE == State.JOINED_G_S && PROTO_STATE != State.CALM){
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
lastMeaningful = System.currentTimeMillis();
      }
    }
  }

  public void array_update(long id, Hypercube h){
    synchronized(STATE_LOCK){
```

```
      if(
 (NODE_STATE == State.JOINING && PROTO_STATE == State.JOIN_CLIENT)
 ){
if(id == SPONSOR_ID){
  LAST_SOURCE  = id;
  PROTO_STATE = State.JOIN_CLIENT_PENDING;
  TEMP_CUBE = h;
  allocate(2);
  BUFFER.putChar(Message_OPC.H_COMMIT);
  writeMessage(BUFFER, id);
  lastMeaningful = System.currentTimeMillis();
}
else if(timedOut()){
  PROTO_STATE = State.CALM;
}
      }
      else if(NODE_STATE == State.JOINED_N_G_S && PROTO_STATE == State.CALM){
if(id == SPONSOR_ID){
  LAST_SOURCE =  id;
  PROTO_STATE = State.JOIN_CLIENT_PENDING;
  TEMP_CUBE = h;
  lastMeaningful = System.currentTimeMillis();
}
else if(timedOut()){
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS = false;
}
      }
      else if (NODE_STATE == State.JOINED_G_S && PROTO_STATE == State.MERGE_CLIENT){
if(id == LAST_SOURCE){

  LAST_SOURCE = id;
  NODE_STATE = State.JOINED_N_G_S;
  TEMP_CUBE = h;

  byte[] out = TEMP_CUBE.getBytes();
  allocate(out.length+14);

  BUFFER.putChar(Message_OPC.SPONSOR_SWITCH);
  BUFFER.putLong(LAST_SOURCE);

  BUFFER.putInt(out.length);
  BUFFER.put(out);

  writeMessage(BUFFER, Message_OPC.BROADCAST);

  // Inform the merge master to commit
  allocate(2);
  BUFFER.putChar(Message_OPC.H_COMMIT);
  writeMessage(BUFFER, id);
  lastMeaningful = System.currentTimeMillis();
  SPONSOR_ID = LAST_SOURCE;
}
else if(timedOut()){
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS = false;
}
      }
      else if(timedOut()){
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
      }
    }
  }

  /* ********************************************************************************** */
  /*                                End Protocol Function Handlers                      */
  /* ********************************************************************************** */
```

```
  /* ********************************************************************************** */
  /*                                  Tree Update Functions                             */
  /* ********************************************************************************** */

  public void key_array(long id, Vector v){
    synchronized(STATE_LOCK){
      // I only care about this if:
      // 1.) I am in the right state
      if(
 (NODE_STATE == State.JOINED_G_S || NODE_STATE == State.JOINED_N_G_S) &&
 (PROTO_STATE == State.UPDATING_KEY)
 ){
// 2.) I was passive in the first round
BigInteger me = new BigInteger(new Integer(CUBE.indexOf(Reactor.ID)).toString());
if(me.testBit(CUBE.dimension()-1)){
  int firstPartner = me.flipBit(CUBE.dimension()-1).intValue();
  // 3.) My partner in the first round exists
  if(firstPartner < CUBE.size()){
    long first = ((Node)CUBE.getVector().elementAt(firstPartner)).id();
    // 4.)  The id of that node is the same as the sender of this
    if(id == first){
      computeKey(v);
    }
  }
}
      }
    }
  }

  public void h_commit(long id){
    synchronized(STATE_LOCK){
      if(id == LAST_SOURCE){
if(
   (NODE_STATE == State.JOINING && PROTO_STATE == State.JOIN_CLIENT_PENDING) ||
   (NODE_STATE == State.JOINED_G_S &&PROTO_STATE == State.MERGE_CLIENT)
   )
  {
    CUBE = TEMP_CUBE;
    NODE_STATE = State.JOINED_N_G_S;
    PROTO_STATE = State.CALM;
    TIMEOUT_MATTERS=false;
    sponsorUpdate();
  }

else if(
(NODE_STATE == State.JOINING || NODE_STATE == State.JOINED_G_S) &&
PROTO_STATE == State.JOIN_SERVER_PENDING
){
  CUBE = TEMP_CUBE;

  allocate(2);
  BUFFER.putChar(Message_OPC.H_COMMIT);
  writeMessage(BUFFER, Message_OPC.BROADCAST);
  NODE_STATE = State.JOINED_G_S;
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS=false;
  sponsorUpdate();
}

else if(NODE_STATE == State.JOINED_G_S && PROTO_STATE == State.MERGE_SERVER_PENDING){
  CUBE = TEMP_CUBE;

  allocate(2);
  BUFFER.putChar(Message_OPC.H_COMMIT);
  writeMessage(BUFFER, Message_OPC.BROADCAST);
  NODE_STATE = State.JOINED_G_S;
```

```
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS=false;
  sponsorUpdate();
}
else if (NODE_STATE == State.JOINED_N_G_S){
  CUBE = TEMP_CUBE;

  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS=false;
  sponsorUpdate();
}
      }
      else if(timedOut()){
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
      }
      lastMeaningful = System.currentTimeMillis();
    }
  }

  /* ********************************************************************************** */
  /*                                    End Tree Functions                              */
  /* ********************************************************************************** */


}
```

5.2.39 v1/HGDH/Reactor.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
 */

import java.nio.channels.DatagramChannel;
import java.nio.channels.Selector;
import java.nio.channels.SelectionKey;
import java.net.InetSocketAddress;

/**
   Class <code>Reactor</code> is the main powerhouse for non-blocking UDP
   datagram broadcasts. It follows a traditional "Reactor" software pattern
   and uses a thread pool to handle the influx of events triggering actions.
 */
public class Reactor implements Runnable{

  /** The port where this <code>Reactor</code> will run.*/
  private static int PORT;

  /** The network Maximum Transmission Unit used in this <code>Reactor</code>.*/
  public static int MTU;

  public static long ID;

  /** The <code>DatagramChannel</code> used as a transport mechanism.*/
  private static DatagramChannel CHANNEL;

  private boolean JUMPSTART;

  public static int MAX_NODES;

  /**
      A </code>Selector</code> used to manage the I/O status of <code>SelectionKeys</code>
      in this <code>Reactor</code>, one for bookeeping measures, another for insecure user
      message capabilities.
   */
  private Selector SELECTOR;
```

```
/** The <code>SelectionKey</code> used for secure group communication I/O bookeeping data. */
public static SelectionKey KEY;

public static Reader READER;

public static MessageProcessor PROCESSOR;

public static Node NODE;

private long accumulated;

public static int DEBUG = 5;

public void setDebugLevel(int level){
  DEBUG = level;
}

/** Constructs a new <code>Reactor</code> on port 12333 with a default MTU of 512 bytes. */
public Reactor(){
  this(12333, 512, false, 100);
}

/**
    Constructs a new <code>Reactor</code> on the specified port with a default MTU of 512 bytes.
    @param port The port to bind the <code>Reactor</code>'s <code>DatagramChannel</code> on.
*/
public Reactor(int port){
  this(port, 512, false, 100);
}

/**
   Constructs a new <code>Reactor</code> on the specified port and MTU.
   @param port The port to bind the <code>Reactor</code>'s <code>DatagramChannel</code> on.
   @param mtu The Maximum Transmission Unit used on the underlying network media.
 */
public Reactor (int port, int mtu){
  this(port, mtu, false, 100);
}

public Reactor(int port, int mtu, boolean jumpstart, int max){
  MAX_NODES = max;
  PORT=port;
  MTU=mtu;
  JUMPSTART=jumpstart;
  ID = (long)(Math.random()*1000000000000000.0);
  NODE = new Node();
  NODE.id(ID);
  accumulated = 0;
  DEBUG=3;
}

/**
   Runs this <code>Reactor</code>.  First, all channels are initialized to be non-blocking,
   then an infinite poll-loop is entered to facilitate reaction-when-necessary event handling.
 */
public void run(){
  if(DEBUG == 5){
    System.out.print("Reactor initializing...");
  }
  initialize();
  if(DEBUG == 5){
    System.out.println("done!");
  }
  if(!JUMPSTART){
    // Attach a new discoverer here...
    PROCESSOR.NODE_STATE = State.JOINING;
    PROCESSOR.PROTO_STATE = State.CALM;
```

```
      if(DEBUG >= 3){
System.out.println("[Reactor]:Probing...");
      }
      PROCESSOR.join_probe();
      KEY.attach(READER);
      KEY.interestOps(KEY.OP_READ);
    }
    else{
      PROCESSOR.NODE_STATE = State.INIT_ONLY;
      PROCESSOR.PROTO_STATE = State.CALM;
      // Otherwise wait for the jumpstart message to propogate to us...
      if (DEBUG >= 3){
System.out.println("Reader Attached!");
      }
      KEY.attach(READER);
      KEY.interestOps(KEY.OP_READ);
    }
    poll();
  }

  /**
     Initializes this <code>Reactor</code>. Channels and selectors are opened and configured for
     non-blocking I/O, and keys are set for reading until later notified otherwise.
   */
  private void initialize(){
    try{
      CHANNEL = DatagramChannel.open();

      // This line allows for many instances of this
      // class to run on the same machine...
      CHANNEL.socket().setReuseAddress(true);
      CHANNEL.socket().setBroadcast(true);
      CHANNEL.configureBlocking(false);
      /* Bind to the broadcast address on the given port */
      CHANNEL.socket().bind(new InetSocketAddress("0.0.0.0", PORT));
      SELECTOR = Selector.open();
      KEY = CHANNEL.register(SELECTOR, SelectionKey.OP_READ);
      /* Create the message processor using out arguments */
      READER = new Reader(KEY);
      PROCESSOR = new MessageProcessor(MTU, JUMPSTART);
      // If we are to not to be jumpstarted,
    }
    catch(Exception e){
      System.out.println("failed!\n\nError occured during initialization:");
      e.printStackTrace();
      System.exit(-1);
    }
  }

  /**
     The main poll loop.  The native hardware of the system is polled to see
     if any I/O bound for this <code>Reactor</code> is ready.  If so, a set
     of <code>SelectionKey</code>s are returned and iterated through, processing
     the I/O events in turn by separate thread-pool threads for maximum event-handling
     throughput.
   */
  private void poll(){
    if(DEBUG >= 3){
      System.out.println("Beginning poll loop...");
    }
    while(!Thread.currentThread().interrupted()){
      try{
if
  (
   (
    PROCESSOR.NODE_STATE == State.JOINING ||
    PROCESSOR.NODE_STATE == State.JOINED_G_S
    ) &&
```

```
   PROCESSOR.PROTO_STATE == State.CALM
   )
  {
    SELECTOR.select((int)(Math.random()*100000.0)%1000);
  }
else if(PROCESSOR.PROTO_STATE != State.CALM){
  SELECTOR.select(1000);
}
else
  {
    SELECTOR.select();
  }
/* ---------------------------------- */
      }
      catch(Exception e){
System.out.print("\nCaught poll interrupt!\nShutting down...");
try{
  KEY.cancel();
  CHANNEL.close();
  SELECTOR.close();
  System.out.println("done!");
  System.exit(-1);
}
catch(Exception f){}
      }
      java.util.Set readyKeys = SELECTOR.selectedKeys();
      java.util.Iterator it = readyKeys.iterator();
      while(it.hasNext()){
SelectionKey sk = (SelectionKey)it.next();
if(sk != null){
  Object o = sk.attachment();
  if(o != null && o instanceof Runnable){
    ((Runnable)o).run();
  }
}
      }
    }
  }

  public static void write(byte[] message, long destination){
    /* Later this needs to be passed to an encoder for breaking into chunks */
    java.nio.ByteBuffer out = java.nio.ByteBuffer.allocate(message.length+16);
    out.putLong(ID);
    out.putLong(destination);
    out.put(message);
    out.position(0);
    int wrote = 0;
    try{
      while(out.position() < out.capacity()-1){
CHANNEL.send(out, new java.net.InetSocketAddress("255.255.255.255", PORT));
      }
    }
    catch(Exception e){
      System.out.println("I/O Error on Channel!");
      e.printStackTrace();
    }
  }

}
```

5.2.40 v1/HGDH/Reader.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
*/
```

```
import java.nio.channels.SelectionKey;
import java.nio.ByteBuffer;
import java.net.SocketAddress;

public class Reader implements Runnable{

  private final SelectionKey KEY;
  private ByteBuffer BUFFER;
  private boolean EXECUTED;

  public Reader(SelectionKey sk){
    KEY=sk;
    EXECUTED=false;
    BUFFER= ByteBuffer.allocate(Reactor.MTU);
  }

  public void run(){
    try{
      BUFFER.position(0);
      SocketAddress a = ((java.nio.channels.DatagramChannel)(KEY.channel())).receive(BUFFER);
      BUFFER.position(0);

      // Either there is something here, useability undetermined...
      if(a != null){
new Decoder(KEY, BUFFER).run();
      }

      else if(
      (Reactor.PROCESSOR.NODE_STATE == State.JOINING && Reactor.PROCESSOR.PROTO_STATE == State.CALM)
      ){
if (Reactor.DEBUG >= 3)
  System.out.println("[Reader]:Join Probing...");
Reactor.PROCESSOR.join_probe();
      }

      else if(
      (Reactor.PROCESSOR.NODE_STATE == State.JOINED_G_S && Reactor.PROCESSOR.PROTO_STATE == State.CALM)
      ){
if (Reactor.DEBUG >= 3)
  System.out.println("[Reader]:Merge Probing...");
Reactor.PROCESSOR.merge_probe();
      }

      // Or we have timed out...
      else{
       Reactor.PROCESSOR.setProtoState(State.CALM);
KEY.selector().wakeup();
      }

      KEY.attach(this);
      KEY.interestOps(KEY.OP_READ);
      return;

    }
    catch(Exception e){
      System.out.println("I/O Error while reading!!!!");
      e.printStackTrace();
    }
  }
}
```

5.2.41 v1/HGDH/State.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
 */
```

```
public class State{

  /* Node States...*/
  public static final char INIT_ONLY = 0x0001;
  public static final char JOINING = 0x0002;
  public static final char JOINED_N_G_S = 0x0003;
  public static final char JOINED_G_S = 0x0004;

  /* Protocol States...*/

  /* The Join States ... */
  public static final char JOIN_SERVER = 0x0010;
  public static final char JOIN_CLIENT = 0x0011;
  public static final char JOIN_SERVER_CONFIRMED = 0x0012;
  public static final char JOIN_SERVER_PENDING = 0x0013;
  public static final char JOIN_CLIENT_PENDING = 0x0014;

  /* The Merge States */
  public static final char MERGE_SERVER = 0x0020;
  public static final char MERGE_CLIENT = 0x0021;
  public static final char MERGE_SERVER_CONFIRMED = 0x0022;
  public static final char MERGE_SERVER_PENDING = 0x0023;
  public static final char MERGE_CLIENT_PENDING = 0x0024;

  /* States relevant during key update */

  public static final char AWAITING_KEY_ARRAY= 0x0031;
  public static final char REQUESTING_TREE = 0x0032;
  public static final char UPDATING_KEY = 0x0033;

  /* The Calm State...*/
  public static final char CALM = 0xffff;

}
```

5.2.42 v1/HGDH/MessageProcessor-OLD.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
*/

import java.nio.channels.SelectionKey;
import java.nio.ByteBuffer;
import java.math.BigInteger;
import java.util.Vector;

public class MessageProcessor{

  /*
    Structural Stuff....
   */

  private Hypercube CUBE;
  private Hypercube TEMP_CUBE;
  private int currentDimension;

  private final int MTU;
  public boolean JUMPSTART;
  private final SelectionKey KEY;

  private final Object STATE_LOCK;

  private boolean IN_EXCHANGE;
  private boolean TIMEOUT_MATTERS;
```

```
private long LAST_SOURCE;

private long SPONSOR_ID;
private long NEW_SPONSOR;

private long MERGER_ID;

public char NODE_STATE;
public char PROTO_STATE;

private ByteBuffer BUFFER;

// Stuff for key agreement....
private int CURRENT_PARTNER;
private long CURRENT_PARTNER_ID;

private BigInteger MY_PRIVATE;
private BigInteger CURRENT_PRIVATE;
private BigInteger ENCRYPTION_KEY;

private BigInteger FIRST_PK;
private BigInteger INITIAL_PK;

// Stuff for gauging key agreement times
private long START_TIME;
private double ELAPSED;

// Diffie-Hellman Parameters...
private BigInteger CURRENT_P;
private BigInteger CURRENT_G;
private java.util.Random GENERATOR;

private DiffieHellmanExchange CURRENT_EXCHANGE;
private Vector COLLECTABLES;

// --------------------------------

long lastMeaningful = 0;

long TIMEOUT = 2000;

public MessageProcessor(int mtu, boolean jumpstart, SelectionKey key, Node node){
  INITIAL_PK = null;
  IN_EXCHANGE = false;
  GENERATOR = new java.util.Random();
  MTU=mtu;
  JUMPSTART=jumpstart;
  NODE_STATE = State.INIT_ONLY;
  KEY=key;
  STATE_LOCK = new Object();

  CUBE = new Hypercube();
  Node n = new Node();
  n.id(Reactor.ID);
  CUBE.add(n);
  System.out.println(CUBE);

}

/* ------------------------------------------------------------------------------------- */
/* ------------------------------ Helper Functions ------------------------------------- */
/* ------------------------------------------------------------------------------------- */

public char getProtoState(){
  return PROTO_STATE;
}

public char getNodeState(){
```

```
  return NODE_STATE;
}

public void setProtoState(char state){
  synchronized(STATE_LOCK){
    PROTO_STATE = state;
  }
}

public void setNodeState(char state){
  synchronized(STATE_LOCK){
    NODE_STATE = state;
  }
}

public void setStates(char nstate, char pstate){
  synchronized(STATE_LOCK){
    NODE_STATE = nstate;
    PROTO_STATE = pstate;
  }
}

private void allocate(int size){
  BUFFER = ByteBuffer.allocate(size);
}

public void writeMessage(ByteBuffer b, long dest){
  byte[] a = new byte[b.limit()];
  b.position(0);
  b.get(a);
  Reactor.write(a, dest);
}

public void printDebug(String message, long src){
  if(Reactor.DEBUG >= 3){
    System.out.println(
"\n"+message+" "+src+"\n"+
"Node State:0x"+Integer.toString(NODE_STATE, 16)+"\n"+
"Proto State:0x"+Integer.toString(PROTO_STATE, 16)+"\n"
);
    }
}

public boolean timedOut(){
  boolean retval = (System.currentTimeMillis()-lastMeaningful > TIMEOUT) && TIMEOUT_MATTERS;
  return retval;
}


/**
   Computes the currently known key from the current tree iteratively...
*/
private void computeKey (BigInteger share, Vector v){
  BigInteger key  = share;
  BigInteger thisShare = share;
  for(int i = 0; i < v.size(); i++){
    key = ((BigInteger)v.elementAt(i)).modPow(thisShare, CURRENT_P);
  }
  ENCRYPTION_KEY = key;
  System.out.println("Encryption Key:"+ENCRYPTION_KEY);
}

/* ------------------------------------------------------------------------------------ */
/* ----------------------------- Message Reaction Functions ----------------------------- */
/* ------------------------------------------------------------------------------------ */

/* ************************************************************************************ */
/*                           Functions to deal with key agreement                       */
```

```
  /* ********************************************************************************** */


  /**
     Causes all "non-global-sponsor" nodes to initiate the "appropriate" action in response
     to a "KEY_UPDATE" request. i.e., if nodes have an even index in their tree positions,
     they must initiate a Diffie-Hellman key exchange with their partners.  Otherwise,
     they sit and wait for a request.
   */
  public void keyUpdate(long src, BigInteger p, BigInteger g){
    if(src == SPONSOR_ID && PROTO_STATE == State.CALM && NODE_STATE == State.JOINED_N_G_S){
      TIMEOUT_MATTERS = false;
      START_TIME = System.currentTimeMillis();

      currentDimension = new java.math.BigInteger(new Integer(CUBE.getVector().size()-1).toString()).bitLength();
      // Refresh our own share...
      CURRENT_P = p;
      CURRENT_G = g;
      MY_PRIVATE = new BigInteger(128, 100, GENERATOR);

      COLLECTABLES = new Vector();

      CURRENT_PRIVATE = MY_PRIVATE;
      INITIAL_PK = null;
      // Set Current Partner
      updatePartner();
      initDiffieHellman();
    }
  }

  /**
      Initialize a Diffie-Hellman key exchange at the current point in the tree,
      with the current key, prime, and personal share. If the node at which this
      exchange is to take place is an intermediary node, we will query it's "sponsor"
      node.
  */
  public void initDiffieHellman(){
    // What is our position in this cubic mess....
    BigInteger pos = new BigInteger(new Integer(CUBE.indexOf(Reactor.ID)).toString());
    int MAXDIM = new BigInteger(new Integer(CUBE.getVector().size()-1).toString()).bitLength();

    // If we are a not in the "outermost" dimension, we wait...
    // This way, if there are some "stragglers," (i.e., the hypercube isn't "full",) then
    // only the few stragglers wait for (dimension-1) inner node's public keys, rather
    // than all inner nodes waiting for one straggler's key...
    if(!pos.testBit(currentDimension-1)){
      // If we work in the outermost dimension, our partner may not exist. If he does not exist,
      // That is OK. Just proceed on as normal...
      if(currentDimension == MAXDIM){
if(CURRENT_PARTNER >= CUBE.getVector().size()){
  // Recurse a level.
  // It can be proven that we will simply wait,
  // but things past dimension 5-6 are impossible
  // to visualize.........make the call to
  // initDiffieHellman for safety....
  currentDimension --;
  updatePartner();
  initDiffieHellman();
}
else{
  // Proceed Forward As Normal!!!
  DiffieHellmanExchange dhe = new DiffieHellmanExchange(Reactor.ID,
CURRENT_PARTNER_ID,
CURRENT_P,
CURRENT_G,
CURRENT_PRIVATE
);
  CURRENT_EXCHANGE = dhe;
```

```
}
      }
      else{
// If we are not active the outermost dimension,
// and our dimension is not full,
// then we do nothing more now....
if(pos.testBit(MAXDIM-1)){
  // Dimension is not full
  if(CUBE.getVector().size() != (int)Math.pow(2.0, (double)MAXDIM)){
    PROTO_STATE = State.AWAITING_KEY_ARRAY;
  }
  // Otherwise, we are at a full dimension (node count is power of 2)
  else{
    // Proceed Forward As Normal!!!
    DiffieHellmanExchange dhe = new DiffieHellmanExchange(Reactor.ID,
  CURRENT_PARTNER_ID,
  CURRENT_P,
  CURRENT_G,
  CURRENT_PRIVATE
  );
    CURRENT_EXCHANGE = dhe;
  }
}
// Otherwise we were active in the outer dimension.... if we are here, we are active now....
else{
  DiffieHellmanExchange dhe = new DiffieHellmanExchange(Reactor.ID,
CURRENT_PARTNER_ID,
CURRENT_P,
CURRENT_G,
CURRENT_PRIVATE   );
  CURRENT_EXCHANGE = dhe;
}
      }

    }
  }

  /**
     Message handler for the request of a Diffie-Hellman key exchange.  From a cryptographic
     perspective, this indicates that this node is "Bob," or the non-initiating party...
   */
  public void diffieHellmanStageOne(long src, BigInteger p, BigInteger g, BigInteger pk){
    if (src == CURRENT_PARTNER_ID){
      DiffieHellmanExchange dhe = new DiffieHellmanExchange(
    src,
    Reactor.ID,
    CURRENT_P,
    CURRENT_G,
    CURRENT_PRIVATE,
    pk
    );

      CURRENT_EXCHANGE = dhe;

      if(currentDimension != new BigInteger(Integer.toString(CUBE.getVector().size()-1)).bitLength()){
COLLECTABLES.add(pk);
      }

      System.out.println(dhe.secret_key);

      // Perform the DH stage 2....
      byte[] a = dhe.public_key_bob.toByteArray();
      allocate(a.length+6);
      BUFFER.putChar(Message_OPC.DH_STAGE_2);
      BUFFER.putInt(a.length);
      BUFFER.put(a);
      writeMessage(BUFFER,src);
```

```
      //Proceed to the next dimension...
      currentDimension--;
      if(currentDimension > 0){
CURRENT_PRIVATE = dhe.secret_key;
updatePartner();
initDiffieHellman();
      }
      else{
// If I had a straggler, he needs my public key vector....
sendVector();
      }
    }
  }

  // Yep, stage 2 is back...
  public void diffieHellmanStageTwo(long src, BigInteger pk_bob){
    CURRENT_EXCHANGE.public_key_bob = pk_bob;
    CURRENT_EXCHANGE.secret_key = pk_bob.modPow(CURRENT_PRIVATE, CURRENT_P);
    System.out.println(CURRENT_EXCHANGE.secret_key);

    if(currentDimension != new BigInteger(Integer.toString(CUBE.getVector().size()-1)).bitLength()){
      COLLECTABLES.add(pk_bob);
    }

    //Proceed to the next dimension...
    currentDimension--;

    if(currentDimension > 0){
      CURRENT_PRIVATE = CURRENT_EXCHANGE.secret_key;
      updatePartner();
      initDiffieHellman();
    }
    else{
      // If I had a straggler, he needs my public key vector....
sendVector();
    }
  }

  private void sendVector(){
    // If I was active in the outer dimension and
    int pos = CUBE.indexOf(Reactor.ID);
    // The index of the last position
    int lastPos = new BigInteger(Integer.toString(CUBE.getVector().size()-1)).bitLength();
    if ( ! new BigInteger(Integer.toString(pos)).testBit(lastPos) ){
      // actually have a partner, he needs COLLECTABLES
      int hisPos =  (new BigInteger(Integer.toString(pos)).flipBit(lastPos-1)).intValue();
      if ( hisPos < CUBE.getVector().size() ){
// And the vector was not a full cube...
int cube = (int)Math.pow(2.0, (double)(lastPos));
if ( CUBE.getVector().size() != cube){
  ByteBuffer temp  = java.nio.ByteBuffer.allocate(17*(COLLECTABLES.size()+4));

  temp.putInt(COLLECTABLES.size());
  int count = 4;

  for(int i = 0; i < COLLECTABLES.size(); i++){
    BigInteger bi = (BigInteger)(COLLECTABLES.elementAt(i));
    byte[] out = bi.toByteArray();
    count += 4;
    temp.putInt(out.length);
    count += (out.length);
    temp.put(out);
  }

  byte[] t = new byte[count];
  temp.position(0);
  temp.get(t);
```

```
  allocate(count+2);
  BUFFER.putChar(Message_OPC.KEY_ARRAY);
  BUFFER.put(t);
  writeMessage(BUFFER, ((Node)CUBE.getVector().elementAt(hisPos)).id());
}
      }
    }

    ENCRYPTION_KEY = CURRENT_EXCHANGE.secret_key;
    System.out.println("Encryption Key:"+ENCRYPTION_KEY);
    PROTO_STATE = State.CALM;
  }

  private void updatePartner(){
    int myPos = CUBE.indexOf(Reactor.ID);
    int index = new java.math.BigInteger(new Integer(myPos).toString()).flipBit(currentDimension-1).intValue();
    CURRENT_PARTNER = index;
    if(CURRENT_PARTNER < CUBE.getVector().size()){
      CURRENT_PARTNER_ID = ((Node)CUBE.getVector().elementAt(index)).id();
    }
  }

  /* ##########################################################################################  */


  /* ***************************************************************************************** */
  /*                                      End key agreement                                    */
  /* ***************************************************************************************** */


  /* ***************************************************************************************** */
  /*                            Functions to delegate a global sponsor                         */
  /* ***************************************************************************************** */


  private void sponsorUpdate(){
    synchronized(STATE_LOCK){
      lastMeaningful = System.currentTimeMillis();

      // Get the sponsor
      long sponsor = CUBE.getSponsor();
      // Literally do the switch......
      SPONSOR_ID = sponsor;


      // Make a conceptual switch of necessary....
      if (NODE_STATE == State.JOINED_G_S && SPONSOR_ID != Reactor.ID){
NODE_STATE = State.JOINED_N_G_S;
PROTO_STATE = State.CALM;
      }
      else if(NODE_STATE == State.JOINED_N_G_S && SPONSOR_ID == Reactor.ID){
NODE_STATE = State.JOINED_G_S;
PROTO_STATE = State.CALM;
      }

      System.out.println("\n"+CUBE);

      if (NODE_STATE == State.JOINED_G_S){

COLLECTABLES = new Vector();

PROTO_STATE = State.UPDATING_KEY;
TIMEOUT_MATTERS = false;

// Reset the public values, p, g, and our share...
CURRENT_P = new BigInteger(128, 100, GENERATOR);
CURRENT_G = new BigInteger(CURRENT_P.bitLength(), GENERATOR);
MY_PRIVATE = new BigInteger(128, 100, GENERATOR);
```

```
// Initially, the private key is our share
CURRENT_PRIVATE = MY_PRIVATE;
// Set out current responsibility to be our sibling

// Broadcast the update request...
allocate(Reactor.MTU);
byte[] pa = CURRENT_P.toByteArray();
byte[] ga = CURRENT_G.toByteArray();
allocate(pa.length+ga.length+10);
BUFFER.putChar(Message_OPC.KEY_UPDATE);
BUFFER.putInt(pa.length);
BUFFER.put(pa);
BUFFER.putInt(ga.length);
BUFFER.put(ga);

// Start the timer to include the write to
// compensate for the missing read...
START_TIME = System.currentTimeMillis();
TIMEOUT_MATTERS = false;
writeMessage(BUFFER, Message_OPC.BROADCAST);


// Update our partner....
currentDimension = new java.math.BigInteger(new Integer(CUBE.getVector().size()-1).toString()).bitLength();
updatePartner();

// Start the melee....
initDiffieHellman();
      }
      else{
PROTO_STATE = State.CALM;
      }
    }
  }

  public void sponsor_switch(long src, long newS, Hypercube h){
    synchronized(STATE_LOCK){
      if(src == SPONSOR_ID && NODE_STATE == State.JOINED_N_G_S){
// This looks ugly????
TEMP_CUBE = h;
LAST_SOURCE = newS;
SPONSOR_ID = newS;
PROTO_STATE = State.JOIN_CLIENT_PENDING;
lastMeaningful = System.currentTimeMillis();
      }
    }
  }


  /* ********************************************************************************** */
  /*                                End global sponsor                                  */
  /* ********************************************************************************** */



  /* ********************************************************************************** */
  /*                                Protocol Function Handlers                          */
  /* ********************************************************************************** */


  public void join_probe(){
    synchronized(STATE_LOCK){
      allocate(2);
      BUFFER.putChar(Message_OPC.JOIN_REQUEST);
      writeMessage(BUFFER, Message_OPC.BROADCAST);
    }
  }
```

```
  public void merge_probe(){
    synchronized(STATE_LOCK){
      allocate(2);
      BUFFER.putChar(Message_OPC.MERGE_REQUEST);
      writeMessage(BUFFER, Message_OPC.BROADCAST);
    }
  }

 public void jumpstart(){
    synchronized(STATE_LOCK){
      if(NODE_STATE == State.INIT_ONLY){
NODE_STATE = State.JOINING;
PROTO_STATE = State.CALM;
join_probe();
KEY.attach(Reactor.READER);
KEY.interestOps(KEY.OP_READ);
lastMeaningful = System.currentTimeMillis();
      }
    }
  }

  public void join_request(long src){
    synchronized(STATE_LOCK){
      if(
 (NODE_STATE == State.JOINING || NODE_STATE == State.JOINED_G_S) &&
 PROTO_STATE==State.CALM
 ){

TIMEOUT_MATTERS=true;
PROTO_STATE = State.JOIN_SERVER;
allocate(2);
BUFFER.putChar(Message_OPC.JOIN_RESPONSE);
writeMessage(BUFFER, src);
LAST_SOURCE = src;
lastMeaningful = System.currentTimeMillis();
      }
      else if(timedOut()){
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
      }
    }
  }

  public void merge_request(long src){
    synchronized(STATE_LOCK){
      if(
 NODE_STATE == State.JOINED_G_S &&
 PROTO_STATE == State.CALM
 ){

TIMEOUT_MATTERS=true;
PROTO_STATE = State.MERGE_SERVER;
allocate(2);
BUFFER.putChar(Message_OPC.MERGE_RESPONSE);
writeMessage(BUFFER, src);
LAST_SOURCE = src;
lastMeaningful = System.currentTimeMillis();
      }
      else if(timedOut()){
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
      }
    }
  }

  public void join_response(long src){
    synchronized(STATE_LOCK){
      if(
```

```
 NODE_STATE == State.JOINING &&
 (PROTO_STATE == State.CALM)
 ){
TIMEOUT_MATTERS=true;
PROTO_STATE = State.JOIN_CLIENT;
SPONSOR_ID = src;

Node n = new Node();
n.id(Reactor.ID);
allocate(n.sizeInBytes()+2);

BUFFER.putChar(Message_OPC.JOIN_CONFIRM);
BUFFER.put(n.getBytes());

writeMessage(BUFFER, src);
LAST_SOURCE = src;
lastMeaningful = System.currentTimeMillis();
      }
      else{
allocate(2);
BUFFER.putChar(Message_OPC.JOIN_REJECT);
writeMessage(BUFFER, src);
if(timedOut()){
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS = false;
}
      }
    }
  }

  public void merge_response(long src){
    synchronized(STATE_LOCK){
      if(
 NODE_STATE == State.JOINED_G_S &&
 PROTO_STATE == State.CALM
 ){
TIMEOUT_MATTERS=true;

PROTO_STATE = State.MERGE_CLIENT;
LAST_SOURCE = src;

byte[] a = CUBE.getBytes();
allocate(a.length + 6);
BUFFER.putChar(Message_OPC.MERGE_CONFIRM);
BUFFER.putInt(a.length);
BUFFER.put(a);

writeMessage(BUFFER, src);
lastMeaningful = System.currentTimeMillis();
      }
      else{
allocate(2);
BUFFER.putChar(Message_OPC.MERGE_REJECT);
writeMessage(BUFFER, src);
if(timedOut()){
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS = false;
}
      }
    }
  }

  public void join_confirm(Node n){
    synchronized(STATE_LOCK){
      if(
 (NODE_STATE == State.JOINING || NODE_STATE == State.JOINED_G_S) &&
 PROTO_STATE == State.JOIN_SERVER
 ){
```

```
if(n.id() == LAST_SOURCE){
  // Uh???/

  TEMP_CUBE  = new Hypercube(CUBE.getBytes());
  TEMP_CUBE.add(n);
  byte[] a = TEMP_CUBE.getBytes();
  allocate(a.length + 6);
  BUFFER.putChar(Message_OPC.H_STRUCTURE);
  BUFFER.putInt(a.length);
  BUFFER.put(a);

  writeMessage(BUFFER, Message_OPC.BROADCAST);
  PROTO_STATE = State.JOIN_SERVER_PENDING;

  lastMeaningful = System.currentTimeMillis();
}
else if(timedOut()){
  PROTO_STATE = State.CALM;
}
      }
      else if(timedOut()){
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
      }
    }
  }

  public void merge_confirm(long src, Hypercube h){
    synchronized(STATE_LOCK){
      if(
 NODE_STATE == State.JOINED_G_S &&
 PROTO_STATE == State.MERGE_SERVER
 ){
if(src == LAST_SOURCE){

  // Grab the array transmitted to us and
  // add it to our own...

  // Grab the array transmitted to us and
  // add it to our own...
  TEMP_CUBE = CUBE;
  TEMP_CUBE.merge(h);
  byte[] a = TEMP_CUBE.getBytes();
  allocate(a.length + 6);
  BUFFER.putChar(Message_OPC.H_STRUCTURE);
  BUFFER.putInt(a.length);
  BUFFER.put(a);

  writeMessage(BUFFER, Message_OPC.BROADCAST);
  PROTO_STATE = State.MERGE_SERVER_PENDING;
  lastMeaningful = System.currentTimeMillis();
}
else if(timedOut()){
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS = false;
}
      }
      else if(timedOut()){
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
      }
    }
  }

  public void join_reject(long src){
    synchronized(STATE_LOCK){
      if(src == LAST_SOURCE && NODE_STATE < 3){
PROTO_STATE = State.CALM;
```

```
TIMEOUT_MATTERS = false;
lastMeaningful = System.currentTimeMillis();
      }
    }
  }

  public void merge_reject(long src){
    synchronized(STATE_LOCK){
      if(src == LAST_SOURCE && NODE_STATE == State.JOINED_G_S && PROTO_STATE != State.CALM){
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
lastMeaningful = System.currentTimeMillis();
      }
    }
  }

  public void array_update(long id, Hypercube h){
    synchronized(STATE_LOCK){
      if(
 (NODE_STATE == State.JOINING && PROTO_STATE == State.JOIN_CLIENT)
 ){
if(id == SPONSOR_ID){
  LAST_SOURCE  = id;
  PROTO_STATE = State.JOIN_CLIENT_PENDING;
  TEMP_CUBE = h;
  allocate(2);
  BUFFER.putChar(Message_OPC.H_COMMIT);
  writeMessage(BUFFER, id);
  lastMeaningful = System.currentTimeMillis();
}
else if(timedOut()){
  PROTO_STATE = State.CALM;
}
      }
      else if(NODE_STATE == State.JOINED_N_G_S && PROTO_STATE == State.CALM){
if(id == SPONSOR_ID){
  LAST_SOURCE =  id;
  PROTO_STATE = State.JOIN_CLIENT_PENDING;
  TEMP_CUBE = h;
  lastMeaningful = System.currentTimeMillis();
}
else if(timedOut()){
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS = false;
}
      }
      else if (NODE_STATE == State.JOINED_G_S && PROTO_STATE == State.MERGE_CLIENT){
if(id == LAST_SOURCE){

  LAST_SOURCE = id;
  NODE_STATE = State.JOINED_N_G_S;
  TEMP_CUBE = h;

  byte[] out = TEMP_CUBE.getBytes();
  allocate(out.length+14);

  BUFFER.putChar(Message_OPC.SPONSOR_SWITCH);
  BUFFER.putLong(LAST_SOURCE);

  BUFFER.putInt(out.length);
  BUFFER.put(out);

  writeMessage(BUFFER, Message_OPC.BROADCAST);

  // Inform the merge master to commit
  allocate(2);
  BUFFER.putChar(Message_OPC.H_COMMIT);
  writeMessage(BUFFER, id);
```

```
  lastMeaningful = System.currentTimeMillis();
  SPONSOR_ID = LAST_SOURCE;
}
else if(timedOut()){
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS = false;
}
      }
      else if(timedOut()){
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
      }
    }
  }

  /* ********************************************************************************** */
  /*                                 End Protocol Function Handlers                     */
  /* ********************************************************************************** */


  /* ********************************************************************************** */
  /*                                     Tree Update Functions                          */
  /* ********************************************************************************** */

  public void key_array(long id, Vector v){
    synchronized(STATE_LOCK){
      // Ensure the person giving us this is the node of next inner dimension...
      computeKey(CURRENT_PRIVATE, v);
    }
  }

  public void h_commit(long id){
    synchronized(STATE_LOCK){
      if(id == LAST_SOURCE){
lastMeaningful = System.currentTimeMillis();
if(
   (NODE_STATE == State.JOINING && PROTO_STATE == State.JOIN_CLIENT_PENDING) ||
   (NODE_STATE == State.JOINED_G_S &&PROTO_STATE == State.MERGE_CLIENT)
   )
  {
    CUBE = TEMP_CUBE;
    System.out.println("Hypercube Commit:"+id);
    NODE_STATE = State.JOINED_N_G_S;
    PROTO_STATE = State.CALM;
    TIMEOUT_MATTERS=false;
    sponsorUpdate();
  }

else if(
(NODE_STATE == State.JOINING || NODE_STATE == State.JOINED_G_S) &&
PROTO_STATE == State.JOIN_SERVER_PENDING
){
  CUBE = TEMP_CUBE;

  allocate(2);
  BUFFER.putChar(Message_OPC.H_COMMIT);
  writeMessage(BUFFER, Message_OPC.BROADCAST);
  NODE_STATE = State.JOINED_G_S;
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS=false;
  sponsorUpdate();
}

else if(NODE_STATE == State.JOINED_G_S && PROTO_STATE == State.MERGE_SERVER_PENDING){
  CUBE = TEMP_CUBE;

  allocate(2);
  BUFFER.putChar(Message_OPC.H_COMMIT);
```

```
  writeMessage(BUFFER, Message_OPC.BROADCAST);
  NODE_STATE = State.JOINED_G_S;
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS=false;
  sponsorUpdate();
}
else if (NODE_STATE == State.JOINED_N_G_S){
  CUBE = TEMP_CUBE;

  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS=false;
  sponsorUpdate();
}
      }
      else if(timedOut()){
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
      }
    }
  }

  /* ********************************************************************************** */
  /*                                    End Tree Functions                              */
  /* ********************************************************************************** */


}
```

5.2.43 v1/LGDH/Decoder.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
 */

import java.nio.channels.SelectionKey;
import java.nio.ByteBuffer;

public class Decoder implements Runnable{

  private SelectionKey KEY;
  private ByteBuffer MESSAGE;
  private boolean EXECUTED;

  int length;
  byte[] tree;
  NodeArray array;


  public Decoder(SelectionKey key, ByteBuffer mesg){
    KEY=key;
    EXECUTED = false;
    MESSAGE=mesg;
  }

  public void run(){
    synchronized(this){
      if(EXECUTED)return;
      else EXECUTED=true;
    }
    /* Decode a message */
    long src = MESSAGE.getLong();
    long dest = MESSAGE.getLong();
    if(src != Reactor.ID && (dest == Message_OPC.BROADCAST || dest == Reactor.ID )){
      final char type = MESSAGE.getChar();
      switch(type){
```

```
      case Message_OPC.JUMPSTART:
Reactor.PROCESSOR.jumpstart();
break;

/* ----------------- Join Messages ----------------- */
      case Message_OPC.JOIN_REQUEST:
Reactor.PROCESSOR.join_request(src);
break;

      case Message_OPC.JOIN_RESPONSE:
Reactor.PROCESSOR.join_response(src);
break;

      case Message_OPC.JOIN_CONFIRM:
byte[] rest = new byte[Reactor.NODE.sizeInBytes()];
MESSAGE.get(rest);
Node n = new Node(rest);
Reactor.PROCESSOR.join_confirm(n);
break;

      case Message_OPC.JOIN_REJECT:
Reactor.PROCESSOR.join_reject(src);
break;

/* ---------------- Merge Messages ------------------ */

      case Message_OPC.MERGE_REQUEST:
Reactor.PROCESSOR.merge_request(src);
break;

      case Message_OPC.KEY_UPDATE:
int plen = MESSAGE.getInt();
byte[] p = new byte[plen];
MESSAGE.get(p);
java.math.BigInteger prime = new java.math.BigInteger(p);

int glen = MESSAGE.getInt();
byte[] g = new byte[glen];
MESSAGE.get(g);
java.math.BigInteger generator = new java.math.BigInteger(g);

Reactor.PROCESSOR.keyUpdate(src, prime, generator);
break;

      case Message_OPC.MERGE_RESPONSE:
Reactor.PROCESSOR.merge_response(src);
break;

      case Message_OPC.MERGE_CONFIRM:
int length = MESSAGE.getInt();
byte[] array = new byte[length];
MESSAGE.get(array);
NodeArray a = new NodeArray(array);
Reactor.PROCESSOR.merge_confirm(src, a);
break;

      case Message_OPC.DH_STAGE_1:

length = MESSAGE.getInt();
byte[] pa = new byte[length];
MESSAGE.get(pa);
prime = new java.math.BigInteger(pa);

length = MESSAGE.getInt();
byte [] ga = new byte[length];
MESSAGE.get(ga);
generator = new java.math.BigInteger(ga);
```

```
length = MESSAGE.getInt();
byte[] pka = new byte[length];
MESSAGE.get(pka);
java.math.BigInteger public_key = new java.math.BigInteger(pka);

length = MESSAGE.getInt();
byte[] vec = new byte[length];
MESSAGE.get(vec);
NodeArray noda = new NodeArray(vec);

Reactor.PROCESSOR.diffieHellmanStageOne(src, prime, generator, public_key, noda);
break;

      case Message_OPC.MERGE_REJECT:
Reactor.PROCESSOR.merge_reject(src);
break;

      case Message_OPC.SPONSOR_SWITCH:
long newS = MESSAGE.getLong();

int len = MESSAGE.getInt();
byte[] a4 = new byte[len];
MESSAGE.get(a4);

Reactor.PROCESSOR.sponsor_switch(src, newS, new NodeArray(a4));
break;


/* ------------------------------------------------ */

      case Message_OPC.A_COMMIT:
Reactor.PROCESSOR.a_commit(src);
break;

      case Message_OPC.A_STRUCTURE:
length = 0;
try{
  int l = MESSAGE.getInt();
  byte[] ab = new byte[l];
  MESSAGE.get(ab);
  NodeArray ar = new NodeArray(ab);
  Reactor.PROCESSOR.array_update(src, ar);
}
catch(Exception e){
  if (e instanceof java.nio.BufferUnderflowException){
    System.out.println("Needed:"+length+"\nAvailable:"+(MESSAGE.limit()-MESSAGE.position()));
    System.out.println("Network MTU Exceeded By Group Structure Transmission!\n"+
       "Fragmentation Not Implemented Yet!"
       );
    System.exit(-1);
  }
}
break;

      case Message_OPC.KEY_ARRAY:
length = 0;
try{
  length = MESSAGE.getInt();
  array = new byte[length];
  MESSAGE.get(array);
  Reactor.PROCESSOR.key_array(new NodeArray(array), src);
}
catch(Exception e){
  if (e instanceof java.nio.BufferUnderflowException){
    System.out.println("Needed:"+length+"\nAvailable:"+(MESSAGE.limit()-MESSAGE.position()));
    System.out.println("Network MTU Exceeded By Group Structure Transmission!\n"+
       "Fragmentation Not Implemented Yet!"
       );
```

```
    System.exit(-1);
  }
}
break;
      }
    }

  }

}
```

5.2.44 v1/LGDH/Device.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
 */

/**
   The main executable class for Kieran S. Hagzan's master's thesis work at R.I.T. .
   This class parses command-line arguments, checks their validity, creates a new
   <code>Reactor</code>and displays usage information if necessary.
 */
public class Device{

  /**
     A newly created <code>Reactor</code> to be run.
   */
  private final Reactor REACTOR;

  /**
      Creates a new <code>Reactor</code> on the given port
      using the specified network MTU.
      @param port The port to bind the <code>Reactor</code> on, (1024 <= port <= 65535).
      @param mtu The Maximum Transmission Unit of the <code>Reactor</code> (1 <= mtu <= 65535).
   */
  private Device(int port, int mtu, boolean jumpstart, int maxNodes){
    REACTOR = new Reactor(port, mtu, jumpstart, maxNodes);
    REACTOR.setDebugLevel(2);
    REACTOR.run();
  }

  /**
     The main method invoked by the VM at program call.
     @param args The command-line arguments passed to this class.
   */
  public static void main(String[] args){
    boolean jumpstart;
    int mtu;
    int port;
    int max;

    // Bomb out if the user cannot type "--help"
    if (args.length > 4)exitWithUsage();
    // grab the outermost argument - the jumpstart mode if present
    if(args.length == 4){
      String s = args[2];

      if ((s.indexOf("true") < 0) && (s.indexOf("false") < 0)){
exitWithUsage();
      }

      if(s.indexOf("true") >= 0){
jumpstart = true;
      }
      else{
jumpstart = false;
```

```
      }
    }
    else{
      jumpstart = false;
    }

    // Grab the first argument - the port if present
    if(args.length == 4){
      if(!java.util.regex.Pattern.matches("[0-9]+", args[3])){
exitWithUsage();
      }
      max = Integer.parseInt(args[3]);
    }
    else{
      max = 100;
    }

    // Grab the secondmost argument - the MTU if present
    if(args.length >= 2){
      if(!java.util.regex.Pattern.matches("[0-9]+", args[1])){
exitWithUsage();
      }
      mtu = Integer.parseInt(args[1]);
      if (mtu < 33 || mtu > 65535){exitWithUsage();}
    }
    else{
      mtu = 512;
    }

    // Grab the first argument - the port if present
    if(args.length >= 1){
      if(!java.util.regex.Pattern.matches("[0-9]+", args[0])){
exitWithUsage();
      }
      port = Integer.parseInt(args[0]);
      if(port < 1024 || port > 65535){exitWithUsage();}
    }
    else{
      port = 12333;
    }

    // Let them know what they have done....
    //    System.out.println("Initializing on port "+port+" using MTU of "+mtu+
    //        (jumpstart?" with":" without")+" jumpstart mode...");

    new Device(port, mtu, jumpstart, max);
  }

  /**
     Displays proper usage of this class.
   */
  private static void exitWithUsage(){
    System.out.println("\nUsage:\n------\n"+
       "\t\"java Device --help\" will display this screen\n"+
       "\t\"java Device <port> <mtu> <jumpstart> \"\n\nWhere:\n------\n"+
       "<port> - is an optionally specified number between 1024 and 65535 (defaults to 12333)\n"+
       "<mtu>  - is an optionally specified number between 33 and 65535 (defaults to 512)\n"+
       "<jumpstart> - is one of \"true\" or \"false\" indicating the jumpstart status of the node\n"
       );
    System.exit(1);
  }
}
```

5.2.45 v1/LGDH/DiffieHellmanExchange.java

```
import java.math.*;
```

```
import java.util.Random;
import java.nio.ByteBuffer;

public class DiffieHellmanExchange{

  public BigInteger p;
  public BigInteger g;
  public BigInteger x;
  public BigInteger public_key_alice;
  public BigInteger public_key_bob;
  public BigInteger secret_key;

  public long ALICE;
  public long BOB;

  // Constructor for bob...
  public DiffieHellmanExchange(
       long alice,
       long bob,
       BigInteger prime,
       BigInteger generator,
       BigInteger share,
       BigInteger alices_public_key
       ){
    ALICE=alice;
    BOB=bob;
    p = prime;
    g = generator;
    public_key_alice = alices_public_key;

    Random rand = new Random();

    x = share;

    // And public key....
    public_key_alice = alices_public_key;
    public_key_bob = g.modPow(x,p);
    secret_key = alices_public_key.modPow(x, p);

  }



  // Constructor for alice
  public DiffieHellmanExchange(long alice,
       long bob,
       BigInteger prime,
       BigInteger generator,
       BigInteger share,
       java.util.Vector array,
       BigInteger firstPK
       ){
    ALICE=alice;
    BOB=bob;

    Random rand = new Random();

    // Generate a private key....
    x = share;
    g = generator;
    p = prime;

    // And public key....
    public_key_alice = generator.modPow(x,prime);

    // Now, things are a little different than in TGDH

    byte[] po = prime.toByteArray();
```

```
    byte[] go = generator.toByteArray();

    ByteBuffer bb = ByteBuffer.allocate(Reactor.MTU);
    bb.putChar(Message_OPC.DH_STAGE_1);

    bb.putInt(po.length);
    bb.put(po);

    bb.putInt(go.length);
    bb.put(go);

    byte[] pko = public_key_alice.toByteArray();
    bb.putInt(pko.length);
    bb.put(pko);

    // Now also put the node vector including me....
    Node me = new Node();
    me.id(Reactor.ID);
    if(firstPK == null){
      me.PUBLIC_KEY = public_key_alice;
    }
    else{
      me.PUBLIC_KEY = firstPK;
    }
    array.add(me);

    int len = 0;
    ByteBuffer vec = ByteBuffer.allocate(Reactor.MTU);
    for(int i = 0; i < array.size(); i++){
      Node n = (Node)(array.elementAt(i));
      len += n.sizeInBytes()+4;
      vec.putInt(n.sizeInBytes());
      vec.put(n.getBytes());
    }

    vec.position(0);
    byte[] v = new byte[len];
    vec.get(v);
    bb.putInt(v.length);
    bb.put(v);

    byte[] out = new byte[bb.position()];
    bb.position(0);
    bb.get(out);

    Reactor.write(out, bob);
  }



}
```

5.2.46 v1/LGDH/Discoverer.java

```
import java.nio.channels.SelectionKey;
import java.nio.ByteBuffer;

public class Discoverer implements Runnable{

  private final SelectionKey KEY;
  private boolean EXECUTED;

  public Discoverer(SelectionKey key){
    KEY=key;
    EXECUTED=false;
  }
```

```
  public void run(){
    synchronized(this){
      if(EXECUTED)return;
      EXECUTED=true;
    }
    ByteBuffer bb = ByteBuffer.allocate(2);
    bb.putChar(Message_OPC.JOIN_REQUEST);
    bb.position(0);
    byte[] a = new byte[bb.limit()];
    bb.get(a);
    Reactor.write(a, Message_OPC.BROADCAST);
    KEY.attach(Reactor.READER);
    KEY.interestOps(KEY.OP_READ);
  }

}
```

5.2.47 v1/LGDH/Jumpstarter.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
 */

public class Jumpstarter{
  public static void main(String[] args){
boolean local;
if(args.length > 0 && args[0].indexOf("local") >= 0){
  local = true;
}
else local=false;
    try{
      java.nio.channels.DatagramChannel c = java.nio.channels.DatagramChannel.open();
      c.socket().setReuseAddress(true);
      c.socket().setBroadcast(true);
      c.socket().bind(new java.net.InetSocketAddress("0.0.0.0", 12333));
      java.nio.ByteBuffer out = java.nio.ByteBuffer.allocateDirect(18);
      out.putLong(0xdeadbeefl);
      out.putLong(0xffffffffffffffffl);
      out.putChar(Message_OPC.JUMPSTART);
      for(int i = 0; i < 3; i++){
out.position(0);
java.net.InetSocketAddress outA;
if(local){
  System.out.println("Yelling locally...");
  outA = new java.net.InetSocketAddress("localhost", 12333);
}
else{
    outA = new java.net.InetSocketAddress("255.255.255.255", 12333);
}
      c.send(out, outA);
      }
      c.close();
    }
    catch(Exception e){
      e.printStackTrace();
      System.exit(-1);
    }
  }
}
```

5.2.48 v1/LGDH/Merger.java

```
import java.nio.channels.SelectionKey;
import java.nio.ByteBuffer;

public class Merger implements Runnable{

  private final SelectionKey KEY;
  private boolean EXECUTED;

  public Merger(SelectionKey key){
    KEY=key;
    EXECUTED=false;
  }

  public void run(){
    synchronized(this){
      if(EXECUTED)return;
      EXECUTED=true;
    }
    ByteBuffer bb = ByteBuffer.allocate(2);
    System.out.println("Requesting Merge...");
    bb.putChar(Message_OPC.MERGE_REQUEST);
    bb.position(0);
    byte[] a = new byte[bb.limit()];
    bb.get(a);
    Reactor.write(a, Message_OPC.BROADCAST);
    KEY.attach(new Reader(KEY));
    KEY.interestOps(KEY.OP_READ);
  }

}
```

5.2.49 v1/LGDH/Message_OPC.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
 */

public class Message_OPC{

  public static final long BROADCAST = 0xffffffffffffffffl;

  public static final char JUMPSTART =0xffff;

  public static final char JOIN_REQUEST =0x0001;
  public static final char JOIN_RESPONSE =0x0002;
  public static final char JOIN_CONFIRM =0x0003;
  public static final char JOIN_REJECT =0x0004;


  public static final char A_COMMIT =0x0005;
  public static final char A_STRUCTURE = 0x0006;

  public static final char MERGE_REQUEST = 0x0007;
  public static final char MERGE_RESPONSE = 0x0008;
  public static final char MERGE_CONFIRM =0x0009;
  public static final char MERGE_REJECT =0x000a;

  public static final char SPONSOR_SWITCH =0x000b;

  public static final char DH_STAGE_1 = 0x000c;
  public static final char DH_STAGE_2 = 0x000d;
  public static final char KEY_UPDATE = 0x000e;
  public static final char DH_WAIT = 0x000f;
  public static final char KEY_ARRAY = 0x0010;

}
```

5.2.50 v1/LGDH/MessageProcessor.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
*/

import java.nio.channels.SelectionKey;
import java.nio.ByteBuffer;
import java.math.BigInteger;

public class MessageProcessor{


  private final int MTU;
  public boolean JUMPSTART;
  private final SelectionKey KEY;

  private final Object STATE_LOCK;

  private boolean TIMEOUT_MATTERS;

  private NodeArray ARRAY;
  private NodeArray TEMP_ARRAY;
  private NodeArray CURRENT_ARRAY;

  private long LAST_SOURCE;

  private long SPONSOR_ID;
  private long NEW_SPONSOR;

  public char NODE_STATE;
  public char PROTO_STATE;

  private ByteBuffer BUFFER;

  private DiffieHellmanExchange CURRENT_EXCHANGE;

  // Stuff for key agreement....
  private int CURRENT_PARTNER;
  private long CURRENT_PARTNER_ID;

  private BigInteger MY_PRIVATE;
  private BigInteger CURRENT_PRIVATE;
  private BigInteger ENCRYPTION_KEY;

  private BigInteger FIRST_PK;
  private BigInteger INITIAL_PK;

  // Stuff for gauging key agreement times
  private long START_TIME;
  private double ELAPSED;

  // Diffie-Hellman Parameters...
  private BigInteger CURRENT_P;
  private BigInteger CURRENT_G;
  private java.util.Random GENERATOR;
  // --------------------------------

  long lastMeaningful = 0;

  long TIMEOUT = 2000;

  public MessageProcessor(int mtu, boolean jumpstart, SelectionKey key, Node node){
    INITIAL_PK = null;
    GENERATOR = new java.util.Random();
    MTU=mtu;
    JUMPSTART=jumpstart;
```

```
    NODE_STATE = State.INIT_ONLY;
    KEY=key;

    STATE_LOCK = new Object();

    ARRAY = new NodeArray();
    ARRAY.add(node);

  }

  /* ------------------------------------------------------------------------------------- */
  /* ----------------------------- Helper Functions ------------------------------------- */
  /* ------------------------------------------------------------------------------------- */

  public char getProtoState(){
    return PROTO_STATE;
  }

  public char getNodeState(){
    return NODE_STATE;
  }

  public void setProtoState(char state){
    synchronized(STATE_LOCK){
      PROTO_STATE = state;
    }
  }

  public void setNodeState(char state){
    synchronized(STATE_LOCK){
      NODE_STATE = state;
    }
  }

  public void setStates(char nstate, char pstate){
    synchronized(STATE_LOCK){
      NODE_STATE = nstate;
      PROTO_STATE = pstate;
    }
  }

  private void allocate(int size){
    BUFFER = ByteBuffer.allocate(size);
  }

  public void writeMessage(ByteBuffer b, long dest){
    byte[] a = new byte[b.limit()];
    b.position(0);
    b.get(a);
    Reactor.write(a, dest);
  }

  public void printDebug(String message, long src){
    if(Reactor.DEBUG >= 3){
      System.out.println(
"\n"+message+" "+src+"\n"+
"Node State:0x"+Integer.toString(NODE_STATE, 16)+"\n"+
"Proto State:0x"+Integer.toString(PROTO_STATE, 16)+"\n"
);
    }
  }

  public boolean timedOut(){
    boolean retval = (System.currentTimeMillis()-lastMeaningful > TIMEOUT) && TIMEOUT_MATTERS;
    return retval;
  }
```

```
  /**
     Computes the currently known key from the current tree iteratively...
  */
  private void computeKey (BigInteger share){
    int myIndex = ARRAY.indexOf(Reactor.ID);
    java.util.Vector nodes = ARRAY.getVector();
    int len = nodes.size();
    BigInteger currentShare = share;

    // Initially, if our index is not zero, we need to do an
    // exchange with the person to our left
    int i = (myIndex != 0)?(myIndex - 1):(myIndex + 1);

    for(; i < len ;i++){
      if(i != myIndex){
Node n = ((Node)(nodes.elementAt(i)));
BigInteger pk;
if(i < myIndex){
  pk = FIRST_PK;
}
else{
  pk = n.PUBLIC_KEY;
}
currentShare = pk.modPow(currentShare, CURRENT_P);
      }
    }

    ENCRYPTION_KEY = currentShare;

    ELAPSED = (double)(System.currentTimeMillis() - START_TIME);
    ELAPSED = ELAPSED / 1000.0;

    System.out.println(ARRAY.getVector().size()+"\t"+ELAPSED);

    PROTO_STATE = State.CALM;

    INITIAL_PK = null;
    FIRST_PK = null;

    if(Reactor.MAX_NODES == ARRAY.getVector().size()){System.exit(0);}
  }

  /* ------------------------------------------------------------------------------------ */
  /* ----------------------------- Message Reaction Functions ---------------------------- */
  /* ------------------------------------------------------------------------------------ */

  /* ************************************************************************************ */
  /*                            Functions to deal with key agreement                      */
  /* ************************************************************************************ */


  /**
     Causes all "non-global-sponsor" nodes to initiate the "appropriate" action in response
     to a "KEY_UPDATE" request. i.e., if nodes have an even index in their tree positions,
     they must initiate a Diffie-Hellman key exchange with their partners.  Otherwise,
     they sit and wait for a request.
   */
  public void keyUpdate(long src, BigInteger p, BigInteger g){
    if(src == SPONSOR_ID && PROTO_STATE == State.CALM && NODE_STATE == State.JOINED_N_G_S){

      CURRENT_ARRAY = new NodeArray();
      TIMEOUT_MATTERS = false;
      START_TIME = System.currentTimeMillis();

      // Refresh our own share...

      CURRENT_P = p;
      CURRENT_G = g;
```

```
      MY_PRIVATE = new BigInteger(128, 100, GENERATOR);

      CURRENT_PRIVATE = MY_PRIVATE;

      INITIAL_PK = null;

// Initially, all nodes but the first node (the sponsor) have partners being their left sibling
      if(ARRAY.indexOf(Reactor.ID) != 0){
CURRENT_PARTNER = (ARRAY.indexOf(Reactor.ID))-1;
CURRENT_PARTNER_ID = ((Node)(ARRAY.getVector().elementAt(CURRENT_PARTNER))).id();
PROTO_STATE = State.UPDATING_KEY;
      }
    }
  }

  /**
      Initialize a Diffie-Hellman key exchange at the current point in the tree,
      with the current key, prime, and personal share. If the node at which this
      exchange is to take place is an intermediary node, we will query it's "sponsor"
      node.
  */
  public void initDiffieHellman(){
    // I begin by sending a Diffie-Hellman stage 1
    // Now I include my vector up to my position...

    java.util.Vector out = new java.util.Vector();
    if(CURRENT_ARRAY == null)CURRENT_ARRAY = new NodeArray();
    java.util.Vector orig = CURRENT_ARRAY.getVector();

    for(int i = 0; i < CURRENT_ARRAY.indexOf(Reactor.ID); i++){
      out.add(orig.elementAt(i));
    }

    // I am alice....
    DiffieHellmanExchange dhe = new DiffieHellmanExchange(
  Reactor.ID,
  CURRENT_PARTNER_ID,
  CURRENT_P,
  CURRENT_G,
  CURRENT_PRIVATE,
  out,
  INITIAL_PK
  );

    // If we are the seroth node, our initial PK is the one that occurs now
    // for alice....
    if(ARRAY.indexOf(Reactor.ID) == 0){
      INITIAL_PK = dhe.public_key_alice;
    }

    CURRENT_PRIVATE = dhe.secret_key;
    // If you initialized an exchange, you won't be the last to broadcast the array,
    // so wait for it...
    PROTO_STATE = State.AWAITING_KEY_ARRAY;
  }

  /**
     Message handler for the request of a Diffie-Hellman key exchange.  From a cryptographic
     perspective, this indicates that this node is "Bob," or the non-initiating party...
   */
  public void diffieHellmanStageOne(long src, BigInteger p, BigInteger g, BigInteger pk, NodeArray precursor){
    // We need to keep this public key around for later.....

    FIRST_PK = pk;

    if(src == CURRENT_PARTNER_ID && PROTO_STATE == State.UPDATING_KEY){
```

```
      // I am bob...
      DiffieHellmanExchange dhe = new DiffieHellmanExchange(src, Reactor.ID, p, g, CURRENT_PRIVATE, pk);

      // We were waiting for the node before us
      // The node at us is us
      // So 2 nodes past our current partner is where we head....

Node me = new Node();
me.id(Reactor.ID);

// The zeroth node never sees this. so, bob's public key is everyone else's initial...
me.PUBLIC_KEY = dhe.public_key_bob;
INITIAL_PK = me.PUBLIC_KEY;

java.util.Vector v = precursor.getVector();
v.add(me);
NodeArray a  = new NodeArray(v);
CURRENT_ARRAY = a;

      // So, now we use iteration, rather than recursion as in TGDH
      if(ARRAY.indexOf(Reactor.ID) == ARRAY.getVector().size()-1){

byte[] out = a.getBytes();
allocate(out.length + 6);
BUFFER.putChar(Message_OPC.KEY_ARRAY);
BUFFER.putInt(out.length);
BUFFER.put(out);

writeMessage(BUFFER, Message_OPC.BROADCAST);
ARRAY=a;
computeKey(MY_PRIVATE);
      }
      else{
CURRENT_PARTNER+=2;
CURRENT_PARTNER_ID = ((Node)(ARRAY.getVector().elementAt(CURRENT_PARTNER))).id();
CURRENT_PRIVATE = dhe.secret_key;
initDiffieHellman();
      }
    }
  }

  /* ############################################################################################  */


  /* ********************************************************************************************* */
  /*                                    End key agreement                                          */
  /* ********************************************************************************************* */


  /* ********************************************************************************************* */
  /*                           Functions to delegate a global sponsor                              */
  /* ********************************************************************************************* */


  private void sponsorUpdate(){
    synchronized(STATE_LOCK){

      long sponsor = ARRAY.getSponsor();

      // Literally do the switch......
      lastMeaningful = System.currentTimeMillis();
      SPONSOR_ID = sponsor;

      // Make a conceptual switch of necessary....
      if (NODE_STATE == State.JOINED_G_S && SPONSOR_ID != Reactor.ID){
NODE_STATE = State.JOINED_N_G_S;
PROTO_STATE = State.CALM;
      }
```

```
      else if(NODE_STATE == State.JOINED_N_G_S && SPONSOR_ID == Reactor.ID){
NODE_STATE = State.JOINED_G_S;
PROTO_STATE = State.CALM;
      }

      // Take the appropriate action
      if (NODE_STATE == State.JOINED_G_S){

PROTO_STATE = State.UPDATING_KEY;
TIMEOUT_MATTERS = false;

// Reset the public values, p, g, and our share...
CURRENT_P = new BigInteger(128, 100, GENERATOR);
CURRENT_G = new BigInteger(CURRENT_P.bitLength(), GENERATOR);
MY_PRIVATE = new BigInteger(128, 100, GENERATOR);
// Initially, the private key is our share
CURRENT_PRIVATE = MY_PRIVATE;
// Set out current responsibility to be our sibling

// Broadcast the update request...
allocate(Reactor.MTU);
byte[] pa = CURRENT_P.toByteArray();
byte[] ga = CURRENT_G.toByteArray();
allocate(pa.length+ga.length+10);
BUFFER.putChar(Message_OPC.KEY_UPDATE);
BUFFER.putInt(pa.length);
BUFFER.put(pa);
BUFFER.putInt(ga.length);
BUFFER.put(ga);

// Start the timer to include the write to
// compensate for the missing read...
START_TIME = System.currentTimeMillis();
CURRENT_ARRAY = new NodeArray();
TIMEOUT_MATTERS = false;

CURRENT_PARTNER = 1;
CURRENT_PARTNER_ID = ((Node)(ARRAY.getVector().elementAt(1))).id();

writeMessage(BUFFER, Message_OPC.BROADCAST);

initDiffieHellman();
      }
      else{
PROTO_STATE = State.CALM;
      }
    }
  }

  public void sponsor_switch(long src, long newS, NodeArray na){
    synchronized(STATE_LOCK){
      if(src == SPONSOR_ID && NODE_STATE == State.JOINED_N_G_S){
// This looks ugly????
TEMP_ARRAY = na;
LAST_SOURCE = newS;
SPONSOR_ID = newS;
PROTO_STATE = State.JOIN_CLIENT_PENDING;
lastMeaningful = System.currentTimeMillis();
      }
    }
  }


  /* ************************************************************************************ */
  /*                                End global sponsor                                    */
  /* ************************************************************************************ */
```

```
  /* ************************************************************************************ */
  /*                                Protocol Function Handlers                            */
  /* ************************************************************************************ */


  public void join_probe(){
    synchronized(STATE_LOCK){
      allocate(2);
      BUFFER.putChar(Message_OPC.JOIN_REQUEST);
      writeMessage(BUFFER, Message_OPC.BROADCAST);
    }
  }

  public void merge_probe(){
    synchronized(STATE_LOCK){
      allocate(2);
      BUFFER.putChar(Message_OPC.MERGE_REQUEST);
      writeMessage(BUFFER, Message_OPC.BROADCAST);
    }
  }

 public void jumpstart(){
    synchronized(STATE_LOCK){
      if(NODE_STATE == State.INIT_ONLY){
NODE_STATE = State.JOINING;
PROTO_STATE = State.CALM;
join_probe();
KEY.attach(Reactor.READER);
KEY.interestOps(KEY.OP_READ);
lastMeaningful = System.currentTimeMillis();
      }
    }
  }

  public void join_request(long src){
    synchronized(STATE_LOCK){
      if(
 (NODE_STATE == State.JOINING || NODE_STATE == State.JOINED_G_S) &&
 PROTO_STATE==State.CALM
 ){
TIMEOUT_MATTERS=true;
PROTO_STATE = State.JOIN_SERVER;
allocate(2);
BUFFER.putChar(Message_OPC.JOIN_RESPONSE);
writeMessage(BUFFER, src);
LAST_SOURCE = src;
lastMeaningful = System.currentTimeMillis();

      }
      else if(timedOut()){
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
join_request(src);
      }
    }
  }

  public void merge_request(long src){
    synchronized(STATE_LOCK){
      if(
 NODE_STATE == State.JOINED_G_S &&
 PROTO_STATE == State.CALM
 ){

TIMEOUT_MATTERS=true;
PROTO_STATE = State.MERGE_SERVER;
allocate(2);
```

```
BUFFER.putChar(Message_OPC.MERGE_RESPONSE);
writeMessage(BUFFER, src);
LAST_SOURCE = src;
lastMeaningful = System.currentTimeMillis();

      }
      else if(timedOut()){
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
merge_request(src);
      }
    }
  }

  public void join_response(long src){
    synchronized(STATE_LOCK){
      if(
 NODE_STATE == State.JOINING &&
 (PROTO_STATE == State.CALM)
 ){
TIMEOUT_MATTERS=true;
PROTO_STATE = State.JOIN_CLIENT;
SPONSOR_ID = src;

Node n = new Node();
n.id(Reactor.ID);
allocate(n.sizeInBytes()+2);

BUFFER.putChar(Message_OPC.JOIN_CONFIRM);
BUFFER.put(n.getBytes());

writeMessage(BUFFER, src);
LAST_SOURCE = src;
lastMeaningful = System.currentTimeMillis();
      }
      else{
allocate(2);
BUFFER.putChar(Message_OPC.JOIN_REJECT);
writeMessage(BUFFER, src);
if(timedOut()){
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS = false;
  join_response(src);
}
      }
    }
  }

  public void merge_response(long src){
    synchronized(STATE_LOCK){
      if(
 NODE_STATE == State.JOINED_G_S &&
 PROTO_STATE == State.CALM
 ){
TIMEOUT_MATTERS=true;

PROTO_STATE = State.MERGE_CLIENT;
LAST_SOURCE = src;
byte[] a = ARRAY.getBytes();
allocate(a.length + 6);
BUFFER.putChar(Message_OPC.MERGE_CONFIRM);
BUFFER.putInt(a.length);
BUFFER.put(a);
writeMessage(BUFFER, src);
lastMeaningful = System.currentTimeMillis();

      }
      else{
```

```
allocate(2);
BUFFER.putChar(Message_OPC.MERGE_REJECT);
writeMessage(BUFFER, src);
if(timedOut()){
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS = false;
  merge_response(src);
}
      }
    }
  }

  public void join_confirm(Node n){
    synchronized(STATE_LOCK){
      if(
 (NODE_STATE == State.JOINING || NODE_STATE == State.JOINED_G_S) &&
 PROTO_STATE == State.JOIN_SERVER
 ){
if(n.id() == LAST_SOURCE){
  TEMP_ARRAY  = new NodeArray(ARRAY.getBytes());
  TEMP_ARRAY.add(n);
  byte[] a = TEMP_ARRAY.getBytes();
  allocate(a.length + 6);
  BUFFER.putChar(Message_OPC.A_STRUCTURE);
  BUFFER.putInt(a.length);
  BUFFER.put(a);

  writeMessage(BUFFER, Message_OPC.BROADCAST);
  PROTO_STATE = State.JOIN_SERVER_PENDING;

  lastMeaningful = System.currentTimeMillis();
}
else if(timedOut()){
  PROTO_STATE = State.CALM;
}
      }
      else if(timedOut()){
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
      }
    }
  }

  public void merge_confirm(long src, NodeArray na){
    synchronized(STATE_LOCK){
      if(
 NODE_STATE == State.JOINED_G_S &&
 PROTO_STATE == State.MERGE_SERVER
 ){
if(src == LAST_SOURCE){

  // Grab the array transmitted to us and
  // add it to our own...
  TEMP_ARRAY = ARRAY;
  TEMP_ARRAY.merge(na);
  byte[] a = TEMP_ARRAY.getBytes();
  allocate(a.length + 6);
  BUFFER.putChar(Message_OPC.A_STRUCTURE);
  BUFFER.putInt(a.length);
  BUFFER.put(a);

  writeMessage(BUFFER, Message_OPC.BROADCAST);
  PROTO_STATE = State.MERGE_SERVER_PENDING;
  lastMeaningful = System.currentTimeMillis();
}
else if(timedOut()){
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS = false;
```

```
}
      }
      else if(timedOut()){
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
      }
    }
  }

  public void join_reject(long src){
    synchronized(STATE_LOCK){
      if(src == LAST_SOURCE && NODE_STATE < 3){
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
lastMeaningful = System.currentTimeMillis();
      }
    }
  }

  public void merge_reject(long src){
    synchronized(STATE_LOCK){
      if(src == LAST_SOURCE && NODE_STATE == State.JOINED_G_S && PROTO_STATE != State.CALM){
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
lastMeaningful = System.currentTimeMillis();
      }
    }
  }

  public void array_update(long id, NodeArray na){
    synchronized(STATE_LOCK){
      if(
 (NODE_STATE == State.JOINING && PROTO_STATE == State.JOIN_CLIENT)
 ){
if(id == SPONSOR_ID){
  LAST_SOURCE  = id;
  PROTO_STATE = State.JOIN_CLIENT_PENDING;
  TEMP_ARRAY = na;
  allocate(2);
  BUFFER.putChar(Message_OPC.A_COMMIT);
  writeMessage(BUFFER, id);
  lastMeaningful = System.currentTimeMillis();
}
else if(timedOut()){
  PROTO_STATE = State.CALM;
}
      }
      else if(NODE_STATE == State.JOINED_N_G_S && PROTO_STATE == State.CALM){
if(id == SPONSOR_ID){
  LAST_SOURCE =  id;
  PROTO_STATE = State.JOIN_CLIENT_PENDING;
  TEMP_ARRAY = na;
  lastMeaningful = System.currentTimeMillis();
}
else if(timedOut()){
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS = false;
}
      }
      else if (NODE_STATE == State.JOINED_G_S && PROTO_STATE == State.MERGE_CLIENT){
if(id == LAST_SOURCE){
  TEMP_ARRAY = na;
  LAST_SOURCE = id;
  NODE_STATE = State.JOINED_N_G_S;

  byte[] out = TEMP_ARRAY.getBytes();
  allocate(out.length+14);
```

```
  BUFFER.putChar(Message_OPC.SPONSOR_SWITCH);
  BUFFER.putLong(LAST_SOURCE);

  BUFFER.putInt(out.length);
  BUFFER.put(out);

  writeMessage(BUFFER, Message_OPC.BROADCAST);

  // Inform the merge master to commit
  allocate(2);
  BUFFER.putChar(Message_OPC.A_COMMIT);
  writeMessage(BUFFER, id);
  lastMeaningful = System.currentTimeMillis();
  SPONSOR_ID = LAST_SOURCE;
}
else if(timedOut()){
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS = false;
}
      }
      else if(timedOut()){
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
      }
    }
  }

  /* ********************************************************************************** */
  /*                                End Protocol Function Handlers                      */
  /* ********************************************************************************** */


  /* ********************************************************************************** */
  /*                                     Tree Update Functions                          */
  /* ********************************************************************************** */

  public void key_array(NodeArray n, long id){
    synchronized(STATE_LOCK){
      java.util.Vector v = ARRAY.getVector();
      long last = ((Node)(v.elementAt(v.size()-1))).id();
      if(
 (PROTO_STATE == State.AWAITING_KEY_ARRAY)&&(id == last)
 ){
ARRAY = n;
computeKey(MY_PRIVATE);

PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
lastMeaningful = System.currentTimeMillis();
      }
      else if(timedOut()){
TIMEOUT_MATTERS = false;
lastMeaningful = System.currentTimeMillis();
      }
    }
  }

  public void a_commit(long id){
    synchronized(STATE_LOCK){
      if(id == LAST_SOURCE){
lastMeaningful = System.currentTimeMillis();
if(
   (NODE_STATE == State.JOINING && PROTO_STATE == State.JOIN_CLIENT_PENDING) ||
   (NODE_STATE == State.JOINED_G_S &&PROTO_STATE == State.MERGE_CLIENT)
   )
  {
    ARRAY = TEMP_ARRAY;
    NODE_STATE = State.JOINED_N_G_S;
```

```
    PROTO_STATE = State.CALM;
    TIMEOUT_MATTERS=false;
    sponsorUpdate();
  }

else if(
(NODE_STATE == State.JOINING || NODE_STATE == State.JOINED_G_S) &&
PROTO_STATE == State.JOIN_SERVER_PENDING
){
  allocate(2);
  ARRAY = TEMP_ARRAY;
  BUFFER.putChar(Message_OPC.A_COMMIT);
  writeMessage(BUFFER, Message_OPC.BROADCAST);
  NODE_STATE = State.JOINED_G_S;
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS=false;
  sponsorUpdate();
}

else if(NODE_STATE == State.JOINED_G_S && PROTO_STATE == State.MERGE_SERVER_PENDING){
  ARRAY = TEMP_ARRAY;
  allocate(2);
  BUFFER.putChar(Message_OPC.A_COMMIT);
  writeMessage(BUFFER, Message_OPC.BROADCAST);
  NODE_STATE = State.JOINED_G_S;
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS=false;
  sponsorUpdate();
}
else if (NODE_STATE == State.JOINED_N_G_S){
  ARRAY = TEMP_ARRAY;
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS=false;
  sponsorUpdate();
}
      }
      else if(timedOut()){
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
      }
    }
  }

  /* ********************************************************************************** */
  /*                                    End Tree Functions                              */
  /* ********************************************************************************** */


}
```

5.2.51 v1/LGDH/Node.java

```
import java.math.BigInteger;

public class Node{

  private long ID;
  private byte INTERMEDIARY;
  // Each node in the tree generates a public key through a diffie-hellman exchange.
  public BigInteger PUBLIC_KEY;

  public Node(){
    PUBLIC_KEY = new BigInteger("-1");
    ID = -1;
  }
```

```
public Node(byte[] b){
  java.nio.ByteBuffer bb = java.nio.ByteBuffer.allocate(b.length);
  bb.put(b);
  bb.position(0);

  if(bb.get() == -1){
    ID = -1;
  }
  else{
    ID = bb.getLong();
  }

  int length = bb.getInt();
  byte[] shareval = new byte[length];
  bb.get(shareval);

  PUBLIC_KEY = new BigInteger(shareval);
}

public byte[] getBytes(){
  java.nio.ByteBuffer out = java.nio.ByteBuffer.allocate(sizeInBytes());

  // Hack to save space!!!
  if(ID == -1){
    out.put((byte)-1);
  }
  else{
    out.put((byte)1);
    out.putLong(ID);
  }

  byte[] a = PUBLIC_KEY.toByteArray();

  out.putInt(a.length);
  out.put(a);

  byte[] retval = new byte[sizeInBytes()];
  out.position(0);
  out.get(retval);

  return retval;
}

public long id(){
  return ID;
}

public void id(long id){
  ID = id;
}

public boolean intermediary(){
  return (ID == -1)?true:false;
}

public void intermediary(boolean value){
  INTERMEDIARY = ((value==true)?(byte)-1:(byte)1);
}

public String toString(){
  String s = new Long(ID).toString();
  return s.substring(((s.length()-7)>=0)?(s.length()-7):0, s.length());
}

// 1 byte intermediate flag
// Possible 8 byte ID
// 16/17 byte public key
public int sizeInBytes(){
```

```
    int pkl = PUBLIC_KEY.toByteArray().length;
    return (ID == -1)?(pkl+5):(pkl+13);
  }

}
```

5.2.52 v1/LGDH/Reactor.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
 */

import java.nio.channels.DatagramChannel;
import java.nio.channels.Selector;
import java.nio.channels.SelectionKey;
import java.net.InetSocketAddress;

/**
   Class <code>Reactor</code> is the main powerhouse for non-blocking UDP
   datagram broadcasts. It follows a traditional "Reactor" software pattern
   and uses a thread pool to handle the influx of events triggering actions.
 */
public class Reactor implements Runnable{

  /** The port where this <code>Reactor</code> will run.*/
  private static int PORT;

  /** The network Maximum Transmission Unit used in this <code>Reactor</code>.*/
  public static int MTU;

  public static long ID;

  /** The <code>DatagramChannel</code> used as a transport mechanism.*/
  private static DatagramChannel CHANNEL;

  private boolean JUMPSTART;

  public static int MAX_NODES;

  /**
      A </code>Selector</code> used to manage the I/O status of <code>SelectionKeys</code>
      in this <code>Reactor</code>, one for bookeeping measures, another for insecure user
      message capabilities.
   */
  private Selector SELECTOR;

  /** The <code>SelectionKey</code> used for secure group communication I/O bookeeping data. */
  private SelectionKey KEY;

  public static Reader READER;

  public static MessageProcessor PROCESSOR;

  public static Node NODE;

  private long accumulated;

  public static int DEBUG = 5;

  public void setDebugLevel(int level){
    DEBUG = level;
  }

  /** Constructs a new <code>Reactor</code> on port 12333 with a default MTU of 512 bytes. */
  public Reactor(){
    this(12333, 512, false, 100);
```

```
  }

  /**
      Constructs a new <code>Reactor</code> on the specified port with a default MTU of 512 bytes.
      @param port The port to bind the <code>Reactor</code>'s <code>DatagramChannel</code> on.
  */
  public Reactor(int port){
    this(port, 512, false, 100);
  }

  /**
     Constructs a new <code>Reactor</code> on the specified port and MTU.
     @param port The port to bind the <code>Reactor</code>'s <code>DatagramChannel</code> on.
     @param mtu The Maximum Transmission Unit used on the underlying network media.
   */
  public Reactor (int port, int mtu){
    this(port, mtu, false, 100);
  }

  public Reactor(int port, int mtu, boolean jumpstart, int max){
    MAX_NODES = max;
    PORT=port;
    MTU=mtu;
    JUMPSTART=jumpstart;
    ID = (long)(Math.random()*1000000000000000.0);
    NODE = new Node();
    NODE.id(ID);
    accumulated = 0;
    DEBUG=3;
  }

  /**
     Runs this <code>Reactor</code>.  First, all channels are initialized to be non-blocking,
     then an infinite poll-loop is entered to facilitate reaction-when-necessary event handling.
   */
  public void run(){
    if(DEBUG == 5){
      System.out.print("Reactor initializing...");
    }
    initialize();
    if(DEBUG == 5){
      System.out.println("done!");
    }
    if(!JUMPSTART){
      // Attach a new discoverer here...
      PROCESSOR.NODE_STATE = State.JOINING;
      PROCESSOR.PROTO_STATE = State.CALM;
      if(DEBUG >= 3){
System.out.println("[Reactor]:Probing...");
      }
      PROCESSOR.join_probe();
      KEY.attach(READER);
      KEY.interestOps(KEY.OP_READ);
    }
    else{
      PROCESSOR.NODE_STATE = State.INIT_ONLY;
      PROCESSOR.PROTO_STATE = State.CALM;
      // Otherwise wait for the jumpstart message to propogate to us...
      if (DEBUG >= 3){
System.out.println("Reader Attached!");
      }
      KEY.attach(READER);
      KEY.interestOps(KEY.OP_READ);
    }
    poll();
  }

  /**
```

```
     Initializes this <code>Reactor</code>. Channels and selectors are opened and configured for
     non-blocking I/O, and keys are set for reading until later notified otherwise.
   */
  private void initialize(){
    try{
      CHANNEL = DatagramChannel.open();

      // This line allows for many instances of this
      // class to run on the same machine...
      CHANNEL.socket().setReuseAddress(true);
      CHANNEL.socket().setBroadcast(true);
      CHANNEL.configureBlocking(false);
      /* Bind to the broadcast address on the given port */
      CHANNEL.socket().bind(new InetSocketAddress("0.0.0.0", PORT));
      SELECTOR = Selector.open();
      KEY = CHANNEL.register(SELECTOR, SelectionKey.OP_READ);
      /* Create the message processor using out arguments */
      READER = new Reader(KEY);
      PROCESSOR = new MessageProcessor(MTU, JUMPSTART, KEY, NODE);
      // If we are to not to be jumpstarted,
    }
    catch(Exception e){
      System.out.println("failed!\n\nError occured during initialization:");
      e.printStackTrace();
      System.exit(-1);
    }
  }

  /**
     The main poll loop.  The native hardware of the system is polled to see
     if any I/O bound for this <code>Reactor</code> is ready.  If so, a set
     of <code>SelectionKey</code>s are returned and iterated through, processing
     the I/O events in turn by separate thread-pool threads for maximum event-handling
     throughput.
   */
  private void poll(){
    if(DEBUG >= 3){
      System.out.println("Beginning poll loop...");
    }
    while(!Thread.currentThread().interrupted()){
      try{
if
  (
   (
    PROCESSOR.NODE_STATE == State.JOINING ||
    PROCESSOR.NODE_STATE == State.JOINED_G_S
    ) &&
   PROCESSOR.PROTO_STATE == State.CALM
   )
  {
    SELECTOR.select((int)(Math.random()*100000.0)%1000);
  }
else if(PROCESSOR.PROTO_STATE != State.CALM){
  SELECTOR.select(1000);
}
else
  {
    SELECTOR.select();
  }
/* ---------------------------------- */
      }
      catch(Exception e){
System.out.print("\nCaught poll interrupt!\nShutting down...");
try{
  KEY.cancel();
  CHANNEL.close();
  SELECTOR.close();
  System.out.println("done!");
```

```
  System.exit(-1);
}
catch(Exception f){}
      }
      java.util.Set readyKeys = SELECTOR.selectedKeys();
      java.util.Iterator it = readyKeys.iterator();
      while(it.hasNext()){
SelectionKey sk = (SelectionKey)it.next();
if(sk != null){
  Object o = sk.attachment();
  if(o != null && o instanceof Runnable){
    ((Runnable)o).run();
  }
}
      }
    }
  }

  public static void write(byte[] message, long destination){
    /* Later this needs to be passed to an encoder for breaking into chunks */
    java.nio.ByteBuffer out = java.nio.ByteBuffer.allocate(message.length+16);
    out.putLong(ID);
    out.putLong(destination);
    out.put(message);
    out.position(0);
    int wrote = 0;
    try{
      while(out.position() < out.capacity()-1){
CHANNEL.send(out, new java.net.InetSocketAddress("255.255.255.255", PORT));
      }
    }
    catch(Exception e){
      System.out.println("I/O Error on Channel!");
      e.printStackTrace();
    }
  }

}
```

5.2.53 v1/LGDH/Reader.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
*/

import java.nio.channels.SelectionKey;
import java.nio.ByteBuffer;
import java.net.SocketAddress;

public class Reader implements Runnable{

  private final SelectionKey KEY;
  private ByteBuffer BUFFER;
  private boolean EXECUTED;

  public Reader(SelectionKey sk){
    KEY=sk;
    EXECUTED=false;
    BUFFER= ByteBuffer.allocate(Reactor.MTU);
  }

  public void run(){
    try{
      BUFFER.position(0);
      SocketAddress a = ((java.nio.channels.DatagramChannel)(KEY.channel())).receive(BUFFER);
      BUFFER.position(0);
```

```
      // Either there is something here, useability undetermined...
      if(a != null){
new Decoder(KEY, BUFFER).run();
      }

      else if(
      (Reactor.PROCESSOR.NODE_STATE == State.JOINING && Reactor.PROCESSOR.PROTO_STATE == State.CALM)
      ){
if (Reactor.DEBUG >= 3)
  System.out.println("[Reader]:Join Probing...");
Reactor.PROCESSOR.join_probe();
      }

      else if(
      (Reactor.PROCESSOR.NODE_STATE == State.JOINED_G_S && Reactor.PROCESSOR.PROTO_STATE == State.CALM)
      ){
if (Reactor.DEBUG >= 3)
  System.out.println("[Reader]:Merge Probing...");
Reactor.PROCESSOR.merge_probe();
      }

      // Or we have timed out...
      else{
       Reactor.PROCESSOR.setProtoState(State.CALM);
KEY.selector().wakeup();
      }

      KEY.attach(this);
      KEY.interestOps(KEY.OP_READ);
      return;

    }
    catch(Exception e){
      System.out.println("I/O Error while reading!!!!");
      e.printStackTrace();
    }
  }
}
```

5.2.54 v1/LGDH/State.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
 */

public class State{

  /* Node States...*/
  public static final char INIT_ONLY = 0x0001;
  public static final char JOINING = 0x0002;
  public static final char JOINED_N_G_S = 0x0003;
  public static final char JOINED_G_S = 0x0004;

  /* Protocol States...*/

  /* The Join States ... */
  public static final char JOIN_SERVER = 0x0010;
  public static final char JOIN_CLIENT = 0x0011;
  public static final char JOIN_SERVER_CONFIRMED = 0x0012;
  public static final char JOIN_SERVER_PENDING = 0x0013;
  public static final char JOIN_CLIENT_PENDING = 0x0014;

  /* The Merge States */
  public static final char MERGE_SERVER = 0x0020;
  public static final char MERGE_CLIENT = 0x0021;
```

```
  public static final char MERGE_SERVER_CONFIRMED = 0x0022;
  public static final char MERGE_SERVER_PENDING = 0x0023;
  public static final char MERGE_CLIENT_PENDING = 0x0024;

  /* States relevant during key update */

  public static final char AWAITING_KEY_ARRAY= 0x0031;
  public static final char REQUESTING_TREE = 0x0032;
  public static final char UPDATING_KEY = 0x0033;

  /* The Calm State...*/
  public static final char CALM = 0xffff;

}
```

5.2.55 v1/LGDH/NodeArray.java

```
import java.util.Vector;
import java.io.*;

public class NodeArray{

  private Vector NODES;

  public static void main(String[] args){
    NodeArray STRUCT = new NodeArray();
    try{
      BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
      while(true){
System.out.print("\nNodeArray >> ");
String command = br.readLine();
String[] operands = command.split("\\s");
if(operands.length == 0){
  operands = new String[1];
  operands[0] = "";
}
final String operator = operands[0].toLowerCase();
if(operator.indexOf("add") >= 0 ){
  if(
     operands.length < 2 ||
     !java.util.regex.Pattern.matches("[0-9]+", operands[1])
     ){
    System.out.println("Add -> Wrong Syntax:\n"+
       "NodeArray>> add <nrNodes>");
  }
  else{
    int nrToAdd = Integer.parseInt(operands[1]);
    for(int i = 0; i < nrToAdd; i++){
      long rand = (long)(Math.random()*1000000000.0);
      Node n = new Node();
      n.id(rand);
      STRUCT.add(n);
    }
  }
}
else if(operator.indexOf("rem") >= 0){
  if(
     operands.length < 2 ||
     !java.util.regex.Pattern.matches("[0-9]+", operands[1])
     ){
    System.out.println("rem -> Wrong Syntax:\n"+
       "NodeArray>> rem <nodeID>");
  }
  else{
    long rem = Integer.parseInt(operands[1]);
    boolean gone = false;
```

```
    for(int i = 0; i < STRUCT.size(); i++){
      Node n = (Node)(STRUCT.getVector().elementAt(i));
      if(n.toString().indexOf(new Long(rem).toString()) >= 0){
STRUCT.remove(n);
gone = true;
      }
    }
    if(!gone){
      System.out.println("No resemblance of "+rem+" could be found!");
    }
  }
}
else if(operator.indexOf("merge") >= 0){
  if(
     operands.length < 2 ||
     !java.util.regex.Pattern.matches("[0-9]+", operands[1])
     ){
    System.out.println("Merge -> Wrong Syntax:\n"+
       "NodeArray>> merge <NodeArraySize>");
  }
  else{
    int nrToAdd = Integer.parseInt(operands[1]);
    NodeArray na = new NodeArray();
    for(int i = 0; i < nrToAdd; i++){
      long rand = (long)(Math.random()*1000000000.0);
      Node n = new Node();
      n.id(rand);
      na.add(n);
    }
    STRUCT.merge(na);
  }
}
else if(operator.indexOf("part") >= 0){

}
else if(operator.indexOf("quit") >= 0){
  System.out.println("Goodbye!");
  System.exit(0);
}
else{
  System.out.println("Unknown Operator! -> ["+operands[0]+"]");
}
System.out.println("Structure:\n"+STRUCT);
      }
    }
    catch (IOException e){
      e.printStackTrace();
      System.exit(0);
    }
  }

  public NodeArray(){
    NODES = new Vector();
  }

  public NodeArray(Vector v){
    NODES = v;
  }

  public NodeArray(byte[] nodes){
    NODES = new Vector();
    int count = 0;
    java.nio.ByteBuffer bb = java.nio.ByteBuffer.allocate(nodes.length);
    bb.put(nodes);
    bb.position(0);
    while (bb.position() < bb.limit()){
      int len = bb.getInt();
      byte[] node = new byte[len];
```

```
      bb.get(node);
      Node n = new Node(node);
      NODES.add(n);
    }
  }

  public void add(Node n){
    synchronized(NODES){
      NODES.add(0, n);
    }
  }

  public void remove(Node n){
    synchronized(NODES){
      NODES.remove(n);
    }
  }

  public void merge(NodeArray others){
    synchronized(NODES){
      NODES.addAll(others.getVector());
    }
  }

  // Functions as Vector.removeAll();
  public void partition(Vector ousted){
    synchronized(NODES){
      NODES.removeAll(ousted);
    }
  }

  public int size(){
    return NODES.size();
  }

  public Vector getVector(){
    synchronized(NODES){
      return NODES;
    }
  }

  public String toString(){
    String retval = "";
    synchronized(NODES){
      int a = NODES.size();
      if(a == 0){
retval = "Empty Set";
      }
      else{
retval += "[ ";
for(int i = 0; i < a; i++){
  String id =  new Long(((Node)NODES.elementAt(i)).id()).toString();
  retval +=id;
  if(i != (a-1)){
    retval +=(" | ");
  }
}
retval += " ]";
      }
    }
    return retval;
  }

  public long getSponsor(){
    return ((Node)(NODES.elementAt((0)))).id();
  }

  public byte[] getBytes(){
```

```
    synchronized(NODES){
      java.nio.ByteBuffer out = java.nio.ByteBuffer.allocate(Reactor.MTU);
      int count = 0;
      for(int i = 0; i < NODES.size(); i++){
byte[] a = ((Node)(NODES.elementAt(i))).getBytes();
out.putInt(a.length);
out.put(a);
count += a.length+4;
      }
      byte[] retval = new byte[count];
      out.position(0);
      out.get(retval);
      return retval;
    }
  }

  public int indexOf(long id){
    int index = -1;
    for(int i = 0; i < NODES.size(); i++){
      Node n = (Node)(NODES.elementAt(i));
      if(n.id() == id){
index = i;
      }
    }
    return index;
  }

}
```

5.2.56 v1/TGDH/Decoder.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
 */

import java.nio.channels.SelectionKey;
import java.nio.ByteBuffer;
import java.net.InetSocketAddress;

public class Decoder implements Runnable{

  private SelectionKey KEY;
  private ByteBuffer MESSAGE;
  private boolean EXECUTED;
  private InetSocketAddress source;

  int length;
  byte[] tree;
  Tree t;


  public Decoder(SelectionKey key, ByteBuffer mesg, InetSocketAddress addr){
    KEY=key;
    EXECUTED = false;
    MESSAGE=mesg;
    source = addr;
  }

  public void run(){
    synchronized(this){
      if(EXECUTED)return;
      else EXECUTED=true;
    }
    /* Decode a message */
    long src = MESSAGE.getLong();
    long dest = MESSAGE.getLong();
```

```
    if(src != Reactor.ID && (dest == Message_OPC.BROADCAST || dest == Reactor.ID )){
      final char type = MESSAGE.getChar();
      switch(type){

      case Message_OPC.JUMPSTART:
Reactor.PROCESSOR.jumpstart();
break;

/* ----------------- Join Messages ----------------- */
      case Message_OPC.JOIN_REQUEST:
Reactor.PROCESSOR.join_request(src, source);
break;
      case Message_OPC.JOIN_RESPONSE:
Reactor.PROCESSOR.join_response(src, source);
break;

      case Message_OPC.JOIN_CONFIRM:
byte[] rest = new byte[Reactor.NODE.sizeInBytes()];
MESSAGE.get(rest);
Node n = new Node(rest);
Reactor.PROCESSOR.join_confirm(n, source);
break;

      case Message_OPC.JOIN_REJECT:
Reactor.PROCESSOR.join_reject(src, source);
break;

/* ---------------- Merge Messages ------------------ */

      case Message_OPC.MERGE_REQUEST:
Reactor.PROCESSOR.merge_request(src, source);
break;

      case Message_OPC.KEY_UPDATE:
int plen = MESSAGE.getInt();
byte[] p = new byte[plen];
MESSAGE.get(p);
java.math.BigInteger prime = new java.math.BigInteger(p);

int glen = MESSAGE.getInt();
byte[] g = new byte[glen];
MESSAGE.get(g);
java.math.BigInteger generator = new java.math.BigInteger(g);

Reactor.PROCESSOR.keyUpdate(src, prime, generator, source);
break;

      case Message_OPC.MERGE_RESPONSE:
Reactor.PROCESSOR.merge_response(src, source);
break;

      case Message_OPC.MERGE_CONFIRM:
int length = MESSAGE.getInt();
byte[] tree = new byte[length];
MESSAGE.get(tree);
Tree t = new Tree(tree);
Reactor.PROCESSOR.merge_confirm(src, t, source);
break;

      case Message_OPC.DH_WAIT:
Reactor.PROCESSOR.dh_wait(src, source);
break;

      case Message_OPC.DH_STAGE_1:

length = MESSAGE.getInt();
byte[] pa = new byte[length];
MESSAGE.get(pa);
```

```
prime = new java.math.BigInteger(pa);

length = MESSAGE.getInt();
byte [] ga = new byte[length];
MESSAGE.get(ga);
generator = new java.math.BigInteger(ga);

length = MESSAGE.getInt();
byte[] pka = new byte[length];
MESSAGE.get(pka);
java.math.BigInteger public_key = new java.math.BigInteger(pka);

Reactor.PROCESSOR.diffieHellmanStageOne(src, prime, generator, public_key, source);
break;

      case Message_OPC.DH_STAGE_2:
length = MESSAGE.getInt();
byte[] pubk = new byte[length];
MESSAGE.get(pubk);

int position = MESSAGE.getInt();

Tree tr = null;
length = MESSAGE.getInt();
byte[] tre = new byte[length];
MESSAGE.get(tre);
tr = new Tree(tre);

Reactor.PROCESSOR.diffieHellmanStageTwo(src, new java.math.BigInteger(pubk), tr, position, source);
break;

      case Message_OPC.MERGE_REJECT:
Reactor.PROCESSOR.merge_reject(src, source);
break;

      case Message_OPC.SPONSOR_SWITCH:
long newS = MESSAGE.getLong();
int len = MESSAGE.getInt();
byte[] a4 = new byte[len];
MESSAGE.get(a4);
Reactor.PROCESSOR.sponsor_switch(src, newS, new Tree(a4), source);
break;


/* ------------------------------------------------ */

      case Message_OPC.T_COMMIT:
Reactor.PROCESSOR.t_commit(src, source);
break;

      case Message_OPC.T_STRUCTURE:
length = 0;
try{
  length = MESSAGE.getInt();
  tree = new byte[length];
  MESSAGE.get(tree);
  Reactor.PROCESSOR.tree_update(new Tree(tree), src, source);
}
catch(Exception e){
  if (e instanceof java.nio.BufferUnderflowException){
    System.out.println("Needed:"+length+"\nAvailable:"+(MESSAGE.limit()-MESSAGE.position()));
    System.out.println("Network MTU Exceeded By Group Structure Transmission!\n"+
       "Fragmentation Not Implemented Yet!"
       );
    System.exit(-1);
  }
}
break;
```

```
      case Message_OPC.KEY_TREE:
length = 0;
try{
  length = MESSAGE.getInt();
  tree = new byte[length];
  MESSAGE.get(tree);
  Reactor.PROCESSOR.key_tree(new Tree(tree), src, source);
}
catch(Exception e){
  if (e instanceof java.nio.BufferUnderflowException){
    System.out.println("Needed:"+length+"\nAvailable:"+(MESSAGE.limit()-MESSAGE.position()));
    System.out.println("Network MTU Exceeded By Group Structure Transmission!\n"+
       "Fragmentation Not Implemented Yet!"
       );
    System.exit(-1);
  }
}
break;
      }
    }

  }

}
```

5.2.57 v1/TGDH/Device.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
 */

/**
   The main executable class for Kieran S. Hagzan's master's thesis work at R.I.T. .
   This class parses command-line arguments, checks their validity, creates a new
   <code>Reactor</code>and displays usage information if necessary.
 */
public class Device{

  /**
     A newly created <code>Reactor</code> to be run.
   */
  private final Reactor REACTOR;

  /**
      Creates a new <code>Reactor</code> on the given port
      using the specified network MTU.
      @param port The port to bind the <code>Reactor</code> on, (1024 <= port <= 65535).
      @param mtu The Maximum Transmission Unit of the <code>Reactor</code> (1 <= mtu <= 65535).
   */
  private Device(int port, int mtu, boolean jumpstart, int maxNodes){
    REACTOR = new Reactor(port, mtu, jumpstart, maxNodes);
    REACTOR.setDebugLevel(2);
    REACTOR.run();
  }

  /**
     The main method invoked by the VM at program call.
     @param args The command-line arguments passed to this class.
   */
  public static void main(String[] args){
    boolean jumpstart;
    int mtu;
    int port;
    int max;
```

```
    // Bomb out if the user cannot type "--help"
    if (args.length > 4)exitWithUsage();
    // grab the outermost argument - the jumpstart mode if present

    // Grab the first argument - the port if present
    if(args.length >= 4){
      if(!java.util.regex.Pattern.matches("[0-9]+", args[3])){
exitWithUsage();
      }
      max = Integer.parseInt(args[3]);
    }
    else{
      max = 100;
    }

    if(args.length >= 3){
      String s = args[2];

      if ((s.indexOf("true") < 0) && (s.indexOf("false") < 0)){
exitWithUsage();
      }

      if(s.indexOf("true") >= 0){
jumpstart = true;
      }
      else{
jumpstart = false;
      }
    }
    else{
      jumpstart = false;
    }

    // Grab the secondmost argument - the MTU if present
    if(args.length >= 2){
      if(!java.util.regex.Pattern.matches("[0-9]+", args[1])){
exitWithUsage();
      }
      mtu = Integer.parseInt(args[1]);
      if (mtu < 33 || mtu > 65535){exitWithUsage();}
    }
    else{
      mtu = 512;
    }

    // Grab the first argument - the port if present
    if(args.length >= 1){
      if(!java.util.regex.Pattern.matches("[0-9]+", args[0])){
exitWithUsage();
      }
      port = Integer.parseInt(args[0]);
      if(port < 1024 || port > 65535){exitWithUsage();}
    }
    else{
      port = 12333;
    }

    // Let them know what they have done....
    //    System.out.println("Initializing on port "+port+" using MTU of "+mtu+
    //        (jumpstart?" with":" without")+" jumpstart mode...");

    new Device(port, mtu, jumpstart, max);
  }

  /**
     Displays proper usage of this class.
   */
  private static void exitWithUsage(){
```

```
    System.out.println("\nUsage:\n------\n"+
       "\t\"java Device --help\" will display this screen\n"+
       "\t\"java Device <port> <mtu> <jumpstart> \"\n\nWhere:\n------\n"+
       "<port> - is an optionally specified number between 1024 and 65535 (defaults to 12333)\n"+
       "<mtu>  - is an optionally specified number between 33 and 65535 (defaults to 512)\n"+
       "<jumpstart> - is one of \"true\" or \"false\" indicating the jumpstart status of the node\n"
       );
    System.exit(1);
  }
}
```

5.2.58 v1/TGDH/DiffieHellmanExchange.java

```
import java.math.*;
import java.util.Random;
import java.nio.ByteBuffer;
import java.net.InetSocketAddress;

public class DiffieHellmanExchange{

  public BigInteger p;
  public BigInteger g;
  public BigInteger x;
  public BigInteger public_key_alice;
  public BigInteger public_key_bob;
  public BigInteger secret_key;

  public long ALICE;
  public long BOB;

  // Constructor for bob...
  public DiffieHellmanExchange(
       long alice, long bob, BigInteger prime, BigInteger generator,
       BigInteger share, BigInteger alices_public_key,java.util.Vector subtree,
       int prune_index
       ){
    ALICE=alice;
    BOB=bob;
    p = prime;
    g = generator;

    if (Reactor.DEBUG >= 3){
      System.out.println("Diffie-Hellman:[Initiated:]"+alice+" and "+bob+" with p.k.:"+alices_public_key);
    }

    Random rand = new Random();

    x = share;

    // And public key....
    public_key_alice = alices_public_key;
    public_key_bob = g.modPow(x,p);
    secret_key = alices_public_key.modPow(x, p);

    if(Reactor.DEBUG >= 3){
      System.out.println("\nCurrent Secret Key:"+x);
      System.out.println("Public Key Alice:"+public_key_alice);
      System.out.println("Public Key Bob:"+public_key_bob);
      System.out.println("Computed Key:"+secret_key+"\n");
    }

    // TODO, attach our key of tree's rooted at our responsibilitie's node...
    ByteBuffer out = ByteBuffer.allocate(Reactor.MTU);
    // The flag
    out.putChar(Message_OPC.DH_STAGE_2);
```

```
  // Bob's public key
  byte[] pua = public_key_bob.toByteArray();
  out.putInt(pua.length);
  out.put(pua);
  // An integer representing the node where the tree is located, if needed

  Tree t = new Tree(subtree);
  byte[] tree = t.getBytes();
  out.putInt(prune_index);
  out.putInt(tree.length);
  out.put(tree);

  pua = new byte[out.position()];
  out.position(0);
  out.get(pua);
  Reactor.write(pua, alice, new InetSocketAddress("255.255.255.255", Reactor.PORT));
}



// The final exchange to alice
public void finishExchange(long bob, BigInteger bobs_public_key){
  public_key_bob = bobs_public_key;
  if(Reactor.DEBUG>=3)
    System.out.println("Finishing Exchange with "+bob);
  secret_key = bobs_public_key.modPow(x, p);
  if(Reactor.DEBUG>=3){
    System.out.println("\nCurrent Secret Key:"+x);
    System.out.println("Public Key Alice:"+public_key_alice);
    System.out.println("Public Key Bob:"+public_key_bob);
    System.out.println("Computed Key:"+secret_key+"\n");
  }
}

// Constructor for alice
public DiffieHellmanExchange(long alice, long bob, BigInteger prime, BigInteger generator, BigInteger share){
  if (Reactor.DEBUG >= 3){
    System.out.println("Performing Diffie-Hellman Exchange between "+alice+" and "+bob);
  }

  ALICE=alice;
  BOB=bob;

  Random rand = new Random();

  // Generate a private key....
  x = share;
  g = generator;
  p = prime;

  // And public key....
  public_key_alice = generator.modPow(x,prime);

  ByteBuffer bb = ByteBuffer.allocate(Reactor.MTU);

  bb.putChar(Message_OPC.DH_STAGE_1);

  byte[] po = prime.toByteArray();
  byte[] go = generator.toByteArray();
  byte[] pko = public_key_alice.toByteArray();

  bb.putInt(po.length);
  bb.put(po);

  bb.putInt(go.length);
  bb.put(go);

  bb.putInt(pko.length);
```

```
    bb.put(pko);

    byte[] out = new byte[bb.position()];
    bb.position(0);
    bb.get(out);

    Reactor.write(out, bob,new InetSocketAddress("255.255.255.255", Reactor.PORT));
  }



}
```

5.2.59 v1/TGDH/Discoverer.java

```
import java.nio.channels.SelectionKey;
import java.nio.ByteBuffer;
import java.net.InetSocketAddress;

public class Discoverer implements Runnable{

  private final SelectionKey KEY;
  private boolean EXECUTED;

  public Discoverer(SelectionKey key){
    KEY=key;
    EXECUTED=false;
  }

  public void run(){
    synchronized(this){
      if(EXECUTED)return;
      EXECUTED=true;
    }
    ByteBuffer bb = ByteBuffer.allocate(2);
    bb.putChar(Message_OPC.JOIN_REQUEST);
    bb.position(0);
    byte[] a = new byte[bb.limit()];
    bb.get(a);
    Reactor.write(a, Message_OPC.BROADCAST,new InetSocketAddress("255.255.255.255", Reactor.PORT));
    KEY.attach(Reactor.READER);
    KEY.interestOps(KEY.OP_READ);
  }

}
```

5.2.60 v1/TGDH/Jumpstarter.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
 */

public class Jumpstarter{
  public static void main(String[] args){
boolean local;
if(args.length > 0 && args[0].indexOf("local") >= 0){
  local = true;
}
else local=false;
    try{
      java.nio.channels.DatagramChannel c = java.nio.channels.DatagramChannel.open();
      c.socket().setReuseAddress(true);
      c.socket().setBroadcast(true);
```

```
      c.socket().bind(new java.net.InetSocketAddress("0.0.0.0", 12333));
      java.nio.ByteBuffer out = java.nio.ByteBuffer.allocateDirect(18);
      out.putLong(0xdeadbeefl);
      out.putLong(0xffffffffffffffffl);
      out.putChar(Message_OPC.JUMPSTART);
      for(int i = 0; i < 3; i++){
out.position(0);
java.net.InetSocketAddress outA;
if(local){
  System.out.println("Yelling locally...");
  outA = new java.net.InetSocketAddress("localhost", 12333);
}
else{
    outA = new java.net.InetSocketAddress("255.255.255.255", 12333);
}
      c.send(out, outA);
      }
      c.close();
    }
    catch(Exception e){
      e.printStackTrace();
      System.exit(-1);
    }
  }
}
```

5.2.61 v1/TGDH/Merger.java

```
import java.nio.channels.SelectionKey;
import java.nio.ByteBuffer;
import java.net.InetSocketAddress;

public class Merger implements Runnable{

  private final SelectionKey KEY;
  private boolean EXECUTED;

  public Merger(SelectionKey key){
    KEY=key;
    EXECUTED=false;
  }

  public void run(){
    synchronized(this){
      if(EXECUTED)return;
      EXECUTED=true;
    }
    ByteBuffer bb = ByteBuffer.allocate(2);
    System.out.println("Requesting Merge...");
    bb.putChar(Message_OPC.MERGE_REQUEST);
    bb.position(0);
    byte[] a = new byte[bb.limit()];
    bb.get(a);
    Reactor.write(a, Message_OPC.BROADCAST,new InetSocketAddress("255.255.255.255", Reactor.PORT));
    KEY.attach(new Reader(KEY));
    KEY.interestOps(KEY.OP_READ);
  }

}
```

5.2.62 v1/TGDH/Message_OPC.java

```
/*
```

```
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
 */

public class Message_OPC{

  public static final long BROADCAST = 0xffffffffffffffffl;

  public static final char JUMPSTART =0xffff;

  public static final char JOIN_REQUEST =0x0001;
  public static final char JOIN_RESPONSE =0x0002;
  public static final char JOIN_CONFIRM =0x0003;
  public static final char JOIN_REJECT =0x0004;


  public static final char T_COMMIT =0x0005;
  public static final char T_STRUCTURE = 0x0006;

  public static final char MERGE_REQUEST = 0x0007;
  public static final char MERGE_RESPONSE = 0x0008;
  public static final char MERGE_CONFIRM =0x0009;
  public static final char MERGE_REJECT =0x000a;

  public static final char SPONSOR_SWITCH =0x000b;

  public static final char DH_STAGE_1 = 0x000c;
  public static final char DH_STAGE_2 = 0x000d;
  public static final char KEY_UPDATE = 0x000e;
  public static final char DH_WAIT = 0x000f;
  public static final char KEY_TREE = 0x0010;

}
```

5.2.63 v1/TGDH/MessageProcessor.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
*/

import java.nio.channels.SelectionKey;
import java.nio.ByteBuffer;
import java.math.BigInteger;
import java.net.InetSocketAddress;

public class MessageProcessor{

  private Tree TREE;

  private final int MTU;
  public boolean JUMPSTART;
  private final SelectionKey KEY;

  private final Object STATE_LOCK;

  private boolean IN_EXCHANGE;
  private boolean TIMEOUT_MATTERS;

  private Tree TEMP_TREE;
  private long LAST_SOURCE;

  private long SPONSOR_ID;
  private long NEW_SPONSOR;

  private long MERGER_ID;

  public char NODE_STATE;
```

```
public char PROTO_STATE;

private ByteBuffer BUFFER;

private DiffieHellmanExchange CURRENT_EXCHANGE;

// Stuff for key agreement....
private int CURRENT_PARTNER;
private long CURRENT_PARTNER_ID;

private BigInteger MY_PRIVATE;
private BigInteger CURRENT_PRIVATE;
private BigInteger ENCRYPTION_KEY;

// Stuff for gauging key agreement times
private long START_TIME;
private double ELAPSED;

// Diffie-Hellman Parameters...
private BigInteger CURRENT_P;
private BigInteger CURRENT_G;
private java.util.Random GENERATOR;
// --------------------------------

long lastMeaningful = 0;

long TIMEOUT = 2000;

public MessageProcessor(int mtu, boolean jumpstart, SelectionKey key, Node node){
  IN_EXCHANGE = false;
  GENERATOR = new java.util.Random();
  MTU=mtu;
  JUMPSTART=jumpstart;
  NODE_STATE = State.INIT_ONLY;
  KEY=key;

  STATE_LOCK = new Object();

  TREE = new Tree();
  TREE.add(node);

  if(Reactor.DEBUG >= 3)
    System.out.println(TREE);
}

/* ********************************************************************************************** */
/* ****************************** Helper Functions ********************************************** */
/* ********************************************************************************************** */

public char getProtoState(){
  return PROTO_STATE;
}

public char getNodeState(){
  return NODE_STATE;
}

public void setProtoState(char state){
  synchronized(STATE_LOCK){
    PROTO_STATE = state;
  }
}

public void setNodeState(char state){
  synchronized(STATE_LOCK){
    NODE_STATE = state;
  }
}
```

```
  public void setStates(char nstate, char pstate){
    synchronized(STATE_LOCK){
      NODE_STATE = nstate;
      PROTO_STATE = pstate;
    }
  }

  private void allocate(int size){
    BUFFER = ByteBuffer.allocate(size);
  }

  public void writeMessage(ByteBuffer b, long dest, InetSocketAddress recipient){
    byte[] a = new byte[b.limit()];
    b.position(0);
    b.get(a);
    Reactor.write(a, dest, recipient);
  }

  public void printDebug(String message, long src){
    //if(Reactor.DEBUG >= 3){
      System.out.println(
 "\n"+message+" "+src+"\n"+
 "Node State:0x"+Integer.toString(NODE_STATE, 16)+"\n"+
 "Proto State:0x"+Integer.toString(PROTO_STATE, 16)+"\n"
 );
      //}
  }

  public boolean timedOut(){
    boolean retval = (System.currentTimeMillis()-lastMeaningful > TIMEOUT) && TIMEOUT_MATTERS;
    return retval;
  }


  /**
     Computes the currently known key from the current tree recursively...
  */
  private void computeKey (BigInteger share, int index){
    int parentIndex = TREE.parentOf(index);
    if(parentIndex < 0){
      ENCRYPTION_KEY = share;

      if(Reactor.DEBUG >= 3)
System.out.println("\n"+TREE+"\n");
      if(Reactor.DEBUG >= 4)
System.out.println("Key is:"+ENCRYPTION_KEY);

      System.out.println(TREE.nodeCount()+"\t"+ELAPSED);
      if(Reactor.MAX_NODES == TREE.nodeCount()){System.exit(0);}
      PROTO_STATE = State.CALM;
    }
    else{
      int sibling;
      if(index % 2 == 0){
sibling = TREE.leftChildOf(parentIndex);
      }
      else{
sibling = TREE.rightChildOf(parentIndex);
      }
      BigInteger publicKey = ((Node)TREE.NODE_ARRAY.elementAt(sibling)).PUBLIC_KEY;
      BigInteger key = publicKey.modPow(share, CURRENT_P);
      computeKey(key, parentIndex);
    }
  }

  private int getMyFinalPosition(int index){
    if(index % 2 != 0){
```

```
      return index;
    }
    else return getMyFinalPosition(TREE.parentOf(index));
  }


  /* ------------------------------------------------------------------------------------ */
  /* ----------------------------- Message Reaction Functions ---------------------------- */
  /* ------------------------------------------------------------------------------------ */

  /* ************************************************************************************ */
  /*                            Functions to deal with key agreement                      */
  /* ************************************************************************************ */


  /**
     Causes all "non-global-sponsor" nodes to initiate the "appropriate" action in response
     to a "KEY_UPDATE" request. i.e., if nodes have an even index in their tree positions,
     they must initiate a Diffie-Hellman key exchange with their partners.  Otherwise,
     they sit and wait for a request.
   */
  public void keyUpdate(long src, BigInteger p, BigInteger g, InetSocketAddress source){
    START_TIME = System.currentTimeMillis();
    // Refresh our own share....
    if(src == SPONSOR_ID && PROTO_STATE == State.CALM && NODE_STATE == State.JOINED_N_G_S){
      lastMeaningful = System.currentTimeMillis();
      CURRENT_P = p;
      CURRENT_G = g;

      MY_PRIVATE = new BigInteger(128, 100, GENERATOR);
      // Initially, the private key is our share
      CURRENT_PRIVATE = MY_PRIVATE;


      if(TREE.indexOf(Reactor.ID) % 2 == 0){
// Initially, even nodes's have partners being their left sibling
CURRENT_PARTNER = TREE.leftChildOf(TREE.parentOf(TREE.indexOf(Reactor.ID)));
CURRENT_PARTNER_ID = ((Node)TREE.NODE_ARRAY.elementAt(CURRENT_PARTNER)).id();
PROTO_STATE = State.UPDATING_KEY;
initDiffieHellman();
      }
      else{
// If I am odd, set current partner to be our rightmost sibling
CURRENT_PARTNER = TREE.rightChildOf(TREE.parentOf(TREE.indexOf(Reactor.ID)));
CURRENT_PARTNER_ID = TREE.getSponsor(CURRENT_PARTNER);
PROTO_STATE = State.UPDATING_KEY;
      }
      CURRENT_PARTNER_ID = ((Node)(TREE.NODE_ARRAY.elementAt(CURRENT_PARTNER))).id();
    }
  }

  /**
      Initialize a Diffie-Hellman key exchange at the current point in the tree,
      with the current key, prime, and personal share. If the node at which this
      exchange is to take place is an intermediary node, we will query it's "sponsor"
      node.
  */
  public void initDiffieHellman(){
    synchronized(STATE_LOCK){
      lastMeaningful = System.currentTimeMillis();
      // Assert - P, G, CURRENT_INDEX, and MY_PRIVATE are properly set...
      if(CURRENT_PARTNER_ID == -1){
CURRENT_PARTNER_ID = TREE.getSponsor(CURRENT_PARTNER);
      }

      // Initialize a diffie-hellman exchange with the partner...

      // I am alice....
```

```
      CURRENT_EXCHANGE = new DiffieHellmanExchange(
   Reactor.ID,
   CURRENT_PARTNER_ID,
   CURRENT_P,
   CURRENT_G,
   CURRENT_PRIVATE
   );
    }
  }

  /**
     Message handler for the request of a Diffie-Hellman key exchange.  From a cryptographic
     perspective, this indicates that this node is "Bob," or the non-initiating party...
   */
  public void diffieHellmanStageOne(long src, BigInteger p, BigInteger g, BigInteger pk,InetSocketAddress source)
    // If I am in an update, we must tell this person to wait...
    if(src != CURRENT_PARTNER_ID){
      // Tell the asking node to wait.....
      allocate(2);
      BUFFER.putChar(Message_OPC.DH_WAIT);
      writeMessage(BUFFER,src, source);
      return;
    }

    // I am bob.....
    // Now, We REALLY represent the left child of the parent of our current position
    int position = getMyFinalPosition(TREE.indexOf(Reactor.ID));
    java.util.Vector v = new java.util.Vector();
    TREE.collectAll(v, 0, position);
    CURRENT_EXCHANGE = new DiffieHellmanExchange(src, Reactor.ID, p, g, CURRENT_PRIVATE, pk, v, position);
    NODE_STATE = State.JOINED_N_G_S;
    PROTO_STATE = State.AWAITING_KEY_TREE;
  }


  /**
     Message handler indicating that the current Diffie-Hellman Key exchange is to be
     posponed for a breif period until "lesser" nodes in the tree have stabilized.
   */
  public void dh_wait(long src, InetSocketAddress source){
    if(src == CURRENT_PARTNER_ID){
      try{Thread.sleep(1000);}catch(Exception e){e.printStackTrace();}
      CURRENT_EXCHANGE = new DiffieHellmanExchange(
   Reactor.ID,
   CURRENT_PARTNER_ID,
   CURRENT_P,
   CURRENT_G,
   CURRENT_PRIVATE
   );
    }
  }


  /* ##########################################################################################  */

  // WARNING WARNING WARNING WARNING WARNING WARNING WARNING WARNING WARNING WARNING  //
  // ------------------------------------------------------------------------------- //
  // The "position" parameter is broken!!! Rely on CURRENT_PARTNER to determine where //
  // subtrees are appended!!!                                                        //
  public void diffieHellmanStageTwo(long src, BigInteger pk, Tree t, int position, InetSocketAddress source ){
    synchronized(STATE_LOCK){
      if(src == CURRENT_PARTNER_ID){
// I am alice.....
CURRENT_EXCHANGE.finishExchange(src, pk);

// I was the initiator of this exchange, therefore, I was even.
// move up a notch
//System.exit(-1);
```

```
TREE.addSubtree(t.NODE_ARRAY, TREE.NODE_ARRAY, 0, CURRENT_PARTNER);
int left = CURRENT_PARTNER;
long left_id = ((Node)TREE.NODE_ARRAY.elementAt(left)).id();
BigInteger left_pk = CURRENT_EXCHANGE.public_key_bob;
Node n = new Node();
n.id(left_id);
n.PUBLIC_KEY = left_pk;
TREE.NODE_ARRAY.set(left, n);

int right = TREE.rightChildOf(TREE.parentOf(CURRENT_PARTNER));
long right_id = ((Node)TREE.NODE_ARRAY.elementAt(right)).id();
BigInteger right_pk = CURRENT_EXCHANGE.public_key_alice;
Node m = new Node();
m.id(right_id);
m.PUBLIC_KEY = right_pk;
TREE.NODE_ARRAY.set(right, m);

// If the parent of our partner is the root node, we can broadcast the key
// tree
if(TREE.parentOf(CURRENT_PARTNER) == 0){
  byte[] a = TREE.getBytes();
  allocate(a.length + 6);
  BUFFER.putChar(Message_OPC.KEY_TREE);
  BUFFER.putInt(a.length);
  BUFFER.put(a);
  BUFFER.position(0);
  writeMessage(BUFFER, Message_OPC.BROADCAST, new InetSocketAddress("255.255.255.255", Reactor.PORT));
  ELAPSED = (double)(System.currentTimeMillis() - START_TIME);
  ELAPSED = ELAPSED / 1000.0;
  computeKey(MY_PRIVATE, TREE.indexOf(Reactor.ID));
  PROTO_STATE = State.CALM;
  return;
}
else if(TREE.parentOf(CURRENT_PARTNER) % 2 != 0){
  CURRENT_PARTNER = TREE.rightChildOf(TREE.parentOf(TREE.parentOf(CURRENT_PARTNER)));
  CURRENT_PARTNER_ID = TREE.getSponsor(CURRENT_PARTNER);
  CURRENT_PRIVATE = CURRENT_EXCHANGE.secret_key;
}
// If we are at an even index, prepare for an exchange with the proper lefthand party
else{
  int newP =  TREE.leftChildOf(TREE.parentOf(TREE.parentOf(CURRENT_PARTNER)));

  CURRENT_PARTNER = newP;
  CURRENT_PARTNER_ID = ((Node)TREE.NODE_ARRAY.elementAt(CURRENT_PARTNER)).id();
  CURRENT_PRIVATE = CURRENT_EXCHANGE.secret_key;
  initDiffieHellman();
}
     }
   }
  }
  /* ##########################################################################################  */


  /* ***************************************************************************************** */
  /*                                    End key agreement                                      */
  /* ***************************************************************************************** */


  /* ***************************************************************************************** */
  /*                            Functions to delegate a global sponsor                         */
  /* ***************************************************************************************** */


  private void sponsorUpdate(){
    synchronized(STATE_LOCK){
      if (Reactor.DEBUG >= 3)
System.out.println("Updating sponsor to:"+TREE.getSponsor(0));
```

```
      // Literally do the switch......
      lastMeaningful = System.currentTimeMillis();
      long sponsor = TREE.getSponsor(0);
      SPONSOR_ID = sponsor;

      // Make a conceptual switch of necessary....
      if (NODE_STATE == State.JOINED_G_S && sponsor != Reactor.ID){
if (Reactor.DEBUG >= 3)
  System.out.println("Demoting myself...");

NODE_STATE = State.JOINED_N_G_S;
PROTO_STATE = State.CALM;
      }
      else if(NODE_STATE == State.JOINED_N_G_S && sponsor == Reactor.ID){
if (Reactor.DEBUG >= 3)
  System.out.println("Promoting myself...");
NODE_STATE = State.JOINED_G_S;
PROTO_STATE = State.UPDATING_KEY;
      }

      // Take the appropriate action
      if (NODE_STATE == State.JOINED_G_S){
PROTO_STATE = State.UPDATING_KEY;

if (Reactor.DEBUG >= 3)
  System.out.println("Sponsor Update Complete -> Sponsor!");

// Reset the public values, p, g, and our share...
CURRENT_P = new BigInteger(128, 100, GENERATOR);
CURRENT_G = new BigInteger(CURRENT_P.bitLength(), GENERATOR);
MY_PRIVATE = new BigInteger(128, 100, GENERATOR);
// Initially, the private key is our share
CURRENT_PRIVATE = MY_PRIVATE;
// Set out current responsibility to be our sibling
CURRENT_PARTNER = TREE.leftChildOf(TREE.parentOf(TREE.indexOf(Reactor.ID)));
CURRENT_PARTNER_ID = ((Node)(TREE.NODE_ARRAY.elementAt(CURRENT_PARTNER))).id();

// Broadcast the update request...
allocate(Reactor.MTU);
byte[] pa = CURRENT_P.toByteArray();
byte[] ga = CURRENT_G.toByteArray();
allocate(pa.length+ga.length+10);
BUFFER.putChar(Message_OPC.KEY_UPDATE);
BUFFER.putInt(pa.length);
BUFFER.put(pa);
BUFFER.putInt(ga.length);
BUFFER.put(ga);
writeMessage(BUFFER, Message_OPC.BROADCAST, new InetSocketAddress("255.255.255.255", Reactor.PORT));
START_TIME = System.currentTimeMillis();
initDiffieHellman();
      }
      else{
if (Reactor.DEBUG >= 3)
  System.out.println("Sponsor Update Complete -> Non-sponsor!");
PROTO_STATE = State.CALM;
      }
    }
  }

  public void sponsor_switch(long src, long newS, Tree t, InetSocketAddress source){
    synchronized(STATE_LOCK){

      if(src == SPONSOR_ID && NODE_STATE == State.JOINED_N_G_S){
TEMP_TREE = t;
LAST_SOURCE = newS;
SPONSOR_ID = newS;
PROTO_STATE = State.JOIN_CLIENT_PENDING;
lastMeaningful = System.currentTimeMillis();
```

```
      }
    }
  }


  /* ********************************************************************************** */
  /*                                End global sponsor                                  */
  /* ********************************************************************************** */



  /* ********************************************************************************** */
  /*                                Protocol Function Handlers                          */
  /* ********************************************************************************** */


  public void join_probe(){
    synchronized(STATE_LOCK){
      allocate(2);
      BUFFER.putChar(Message_OPC.JOIN_REQUEST);
      writeMessage(BUFFER, Message_OPC.BROADCAST, new InetSocketAddress("255.255.255.255", Reactor.PORT));
      if(Reactor.DEBUG >=3){
System.out.println("Sent join probe...");
      }
    }
  }

  public void merge_probe(){
    synchronized(STATE_LOCK){
      allocate(2);
      BUFFER.putChar(Message_OPC.MERGE_REQUEST);
      writeMessage(BUFFER, Message_OPC.BROADCAST, new InetSocketAddress("255.255.255.255", Reactor.PORT));
      if(Reactor.DEBUG >=3){
System.out.println("Sent merge probe...");
      }
    }
  }

 public void jumpstart(){
    synchronized(STATE_LOCK){
      if(NODE_STATE == State.INIT_ONLY){
printDebug("Jumpstart", 0);
NODE_STATE = State.JOINING;
PROTO_STATE = State.CALM;
if(Reactor.DEBUG >= 3){
  System.out.println("[Reactor]:Join Probing...");
}
join_probe();
KEY.attach(Reactor.READER);
KEY.interestOps(KEY.OP_READ);
      }
    }
  }

  public void join_request(long src, InetSocketAddress source){
    synchronized(STATE_LOCK){
      if(
 (NODE_STATE == State.JOINING || NODE_STATE == State.JOINED_G_S) &&
 PROTO_STATE==State.CALM
 ){
TIMEOUT_MATTERS=true;
printDebug("Join Request ->",src);
PROTO_STATE = State.JOIN_SERVER;
allocate(2);
BUFFER.putChar(Message_OPC.JOIN_RESPONSE);
writeMessage(BUFFER, src, source);
LAST_SOURCE = src;
```

```
lastMeaningful = System.currentTimeMillis();

      }
      else if(timedOut()){
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
join_request(src, source);
      }
    }
  }

  public void merge_request(long src, InetSocketAddress source){
    synchronized(STATE_LOCK){
      if(
 NODE_STATE == State.JOINED_G_S &&
 PROTO_STATE == State.CALM
 ){
if(Reactor.DEBUG >= 3){
  System.out.println("Merge Request "+src);
}
TIMEOUT_MATTERS=true;
PROTO_STATE = State.MERGE_SERVER;
allocate(2);
BUFFER.putChar(Message_OPC.MERGE_RESPONSE);
writeMessage(BUFFER, src, source);
LAST_SOURCE = src;
lastMeaningful = System.currentTimeMillis();

      }
      else if(timedOut()){
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
merge_request(src, source);
      }
    }
  }

  public void join_response(long src, InetSocketAddress source){
    synchronized(STATE_LOCK){
      if(
 NODE_STATE == State.JOINING &&
 (PROTO_STATE == State.CALM)
 ){
TIMEOUT_MATTERS=true;
printDebug("Join Response ->",src);
PROTO_STATE = State.JOIN_CLIENT;
SPONSOR_ID = src;

Node n = new Node();
n.id(Reactor.ID);
allocate(n.sizeInBytes()+2);

BUFFER.putChar(Message_OPC.JOIN_CONFIRM);
BUFFER.put(n.getBytes());

writeMessage(BUFFER, src, source);
LAST_SOURCE = src;
lastMeaningful = System.currentTimeMillis();
      }
      else{
allocate(2);
BUFFER.putChar(Message_OPC.JOIN_REJECT);
writeMessage(BUFFER, src, source);
if(timedOut()){
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS = false;
  join_response(src, source);
}
```

```
      }
    }
  }

  public void merge_response(long src, InetSocketAddress source){
    synchronized(STATE_LOCK){
      if(
 NODE_STATE == State.JOINED_G_S &&
 PROTO_STATE == State.CALM
 ){
TIMEOUT_MATTERS=true;
if(Reactor.DEBUG >= 3){
  System.out.println("Merge Response "+src);
}
PROTO_STATE = State.MERGE_CLIENT;
LAST_SOURCE = src;
// Insert the tree, not just the node this time.....
byte[] a = TREE.getBytes();
allocate(a.length+6);
BUFFER.putChar(Message_OPC.MERGE_CONFIRM);
BUFFER.putInt(a.length);
BUFFER.put(a);
writeMessage(BUFFER, src, source);

lastMeaningful = System.currentTimeMillis();

      }
      else{
allocate(2);
BUFFER.putChar(Message_OPC.MERGE_REJECT);
writeMessage(BUFFER, src, source);
if(timedOut()){
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS = false;
  merge_response(src, source);
}
      }
    }
  }

  public void join_confirm(Node n, InetSocketAddress source){
    synchronized(STATE_LOCK){
      if(
 (NODE_STATE == State.JOINING || NODE_STATE == State.JOINED_G_S) &&
 PROTO_STATE == State.JOIN_SERVER
 ){
if(n.id() == LAST_SOURCE){
  printDebug("Join Confirm -> ", n.id());
  byte[] a = TREE.getBytes();
  TEMP_TREE = new Tree(a);
  TEMP_TREE.add(n);
  a = TEMP_TREE.getBytes();
  allocate(a.length + 6);
  System.out.println(TEMP_TREE);
  BUFFER.putChar(Message_OPC.T_STRUCTURE);
  BUFFER.putInt(a.length);
  BUFFER.put(a);
  writeMessage(BUFFER, Message_OPC.BROADCAST, new InetSocketAddress("255.255.255.255", Reactor.PORT));
  PROTO_STATE = State.JOIN_SERVER_PENDING;

  lastMeaningful = System.currentTimeMillis();
}
else if(timedOut()){
  PROTO_STATE = State.CALM;
}
      }
      else if(timedOut()){
PROTO_STATE = State.CALM;
```

```
TIMEOUT_MATTERS = false;
      }
    }
  }

  public void merge_confirm(long src, Tree t, InetSocketAddress source){
    synchronized(STATE_LOCK){
      if(
 NODE_STATE == State.JOINED_G_S &&
 PROTO_STATE == State.MERGE_SERVER
 ){
if(src == LAST_SOURCE){
  if(Reactor.DEBUG >= 3){
    System.out.println("Merge Confirm "+src);
  }
  // Grab the tree transmitted to us and
  // add it to our own...
  TEMP_TREE = new Tree(TREE.getBytes());
  TEMP_TREE.merge(t.NODE_ARRAY);
  byte[] a = TEMP_TREE.getBytes();
  allocate(a.length + 6);
  BUFFER.putChar(Message_OPC.T_STRUCTURE);
  BUFFER.putInt(a.length);
  BUFFER.put(a);
  writeMessage(BUFFER, Message_OPC.BROADCAST, new InetSocketAddress("255.255.255.255", Reactor.PORT));
  PROTO_STATE = State.MERGE_SERVER_PENDING;
  lastMeaningful = System.currentTimeMillis();
}
else if(timedOut()){
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS = false;
}
      }
      else if(timedOut()){
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
      }
    }
  }

  public void join_reject(long src, InetSocketAddress source){
    synchronized(STATE_LOCK){
      if(src == LAST_SOURCE && NODE_STATE < 3){
//printDebug("Join Reject From ",src);
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
      }
    }
  }

  public void merge_reject(long src, InetSocketAddress source){
    synchronized(STATE_LOCK){
      if(src == LAST_SOURCE && NODE_STATE == State.JOINED_G_S && PROTO_STATE != State.CALM){
if(Reactor.DEBUG >= 3){
  System.out.println("Merge Reject "+src);
}
PROTO_STATE = State.CALM;
TIMEOUT_MATTERS = false;
      }
    }
  }

  public void tree_update(Tree t, long id, InetSocketAddress source){
    synchronized(STATE_LOCK){
      if(
 (NODE_STATE == State.JOINING && PROTO_STATE == State.JOIN_CLIENT)
 ){
if(id == SPONSOR_ID){
```

```
  LAST_SOURCE  = id;
  PROTO_STATE = State.JOIN_CLIENT_PENDING;
  TEMP_TREE = t;
  allocate(2);
  BUFFER.putChar(Message_OPC.T_COMMIT);
  writeMessage(BUFFER, id, source);
  printDebug("Joining Tree Update ->",id);
  lastMeaningful = System.currentTimeMillis();
}
else if(System.currentTimeMillis()-lastMeaningful > TIMEOUT){
  PROTO_STATE = State.CALM;
}
      }
      else if(NODE_STATE == State.JOINED_N_G_S && PROTO_STATE == State.CALM){
if(id == SPONSOR_ID){
  LAST_SOURCE =  id;
  PROTO_STATE = State.JOIN_CLIENT_PENDING;
  TEMP_TREE = t;
  printDebug("Plain Tree Update ->",id);
  lastMeaningful = System.currentTimeMillis();
}
else if(System.currentTimeMillis()-lastMeaningful > TIMEOUT){
  PROTO_STATE = State.CALM;
}
      }
      else if (NODE_STATE == State.JOINED_G_S && PROTO_STATE == State.MERGE_CLIENT){
if(id == LAST_SOURCE){
  TEMP_TREE = t;

  // Inform our tree of the new temporary tree and leave...
  if(Reactor.DEBUG >= 2){
  }
  byte[] out = TEMP_TREE.getBytes();
  allocate(out.length+14);
  BUFFER.putChar(Message_OPC.SPONSOR_SWITCH);
  BUFFER.putLong(LAST_SOURCE);
  BUFFER.putInt(out.length);
  BUFFER.put(out);
  writeMessage(BUFFER, Message_OPC.BROADCAST, new InetSocketAddress("255.255.255.255", Reactor.PORT));

  // Inform the merge master to commit
  allocate(2);
  BUFFER.putChar(Message_OPC.T_COMMIT);
  writeMessage(BUFFER, id, source);
  NODE_STATE = State.JOINED_N_G_S;
  lastMeaningful = System.currentTimeMillis();
}
else if(System.currentTimeMillis()-lastMeaningful > TIMEOUT){
  PROTO_STATE = State.CALM;
  lastMeaningful = System.currentTimeMillis();
}
      }
      else if(System.currentTimeMillis()-lastMeaningful > TIMEOUT){
PROTO_STATE = State.CALM;
lastMeaningful = System.currentTimeMillis();
      }
    }

  }

  /* ************************************************************************************ */
  /*                                 End Protocol Function Handlers                       */
  /* ************************************************************************************ */


  /* ************************************************************************************ */
  /*                                    Tree Update Functions                             */
  /* ************************************************************************************ */
```

```
  public void key_tree(Tree t, long id, InetSocketAddress source){
    synchronized(STATE_LOCK){
      if(
 (PROTO_STATE == State.AWAITING_KEY_TREE || PROTO_STATE == State.CALM)&&
 (id == SPONSOR_ID)
 ){
TREE = t;
ELAPSED = (double)(System.currentTimeMillis() - START_TIME);
ELAPSED = ELAPSED / 1000.0;
computeKey(MY_PRIVATE, TREE.indexOf(Reactor.ID));
PROTO_STATE = State.CALM;
      }
    }
  }

  public void t_commit(long id, InetSocketAddress source){
    synchronized(STATE_LOCK){
      if(id == LAST_SOURCE){
lastMeaningful = System.currentTimeMillis();
if(
   (NODE_STATE == State.JOINING && PROTO_STATE == State.JOIN_CLIENT_PENDING) ||
   (NODE_STATE == State.JOINED_N_G_S &&PROTO_STATE == State.MERGE_CLIENT)
   )
  {
    TREE = TEMP_TREE;
    NODE_STATE = State.JOINED_N_G_S;
    PROTO_STATE = State.CALM;
    TIMEOUT_MATTERS=false;
    sponsorUpdate();
  }

else if(
(NODE_STATE == State.JOINING || NODE_STATE == State.JOINED_G_S) &&
PROTO_STATE == State.JOIN_SERVER_PENDING
){
  TREE = TEMP_TREE;
  printDebug("Pushing Commit from ",id);
  allocate(2);
  BUFFER.putChar(Message_OPC.T_COMMIT);
  writeMessage(BUFFER, Message_OPC.BROADCAST, new InetSocketAddress("255.255.255.255", Reactor.PORT));
  NODE_STATE = State.JOINED_G_S;
  PROTO_STATE = State.CALM;

  TIMEOUT_MATTERS=false;
  sponsorUpdate();
}

else if(NODE_STATE == State.JOINED_G_S && PROTO_STATE == State.MERGE_SERVER_PENDING){
  TREE = TEMP_TREE;
  allocate(2);
  BUFFER.putChar(Message_OPC.T_COMMIT);
  writeMessage(BUFFER, Message_OPC.BROADCAST, new InetSocketAddress("255.255.255.255", Reactor.PORT));
  NODE_STATE = State.JOINED_G_S;
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS=false;
  sponsorUpdate();
}
else if (NODE_STATE == State.JOINED_N_G_S){
  TREE = TEMP_TREE;
  printDebug("Applying Commit from ",id);
  PROTO_STATE = State.CALM;
  TIMEOUT_MATTERS=false;
  sponsorUpdate();
}
      }
      else if(timedOut()){
PROTO_STATE = State.CALM;
```

```
TIMEOUT_MATTERS = false;
      }
    }
  }

  /* ************************************************************************************ */
  /*                                    End Tree Functions                                */
  /* ************************************************************************************ */


}
```

5.2.64 v1/TGDH/Node.java

```
import java.math.BigInteger;

public class Node{

  private long ID;
  private byte INTERMEDIARY;
  // Each node in the tree generates a public key through a diffie-hellman exchange.
  public BigInteger PUBLIC_KEY;

  public Node(){
    PUBLIC_KEY = new BigInteger("-1");
    ID = -1;
  }

  public Node(byte[] b){
    java.nio.ByteBuffer bb = java.nio.ByteBuffer.allocate(b.length);
    bb.put(b);
    bb.position(0);

    if(bb.get() == -1){
      ID = -1;
    }
    else{
      ID = bb.getLong();
    }

    int length = bb.getInt();
    byte[] shareval = new byte[length];
    bb.get(shareval);

    PUBLIC_KEY = new BigInteger(shareval);
  }

  public byte[] getBytes(){
    java.nio.ByteBuffer out = java.nio.ByteBuffer.allocate(sizeInBytes());

    // Hack to save space!!!
    if(ID == -1){
      out.put((byte)-1);
    }
    else{
      out.put((byte)1);
      out.putLong(ID);
    }

    byte[] a = PUBLIC_KEY.toByteArray();

    out.putInt(a.length);
    out.put(a);

    byte[] retval = new byte[sizeInBytes()];
    out.position(0);
```

```
    out.get(retval);

    return retval;
  }

  public long id(){
    return ID;
  }

  public void id(long id){
    ID = id;
  }

  public boolean intermediary(){
    return (ID == -1)?true:false;
  }

  public void intermediary(boolean value){
    INTERMEDIARY = ((value==true)?(byte)-1:(byte)1);
  }

  public String toString(){
    String s = new Long(ID).toString();
    return s.substring(((s.length()-7)>=0)?(s.length()-7):0, s.length());
  }

  // 1 byte intermediate flag
  // Possible 8 byte ID
  // 16/17 byte public key
  public int sizeInBytes(){
    int pkl = PUBLIC_KEY.toByteArray().length;
    return (ID == -1)?(pkl+5):(pkl+13);
  }

}
```

5.2.65 v1/TGDH/Reactor.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
 */

import java.nio.channels.DatagramChannel;
import java.nio.channels.Selector;
import java.nio.channels.SelectionKey;
import java.net.InetSocketAddress;

/**
   Class <code>Reactor</code> is the main powerhouse for non-blocking UDP
   datagram broadcasts. It follows a traditional "Reactor" software pattern
   and uses a thread pool to handle the influx of events triggering actions.
 */
public class Reactor implements Runnable{

  /** The port where this <code>Reactor</code> will run.*/
  public static int PORT;

  /** The network Maximum Transmission Unit used in this <code>Reactor</code>.*/
  public static int MTU;

  public static long ID;

  /** The <code>DatagramChannel</code> used as a transport mechanism.*/
  private static DatagramChannel CHANNEL;

  private boolean JUMPSTART;
```

```
public static int MAX_NODES;

/**
    A </code>Selector</code> used to manage the I/O status of <code>SelectionKeys</code>
    in this <code>Reactor</code>, one for bookeeping measures, another for insecure user
    message capabilities.
 */
private Selector SELECTOR;

/** The <code>SelectionKey</code> used for secure group communication I/O bookeeping data. */
private SelectionKey KEY;

public static Reader READER;

public static MessageProcessor PROCESSOR;

public static Node NODE;

private long accumulated;

public static int DEBUG = 5;

public void setDebugLevel(int level){
  DEBUG = level;
}

/** Constructs a new <code>Reactor</code> on port 12333 with a default MTU of 512 bytes. */
public Reactor(){
  this(12333, 512, false, 100);
}

/**
    Constructs a new <code>Reactor</code> on the specified port with a default MTU of 512 bytes.
    @param port The port to bind the <code>Reactor</code>'s <code>DatagramChannel</code> on.
*/
public Reactor(int port){
  this(port, 512, false, 100);
}

/**
   Constructs a new <code>Reactor</code> on the specified port and MTU.
   @param port The port to bind the <code>Reactor</code>'s <code>DatagramChannel</code> on.
   @param mtu The Maximum Transmission Unit used on the underlying network media.
 */
public Reactor (int port, int mtu){
  this(port, mtu, false, 100);
}

public Reactor(int port, int mtu, boolean jumpstart, int max){
  MAX_NODES = max;
  PORT=port;
  MTU=mtu;
  JUMPSTART=jumpstart;
  ID = (long)(Math.random()*1000000000000000.0);
  NODE = new Node();
  NODE.id(ID);
  accumulated = 0;
  DEBUG=3;
}

/**
   Runs this <code>Reactor</code>.  First, all channels are initialized to be non-blocking,
   then an infinite poll-loop is entered to facilitate reaction-when-necessary event handling.
 */
public void run(){
  if(DEBUG == 5){
    System.out.print("Reactor initializing...");
```

```
    }
    initialize();
    if(DEBUG == 5){
      System.out.println("done!");
    }
    if(!JUMPSTART){
      // Attach a new discoverer here...
      PROCESSOR.NODE_STATE = State.JOINING;
      PROCESSOR.PROTO_STATE = State.CALM;
      if(DEBUG >= 3){
System.out.println("[Reactor]:Probing...");
      }
      PROCESSOR.join_probe();
      KEY.attach(READER);
      KEY.interestOps(KEY.OP_READ);
    }
    else{
      PROCESSOR.NODE_STATE = State.INIT_ONLY;
      PROCESSOR.PROTO_STATE = State.CALM;
      // Otherwise wait for the jumpstart message to propogate to us...
      if (DEBUG >= 3){
System.out.println("Reader Attached!");
      }
      KEY.attach(READER);
      KEY.interestOps(KEY.OP_READ);
    }
    poll();
  }

  /**
     Initializes this <code>Reactor</code>. Channels and selectors are opened and configured for
     non-blocking I/O, and keys are set for reading until later notified otherwise.
   */
  private void initialize(){
    try{
      CHANNEL = DatagramChannel.open();

      // This line allows for many instances of this
      // class to run on the same machine...
      CHANNEL.socket().setReuseAddress(true);
      CHANNEL.socket().setBroadcast(true);
      CHANNEL.configureBlocking(false);
      /* Bind to the broadcast address on the given port */
      CHANNEL.socket().bind(new InetSocketAddress("0.0.0.0", PORT));
      SELECTOR = Selector.open();
      KEY = CHANNEL.register(SELECTOR, SelectionKey.OP_READ);
      /* Create the message processor using out arguments */
      READER = new Reader(KEY);
      PROCESSOR = new MessageProcessor(MTU, JUMPSTART, KEY, NODE);
      // If we are to not to be jumpstarted,
    }
    catch(Exception e){
      System.out.println("failed!\n\nError occured during initialization:");
      e.printStackTrace();
      System.exit(-1);
    }
  }

  /**
     The main poll loop.  The native hardware of the system is polled to see
     if any I/O bound for this <code>Reactor</code> is ready.  If so, a set
     of <code>SelectionKey</code>s are returned and iterated through, processing
     the I/O events in turn by separate thread-pool threads for maximum event-handling
     throughput.
   */
  private void poll(){
    if(DEBUG >= 3){
      System.out.println("Beginning poll loop...");
```

```
    }
    while(!Thread.currentThread().interrupted()){
      try{
if
  (
   (
    PROCESSOR.NODE_STATE == State.JOINING ||
    PROCESSOR.NODE_STATE == State.JOINED_G_S
    ) &&
   PROCESSOR.PROTO_STATE == State.CALM
   )
  {
    SELECTOR.select((int)(Math.random()*100000.0)%1000);
  }
else if(PROCESSOR.PROTO_STATE != State.CALM){
  SELECTOR.select(1000);
}
else
  {
    SELECTOR.select();
  }
/* ---------------------------------- */
      }
      catch(Exception e){
System.out.print("\nCaught poll interrupt!\nShutting down...");
try{
  KEY.cancel();
  CHANNEL.close();
  SELECTOR.close();
  System.out.println("done!");
  System.exit(-1);
}
catch(Exception f){}
      }
      java.util.Set readyKeys = SELECTOR.selectedKeys();
      java.util.Iterator it = readyKeys.iterator();
      while(it.hasNext()){
SelectionKey sk = (SelectionKey)it.next();
if(sk != null){
  Object o = sk.attachment();
  if(o != null && o instanceof Runnable){
    ((Runnable)o).run();
  }
}
      }
    }
  }

  public static void write(byte[] message, long destination, InetSocketAddress recipient){
    /* Later this needs to be passed to an encoder for breaking into chunks */
    java.nio.ByteBuffer out = java.nio.ByteBuffer.allocate(message.length+16);
    out.putLong(ID);
    out.putLong(destination);
    out.put(message);
    out.position(0);
    int wrote = 0;
    try{
      while(out.position() < out.capacity()-1){
CHANNEL.send(out, recipient);
      }
    }
    catch(Exception e){
      System.out.println("I/O Error on Channel!");
      e.printStackTrace();
    }
  }

}
```

5.2.66 v1/TGDH/Reader.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
*/

import java.nio.channels.SelectionKey;
import java.nio.ByteBuffer;
import java.net.SocketAddress;

public class Reader implements Runnable{

  private final SelectionKey KEY;
  private ByteBuffer BUFFER;
  private boolean EXECUTED;

  public Reader(SelectionKey sk){
    KEY=sk;
    EXECUTED=false;
    BUFFER= ByteBuffer.allocate(Reactor.MTU);
  }

  public void run(){
    try{
      BUFFER.position(0);
      SocketAddress a = ((java.nio.channels.DatagramChannel)(KEY.channel())).receive(BUFFER);
      BUFFER.position(0);

      // Either there is something here, useability undetermined...
      if(a != null){
new Decoder(KEY, BUFFER, (java.net.InetSocketAddress)a).run();
      }

      else if(
      (Reactor.PROCESSOR.NODE_STATE == State.JOINING && Reactor.PROCESSOR.PROTO_STATE == State.CALM)
      ){
if (Reactor.DEBUG >= 3)
  System.out.println("[Reader]:Join Probing...");
Reactor.PROCESSOR.join_probe();
      }

      else if(
      (Reactor.PROCESSOR.NODE_STATE == State.JOINED_G_S && Reactor.PROCESSOR.PROTO_STATE == State.CALM)
      ){
if (Reactor.DEBUG >= 3)
  System.out.println("[Reader]:Merge Probing...");
Reactor.PROCESSOR.merge_probe();
      }

      // Or we have timed out...
      else{
       Reactor.PROCESSOR.setProtoState(State.CALM);
KEY.selector().wakeup();
      }

      KEY.attach(this);
      KEY.interestOps(KEY.OP_READ);
      return;

    }
    catch(Exception e){
      System.out.println("I/O Error while reading!!!!");
      e.printStackTrace();
    }
  }
}
```

5.2.67 v1/TGDH/State.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
 */

public class State{

  /* Node States...*/
  public static final char INIT_ONLY = 0x0001;
  public static final char JOINING = 0x0002;
  public static final char JOINED_N_G_S = 0x0003;
  public static final char JOINED_G_S = 0x0004;

  /* Protocol States...*/

  /* The Join States ... */
  public static final char JOIN_SERVER = 0x0010;
  public static final char JOIN_CLIENT = 0x0011;
  public static final char JOIN_SERVER_CONFIRMED = 0x0012;
  public static final char JOIN_SERVER_PENDING = 0x0013;
  public static final char JOIN_CLIENT_PENDING = 0x0014;

  /* The Merge States */
  public static final char MERGE_SERVER = 0x0020;
  public static final char MERGE_CLIENT = 0x0021;
  public static final char MERGE_SERVER_CONFIRMED = 0x0022;
  public static final char MERGE_SERVER_PENDING = 0x0023;
  public static final char MERGE_CLIENT_PENDING = 0x0024;

  /* States relevant during key update */

  public static final char AWAITING_KEY_TREE= 0x0031;
  public static final char REQUESTING_TREE = 0x0032;
  public static final char UPDATING_KEY = 0x0033;

  /* The Calm State...*/
  public static final char CALM = 0xffff;

}
```

5.2.68 v1/TGDH/Tree.java

```
/*
  This code is Copyright Kieran S. Hagzan. NO unauthorized duplication without permission.
 */

import java.util.Vector;
import java.util.StringTokenizer;

/**
   This class represents a fully-binary tree as specified by Kim, Perrig, and Tsudik in
   <i>Tree-Based Group Key Agreement</i>. Nodes have either 2 or no children exactly,
   and are member nodes if they have no children. The tree is stored in an array, with
   the rules for node <i>n</i>:<br><br>parent nodes are at indices <i>(n-1)/2</i><br>right and left children
   are at <i>(2n)+1</i> and <i>2(n + 1)</i> respectively<br>"trim points" are at non-zero
   indexed nodes satisfying the property <i>2^(depth(n)-1)+2</i><br>The tree is fully balanced
   when the lowest point of insertion is at <i>2^(depth(n))-1</i>.
 */
public class Tree{

  /**
     A vector holding the tree itself.
   */
  public Vector NODE_ARRAY;
```

```
  /**
     A string-representation of the tree, for the printSideways method.
     DO NOT USE!!!
   */
  private String TREE_STR = "";

  /**
     Creates a new, empty tree.
   */
  public Tree(){
    NODE_ARRAY = new Vector(0);
  }

  public Vector toVector(){
    return NODE_ARRAY;
  }

  public int nodeCount(){
    int count = 0;
    for(int i = 0; i < NODE_ARRAY.size(); i++){
      Node n = ((Node)NODE_ARRAY.elementAt(i));
      if (n != null && n.id()>=0){
count++;
      }
    }
    return count;
  }

  public static void main(String[] args){
    java.io.BufferedReader br = new java.io.BufferedReader(new java.io.InputStreamReader(System.in));
    Tree t = new Tree();
    while(true){
      System.out.print(">>> ");
      String s = "";
      try{
s = br.readLine().toLowerCase();
      }
      catch(Exception e){}
      java.util.StringTokenizer tok = new java.util.StringTokenizer(s);
      String directive = tok.nextToken();
      int op = 0;
      if(directive.indexOf("quit") >= 0){System.exit(1);}
      else if(directive.indexOf("rem") >= 0){
op = 1;
      }
      else if(directive.indexOf("add") >= 0){
op = 2;
      }
      else if(directive.indexOf("part") >= 0){
op = 3;
      }
      try{
final int opc = op;
String quant = tok.nextToken();
int quantity = Integer.parseInt(quant);
switch(opc){
case 3:
  System.out.println("Partitioning at array index "+quantity+"...");
  t.partition(quantity);
  break;
case 2:
  System.out.println("Adding "+quantity+" node(s)...");
  for(int i = 0; i < quantity; i++){
    Node o = new Node();
    o.id((int)(Math.random()*10000000.0));
    t.add(o);
  }
```

```
  break;
case 1:
  System.out.println("Removing node "+quantity+"...");
  t.remove(quantity);
  break;
}
System.out.println(t);
System.out.println("Sponsor:"+t.getSponsor(0));
      }
      catch(Exception e){e.printStackTrace();}
    }
  }

  public byte[] getBytes(){
    synchronized (NODE_ARRAY){
      int length = 0;
      java.nio.ByteBuffer out = java.nio.ByteBuffer.allocate(Reactor.MTU);
      for(int i = 0; i < NODE_ARRAY.size(); i++){
Node n = (Node)NODE_ARRAY.elementAt(i);
if(n == null){
  out.put((byte)-1);
  length++;
}
else{
  out.put((byte)1);
  length++;
  out.put(n.getBytes());
  length+=(n.sizeInBytes());
}
      }
      byte[] retval = new byte[length];
      out.position(0);
      out.get(retval);
      return retval;
    }
  }

  public Tree(byte[] src){
      java.nio.ByteBuffer in = java.nio.ByteBuffer.allocate(src.length);
      in.put(src);
      in.position(0);
      Vector v = new Vector();
      while(in.position() < in.limit()){
byte nul = in.get();
if(nul != (byte)-1){

  // Construct the node as it was destructed to accomodate
  // The space saving hack....
  Node n;
  if(in.get() == (byte)-1){
    n = new Node();
    n.id(-1);
    int len = in.getInt();
    byte[] pk = new byte[len];
    in.get(pk);
    n.PUBLIC_KEY = new java.math.BigInteger(pk);
    v.add(n);
  }
  else{
    n = new Node();
    long id = in.getLong();
    n.id(id);
    int len = in.getInt();
    byte[] pk = new byte[len];
    in.get(pk);
    n.PUBLIC_KEY = new java.math.BigInteger(pk);
    v.add(n);
  }
```

```
}
else v.add(null);
      }
      NODE_ARRAY = v;
  }

  public Tree(Vector src){
      NODE_ARRAY = src;
  }

  /* ----------------------------------------------------------------------- */
  /* Functions exploiting the mathematical properties of array-based trees.*/
  /* ----------------------------------------------------------------------- */

  public int indexOf(long id){
    for(int i = NODE_ARRAY.size()-1; i >=0; i--){
      Node n = (Node)NODE_ARRAY.elementAt(i);
      if(n != null){
if(n.id() == id) return i;
      }
    }
    System.out.println("[Tree] Meef???");
    System.exit(-1);
    return -1;
  }


  public long getSponsor(int index){
    int rightChild = rightChildOf(index);
    int leftChild = leftChildOf(index);
    if(NODE_ARRAY.size() > rightChild && NODE_ARRAY.elementAt(rightChild) != null){
      return getSponsor(rightChild);
    }
    else if(NODE_ARRAY.size() > leftChild && NODE_ARRAY.elementAt(leftChild) != null){
      return getSponsor(leftChild);
    }
    else{
      return ((Node)NODE_ARRAY.elementAt(index)).id();
    }
  }

  /**
     Returns the current depth of the tree.
     @return The current depth of the  tree.
   */
  private int depth(){
    return depth(NODE_ARRAY.size());
  }


  /**
     Returns the current depth of the specified index in the vector holding the tree.
     @return The current depth of the specified index in the vector holding the tree.
     @param index The index of the node to compute on.
   */
  private int depth(int index){return (int)(Math.ceil((Math.log(index+2)/Math.log(2))-1));}


  /**
     Returns the left child index of the specified index in the vector holding the tree.
     @return The left child index of the specified index in the vector holding the tree.
     @param index The index of the node to compute on.
   */
  public int leftChildOf(int index){return (2*index)+1;}


  /**
```

```
     Returns the right child index of the specified index in the vector holding the tree.
     @return The right child index of the specified index in the vector holding the tree.
     @param index The index of the node to compute on.
   */
  public int rightChildOf(int index){return 2*(index+1);}

  /**
     Returns the parent index of the specified index in the vector holding the tree.
     @return The parent index of the specified index in the vector holding the tree.
     @param index The index of the node to compute on.
   */
  public int parentOf(int index){
    if(index <= 0)return -1;
    return (index-1)/2;
  }

  /**
     Returns the point at which a current computation involving promotion and rearrangemet
     to an odd-index valued node should trim the subtree to rearrange.
     @return The point at which a current computation involving promotion and rearrangemet
     to an odd-index valued node should trim the subtree to rearrange.
     @param index The index of the node to compute on.
   */
  private int trimPoint(int insert){
    int retval = insert;
    // Ahh a beautiful mathematical property fulfilled again....
    // Keep iterating through "lineage" (parent()  method) until
    // the index of that parent satisfies the following equation,
    // representative of the fact that we at a node in the "rightmost"
    // branch from the root.
    while(retval != (int)(Math.pow(2, depth(retval)+1)-2)){
      retval = parentOf(retval);
    }
    return retval;
  }

  public Vector collect(Vector src, Vector dest, int index){
    if(index > src.size()-1)return dest;
    if(src.elementAt(index) instanceof Node){
      Node n = (Node)src.elementAt(index);
      if(!n.intermediary()){
dest.add(n);
      }
    }
    dest = collect(src, dest, leftChildOf(index));
    dest = collect(src, dest, rightChildOf(index));
    return dest;
  }

  public Vector collectAll(Vector dest, int destIndex, int position){
    // Insert this node (It must be there if we got here...
    Object o = NODE_ARRAY.elementAt(position);
    addNode(((o==null)?null:o), dest, destIndex);
    // Collect the left children if they exist...
    if(leftChildOf(position) < NODE_ARRAY.size()){
      dest = collectAll(dest, leftChildOf(destIndex), leftChildOf(position));
    }
    // Collect the right children if they exist...
    if(rightChildOf(position) < NODE_ARRAY.size()){
      dest = collectAll(dest, rightChildOf(destIndex), rightChildOf(position));
    }
    return dest;
  }

  /* ------------------------------------------------------------------- */
  /*                          Bulk Add Function                          */
  /* ------------------------------------------------------------------- */
```

```
  public boolean merge(Vector tree){
    boolean retval = false;
    synchronized(NODE_ARRAY){
      Vector temp = new Vector();
      temp = collect(tree, temp, 0);
      for(int i = 0; i < temp.size(); i++){
add((Node)temp.elementAt(i));
      }
      retval=true;
    }
    return retval;
  }

  /* ------------------------------------------------------------------------ */
  /*                           Bulk Remove Function                           */
  /* ------------------------------------------------------------------------ */

  public boolean partition(int index){
    boolean retval = false;
    synchronized(NODE_ARRAY){
      Vector temp = new Vector();
      temp = collect(NODE_ARRAY, temp, index);
      for(int i = 0; i < temp.size(); i++){
remove(((Node)temp.elementAt(i)).id());
      }
      retval=true;
    }
    return retval;
  }

  /* ------------------------------------------------------------------------ */
  /*                       Functions to Add to the Tree                       */
  /* ------------------------------------------------------------------------ */

  /**
     Adds the specified node to the specified index in the specified vector.
     Room is made in the Vector if it is not big enough to hold the added
     element.
     @param o The object to add to this tree.
     @param v The vector holding the tree.
     @param index The index in the vector of the node to add to the tree.
   */
  private void addNode(Object o, Vector v, int index){
    if(index >= v.size())v.setSize(index+1);
    v.set(index, o);
  }

  /**
     Appends a subtree of the source tree to the destination tree at the given indices
     by calling itself recursively.
     @param src A vector containing the source subtree.
     @param dest A vector containing the destination subtree.
     @param srcIndex The index in the source vector of the root of the subtree to add.
     @param destIndex The index in the destination where the root of the added subtree
     should be appended.
   */
  public void addSubtree(Vector src, Vector dest, int srcIndex, int destIndex){
    if(srcIndex >= src.size())return;
    else{
      addNode(src.elementAt(srcIndex), dest, destIndex);
      addSubtree(src, dest, leftChildOf(srcIndex), leftChildOf(destIndex));
      addSubtree(src, dest, rightChildOf(srcIndex), rightChildOf(destIndex));
    }
  }

  /**
      "Kills," or nullifies elements in the tree contained in the specified vector
      beginning at the specified index recursively.
```

```
      @param src The source vector holding the tree.
      @param index The index in the vector of the root of the subtree to nullify.
   */
  private void killSubtree(Vector src, int index){
    if(index >= src.size() || src.elementAt(index) == null)return;
    else{
      addNode(null, src, index);
      killSubtree(src, leftChildOf(index));
      killSubtree(src, rightChildOf(index));
    }
  }

  /**
     Breadth-first leftmost search for this tree to determine optimal insertion point.
   */
  private int lowest(int index){
    if(index >= NODE_ARRAY.size())return Integer.MAX_VALUE;
    else if(NODE_ARRAY.elementAt(index) == null)return index;
    else{
      int left = lowest(leftChildOf(index));
      int right = lowest(rightChildOf(index));
      if(right > left)return left;else return right;
    }
  }

  /**
    Adds an object to this tree according to the protocol specified by
    Kim, Perrig, and Tsudik. The tree is maintained to be fully binary,
    such that all nodes have either zero (member nodes) or two (intermediary nodes)
    children.
    @param o The object to add, can be null for testing purposes.
   */
  public void add(Node o){
    // Lock the NODE_ARRAY structure against remove...
    synchronized(NODE_ARRAY){
      o.intermediary(false);

      // Prepare a new intermediary node, it is most likely needed...
      Node inter = new Node();
      inter.intermediary(true);

      int lowest = lowest(0);

      // If the tree array was full, we need to insert at the next available
      // array slot...
      if(lowest == Integer.MAX_VALUE)lowest = NODE_ARRAY.size();

      // Otherwise, if the lowest is 0, the tree is empty, so just insert...
      if(lowest == 0){
NODE_ARRAY.add(o);
      }

      // Else, determine if the depth of the tree is to be increased,
      // if so, insert at the root...we know this when the tree is fully
      // balanced, or the following equation holds...
      else if(// The index must be one the edge
      lowest == ((int)Math.pow(2, depth(lowest)))-1 &&
      //and greater than the current size
      lowest > NODE_ARRAY.size()-1
      ){
Vector temp = new Vector();
temp.add(0, inter);
addSubtree(NODE_ARRAY, temp, 0, 1);
addNode(o, temp, 2);
NODE_ARRAY = temp;
      }

      // If the depth is not to be increased, but we need to add another leaf,
```

```
      // then we need to promote and rearrange
      else{
// Create a new temporary tree...
Vector temp = (Vector)NODE_ARRAY.clone();

// If the point of insertion's parent is even indexed,
// Simply make this node a right-sibling of it and replace
// it's former location with an intermediate
if(parentOf(lowest)%2 == 0){
  Object parent = NODE_ARRAY.elementAt(parentOf(lowest));
  addNode(inter, temp, parentOf(lowest));
  addNode(parent, temp, lowest);
  addNode(o, temp, lowest+1);
}

// Otherwise we have to move the first "outermost" subtree
// to a new intermediary nodes left child, and
// insert this node at the intermediary's right sibling
else{
  if(lowest > NODE_ARRAY.size()-1){
    // This is the point to insert at, see method trimPoint();
    int tp = trimPoint(lowest);
    int untrim = leftChildOf(tp);
    int insertPoint = rightChildOf(tp);
    killSubtree(temp, tp);
    addNode(inter, temp, tp);
    addSubtree(NODE_ARRAY, temp, tp, untrim);
    addNode(o, temp, insertPoint);
  }
  else{
    if(lowest < NODE_ARRAY.size() -1 && NODE_ARRAY.elementAt(lowest) == null){
      // Replace parent with new intermediary and make parent sibling of added node
      Node n = new Node();
      n.intermediary(true);
      Object parent = NODE_ARRAY.elementAt(parentOf(lowest));
      temp.set(parentOf(lowest), n);
      temp.set(lowest, o);
      temp.set(rightChildOf(parentOf(lowest)), parent);
    }
  }
}
NODE_ARRAY = temp;
      }
    }
  }

  /* ------------------------------------------------------------------------ */
  /*                   Functions to Remove from the Tree                      */
  /* ------------------------------------------------------------------------ */

  /**
     Removes the leaf node at the specified index.
   */
  public boolean remove(long id){
    // Lock the NODE_ARRAY structure against add...
    synchronized(NODE_ARRAY){
      for(int index = NODE_ARRAY.size()-1; index >=0; index--){
Node n = (Node)NODE_ARRAY.elementAt(index);
if(n != null && n.id() == id){
  Vector temp = null;
  NODE_ARRAY.set(index, null);
  // Promote this node's parent up one level...
  // If this node's parent was root, root's other child is now root
  if(parentOf(index) == 0){
    temp = new Vector();
    if(leftChildOf(0) == index){
      addSubtree(NODE_ARRAY, temp, rightChildOf(0), 0);
    }
```

```
    else{
      addSubtree(NODE_ARRAY, temp, leftChildOf(0), 0);
    }
    NODE_ARRAY = temp;
    return true;
  }

  else if(parentOf(index) >0){
    // Make new Clone of the Tree...
    temp = (Vector)NODE_ARRAY.clone();
    int grandparent = parentOf(parentOf(index));
    // Kill the tree at the grandparent...
    killSubtree(temp, grandparent);
    // Add The subtree from this node's parent to the grandparent...
    addSubtree(NODE_ARRAY, temp, parentOf(index), grandparent);
    // Determine if the parent of this node was the left child of the grndparent
    boolean parentLeftChild = (parentOf(index) == leftChildOf(grandparent));
    boolean thisLeftChild = (index == leftChildOf(parentOf(index)));
    if(parentLeftChild){
      if(thisLeftChild){
addSubtree(NODE_ARRAY, temp, rightChildOf(grandparent), leftChildOf(grandparent));
      }
      else{
addSubtree(NODE_ARRAY, temp, rightChildOf(grandparent), rightChildOf(grandparent));
      }
    }
    else{
      if(thisLeftChild){
addSubtree(NODE_ARRAY, temp, leftChildOf(grandparent), leftChildOf(grandparent));
      }
      else{
addSubtree(NODE_ARRAY, temp, leftChildOf(grandparent), rightChildOf(grandparent));
      }
    }
    NODE_ARRAY = temp;
    return true;
  }

}
      }
      return false;
    }
  }

  /* ------------------------------------------------------------------------ */
  /*                   Functions to ASCII-print the Tree.                     */
  /* ------------------------------------------------------------------------ */

  /**
     Returns a copy of this tree as a human-readable String.
     @return A copy of this tree as a human-readable String.
   */
  public String toString(){
    String retval = "";
    retval += "Array View:\n-----------\n|";
    for(int i = 0; i < NODE_ARRAY.size(); i++){
      Object o = NODE_ARRAY.elementAt(i);
      String id = ((o==null) ? "X" : ((Node)o).intermediary() ? " I " : ((Node)o).toString());
      retval += (id+"|");
    }
    retval += "\n\n"+"Tree View:\n----------\n"+(this.filter())+"\n";
    return retval;
  }

  /**
     Prints the tree in human-readable ascii format.
     @return A String representing a human-readable ascii-version of the current tree.
   */
```

```
  private String printTree(){
    String retval = "";
    retval += ("\n\nTree:\n-----\n");
    /* Since printSideways(...) is recursive, we need this little global
       variablle hack...
     */
    TREE_STR="";
    printSideways(0,7,0);
    return TREE_STR;
  }

  /**
     A recursive method to print the tree rotated 90* counter-clockwise.  One more pass
     through flip() and the tree is human-readable.  These two passes are proven through
     graphics theory to be the minimal number of passes needed to accomplish this task.
     @param node The node in the tree to begin with during traversal.
     @param space The space to use around a particular level's elements.
     @param inc The increment total of spaces to be added around a depth level's elements per
     iteration.
   */
  private void printSideways(int node, int inc, int space){
    if(node < 0 || node >= NODE_ARRAY.size() || NODE_ARRAY.elementAt(node) == null){return;}
    printSideways(rightChildOf(node), inc, space+inc);
    for(int i= 0; i < space; i++)
      TREE_STR += "_";
    Node n = (Node)NODE_ARRAY.elementAt(node);
    if(n.intermediary())TREE_STR += "__/X\\__\n";
    else{
      String s = n.toString();
      while(s.length() < 7){s = "0"+s;}
      s += "\n";
      TREE_STR+=s;
    }
    printSideways(leftChildOf(node), inc, space+inc);
  }


  /**
     Converts it's argument to a String[], of whom all entries
     are padded with whitespace to the same length. (I.E., make
     Java version of a a C-Style char[][] (character matrix) that
     is regular.
     @param tree A tree-representation from the print-tree method.
   */
  private String[] arrayize(String tree){
    StringTokenizer toker = new StringTokenizer(tree, "\n", false);
    String [] out = new String[toker.countTokens()];
    int count = 0;
    int maxLength = -1;
    while(toker.hasMoreTokens()){
      String s = toker.nextToken();
      if(s.length() > maxLength)maxLength = s.length();
      out[count++] = s;
    }

    // Pad to the max length
    for(int i = 0; i < out.length; i++){
      String s = out[i];
      while(s.length() < maxLength){
s +=" ";
      }
      out[i] = s;
    }
    return out;
  }

  /**
     Flips it's argument 90* clockwise. The argument must be an array'ized tree
```

```
     representation from arrayize() and printTree respectively, or the output
     will be goook.
     @param tree A String[] from the arrayize method.
   */
  private String flip(String[] tree){
    String retval = "";
    if(tree.length == 0)throw new IllegalArgumentException("Tree size must be greater than zero!");
    for(int i = 0; i < tree[0].length(); i+=7){
      for(int j = tree.length-1; j >=0; j--){
String substring = tree[j].substring(i, i+7);
retval += substring;
      }
      retval += "\n";
    }
    return retval;
  }

  /**
     Returns a "graphically filtered" (ASCII) version of the tree (more readable).
     @return A "graphically filtered" (ASCII) version of the tree (more readable).
   */
  private String filter(){
    String retval = "";
    String[] a = this.arrayize(this.flip(this.arrayize(this.printTree())));
    for(int i = 0; i < a.length-1; i++){
      String current = a[i];
      boolean underscore = false;
      for(int j = 0; j < current.length(); j+=7){
String chunk = current.substring(j, j+7);
String underchunk = a[i+1].substring(j, j+7);
// The following cases exhaust all possibilities of input passed to us
// This suffices for debugging purposes until a better algorithm is
// found...
if(!underscore && underchunk.equals("__/X\\__")){
  underscore = true;
  retval += "_______";
}
else if(chunk.equals("_______") && !underscore && underchunk.equals("_______")){
  retval += "       ";
}
else if(chunk.equals("_______") && underscore && underchunk.equals("_______")){
  retval += "_______";
}
else if(chunk.equals("__/X\\__")){
  underscore = true;
  retval += "__/X\\__";
}
else if (underscore && underchunk.equals("__/X\\__")){
  underscore = false;
  retval += "_______";
}
else if (chunk.equals("       ")){
  retval += "       ";
}
else if ( java.util.regex.Pattern.matches("[0-9]+", chunk) ){
  retval += chunk;
}
else if ( chunk.equals("_______") && java.util.regex.Pattern.matches("[0-9]+", underchunk)){
  retval += "_______";
}
else{
  // This should never happen here...
}
      }
      retval += "\n";
    }
    retval += a[a.length-1]+"\n";
    return retval;
```

```
  }

}
```

References

B. Adamson, C. Bormann, S. Floyd, M. Handley, and J. Macker. Nack-oriented reliable multicast protocol (norm), ietf internet draft, 2000.

Yair Amir, Yongdae Kim, Cristina Nita-Rotaru, John Schultz, Jonathan Stanton, and Gene Tsudik. Exploring robustness in group key agreement. In *21st IEEE International Conference on Distributed Computing Systems*, April 2001.

Yair Amir, Yongdae Kim, Christina Nita-Rotaru, and Gene Tsudik. On the performance of group key agreement protocols. In *22nd IEEE International Conference on Distributed Computing Systems*, June 2002.

Yair Amir, Cristina Nita-rotaru, Jonathan Stanton, and Gene Tsudik. Scaling secure group communication systems: Beyond peer-to-peer. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX'03)*, 2003. URL `citeseer.ist.psu.edu/573628.html`.

Klaus Becker and Uta Wille. Communication complexity of group key distribution. In *ACM Conference on Computer and Communications Security*, pages 1–6, 1998. URL `citeseer.ist.psu.edu/becker98communication.html`.

Yongdae Kim, Adrian Perrig, and Gene Tsudik. Tree-based group key agreement. Technical Report 2002/009, February 2002.

P. Lee, J. Lui, and D. Yau. Distributed collaborative key agreement protocols for dynamic peer groups. In *22nd IEEE International Conference on Distributed Computing Systems*. IEEE Press, July 2002.

David A. McGrew and Alan T. Sherman. Key establishment in large dynamic groups using one-way function trees. In *18th International Conference on Distributed Computing Systems*, May 1998.

Katia Obraczka, Gene Tsudik, and Kumar Viswanath. Pushing the limit of multicast in ad hoc networks. In *21st International Conference on Distributed Computing Systems*, April 2001.

Michael Steiner, Gene Tsudik, and Michael Waidner. Diffie-hellman key distribution extended to groups. In *ACM Conference on Computer and Communications Security*, pages 31–37, March 1996.

Michael Steiner, Gene Tsudik, and Michael Waidner. Cliques: A new approach to group key agreement. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS'98)*, pages 380–387. IEEE Computer Society Press, May 1998. URL `citeseer.ist.psu.edu/steiner98cliques.html`.

Wade Trappe and Lawrence C. Washington. *Introduction to cryptography: with coding theory*. 2002. ISBN 0-13-061814-4.