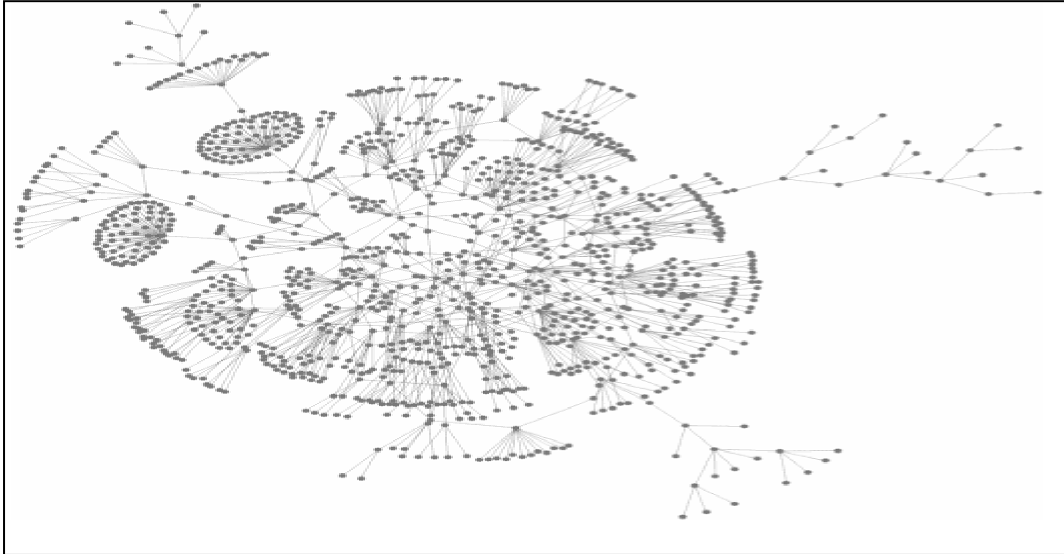
CHALMERS



Robust Overlay networks for Volunteer Computing

Decentralized Volunteer Computing Architecture with Fault-tolerance Design

Master of Science Thesis in Secure and Dependable Computer Systems

HAO NING

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineerings
Göteborg, Sweden, 2010

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Robust Overlay networks for Volunteer Computing
Decentralized Volunteer Computing Architecture with Fault-tolerance Designs
NING. HAO,

© NING. HAO, 2010.

Examiner: Marina, Papatriantafilou

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

[Cover:
Multi-level Dispatching Tree Structure for Volunteer Computing]

Department of Computer Science and Engineering
Göteborg, Sweden 2010

Abstract

The potential vulnerability of Centralized Volunteer Computing systems is the single point of failure, which can be triggered by attacks on malicious purpose, or even by the normal use which may lead to the resource exhaustion on the central server. To originally eliminate this vulnerability, a variety of decentralized architecture designs have been proposed for Volunteer Computing. These solutions widely adopted some decentralized design models, such as hierarchical design with super peers, and equal peer design in form of peer-to-peer networks. Regardless of the variation among them, certain vulnerabilities still exist due to the decentralization of the architecture, which makes the architecture less robust when network size changes fast due to volunteers' join and leave. These proposals then also provided certain fault-tolerant mechanisms to against these vulnerabilities of the decentralized architecture, in order to improve the robustness of the system. But none of them paid attention on the locality of influence of the system changes, which should be taken into account as a key factor of the fault-tolerance design to strengthen the decentralized architecture of the system.

This thesis proposes a decentralized architecture design, which originally eliminates the single point of failure hazard, and breaks through the bottleneck of resource exhaustion. The proposed design is in form of a layered system: it uses the principle of MapRecude to construct a multi-level tree structure at the top layer. This structure is decentralized by distributing also the management and control of Volunteer Computing to volunteers. But it has a vulnerability that makes the tree structure fragile. To improve the robustness of the system, this thesis constructs a second layer – a peer-to-peer overlay network layer, coupling it with the top layer for fault-tolerance, and the construction of the second layer is guided by the consideration about the locality of influence of the system changes. A prototype has been built up where some simulations are launched to study the fault-tolerance of this decentralized system design for Volunteer Computing.

Key words: Volunteer Computing, Desktop Grid, MapReduce, Peer-to-Peer Networks

// Acknowledgements

Here I would like to thank my examiner, Professor Marina Papatriantafilou, and her Ph.D. Students Georgios Georgiadis and Zhang Fu for their direction, guidance and assistance during my master thesis work. They not only directed and guided me on the content of this thesis, but much important also on the attitude and the methodology of research. They spent their time and patience to enlighten me of the research with a plenty of valuable and professional suggestions that did broaden my mind. Thank them for all the things they have done during the last few months for my master thesis work.

Here I also would like to thank all my friends for their supports during my thesis work.

Contents

1. Introduction ....5.

1.1. Existing Application Architecture for Volunteer Computing...........6.

1.2. Research Question...7.

2. Backgound..9.

2.1. Relevant Concepts...9.

2.1.1 MapReduce ....9.

2.1.2 Chord Protocol ....9.

2.2. Related Works ....11.

3. System Model ....13.

3.1. MapReduce Relevant...13.

3.2. Peer-to-Peer Overlay Network Relevant ....13.

3.3. System Relevant ....14.

4. Network Application Architecture Design ....15.

4.1. Design Overview..15.

4.2. A Simple Approach for Top Layer Architecture Design..................16.

4.3. Vulnerability of Top Layer Architecture Design...............................18.

4.4. The Second layer Design for The Robustness of Top Layer Architecture 19.

4.5. Fault-tolerance of The Layered System Design ....22.

4.5.1 Distribution Policy ....22.

4.5.2 Analysis of Setting Distribution Policy ....23.

4.5.3 Reconnection..25.

5. Simulations...28.

5.1. Simulation Tools and Prototype ....28.

5.1.1 Simulation Tools ....28.

5.1.2 Prototype..29.

5.2. Simulation Design .... 29.
5.2.1 Reconnecting Operations .... 30.
5.2.2 Setting Distribution Policies .... 31.
5.3. Configuration .... 33.
5.3.1 Parameters .... 33.
5.3.2 Considerations .... 33.
5.4. Simulation One: Study The Success of Fault Tolerance of The System 34.
5.4.1 Configuration .... 34.
5.4.2 Simulation Results .... 36.
5.4.3 Analysis .... 38.
5.5. Simulation Two: Study The Tradeoff of Fault Tolerance of The System 38.
5.5.1 Configuration .... 39.
5.5.2 Simulation Results .... 39.
5.5.3 Analysis .... 41.
6. Conclusion .... 43.
Appendix .... 44.
Bibliography .... 111.

1. Introduction

Volunteer Computing [1] as an attractive branch of Desktop Grid Computing, has been well researched in papers and also developed in the real market to feed the great demand mainly residing in academics. Volunteer Computing regards each public computational resource provider as a volunteer – i.e. an individual Desktop Computer, a laptop, or even a mobile phone – where possess spare CPU utilization, free memory, free disk space and network bandwidth, etc. then voluntarily contribute these capacities into data-intensive computations. The growing attractiveness of Volunteer Computing systems is promoted by several aspects:

I. ***Balance the utilization of the public computational resources***. Under the circumstance that individual computation devices are new-generating in a high speed and density, and are widely spreaded around the world, the existence of public computational devices can be viewed as a global pool full of computation power from every geographical location. Actually the resource utilization in this pool is varied geographically and over time, spare resources may exist at any place in any second but those free capacity may be idle for a long time and hidden behind the box; at the meanwhile, some computing devices are overwhelmed by data processing, which is just at the right place where an appropriate off-the-peg approach should do its job to mitigate this unexpected situation. Volunteer Computing is on hand, it can be a good solution to balance the global resource utilization as well as against the waste of computational resources.

II. ***The mature and robust Internet technologies enable the sharing activity among devices, and organize resources under different models, e.g. Centralized model (Client-Server), or Decentralized model (Peer-to-Peer)***. The latter one enables the distribution of resources based on the communication and the cooperation.

III. ***Academic research areas, such as gene analysis, weather prediction, are based on and bound to huge volumes of data. Therefore giving rise to the data-intensive computing.*** Those demands have gained stressed attentions and can be satisfied by the blossom of the IT technologies, the concept of Volunteer Computing hits the answer. When volunteer computing is used for academic research, it also drives the public attention to the related academic issues, which were limited only inside the academia before.

1.1 Existing Application Architecture for Volunteer Computing

Existing Volunteer Computing platform in running, well known as BOINC [2], a middleware for Volunteer Computing, under MapReduce [3] framework, is in a **centralized design**(figure 1-a): a bunch of data-intensive projects are registered on the server, then be partitioned and distributed among volunteers who are interested in and qualified for offering their own computational resources for data processing. Now over 60 volunteer computing projects are running on BOINC, such as AQUA@HOME for quantum computing, CHESS960@HOME for game playing, ABC@HOME for mathematics, etc. Most of them are academic oriented, and the volunteers working for some projects are from even more than 200 different countries. The average floating point operations per second achieved at 3.269 PFLOPS as of Oct 26, 2010 [4]. In spite of these achievements, BOINC faces some challenges, among which, ***the unstable clients*** and the ***high server load*** are stressed here. The former one is due to the voluntariness of Volunteer Computing, and the latter one is due to the centralized design of BOINC. And it should not forget the third one, which is also the most natural disadvantage due to the centralized architecture – ***the single point of failure***. BOINC has done a good job to secure its servers by some conventional security mechanisms, such as setting firewalls, using SSH, etc. but no guarantee that the single point of failure has gone forever as long as it is in the centralized architecture design. Therefore, removing the existence of the central server becomes the consideration. A plenty of **decentralized designs** then are proposed based on this consideration, they are mainly in two categories: ***Collaborative Desktop Grid*** with super peers (in figure 1-b) and ***Peer-to-Peer Desktop Grid*** as in PastryGrid [5] with a flat architecture consisting of equal peers (in figure 1-c). There are also many research proposals trying to design a layered structure to avoid the existence of the central server by adopting peer-to-peer overlay network technology, as P2P-Tuple, PPVC, but none of them is on the state of the art, when combining of Volunteer Computing and distributed network technology, they didn't take into accout the locality of the influence of system changes, which makes the system less robust and reliable especially when the network size is quite large and changes fast due to volunteers' join and leave.

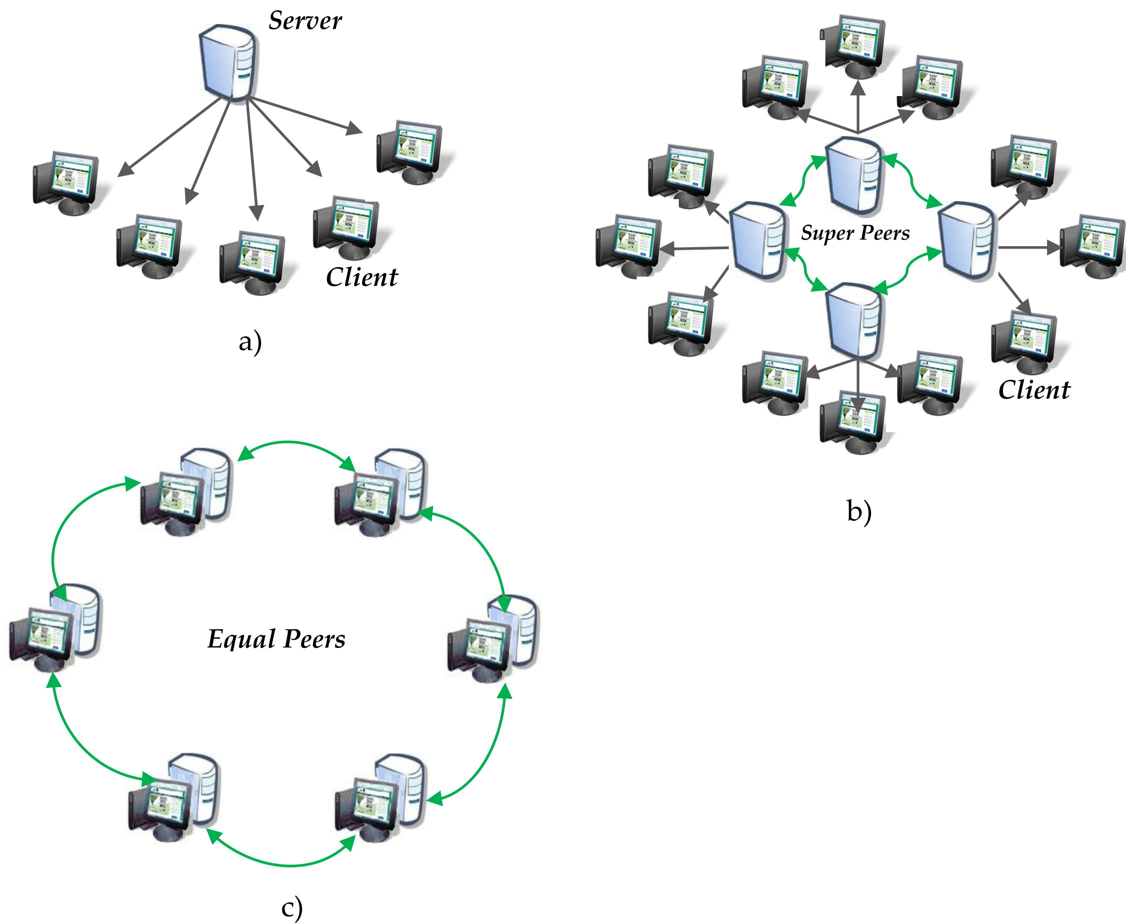


Figure 1: *a) centralized design b) cooperative network with super peers c) peer-to-peer network with peers distributed equally*

1.2 Research Question

If a design wants to obey the nature of Volunteer Computing, it should take full advantage of voluntariness, then the volunteers should provide computing power not only for data processing, but also for the management and control, consequently, they free the central server from high load situation caused by management overheads, and also originally eliminate the single point of failure. Based on this point of view, the decentralized application architecture for the distribution of volunteer computing

projects is obviously better than the centralized architecture. Thus, the first part of the reseach question is to ***design a decentralized application architecture for Volunteer Computing.*** But in another hand, the decentralization has its own problems – there is no central point to globally organize and manage the entire system, especially when the system grows large and changes quickly. This is why in decentralized distributed computing, self-organization, self-adjusting, and self-maintenance are quite crucial. If a volunteer computing application is built upon a decentralized architecture, the situation could be harsher, due to the voluntariness of the system, participants do not have any compelled obligation for staying in the system till the end of the project processing, which makes the system less robust and less reliable than the centralized design. Therefore ***based on the decentralized application architecture, how to improve the system robustness for volunteer computing, how to tolerate the volunteers' failures during the system processing*** is the second part of the research question. By learning from and comparing with a bunch of existing decentralized approaches, it's found that no fault-tolerant mechanism in those approaches took into account the influence of system changes (mainly due to the volunteers' failures), finally, the third part of the reseach question is fixed to ***based on the former two parts, how to design a fault-tolerant mechanism that can not only tolerate the volunteers' failures but also limit the influence of those failures as local as possible.***

The rest of this thesis is organized as follows: some relevant concepts and related works are introduced as the background in section 2. Before the design phase, a system model is given in section 3 represented as a set of design assumptions. Then the system design is described in section 4 in details. In section 5, this thesis builds up a prototype in a simulation environment, and a set of simulations are launched on this prototype to study the fault-tolerance of this decentralized architecture design for Volunteer Computing. Finally, Section 6 contains the conclusion and some future works related to design complementary aspects in the practice, such as data storage, security aspects of the system design in real implementation for Volunteer Computing.

2. Background

2.1 Relevant Concepts

This thesis refers to some concepts that are crucial for the system design. A concise view of these concepts is given in this section.

2.1.1 MapeReduce

This thesis uses the principle of **MapReduce** to construct the top layer model for Volunteer Computing. As the definition in [3], "*MapReduce is a programming model and an associated implementation for processing and generating large data sets*". It has been used by Google to solve problems residing in different categories of projects, such as in large-scale graph computaions and large-scale machine learnings. It mainly consists of two phases: **Map** and **Reduce**. In **Map** phase, a master partitions a problem into a set of sub problems and distributes them to its workers; in **Reduce** phase, a set of workers submit answers back to the master, then the master combines those answers in a way to produce the result. As described in its definition, it is worth to mention that after a master "maps" a set of sub problems to a worker, the worker can continue this **Map** procedure to find its workers. Consequently, the entire system under **MapReduce** framework will form a multi-level tree. The well-known Volunteer Computing platform BOINC is using **MapReduce** framework, and it is in a centralized design, the **Map** and **Reduce** procedures only exist between one level master and workers, the recursive feature of **MapReduce** therefore does not suit to be applied in BOINC.

2.1.2 Chord Protocol

The second layer in this thesis is in form of **peer-to-peer overlay networks.** As a distributed application architecture, **peer-to-peer overlay networks** are often implemented as an abstract overlay network layer, which decouples the system from the underlying physical network topology. A plenty of distributed protocols have been researched and utilized for indexing and efficient peer discovering on the overlay network layer. Such as Chord, Pastry, CAN, P-Grid, BATON, DHR-Trees, etc. each of them has own emphasis point of design. **Chord** [6] has been chosen as the distributed protocol in this thesis. In Chord, each peer requires ***O(log(N))*** references to other nodes

and ***O(log(N))*** messages cost for efficient lookups at worst case (***N*** is the number of peers on the Chord ring). Chord orders and distributes node identifiers along the ring modulo 2^m(*m is the number of bits of each node identifier*), and each node on the Chord ring tasks charge of keys that bigger than the identifier of its predecessor and smaller than or equal to its own identifier. When a query for certain resource is routed to a node on the ring, this node checks the key (resource identifier) against its key space, if it is out of own key space then continues to route this query to its ***successor*** (figure 2).

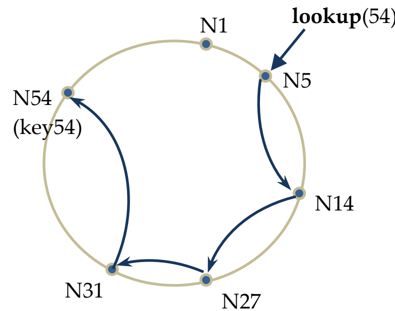


Figure 2: *Looking up key 54 on the Chord ring by routing this lookup from one node to its successor.*

For accelerating the searching procedure, each node on the ring keeps a ***finger table*** and a ***successor list*** instead of a ***successor.*** A ***finger table*** maintains up to ***m*** (the number of bits in the key/node identifiers) entries (actually, only ***O(log(N))*** distinct entries), which shortens the search path for each query by enabling longer distance jumps from each node to the node that is the closest one to the search key in its ***finger table***. And **a *successor list*** that keeps a number of successive nodes after this node on the ring is created to enhance the robustness (the detailed desceiption of this dynamic accelerated searching mechanism of Chord can be found in [6]).

This thesis benefits from a set of advantages of Chord:

1) **Simplicity and Availability:** Chord uses a simple stabilization algorithm to automatically adjust its internal tables to reflect network changes (node joining and leaving). It guarantees that the peer holding the required resource always can be found, even if continuous system changes occur on the ring.
2) **Decentralization**: Chord is fully distributed, which benefits the loosely-organized p2p applications.
3) **Flexible naming**: Chord places no constrains on the structure of the key, it is more flexible to map names to chord keys.

For example, when compared with Tapestry and Pastry, three of them have the same reference and message cost, but Tapestry and Pastry use a more complicated join protocol for reducing the routing latency. Other protocols as BATON [7], P-Grid [8], P-

Tree [9], they all organize a tree structure, the nodes in the tree are not equally exist, which puts much more constraints on applications than Chord.

2.2 Related Works

Many interesting proposals for Volunteer Computing have been researched without the existence of the central server, such as P2P-Tuple [10], PPVC [11] and [12]. They defined different architectures with fault-tolerant mechanisms to improve the system robustness. Their variation in details represents their achievements and also the vulnerability against those two issues: taking full advantage of voluntariness, and implementing the locality of the influence of system changes due to volunteers' silent failures.

I. ***P2P-Tuple*** is a P2P-based Volunteer Computing platform, all peers formed a DHT overlay (PAST) on which the Tuple-Space is constructed by using the storage capacities contributed by peers. Each peer offered two top level services: Tuple Space storage implemented by the Peer Daemon; Job Execution implemented in the Job Executor Daemon. It allowed all districted nodes to disconnect or crash at anytime and will save the submitted task, but as the number of volunteers grew up, the DHT stabilization according to the high churn rate of volunteers infected more other volunteers on the network, and thud cost much longer time for the system , which did not resolve the second issue. And P2P-Tuple guaranteed that submitted tasks will not fail, but for the tasks in processing, there was no collection mechanism.

II. ***In PPVC***, a decentralized job scheduling method was defined, and the overlay layer was formed by all volunteers. The functions on all volunteers were equal – a volunteer can be both Master and Worker at the same time. It required an application be recursively split, each volunteer split its own job into several sub-jobs and distributed them to its neighbors according to its own knowledge about the network, which exactly took full advantage of Volunteer Computing by distributing also the management and control. In PPVC, if a leaf node disjointed the overlay, its parent then reassigned its workload to its siblings; if a middle node disjointed the overlay, its children submitted to its parent node by each child of this leaving node recorded its distribution path. This fault-tolerant policy didn't take into account the influence of the system changes, since the volunteers on each level may disconnect at any time, which is not convenient to maintain a correct root-to-leaf path in each volunteer, especially, when this tree structure grew taller, a longer list would be kept in each volunteer in the system, and it is much harder for each volunteer to keep this list correct over time.

III. [12] defined a peer-to-peer model under MapReduce framework, each node performed as either a Master or a Worker, and the role assigned on each peer changed dynamically overtime but no peer can be both Master and Worker at the same time – the set of Masters formed a peer-to-peer overlay network, and the set of Workers formed another peer-to-peer overlay network– which clearly constrained the management to be taken by a specific set of volunteers at certain moment, and when these two sets of volunteers grew, changed fast, these two peer-to-peer overlay networks would low down the reliability and robustness of the system. In [12] For achieving fault-tolerance, each Master acted as a backup node for other Masters, and relevant Masters periodically made checkpoints of the status of the job on its backup nodes, If the backups detected the failure of the Master, they elected a new Master among them and restarted the job from the latest available checkpoint. This multi-backup fault-tolerant policy took up a lot of resources for data redundancy which in contract can be used for data processing.

3. System Model

This thesis designs the system under **a set of assumptions**, which simplifies the functionality of the entire system, and excludes some insignificant factors to keep attention on certain key mechanisms.

3.1 MapReduce Relevant

This thesis uses a MapReduce-like model at the top layer. This model is a simplified Map-Reduce procedure with a set of specified rules. It is under a set of assumptions:

- Each volunteer on the network has the contact information of some other volunteers (e.g. by holding a preference neighbor list, or by broadcasting on the network). Therefore, in Map phase, volunteers can contact each other to establish Master-Worker relationship in the system.
- In Map phase, it is assumed that all tasks partitioned from a project are in the same size. And all volunteers on the network have the same amount of idle computational resources. These two assumptions simplify the system situation for setting rules at the top layer.
- In MapReduce framework, a DFS is used to store data blocks. In this thesis, a prototype has been built up in a simulation environment, where a single machine is used for a set of simulations. Therefore, all data storage is implemented locally on this machine.
- During the system processing, no time limitation is laid on the tasks or the project, therefore no mechanisms such as task abortion and task reassignment due to timeouts are adopted in the system. This enables a more relax environment for the system to mainly focus on the study of the fault-tolerant mechanism of this system design.

3.2 Peer-to-Peer Overlay Network Relevant

Peer-to-peer overlay network is used in this thesis to achieve fault-tolerance of the system. The construction of the overlay network layer is performed under a set of assumptions, which simplifies the implementation of the prototype, and fixes more parameters in system configuration affecting the capability of the fault-tolerance,

consequently simplifies the study for certain objective about fault-tolerance of the system by narrowing down the scale of determining factors.

- During the peer-to-peer network layer construction phase, volunteers on the network form new Chord rings by following the join procedure of Chord protocol. Chord uses ***consistent hashing*** to produce key/value pairs, which intends to provide load balancing, thus each peer on the same Chord ring receives approximately the equal-sized key space, and makes a relatively few movement of keys when network size changes by peers' join and leave. In this thesis, the key space is assigned according to the dispatching of tasks on the top layer, and the network size in simulations has not been configured to be extremely large, so the consistent hashing is not necessary to be used in the prototype. In this thesis, the task identifier is used as Chord identifier under a set of rules. This solution simplifies the construction of the overlay network layer, and obeys the original design of the top layer.

3.3 System Relevant

Viewing the system as a whole, it is assumed that no malicious behaviors happen during system processing, such as malicious executable distribution, result falsification, project information theft, etc. which are the main security issues in Volunteer Computing; the only unstable factor in the system is that each participant may silently leave the system due to failures, which may be caused by power outage, network disconnection, etc. Based on these assumptions, next section will give detailed description of the system design.

4. Network Application Architecture Design

4.1 Design Overview

In this section, the system design is described in details. It begins with a simple composition of decentralized application architecture at the top layer in 4.2. The decentralization feature of this top layer architecture not only benefits Volunteer Computing, but also makes it vulnerable, and the vulnerability is represented in 4.3. Taking into account this vulnerability, the thesis constructs a second layer – a peer-to-peer overlay network layer in 4.4, devising to couple these two layers to make the decentralized architecture robust, and this coupling mechanism is described in 4.5.

Before the system design, two guidelines should be held according to the research question**:**

1) **Taking full advantage of Voluntariness:** it means that the Volunteer Computing system should also distribute out the management and control to volunteers, as well as data processing. This guideline guarantees the decentralization feature of the design.
2) **Absorbing the influence of system changes as local as possible**: it means that the Volunteer Computing system should be able to limit the influence of each volunteer's silent failure as local as possible. This guideline is crucial to the system design, since on a decentralized distributed network, there is no global view of the whole system, and in a Volunteer Computing system, the network size can be quite large. If a volunteer's failure affects a large part of or even the whole system, the system will endure a quite long stabilization time, which makes the system less robust and reliable with the higher possibility to produce incorrect results and inconsistent status.

These two guidelines provide decentralized thinking and put the constraint on the fault-tolerance mechanism to produce a robust Volunteer Computing system.

4.2 A Simple Approach for Top Layer Architecture Design

The primary goal is to remove the central server, building up a decentralized architecture for Volunteer Computing. In this decentralized model, volunteers should also provide computing resources for management and control according to the guideline one. And this thinking also benefits such a scenario: each volunteer can hold the demand of requiring computational resources from others for its own projects, such resource requests can be communicated among neighborhoods, according to each volunteer's knowledge of the network, instead of asking a third party for such services as in BOINC. Based on these considerations, each volunteer in the system should be both master and worker, which can be achieved by using the recursive feature of MapReduce. Therefore, the principle of MapReduce is adopted to build a dispatching model at the top layer by following the Volunteer Computing dispatching rules:

Dispatching Rules

1) A client, outside the Volunteer Computing dispatching system, partitions a project into a set of equal seized tasks. Then it launches those tasks onto a volunteer or onto a set of volunteers in the system. This volunteer or the set of volunteers become(s) the root level master(s). Here let's focus on one root level master $\boldsymbol{M}_{root}$, it records the range of these tasks as ***Task Range***, and takes only the first task for itself to compute, recording it as ***Load Range***. Finally $\boldsymbol{M}_{root}$ marks its ***Dispatch Level*** of this project as "0", which is the first dispatching level of this project in the system.
2) Then, $\boldsymbol{M}_{root}$ begins to find voluntary workers that can work for it. This worker discovering procedure can be done according to a known preference list where $\boldsymbol{M}_{root}$ puts contact information of its neighbors, and prioritizes it according to a set of preference, a related approach has been proposed in [13]. Here, e.g. volunteers $\boldsymbol{W}_1 - \boldsymbol{W}_n$ have been chosen, $\boldsymbol{M}_{root}$ therefore sends task requests to them with the information and requirements of the project and tasks (e.g. the author, topic and content of a project, the amount of CPU utilization and storage that each task needs to consume). When each of $\boldsymbol{W}_1 - \boldsymbol{W}_n$ receives this request, it will check the contained information to see if it is interested in the project, and if the answer is yes, it continues to examine own spare computational capacity against the task requirements. Finally it replies to $\boldsymbol{M}_{root}$ to identify its capability of working for $\boldsymbol{M}_{root}$ on this project.
3) After $\boldsymbol{M}_{root}$ has got the answers from $\boldsymbol{W}_1 - \boldsymbol{W}_n$, and validated their qualification, $\boldsymbol{M}_{root}$ will equally divide the rest tasks among them and dispatch out each set of tasks separately (in practice, the master can partition the untaken tasks according to each worker's computing capability, but here, it's assumed that all volunteers

have the same spare computing capability). When the whole stuff is completed, $\boldsymbol{M}_{root}$ can begin to compute its own task. Another situation may exist: if no volunteers have replied to work for $\boldsymbol{M}_{root}$ on this project, then $\boldsymbol{M}_{root}$ will record the range of all tasks in its Load Range as well as in its Task Range, and compute all of them itself.

4) When each of $\boldsymbol{W}_1 - \boldsymbol{W}_n$ receives its own set of tasks, it will trigger the same procedure just as the one $\boldsymbol{M}_{root}$ has: recording own **Task Range, Load Range** and **Dispatch Level** of the project, the dispatch level is one level higher than its master's. After this, it can find new volunteers that can work for it.

In my scenario, this recursive procedure spreads on the network among volunteers; each volunteer can be both master and worker at the same moment. Consequently, along with the dispatching procedure, all volunteers in the system at the top layer form a multi-level tree that grows downward from the top root to the bottom leaves. The dispatching procedure at the top layer is showed in figure 3.

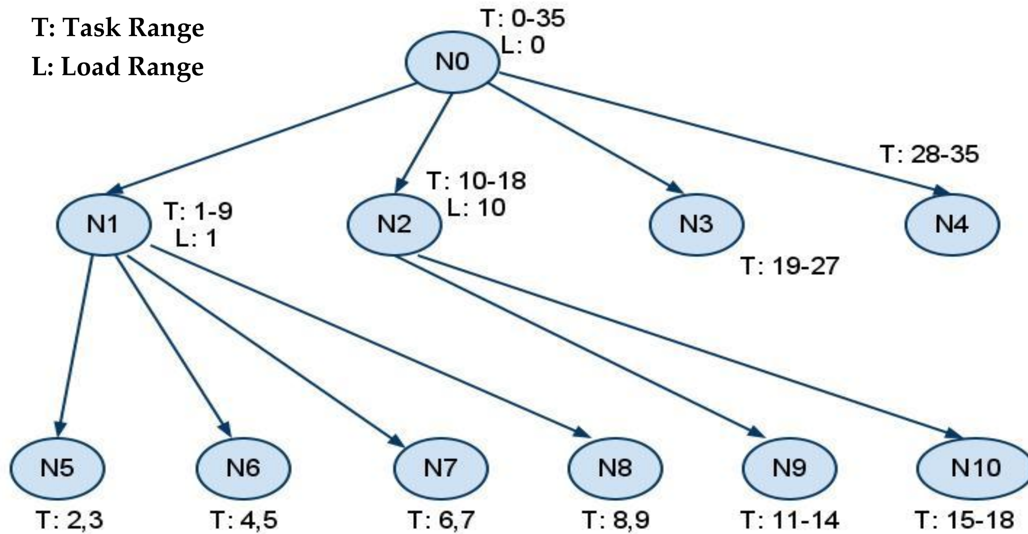


Figure 3: *a client launches a set of tasks 0-35 of a project to volunteer $\boldsymbol{N}_0$, then, $\boldsymbol{N}_0$ records its **Task Range** as 0-35, and **Load Range** as 0, since $\boldsymbol{N}_0$ only keeps the first task (task 0) for itself to compute. $\boldsymbol{N}_0$ continues to find $\boldsymbol{N}_1$- $\boldsymbol{N}_4$ as its workers, then it equally splits the rest tasks 1-35 among $\boldsymbol{N}_1$- $\boldsymbol{N}_4$, $\boldsymbol{N}_1$ gets tasks 1-9, keeps task 1 for itself, $\boldsymbol{N}_2$ holds tasks 10-18, keeps task 10 to compute, and $\boldsymbol{N}_3$, $\boldsymbol{N}_4$ get tasks 19-27, 28-35 separately, since neither of them has workers, they keep whole task set for themselves. $\boldsymbol{N}_1$ further gets $\boldsymbol{N}_5$- $\boldsymbol{N}_8$ as its workers, and $\boldsymbol{N}_2$ finds $\boldsymbol{N}_9$, $\boldsymbol{N}_{10}$ as its workers, the same procedure will happen between $\boldsymbol{N}_1$, $\boldsymbol{N}_2$ and their workers respectively.*

4.3 Vulnerability of Top Layer Architecture Design

The decentralized application architecture in Volunteer Computing layer, as described in the former section, distributes out also the management and control of Volunteer Computing by enabling each volunteer in the system to be both master and worker. But it has vulnerability under this decentralized dispatching model. When the dispatching procedure is spreading over the network, the tree will grow vertically as well as horizontally. The more volunteers join the system, which makes the former workers become both workers and masters, the more fragile the multi-level dispatching tree will be, since the growing set of middle level volunteers means that more middle level volunteers may leave the tree silently under the high churn rate, directly leading to the cut offs of the whole tree structure (the scenario is simply presented in figure 4). And the Volunteer Computing under the MapReduce-like dispatching model needs to collect results from the bottom back to the root through the entire tree structure. Therefore, if the tree is cut into parts, the Reduce phase will be stuck, resulting in the collapse of the entire computing. The application architecture therefore seems less robust and less valuable. ***For this issue, it is necessary and inevitable to create certain mechanism for maintaining the integrality, the connectivity of this dispatching multi-level tree structure, under the condition that the middle level volunteers may leave the system silently due to failures.*** And as mentioned previously, in some papers, only reassignment was adopted for fault-tolerance, which produced huge amount of computation resources wasted when the number of participants is large, since all workers of a failed master had worked in vain. In some other papers, certain fault-tolerant mechanisms were introduced as well as reassignment, but the influence of volunteers' failures in the system was not taken into account as a key factor. This thesis tries to fix this architectural vulnerability with the consideration of the locality of the influence produced by volunteers' failures in the following section.

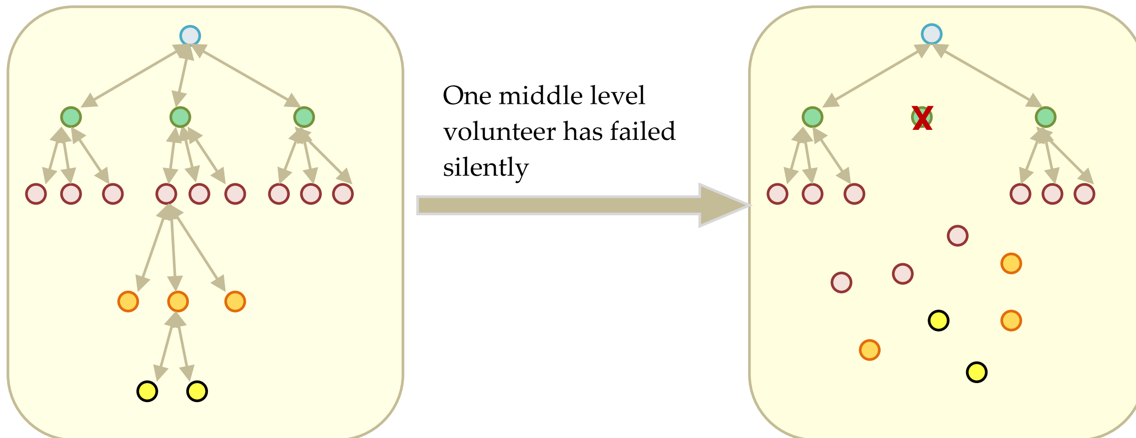


Figure 4: *the volunteer computing dispatching tree structure on the left side, will change to the one on the right side when a middle level volunteer fails silently – all participants derived from the failed volunteer will be cut off from the tree structure.*

4.4 The Second layer Design for The Robustness of Top Layer Architecture

The structured peer-to-peer networks mainly have been designed for efficiently routing lookups of some resources to the right point in the network, and the lookups can be launched onto any node on the overlay network. These features can solve the reunion problem in such a scenario: when a master has failed silently, in case that some live volunteer tries to collect all the workers of the failed master, and if those workers finish their tasks, they can submit the results to a new master. The structured peer-to-peer networks have been used in many proposals for creating decentralized Volunteer Computing systems. Those proposals composed structured overlay networks by all volunteers working in the system. In contract, this thesis constructs the peer-to-peer network layer in a different way.

Before the construction, by following the second guideline to limit the influence of each middle level volunteer's failure as local as possible, it is required to narrow down the influence scope of a middle level volunteer's failure in the tree structure at the top layer. Apparently, under the dispatching model defined in the former section, ***a middle level volunteer communicates only with its master and all its workers, so the affected area of a middle level volunteer should be fixed between its master and its workers***. If this volunteer silently fails in the middle of processing, its workers should be re-gathered by some other volunteer still alive in the system. The results produced by these workers

therefore can be submitted back to the root, instead of being aborted. Based on this thinking, it's given the following peer-to-peer network layer construction rules :(The establishment is shown in Figure 5.)

Peer-to-Peer Network Layer Construction Rules:

1) Each volunteer ***V*** after dispatching out a set of tasks to the first worker W at the volunteer computing application layer, ***V*** and ***W*** will begin to establish a Chord Ring under the top layer. ***W*** as a worker sends out a ***Chord Ring Join Request*** to the master ***V***. This Chord Ring, is viewed by the master ***V*** as a ***Lower Chord Ring***, and is viewed by the worker ***W*** as its ***Upper Chord Ring***, so the Upper or Lower really depends on the roles of volunteers in volunteer computing layer.
2) On this Chord Ring, the master ***V***'s chord id is the task id in its Load Range (the task kept by ***V*** for itself to compute); the worker ***W***'s chord id is the task id of the last task in its Task Range (the last task ***W*** received from ***V***).
3) Then, more volunteers may become ***V***'s workers, those new workers will join the same Chord ring just as ***W***. Consequently, on this chord ring, master ***V*** will takes charge of the key space of only one task, and each other worker volunteer on this chord ring will take charge of the key space of tasks that received from master ***V***.
4) If the worker ***W*** further finds volunteers working for it, the first new worker (e.g. ***R)***, after ***R*** receives a set of tasks from ***W*** (now ***W*** is a master of ***R***, and a worker of ***V***), ***R*** sends out a ***Chord Ring Join Request*** to ***W*** to establish a new Chord ring with ***W***. This new Chord ring, is viewed by ***W*** as its ***Lower Chord Ring***, and is viewed by ***R*** as its ***Upper Chord Ring***. Then, using the same rules to fix each own Chord id on this new Chord ring.
5) The root level master and the leaf level workers of the whole volunteer computing dispatching tree are special: the root level master has only a Lower Chord DHT since it has no master; the leaf level workers have only an Upper Chord DHT since they have no workers.
6) When a volunteer ***V*** has left this volunteer computing application by failures, both its Upper Chord Ring and Lower Chord Ring will detect its leaving and then automatically adjust themselves respectively (***V***'s key space will change hands to ***V***'s successor on each Chord ring. And all of the rest nodes will stabilize to reflect ***V***'s leaving) - all these self-adjusting actions are automatically taken by the fault-tolerant mechanism of Chord Protocol.

For more detailed explanation, it can focus on the step 4). By taking step 4), ***W*** as a middle level volunteer at the top layer holds two Chord DHT tables in the peer-to-peer overlay network layer: one for its attending in the Upper Chord Ring, and one for its attending in the Lower Chord Ring. Let's see these two Chord rings from a

different angle: The Upper Chord Ring of ***W*** is formed by ***W***, its master ***V***, and all other workers of ***V*** (those workers can be viewed as ***W***'s siblings). ***V***, ***W*** and other workers of ***V*** are equally peers on this Chord ring unlabeled with mater/worker roles. This Chord ring is viewed by ***V*** as a Lower Chord ring, but is viewed by ***W*** and all other workers of ***V*** as an Upper Chord ring. The second Chord ring – the Lower Chord Ring of ***W*** – is formed by ***W*** and all its workers. ***W*** and all its workers are equal peers on this ring. ***W*** treats this Chord Ring as its Lower Chord Ring, and all workers of ***W*** on this ring view this Chord ring as their Upper Chord Ring. ***W*** covers different key spaces (holding different Chord identifiers) on these two Chord rings. On the Upper Chord Ring, ***W*** takes charge of the key space as the range in its Task Range, which means that ***W*** takes charge of all tasks received from its master on the Upper Chord Ring, since this Chord ring is formed by its master and all its siblings as well as itself. And on the Lower Chord ring, ***W***'s key space only contains the task in its Load Range, since the rest tasks have been dispatched out to its workers. And those workers are also on this Lower Chord Ring, each of them will take the key space according to the tasks got from ***W***.

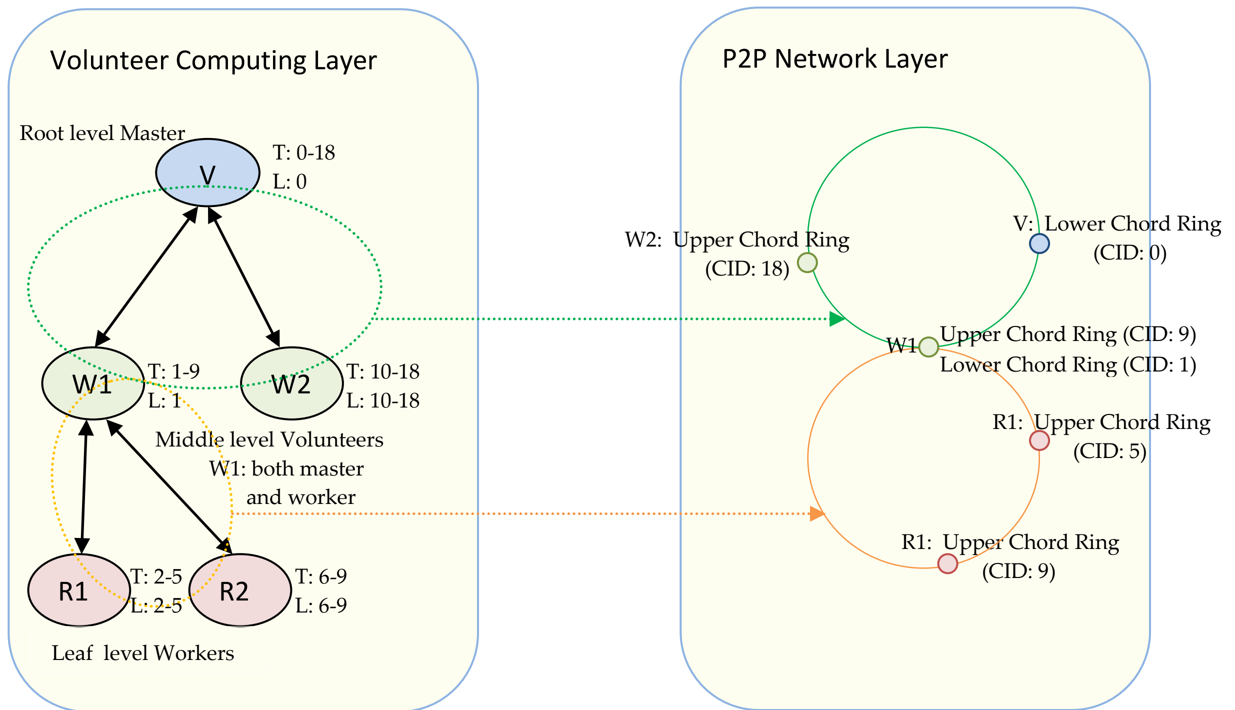


Figure 5: *The peer-to-peer network layer is constructed according to the dispatching procedure running on the Volunteer Computing layer. If $\boldsymbol{w_2}$ can find new workers, then a new Chord ring will be established among $\boldsymbol{w_2}$ and its new workers, which can be added under the same construction rules at the second layer described in the right part.*

4.5 Fault-tolerance of The Layered System Design

As described in the previous sections, this thesis has introduced the decentralized network application architecture for Volunteer Computing, which is in form of a distributed multi-level tree. It removes the existence of the central server by enabling each volunteer in the system to be both master and worker. Then this thesis has also constructed a second layer – a peer-to-peer overlay network layer – to guarantee that the recollection of all workers of a failed master is possible and can be performed efficiently. But when a middle level volunteer fails silently, before the recollection is triggered, there should be a way to make some other volunteers that are still alive in the system find at least a piece of information of one of those workers of the failed volunteer. Without this, no recollection can be triggered since no extra contact information exists between those workers and any live volunteers in the system.

4.5.1 Distribution Policy

Based on the discussion above, this thesis solves the question by coupling these two layers under a set of rules, which enables the trigger of the reconnection procedure, and is crutial for the fault-tolerance of the system:

Sibling Information Distribution Rules:

1) For each volunteer $\boldsymbol{V}$ processing data in this system, if it is in both a Upper Chord Ring (it has a Upper DHT) and a Lower Chord Ring (it has a Lower DHT), which means that $\boldsymbol{V}$ is a middle level volunteer at the top layer, then whenever the automatic stabilization procedure is triggered in both $\boldsymbol{V}$'s Upper and Lower Chord Ring at the second layer, it also triggers another function in $\boldsymbol{V}$: this function is called ***DistributeUpperToLower*** that distributes the contact information (identifiers) of volunteers in $\boldsymbol{V}$'s Upper DHT to the volunteers in $\boldsymbol{V}$'s Lower DHT. Therefore, part of $\boldsymbol{V}$'s workers will hold contact information of part of $\boldsymbol{V}$'s siblings. For example, in Figure 5, the Upper Chord DHT of $\boldsymbol{W}_1$ may contain the contact information of $\boldsymbol{W}_2$, and the Lower Chord DHT of $\boldsymbol{W}_1$ may contain the contact information of $\boldsymbol{R}_2$, when either chord ring triggers the stabilization procedure, it also triggers a function in $\boldsymbol{W}_1$, distributing the contact information of $\boldsymbol{W}_2$ to $\boldsymbol{R}_2$, then if $\boldsymbol{W}_1$ fails silently, $\boldsymbol{R}_2$ can try to contact $\boldsymbol{W}_2$ for reconnection.
2) For the non-middle level volunteers such as the root level volunteer $\boldsymbol{N}_0$ and the leaf level volunteers on the dispatching tree, the solution for their silent leaving is not the key issue in this thesis. And it should be easier than the middle volunteers' unexpected leaving: if a leaf level volunteer $\boldsymbol{V}$ has failed silently, its Upper Chord Ring then adjusts itself to repair the whole ring, $\boldsymbol{V}$'s key space will

change hands to ***V***'s successor on its Upper Chord Ring. After this Chord ring has stabilized, the master of ***V*** at the volunteer computing layer will reassign ***V***'s tasks to the volunteer who currently takes charge of ***V***'s key space (the successor of ***V***), and there is no need to consider about the ***V***'s Lower Chord Ring, since it does not exist. For the root $\boldsymbol{N_0}$, it has only a Lower DHT, so in case of failing silently, $\boldsymbol{N_0}$ can pre-distribute the client contact information to its workers whose contact information is kept in $\boldsymbol{N_0}$'s Lower Chord DHT; or it can pre-send the contact information of one or more workers in its Lower Chord DHT to the client. Either way enables the reconnection in case that $\boldsymbol{N_0}$ fails silently.

4.5.2 Analysis of Setting Distribution Policy

The policy implemented in the ***DistributeUpperToLower*** function determines the fault-tolerant capability, affects the robustness of the system. Some policy may also use the information in the successor list as well as the information in the finger table of Chord protocol. Due to the features of Chord protocol and the position of volunteers in the dispatching tree, for a middle level volunteer, how to select sibling information to distribute (i.e. in what order) between its two Chord DHTs is also a factor in setting policies. But in spite of what kind of policy is adopted, ***the tradeoff always exists, between the possibility of fault-tolerance and the information redundancy***.
Given each volunteer in the system ***50% possibility to fail***, for a middle level volunteer ***V,*** there are $\boldsymbol{N}_{\mathrm{U}}$ volunteers in its Upper Chord ring, and $\boldsymbol{N}_{\mathrm{L}}$ volunteers in its Lower Chord ring; ***V's*** Upper DHT contains the information of $\boldsymbol{N}_{\mathrm{DU}}$ distinct volunteers ($\boldsymbol{N}_{\mathrm{DU}} < \boldsymbol{N}_{\mathrm{U}}$), and ***V's*** Lower DHT contains the information of $\boldsymbol{N}_{\mathrm{DL}}$ distinct volunteers ($\boldsymbol{N}_{\mathrm{DL}} < \boldsymbol{N}_{\mathrm{L}}$). $\boldsymbol{N}_{\mathrm{DU}}$ and $\boldsymbol{N}_{\mathrm{DL}}$ will change with $\boldsymbol{N}_{\mathrm{U}}$ and $\boldsymbol{N}_{\mathrm{L}}$ respectively, according to the storage cost of each peer in Chord for efficient routing, it should be a logarithmic relation between each pair. Thus, when given $\boldsymbol{N}_{\mathrm{U}}$ and $\boldsymbol{N}_{\mathrm{L}}$, $\boldsymbol{N}_{\mathrm{DU}}$ and $\boldsymbol{N}_{\mathrm{DL}}$ can also be fixed. It's also assumed that ***V*** has failed silently after the latest trigger of its distribution function, and both Upper and Lower Chord rings of ***V*** have stabilized due to the latest system change.
If the consideration is to ***achieve the highest possibility of fault-tolerance of this layered system for one middle level volunteer V's silent failure***, the sibling information distribution policy should be set to: whenever the distribution function is triggered, ***V*** will send the contact information of all $\boldsymbol{N}_{\mathrm{DU}}$ distinct volunteers in its Upper DHT to each of $\boldsymbol{N}_{\mathrm{DL}}$ distinct volunteers in its Lower DHT. Based on this policy, the system should fail to reconnect itself under only two cases:

1) When all of $\boldsymbol{N}_{\mathrm{DL}}$ distinct volunteers in ***V's*** Lower DHT have failed with ***V*** concurrently, it means that the rest workers of volunteer ***V*** who are still alive in the system, have no contact information of ***V's*** siblings, therefore, it's impossible

for any of $\boldsymbol{V's}$ rest workers to contact an live volunteer, the reconnection cannot be triggered.

2) When all of $\boldsymbol{N}_{\mathrm{DU}}$ distinct volunteers in $\boldsymbol{V's}$ Upper DHT have failed with $\boldsymbol{V}$ concurrently, it means that although one or more workers of $\boldsymbol{V}$ are still alive and keep the contact information of $\boldsymbol{N}_{\mathrm{DU}}$ distinct volunteers in $\boldsymbol{V's}$ Upper DHT, but no siblings will respond to the reconnection requests, so the reconnection procedure cannot be triggered successfully.

So the possibility of fault-tolerance of this layered system due to $\boldsymbol{V's}$ failure, and under the distribution policy described above should be:

$$\boldsymbol{P = 1 - (2^{-N_{DU}} + 2^{-N_{DU}})}$$

When $\boldsymbol{N}_{\mathrm{U}} = \boldsymbol{N}_{\mathrm{L}} = \boldsymbol{N}$ ***and*** $\boldsymbol{N}_{\mathrm{DU}} = \boldsymbol{N}_{\mathrm{DL}}$, by using Chord protocol, each node in a *N*-nodes Chord ring requires information about ***O(logN)*** other nodes for efficient routing. Then the formula above can be changed into:

$$\boldsymbol{P = 1 - 2/N}$$

And the information redundancy $\boldsymbol{R}$ made by this policy due to $\boldsymbol{V}$'s failure is measured by the total number of the failed master's siblings, whose contact information is kept by the workers of this failed master:

$$\boldsymbol{R = (logN)^2}$$

P and ***R*** are showed in figure 6:

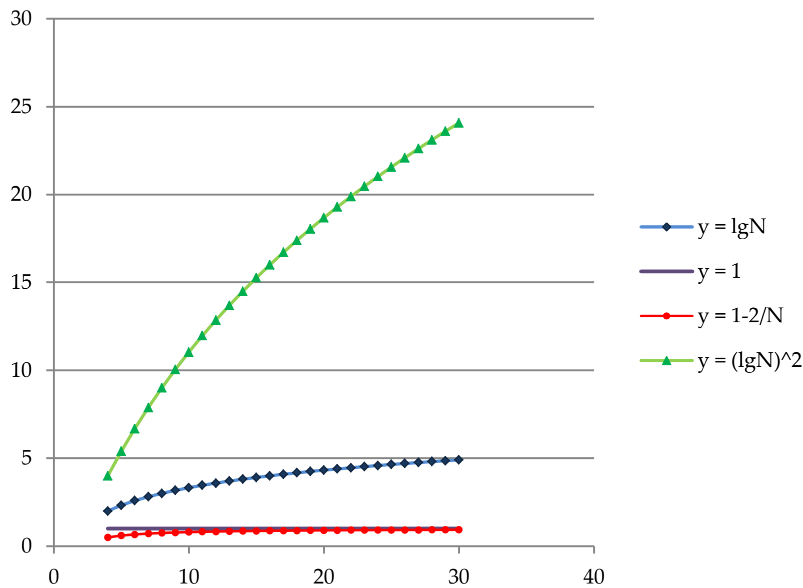


Figure 6: *the curve marked with round points stands for* ***P****, when* ***N*** *grows, the value of* ***P*** *indefinitely approaches* ***1****(presented as a straight line)***,** *at the meantime, the cost of Chord ring grows as the curve marked with short vertical lines, and the* ***R*** *grows as the curve marked with arrows.*

4.5.3 Reconnection

All necessary "tools" are prepared for reconnecting the dispatching tree structure when middle level volunteers fail silently in the system. The reconnection procedure is the core of the fault-tolerance design. It's given the following set of rules using both the layered architecture and the distribution function:

Reconnection Rules:

1) Each volunteer periodically checks that if all its tasks are completed, and if its master of a project is still alive.
2) If the master of a project has gone silently, this volunteer will check that if it has the contact information of the failed master's siblings. If it has, it will send a ***Reconnection Request*** to this sibling with its own chord id.
3) If the sibling of the failed master is alive, when it gets the ***Reconnection Request***, it firstly checks that if the requested chord id is in own key space. If not, this volunteer will continue to route this ***Reconnection Request*** on its Upper Chord Ring (or on its Lower Chord Ring, depends on whether this sibling volunteer is a sibling of the failed master, or is the master of the failed master) to the replacing node (the node who currently takes charge of the failed master's key space)
4) Then the replacer will try to recollect the failed master's workers by launching lookups on the requested volunteer's Upper Chord Ring. Here, since Chord does

not support Range Query, in this thesis, it utilizes the successor mechanic to gather all workers on the Chord ring. When all workers are recollected, the replacer will send each of them an ***UpdateWithNewMaster*** message.

5) When each worker of the failed master receives the ***UpdateWithNewMaster*** message, it is required to follow these two steps:
    I. Firstly, It needs to send necessary information of the tasks under its own management to the new master, so the new master can update its task management scope at the top layer (before this action, the new master only updated the key space at the second layer by the automatic stabilization of Chord, and no updates have been performed on the new master at the Volunteer Computing layer).
    II. After the updating at Volunteer Computing layer, the new master responds to each new worker with ***Join Notification***. Therefore, this new worker can join the new master's Lower Chord Ring with its chord id unchanged since it will take charge of the same set of tasks and has no confliction with the key spaces of any old worker of this new master on the Chord ring. And this joining action also resets this new worker's Upper Chord DHT, since the Chord ring it has newly joined into, will be viewed as its new Upper Chord Ring.

6) If there are two workers that belong to the same failed master have detected that their master has gone silently, and both of them have the contact information of the failed master's siblings, they will then both contact the siblings respectively, and both reconnection requests will be routed to the replacement of the failed master. Then the replacing volunteer will handle only the first request it has received, triggering the reconnect procedure. For the further arrived same type requests for the same failed master, this replacing volunteer just ignores them. Two cases may exist in the reconnection procedure:
    I. The replacing volunteer is at the same dispatch level as the failed master: in this case, the reconnection procedure is performed as described above.
    II. The replacing volunteer is one level higher on the dispatch tree than the failed master: which means that the replacement of the failed master is the failed master's master on the dispatching tree. In this case, the replacing volunteer will try to give the key space of the failed master to its predecessor on its Lower Chord Ring (the predecessor is on the same dispatch level as the failed master since they are both workers of the current replacing volunteer), and then this predecessor will stabilize the expansion of the key space on its Upper Chord Ring for becoming the

replacement of the failed master. Finally this new master can begin to recollect all the workers in the same procedure as in case I.

Former sections have described four sets of rules to provide a fault-tolerant decentralized application architecture design for Volunteer Computing, based on this design, this thesis created a prototype where all the key rules are implemented, but in a simplified way under certain assumptions.

5. Simulations

In this thesis, a prototype has been built up in a simulation environment. A set of simulations were run on this prototype to study the fault-tolerant capability under the system design.

5.1 Simulation Tools and Prototype

5.1.1 Simulation Tools

A prototype of this layered system has been built on a peer-to-peer network simulator – Peersim [14], which is written in Java. Compared with other overlay networker simulators (comprehensive comparisons were made in [15] [16]), Peersim provides ***strong network scalability***. It can be used to simulate a network with millions of nodes, and nodes may join and leave continuously. Peersim has a ***component-based structure***, which enables to ***very quickly and flexibly prototype a protocol***, combining and rearranging different pluggable modules. And Peersim has also implemented some common, well-known protocols, such as Pastry, BitTorrent, SG-1, T-man and Chord (although Chord protocol has been already implemented in Peersim, but the implemented version is still not quite suitable for my application).

Two simulation models are defined in Peersim: ***Cycle-based simulation model*** and ***Event-based simulation model*** *(both models were used in this thesis)*.

- ***Cycle-based simulation model:*** it does not contain the transport layer simulation, in other words, no communication latency has been simulated in this model. All nodes simulated in this model periodically gain the control sequentially.
- ***Event-based simulation model:*** it contains the simulation of transport layer with communication latency. The messages passing over the network are stored in a queue in their sending orders combined with their latencies.

As described above, Peersim does not support distributed simulations. All actions taken by nodes in a network simulated in Peersim, are in a sequential order under both two simulation models. This thesis does not require a distributed simulator, but focuses on the scalability and flexibility of a simulator for creating a prototype of this layered system design within a scalable network.

To demonstrate the system status with fault-tolerance capability, Ubigraph [17] was used as the system result observation GUI. It provides a set of easy-to-use Java API and related documents.

5.1.2 Prototype

Some aspects of the prototype should be stressed here for good understanding of the simulations in the following sections.

- In this prototype, at the beginning of a simulation, all nodes on the network simulated by Peersim, hold the same **degree** value, it represents the maximum number of neighbors each node can have on the network. And the node identifier begins at "0". The assignment of neighbor relationship between two nodes follows the increasing of node identifiers: e.g. the **degree** is 3, then ***node 0*** has ***node 1, node 2*** and ***node 3*** as its neighbors; and ***node 1*** has ***node 4, node 5*** and ***node 6*** as its neighbors, ***node 2*** has ***node 7, node 8*** and ***node 9*** as its neighbors… and so on.
- In prototype, the communication during stabilization and lookup procedures on the peer-to-peer network layer is latency-free. At the volunteer computing layer, each message is sent with a latency randomly generated ranging from **min_latency** to **max_latency**. The prototype uses **Event-based engine** for message communication; and it uses **Cycle-based engine** for periodically triggered functions: Chord ring stabilization in peer-to-peer network layer, sibling information distribution and process status checking in volunteer computing layer.

All above parameters are configurable in Peersim, and they all have influence to the fault-tolerant performance of the prototype.

5.2 Simulation Design

A set of simulations were launched to study the fault-tolerant capability of this layered system for volunteer computing. As mentioned previously, the fault-tolerant mechanism is designed to against the middle level volunteers' failures. In simulations, a set of middle level volunteers were configured to fail concurrently during system processing, under this situation, the success of the core of fault-tolerance – the **Reconnection procedure** – under a specific **Sibling Information Distribution Policy** is one objective, and to study the influence of this policy on both the fault-tolerant capability and

information redundancy of the system is the further objective. At the end of this section, a simple Volunteer Computing application running on the prototype is briefly described.

5.2.1 Reconnecting Operations

To study the success of the reconnection procedure, it's programmed to calculate the number of operations cost during the entire reconnection phase:

1) **Reconnection Trigger Operation**: this type records the number of Reconnection Request messages which have been successfully sent to some siblings of the failed master. – *this number should be equal or larger than the number of the failed masters; otherwise, at least one failed master does not have any alive worker that can contact an alive sibling volunteer of this failed master.*
2) **Reconnection Lookup Operation**: this type records the number of lookups launched in the peer-to-peer network layer for reconnection: one sort of this type is for searching the replacement of the failed master; another sort of this type is for searching the first and last workers in the failed master's Lower Chord. – *This number shows the performance of single queries of the underlying distributed protocol, here is Chord. And this number also shows that the relation between the positions of the volunteer computing layer and the number of lookups in the peer-to-peer network layer.*
3) **Reconnection Update Operation**: this type records the number of operations spent in recollecting all workers of the failed master after the first and the last worker have both been found by the new master (actually the operations of range query), plus the number of the updates in both workers and the new master after the new master have got contact with each worker. The updates all are about tasks under management (i.e. updating the task information with the new master). Each worker or the new master launches a serial of updates at a time, and it is count as one update operation. – *this number shows the recollection cost, combined with the next operation number, to examine if all workers of the failed master have been recollected, and then whether all of those workers have joined the new master's Lower Chord Ring or some of them have just submitted the results to the new Master without this join action.*
4) **Reconnection Join Operation**: this type records the times of triggering re-join functions. Actually, this number should equal to the number of workers of the failed master as long as none of them has failed and none of them has completed. After updating with new masters, each worker of the failed master will wait for a join notify message from the new master. Once the worker gets this message, it can rejoin the new master's Lower Chord Ring. – *this number shows that whether the correct number of workers have joined the new master's Lower Chord Ring, and this number also varies with whether a master has failed and a new master has contacted a*

worker of the failed master before or after this worker has completed its tasks, even in another situation, before or after this worker has failed too (but in this paper, the experiments were configured to exclude those situations, more complicated cases will be put in the future work).

5.2.2 Setting Distribution Policies

The **Sibling Information Distribution Policy** is crucial for fault-tolerance of the system. To study the tradeoff given by this policy between the fault-tolerant capability and information redundancy of the system, two different distribution policies are given in this section:

For a middle level volunteer ***V*** and a project ***P***:

- A ***distribution_list*** is created to hold the contact information of all distinct volunteers in ***V***'s Upper DHT (in both ***finger table*** and ***successor list***);
- A ***contact_list*** is constructed to hold the contact information of all distinct volunteers i***n V***'s Lower DHT (in both ***finger table*** an***d successor list***).
- Divides the information in ***distribution_list*** among the volunteers in ***contact_list.*** – the difference between two policies resides in this part, and is given in form of pseudocode.
- Sends the divided sets of information to corresponding volunteers.

Distribution Policy One:

```
distributeUpperToLower (volunteer v , project p)
begin
    *the distributing policy here is depends on the size of distinct, alive volunteer in v's upper
    *DHT. If this size if lager than the size of the contact list, the longer part will be distributed
    *from the beginning of the contact list
    for j = 0, i = 0 to distribution_list.size() do
        sibling = distribution_list.get(i);
        if j >= contact_list.size() then
            j = 0;
        end
        contactor = contact_list.get(j);
        sibling_list(contactor).add(sibling);
    end
end
```

Distribution Policy Two:

```
distributeUpperToLower (volunteer v , project p)
begin
    * the distributing policy here is depends on the size of the contact list.
    * each volunteer in the contact list will get addresses of all volunteers
    * in distribution list
    for i = 0 to contact_list.size() do
        contactor = contact_list.get(j);
        for j = 0 to distribution_list.size() do
            sibling = distribution_list.get(j);
            sibling_list(contactor).add(sibling);
        end
    end
end
```

5.2.3 Volunteer Computing Application

An English letter counting application was launched on the system. The project file is an English document. When the system starts to run, this project file will be partitioned into a set of equal sized task files. When each task has been finished by a volunteer, the result of the task will be written to a result file. When all tasks have been finished, the final result file will be generated consisting of pairs in the format: *(an English letter: the number of this English letter)*.

5.3 Configuration

5.3.1 Parameters

Simulations in this thesis were run under a set of parameters. Each simulation has a specific configuration of these parameters.

- **Network Size**: the number of volunteers on the network at the beginning of a simulation.
- **Task Size and Task Range**: the size of each task in characters, and the number of all tasks belonging to one project at the beginning of a simulation.
- **Degree**: the maximum number of neighbors that each volunteer on the network can have at the beginning of a simulation.
- **Chord Stabilization Cycle**: the interval between two successive stabilizations of Chord ring.
- **Finger Table Size**: the size of the **finger table** in each peer on a Chord ring.
- **Successor List Size**: the size of the **successor list** in each peer on a Chord ring. For each volunteer, it has been programmed as half of the number of its current workers.
- **Volunteer Computing Application Cycle**: the interval between two successive statuses checking on each volunteer.
- **Distribution Policy**: the **Sibling Information Distribution Policy** adopted in the reconnection procedure.

5.3.2 Considerations

Several considerations should be taken into account when configuring each simulation:

- Each volunteer in the system should have at least one task to process.
- **Chord Stabilization Cycle** should be set as the base cycle of **Volunteer Computing Application Cycle.** Since a change in the volunteer computing application layer on each peer will require the stabilization of all peers on the same Chord ring at the second layer.
- Under each configuration, a set of middle level volunteers fails concurrently. ***The set of concurrent failures occur after the construction of both two layers***, which means that all dispatching work and Chord ring construction work have been completed, and all Chord rings at the second layer have stabilized due to the latest system change, ***and before each volunteer in the system begin to compute its own tasks.***

5.4 Simulation One: Study The Success of Fault Tolerance of The System

This simulation has been launched to study the success of the fault-tolerance of the layered system. The network size was quite small, which enables a convenient observation on the system processing and fault-tolerant results. The success was mainly measured in the number of **Reconnection Operations** and was visualized by Ubigraph [17].

5.4.1 Configuration

Network Size	***36 (volunteer 0 – volunteer 35)***
Task Size and Task Range	***5000, 141 (from tasks 0 to task 140)***
Degree	***4***
Chord Stabilization Cycle	***Cc***
Volunteer Computing Application Cycle	***Cc*Network Size***
Finger Table size	***4***
Distribution Policy	***Policy one***

Based on the configuration, the volunteer computing dispatching tree is showed in Figure 7:

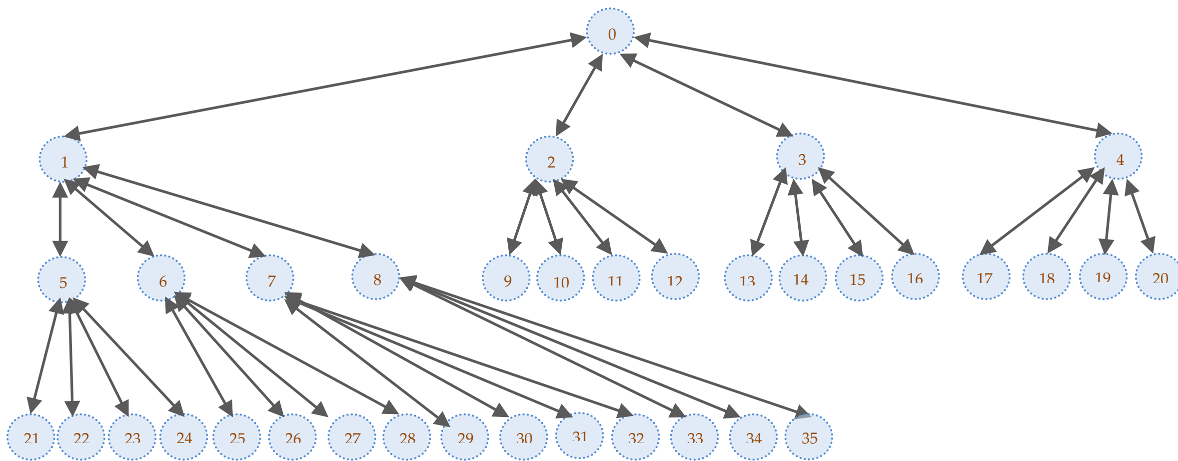


Figure 7: *each volunteer had at most four workers. There were two middle levels in the dispatching tree, together there were 8 middle level volunteers in the system.*

And the peer-to-peer network layer is showed by Ubigraph in figure 8:

Figure 8: *the label of each node in this graph in the format of "upper " or "lower" followed by a number and "(N…)". the "upper" or "lower" points out that the ring this node currently be a part of is viewed by this node as its Lower Chord Ring or its Upper Chord Ring. The number is the Chord id of this node on the ring. "(N..)" is the Volunteer id of this node on the network. The blue color marks the same node but appears as two, one for its attending in the Upper Chord Ring, and one for its attending in the Lower Chord Ring, which can be told from the "upper"/"lower" and the volunteer id.*

As shown in figure 7, the middle level volunteer list contains *volunteer 1 – 8*. The number of failed middle level volunteer (N_f) can be theoretically vary from *one* to *five*. For each Chord ring, the number of peers is at most *five*, and one of them is the master at the volunteer computing layer, therefore, the maximal allowable failed number should be *three* of each Chord ring. But for the Chord ring formed by *volunteer 1* and *volunteer 5 – 8*, *volunteer 1* belongs to both Chord rings, so the theoretical maximal failed number allowed by the system under the configuration should be *five* – *three* from the dispatch level 1 and *two* from the dispatch level 2.($N_f \in \{1,2,3,4,5\}$)

The number of the average stabilization cycles of each volunteer before a set of middle level volunteers fail concurrently is *five*.

5.4.2 Simulation Results

Following shows the success produced under some harsh conditions of a set of failed middle level volunteers (the analysis in the former section showed that the largest number of failed middle level volunteers should be 5, the harsh conditions therefore were sampled according to this number).

N_f = 4, **Cc** = 5

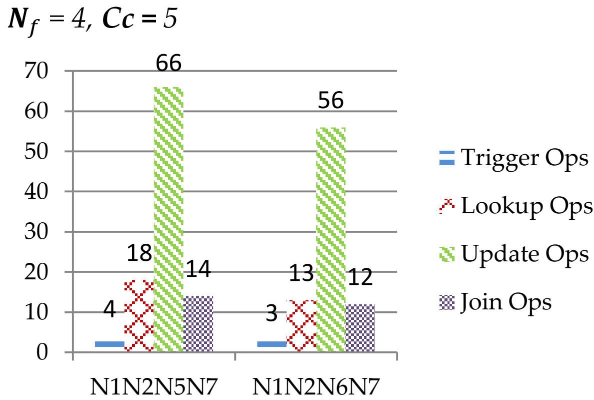


Figure 9: *The columns stand for four types of reconnecting operations respectively, under two harsh failure situations, each situation had four volunteers from different middle levels failed concurrently (volunteer N_1, N_2, N_5 and N_7 failed concurrently; volunteer N_1, N_2, N_6 and N_7 failed concurrently). The system succeeded to reconnect itself only under the first failure situation.*

N_f = 5, **Cc** = 5

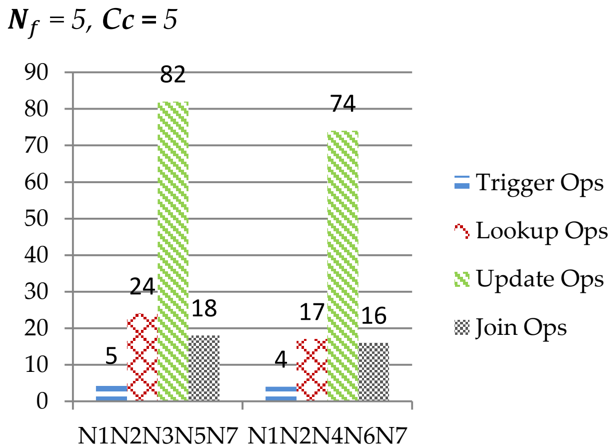


Figure 10: *The columns stand for four types of reconnecting operations respectively, under two harsh failure situations, each situation had five volunteers from different middle levels failed concurrently(volunteer N_1, N_2, N_3, N_5 and N_7 failed concurrently; volunteer N_1, N_2, N_4, N_6 and N_7 failed concurrently). The system succeeded to reconnect itself only under the first failure situation.*

When the failed volunteer sets are {N_1, N_2, N_3, N_5, N_7} and {N_1, N_2, N_3, N_6, N_7}, the system reconnection results as showed in figure 11 and figure 12 respectively:

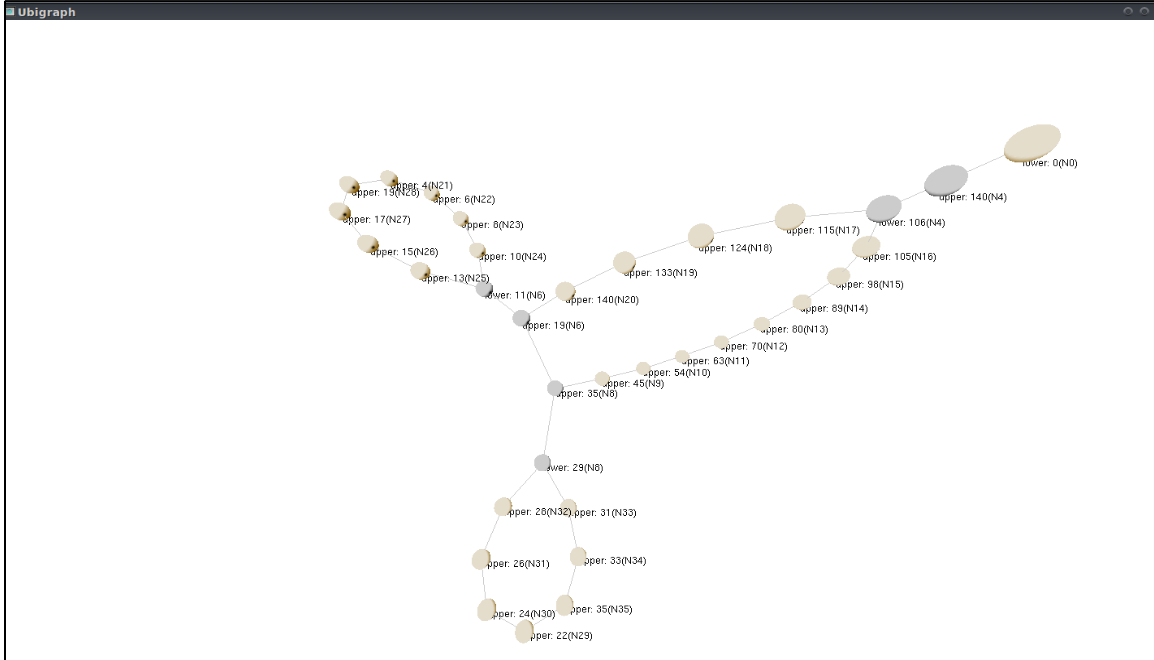


Figure 11: *Volunteer N_1, N_2, N_3, N_5 and N_7 failed concurrently, in the graph, N_0 only had N_4 as its worker; N_4 took all workers of N_1, N_2 and N_3. N_6 took all workers of N_5, and N_8 took all workers of N_7.*

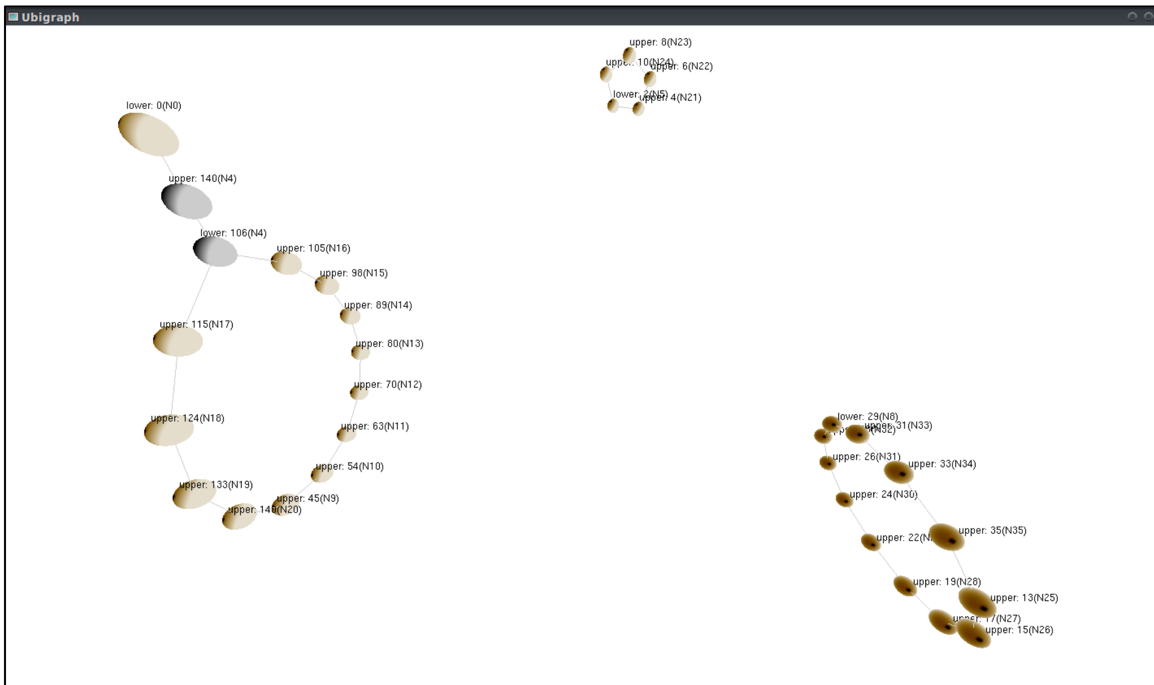


Figure 12: *Volunteer N_1, N_2, N_3, N_6 and N_7 failed silently, now in the graph, the set of Chord rings were drifted away from each other, the system has failed to reconnect itself.*

5.4.3 Analysis

As the results showed in figure 9 and figure 10, they both showed one successful case and one unsuccessful case under two **harsh failure situations** respectively (harsh failure situation is the situation where equal to or larger than half number of volunteers on each middle level fail concurrently). For example, in figure 10, the first set of columns showed that the system succeeded to reconnect itself under the failure situation where volunteer 1 – 3 from the first middle level and volunteer 5, 7 from the second middle level of the dispatching tree failed concurrently, and the final result file was also correctly produced. As presented in the figure 11, the system at the end was still connected.
But in contract, under the failure situations, for example, volunteer 1, 2, 6, 7 failed together (the second set of columns in figure 9), the system was failed to reconnect itself, therefore, no final result was produced, and the Chord rings on the peer-to-peer network were dissociated as showed in figure 12 by Ubigraph. It's obviously showed that the successful case had different values of each column from the unsuccessful case. In the failure situation that four middle level volunteers failed silently, the number of Reconnection Trigger Operations should be larger than four, since to make the system successfully reconnect itself, at least one worker of each failed volunteers should take the Reconnection Trigger operation, but in the unsuccessful case, the result was less than four, which apparently showed that at least one failed volunteer got no workers to successfully trigger the reconnection procedure after its leaving. Therefore, by debugging into the system processing, it's found that: after these four volunteers left the network, $\boldsymbol{N}_1$'s remain workers should be able to contact its currently online siblings – $\boldsymbol{N}_3$ and $\boldsymbol{N}_4$. The currently existing workers of $\boldsymbol{N}_1$ were $\boldsymbol{N}_5$ and $\boldsymbol{N}_8$. For $\boldsymbol{N}_5$, it only held the information of $\boldsymbol{N}_2$, who failed with $\boldsymbol{N}_1$; and for $\boldsymbol{N}_8$, it held nothing. So the reconnection phase failed under the case. Since the system was lack of enough sibling information.

5.5 Simulation Two: Study The Tradeoff of Fault Tolerance of The System

This simulation has been launched to study the influence of the Distribution Policy both on the fault-tolerance capability and the information redundancy of the layered system, and also combined with the success. In this simulation, the network size was relative big, and two different distribution policies defined previously were used. The tradeoff was presented mainly by three types of records related to the distributed sibling information:

1) $\boldsymbol{N}_s$: **The number of siblings** whose information is held by a volunteer that has detected the failure of its master, and tries to trigger the reconnection procedure.

2) N_{as}: **The number of live siblings** whose information is held by a volunteer that has detected the failure of its master, and tries to trigger the reconnection procedure.
3) N_v: **The number of volunteers** who are still alive on the network, and has detected the failure of its master, then tried to trigger the reconnection procedure with non-empty sibling information list.

In the simulation results part, the average of N_s and the average of N_{as} were calculated to show the average information redundancy in each volunteer as a worker for fault-tolerance of the system under a specific distribution policy.

5.5.1 Configuration

Network Size	*1111 (volunteer 0 – volunteer 1110)*
Task Size and Task Range	***600, 1175 (from tasks 0 to task 1174)***
Degree	***10***
Chord Stabilization Cycle	***Cc***
Volunteer Computing Application Cycle	***Cc*Network Size***
Finger Table size	***5***
Distribution Policy	***both policy one and policy two***

Based on the configuration, the volunteer computing dispatching tree is showed in Figure 13:

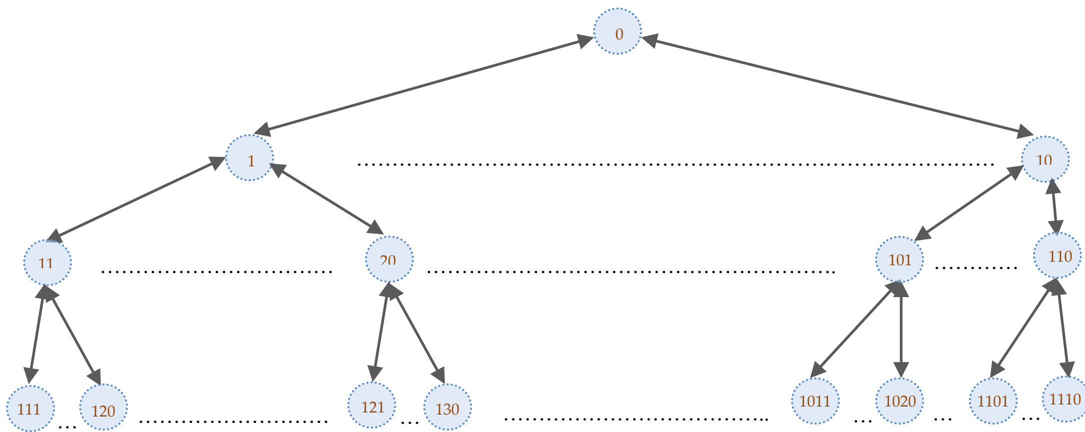


Figure 13: *each volunteer had ten workers. There were two middle levels in the full dispatching tree, together there were 110 middle level volunteers in the system.*

5.5.2 Simulation Results

The failure situation can be any arrangement of *110* middle level volunteers from *volunteer 1* to *volunteer 110* failing concurrently during system processing. The

simulation sampled a set of arrangements that gave a harsh failure situation to the system. Each combination of middle level volunteers was made from two sub arrangements: one was taken from the first middle level where *volunteer 1* to *volunteer 10* held their positions; another sub arrangement was taken from the second middle level on this tree structure, from *volunteer 11* to *volunteer 110.*

Failure Situation 1:

Sub arrangement 1 (from the first middle level)	***{1,2,4,7,9,10}***
Sub arrangement 2 (from the second middle level)	***{1th , 2th, 5th, 7th, 9th, 10th workers of each volunteer at the first middle level}***
The total amount of failed middle level volunteers	***66***

	Succeeded	N_s	N_{as}	N_v	***Avg of*** N_s	***Avg of*** N_v
Policy one	***yes***	***146***	***66***	***145***	***1.00***	***0.45***
Policy two	***yes***	***390***	***172***	***66***	***5.91***	***2.60***

Failure Situation 2:

Sub arrangement 1 (from the first middle level)	***{1,2,4,6,7,8,9,10}***
Sub arrangement 2 (from the second middle level)	***{1th , 2th, 5th, 6th, 7th, 9th, 10th workers of each volunteer at the first middle level}***
The total amount of failed middle level volunteers	***78***

	Succeeded	N_s	N_{as}	N_v	***Avg of*** N_s	***Avg of*** N_v
Policy one	***no***	***220***	***76***	***219***	***1.00***	***0.35***
Policy two	***yes***	***460***	***143***	***78***	***5.90***	***1.83***

Failure Situation 3:

Sub arrangement 1 (from the first middle level)	*{1,2,4,6,7,8,9,10}*
Sub arrangement 2 (from the second middle level)	***{1th , 2th, 5th, 6th, 7th, 8th,9th, 10th workers of each volunteer at the first middle level}***
The total amount of failed middle level volunteers	*88*

	Succeeded	N_s	N_{as}	N_v	$Avg\ of\ N_s$	$Avg\ of\ N_v$
Policy one	***no***	***311***	***61***	***310***	***1.00***	***0.20***
Policy two	***no***	***1240***	***113***	***208***	***5.96***	***0.54***

5.5.3 Analysis

The system under both policies was successful to tolerate the concurrent failures of *66* middle level volunteers in situation 1; in situation 2, the system under policy two was successful to tolerate the concurrent failures of *78* middle level volunteers, but was unsuccessful under policy one; in the last situation, the system was unsuccessful under both policies to tolerate *88* middle level volunteers' failures.

Although both policies enabled the system to tolerate the failures of more than half middle level volunteers, the second policy seems better than the first policy when failure situations become harsher (more middle level volunteers failed concurrently). But in the column "$Avg\ of\ N_s$" (*the average number of siblings whose information was kept in each volunteer that has detected the failure of its master*), under policy two, the values in all the three situations were around *5.9*, which means that ***under policy two, each volunteer as a worker kept the information of averagely 5 or 6 siblings of its master for fault-tolerance***; contrarily, the values under policy one in all these situations were around *1*, which means that ***under policy one, each volunteer as a worker kept the information of averagely one sibling of its master for fault-tolerance.*** The policy two gained higher possibility of fault-tolerance at the price of higher information redundancy.

When the failure situation became harsher, the value in column "$Avg\ of\ N_v$" (*the average number of live siblings whose information was kept in each volunteer that has detected the failure of its master)* decreased from *0.45* to *0.20* under policy one, decreased from *2.60* to *0.54* under policy two. **The thinking is: although higher information redundancy gives higher fault-tolerant capability, when given a certain capacity of information redundancy, how to adjust the distribution policy to improve or maintain the average number of live siblings kept in each worker, in other words, how to improve the**

fault-tolerant possibility without raising the information redundancy, this is a meaningful consideration in future,

6. Conclusion

In this thesis, a decentralized application architecture design for volunteer computing was proposed, where a multi-level tree structure is coupled with a peer-to-peer network layer. The layered design aims at providing the fault-tolerant capability to reconnect this multi-level tree, mainly due to the failures of the middle level volunteers, which lead to the cutoffs of the tree structure, and the influence of such kind of failures during the fault-tolerant procedure can be limited only between a master and all its workers. To observe the fault-tolerant behavior, a system prototype was built up in a simulation environment. Through a set of simple experiments, the results showed that:

- If the distribution policy is properly set, the system is able to tolerate more than half middle level volunteers' concurrent failures, except that all volunteers from the same middle level fail concurrently.
- To achieve the fault-tolerant capability mentioned above, compared with the size of the network, each volunteer as a worker needs to keep the information of only a quite small number of siblings of its failed master. This relatively small amount of information redundancy for fault-tolerance of the system is guaranteed by the guideline two, which requires the second layer design in a way to limit the influence of each middle level volunteer's failure as local as possible.

This thesis only focuses on the fault-tolerance for a decentralized application architecture design of volunteer computing, therefore, some other considerations need taking into account in future. For example, the storage of data files of a Volunteer Computing project should be designed and managed in a way to efficiently associate with this layered system. And if this layered design can be implemented in real distributed environment, a set of things should be added in the implementation, such as the security aspects of Volunteer Computing (e.g. data confidentiality and integrity, result and credit checking). More important is to study the real failure situations happened in the running of a specific Volunteer Computing application, which can produce more decisive conclusions for setting distribution policy, and also for studying the fault-tolerance of the system under different distribution policies.

Appendix

VCMatching.class

```
/*
 * @author: Hao Ning
 * 2010 Master Thesis:
 * Robust Overlay Networks for Volunteer Computing
 */

package example.vc;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.math.BigInteger;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Vector;

import example.vc.messages.Message;
import example.chord.ChordProtocol;
import example.chord.LookUpMessage;

import peersim.cdsim.CDProtocol;
import peersim.config.Configuration;
import peersim.config.FastConfig;
import peersim.core.*;
import peersim.transport.Transport;
import peersim.vector.MultiValueHolder;
import peersim.edsim.EDProtocol;
import peersim.edsim.EDSimulator;

import example.vc.messages.*;
```

```
/**
 *
 * This class describes the volunteer computing application performing on the top layer
 *
 */
public class VCMatching extends MultiValueHolder implements EDProtocol,
CDProtocol {

   // ---------------------------------------------------------------------
   // Parameters
   // ---------------------------------------------------------------------
        /**
         *      The Transport protocol used by the the protocol.
         */
        private static final String PAR_TRANSPORT="transport";

   // ---------------------------------------------------------------------
   // Fields
   // ---------------------------------------------------------------------
        //the current capacity of this volunteer
        private Capacity currentCapacity;

        //the current node
        private long thisNode;

        // the project manager in each volunteer
        //takes charge of a set of projects this node has recerived tasks from
        private LinkedList<ProjectManager> pms;

        // transport layer protocol ID
        private int tid;

        // the Chord ID of this volunteer in its Upper Chord ring
        private int upChordID;

        // the Chord ID of this volunteer in its Lower Chord ring
        private int lowChordID;

        // the data structure recoding the range of tasks
        //of this volunteer got from its master under a project
        private List<Long> taskRange;
```

```
// the data structure recoding the range of tasks
//of this volunteer kept for itself to compute under a project
private List<Long> loadRange;

// the DHT of the Upper Chord ring of this volunteer
private ChordProtocol upperDHT;

// the DHT of the Lower Chord ring of this volunter
private ChordProtocol lowerDHT;

// the list recording the contact information of the siblings of this volunteer's
master
private LinkedList<Long> parentSiblingID;

// the number of tasks that this volunteer's master takes charge for
private int taskNum;

// the first task id of the tasks taken charge by this volunteer's master
//this number combined with taskNnm enable the range queries on Chord ring
for reconnection
private int firstTaskNum;

// the time when this volunteer join a project
private long startTime;

//the time when this volunteer finishes all the tasks of a project
private long endTime;

// records the reduce process to check the final result
public String reduceMonitor = ",";

// recording the numbers of four types of reconnecting operations for fault-
tolerance
private int reconnectLookups;
private int reconnectUpdates;
private int reconnectJoins;
private int reconnectTriger;

//recording the information redundancy to study the fault-tolerance
private int siblingRedundancy;
private int aliveSiblingRedundancy;
```

```
//recording the counting results in <letter, count> format
private HashMap<String, Integer> finalResult;

//recording the reconnection requests sent to this volunteer for a failed master
private HashMap<BigInteger, Node> replceRecord;

//recording the reunited workers of a failed master
private HashMap<Long, String> reunionRecord;

/**
 * @return the siblingRedundancy
 */
public int getSiblingRedundancy() {
	return siblingRedundancy;
}

/**
 * @param siblingRedundancy the siblingRedundancy to set
 */
public void setSiblingRedundancy(int siblingRedundancy) {
	this.siblingRedundancy = siblingRedundancy;
}

/**
 * @return the aliveSiblingRedundancy
 */
public int getAliveSiblingRedundancy() {
	return aliveSiblingRedundancy;
}

/**
 * @param aliveSiblingRedundancy the aliveSiblingRedundancy to set
 */
public void setAliveSiblingRedundancy(int aliveSiblingRedundancy) {
	this.aliveSiblingRedundancy = aliveSiblingRedundancy;
}

public long getStartTime() {
	return startTime;
}
```

```
public void setStartTime(long startTime) {
        this.startTime = startTime;
}

public long getEndTime() {
        return endTime;
}

public void setEndTime(long endTime) {
        this.endTime = endTime;
}

public HashMap<Long, String> getReunionRecord() {
        return reunionRecord;
}

public void setReunionRecord(HashMap<Long, String> reunionRecord) {
        this.reunionRecord = reunionRecord;
}

public HashMap<BigInteger, Node> getReplceRecord() {
        return replceRecord;
}

public void setReplceRecord(HashMap<BigInteger, Node> replceRecord) {
        this.replceRecord = replceRecord;
}

public void setFinalResult(HashMap<String, Integer> finalResult) {
        this.finalResult = finalResult;
}

public long getThisNode() {
        return thisNode;
}

public void setFirstTaskNum(int firstTaskNum) {
        this.firstTaskNum = firstTaskNum;
}

public LinkedList<Long> getParentSiblingID() {
        return parentSiblingID;
```

```
}

public void setParentSiblingID(LinkedList<Long> parentSiblingID) {
	this.parentSiblingID = parentSiblingID;
}

public int getTaskNum() {
	return taskNum;
}

public void setTaskNum(int taskNum) {
	this.taskNum = taskNum;
}

public void setThisNode(long thisNode) {
	this.thisNode = thisNode;
}

public Capacity getCurrentCapacity() {
	return currentCapacity;
}

public void setCurrentCapacity(Capacity currentCapacity) {
	this.currentCapacity = currentCapacity;
}

public LinkedList<ProjectManager> getPms() {
	return pms;
}

public void setPms(LinkedList<ProjectManager> pms) {
	this.pms = pms;
}

public void setUpperDHT(ChordProtocol dht){
	this.upperDHT = dht;
}

public void setLowerDHT(ChordProtocol dht){
	this.lowerDHT = dht;
}
```

```
public ChordProtocol getUpperDHT(){
	return this.upperDHT;
}

public ChordProtocol getLowerDHT(){
	return this.lowerDHT;
}

public int getUpChordID() {
	return upChordID;
}

public void setUpChordID(int upChordID) {
	this.upChordID = upChordID;
}

public int getLowChordID() {
	return lowChordID;
}

public List<Long> getTaskRange() {
	return taskRange;
}

public void setTaskRange(List<Long> taskRange) {
	this.taskRange = taskRange;
}

public List<Long> getLoadRange() {
	return loadRange;
}

public void setLoadRange(List<Long> loadRange) {
	this.loadRange = loadRange;
}

public void setLowChordID(int lowChordID) {
	this.lowChordID = lowChordID;
}

public int getReconnectLookups() {
	return reconnectLookups;
```

```
	}

	public void setReconnectLookups(int reconnectLookups) {
		this.reconnectLookups = reconnectLookups;
	}

	public int getReconnectUpdates() {
		return reconnectUpdates;
	}

	public void setReconnectUpdates(int reconnectUpdates) {
		this.reconnectUpdates = reconnectUpdates;
	}

	public int getReconnectJoins() {
		return reconnectJoins;
	}

	public void setReconnectJoins(int reconnectJoins) {
		this.reconnectJoins = reconnectJoins;
	}

	public int getReconnectTriger() {
		return reconnectTriger;
	}

	public void setReconnectTriger(int reconnectTriger) {
		this.reconnectTriger = reconnectTriger;
	}

	// --------------------------------------------------------------------------
// Initialization
// --------------------------------------------------------------------------
/**
 * Standard constructor that reads the configuration parameters. Invoked by
 * the simulation engine.
 *
 * @param prefix
 *          the configuration prefix for this class.
 */
public VCMatching(String prefix) {
	super(prefix);
```

```
    tid = Configuration.getPid(prefix+"."+PAR_TRANSPORT);
    pms = new LinkedList<ProjectManager>();
    upperDHT = null;
    lowerDHT = null;
    upChordID = -1;
    lowChordID = -1;
    taskNum = 0;
    firstTaskNum = 0;
    parentSiblingID = null;
    loadRange = new ArrayList<Long>();
    taskRange = new ArrayList<Long>();
    finalResult = new HashMap<String, Integer>();
    this.replceRecord = null;
    this.reunionRecord = null;
    this.startTime = 0L;
    this.endTime = 0L;
    reconnectLookups = 0;
    reconnectJoins = 0;
    reconnectUpdates = 0;
    reconnectTriger = 0;
}

// The clone() method is inherited.
public Object clone(){
            Object prot = null;
            try{
                prot = (VCMatching)super.clone();
            }catch(Exception e){}
            ((VCMatching)prot).pms = new LinkedList<ProjectManager>();
            ((VCMatching)prot).currentCapacity = null;
            ((VCMatching)prot).upperDHT = null;
            ((VCMatching)prot).lowerDHT = null;
            ((VCMatching)prot).upChordID = -1;
            ((VCMatching)prot).lowChordID = -1;
            ((VCMatching)prot).taskNum = 0;
            ((VCMatching)prot).firstTaskNum = 0;
            ((VCMatching)prot).parentSiblingID = null;
            ((VCMatching)prot).loadRange = null;
            ((VCMatching)prot).taskRange = null;
            ((VCMatching)prot).finalResult = null;
            ((VCMatching)prot).replceRecord = null;
            ((VCMatching)prot).reunionRecord = null;
```

```
                ((VCMatching)prot).startTime = 0L;
                ((VCMatching)prot).endTime = 0L;
                ((VCMatching)prot).reconnectLookups = 0;
                ((VCMatching)prot).reconnectJoins = 0;
                ((VCMatching)prot).reconnectUpdates = 0;
                ((VCMatching)prot).reconnectTriger = 0;
                return prot;
        }

    // --------------------------------------------------------------------
    // Methods
    // --------------------------------------------------------------------
    //a flag to control each volunteer begin to compute after the concurrent failures occure
    public int count = -2;

        @Override
        /**
     * Periodically (each cycle) triggered on each volunteer,
     * to check if need to compute tasks or to submit results,
     * and more important, to check if the master of a project is still alive
     *
     * @param node: the current node.
     * @param protocolID: the ID of this protocol
     *
     */
        public void nextCycle(Node node, int protocolID) {
                if(count>=-1){
                        this.computeSubmit(0, protocolID);
                        ProjectManager pm = this.containsPro(pms, 0);
                        if(pm == null)return;
                        if(this.endTime<=0L){
                HashMap<Long, String> results = new HashMap<Long, String>();
               Message newMsg = new Message(thisNode, new TaskSubmitMessage(0,
results, 0, null));
                this.reduceResults(newMsg,protocolID);
            }
                        count++;
                }
        }

        /**
     * message switcher
```

```
 *
 * @param node: the current node who is the destination of the incoming message.
 * @param pid: the ID of this protocol
 * @param event: the incoming message
 *
 */
public void processEvent( Node node, int pid, Object event ) {
	Message msg = (Message)event;
	int mType = msg.getType();
	switch(mType){
	case 1://TaskRequestMessage@
		this.answerTaskRequest(msg, pid);
		break;
	case 2://TaskResponseMessage@
		this.dispatchTasks(msg, pid);
		break;
	case 3://TaskDispatchMessage@
		this.beginComputing(msg, pid);
		break;
	case 4://TaskSubmitMessage@
		this.reduceResults(msg,pid);
		break;
	case 5://ParentSiblingNotify@
		this.setParentSibling(msg);
		break;
	case 6://MasterFailedNotifySibling@
		this.lookupNewHandlers(msg, pid);
		break;
	case 7://NewMasterPM@
		this.updatePMSRse(msg, pid);
		break;
	case 8://NewMasterPMRes@
		this.updatePM(msg, pid);
		break;
	case 9://JoinNotify@
		this.joinToReconnect(msg);
		break;
	case 10://RessignReq@
		this.reassign(msg,pid);
		break;
	case 11://RegisterTaskReq@
		this.registerTaskNotify(msg,pid);
```

```
                break;
        case 12://RegisterTaskRes@
                this.registerTasks(msg, pid);
                break;
        default:
                System.out.println("Unrecognized message type: " + mType);
                break;
        }
    }

    /**
     * dealing with the Task Request Message got from a master volunteer for a project
     *
     * @param msg: Task Request Message
     * @param pid: the ID of this protocol
     *
     */
    void answerTaskRequest(Message msg, int pid){
                //check if the received the requiring tasks are already
        this.startTime = System.currentTimeMillis();
                TaskRequestMessage data = (TaskRequestMessage)msg.getData();
                Requirement req = data.getRequire();
                int proId = data.getProId();
                Node sourceNode = getNode(thisNode);
                Node targetNode = getNode(msg.getSource());
                System.out.println("MSG EXECUTION: TaskRequestMessage received by
volunteer "+ thisNode+" from volunteer "+msg.getSource());
                if(containsTasks(this.pms, proId, data.getTaskIds())==null){
        if(this.getCurrentCapacity().getFreeDisk()<=req.getDisk()||this.getCurrentCapac
ity().getFreeMemory()<=req.getCpuUtil())return;
                        msg = new Message(thisNode,new
TaskResponseMessage(this.getCurrentCapacity(),proId, data.getTaskIds()));
                        long latency =
((Transport)sourceNode.getProtocol(tid)).getLatency(sourceNode, targetNode);
                        EDSimulator.add(latency, msg, targetNode, pid);
                        System.out.println("MSG ENQUEUE: TaskResponseMessage sent
by volunteer "+ thisNode+" to volunteer "+targetNode.getID());
                }
        }

    /**
```

```
    * dealing with the Task Response Message got from a volunteer may work for this
node for a project
    *
    * @param msg: Task Response Message
    * @param pid: the ID of this protocol
    *
    */
   void dispatchTasks(Message msg, int pid){
		TaskResponseMessage data = (TaskResponseMessage)msg.getData();
		Capacity workerCapacity = data.getCapacity();
		int proId = data.getProId();
		System.out.println("MSG EXECUTION: TaskRequestMessage received by
volunteer "+ thisNode+" from volunteer "+msg.getSource());
		ProjectManager pm = containsTasks(this.pms, proId, data.getTaskIds());
		Node sourceNode = getNode(thisNode);
		if(pm!=null){
			LinkedList<Task> outtasks = new LinkedList<Task>();
			String taskids = "";
			for(int id:data.getTaskIds()){
				taskids += ","+id+",";
			}
			for(Task task:pm.getDispatchedTasks()){
				if(taskids.contains(","+task.getId()+",")){
					task.setAssigner((Volunteer)sourceNode);

	task.setExecuter((Volunteer)getNode(msg.getSource()));
					task.setStatus("dispatched");
					outtasks.add(task);
				}
			}
			Node targetNode = getNode(msg.getSource());
			msg = new Message(thisNode, new
TaskDispatchMessage(outtasks, pm.getDistrLevel()+1));
			long latency =
((Transport)sourceNode.getProtocol(tid)).getLatency(sourceNode,targetNode);
			EDSimulator.add(latency,msg,targetNode,pid);
			System.out.println("MSG ENQUEUE: TaskDispatchMessage sent
by volunteer " + thisNode
					+ " to volunteer " + targetNode.getID());
		}
	}
```

```
   /**
    * dealing with the Task Dispatch Message got from the master volunteer for this node
for a project
    *
    * @param msg: Task Dispatch Message
    * @param pid: the ID of this protocol
    *
    */
        void beginComputing(Message msg, int pid){
                TaskDispatchMessage data = (TaskDispatchMessage)msg.getData();
                LinkedList<Task> tasks = data.getTasks();
                int proId = tasks==null||tasks.size()<1?-1:tasks.get(0).getProId();
                System.out.println("MSG EXECUTION: TaskDispatchMessage received
by volunteer "+ thisNode +" from volunteer "+msg.getSource());
                //create or update a new Project manager
                if(this.pms == null)this.pms = new LinkedList<ProjectManager>();
                ProjectManager pm = this.containsPro(pms, proId);
                Node node = getNode(thisNode);
                long source = msg.getSource();
                if(pm == null){
                        LinkedList<Task> disT = new LinkedList<Task>();
                        LinkedList<Task> exeT = new LinkedList<Task>();
                        if(tasks!=null||tasks.size()>0){
                                tasks.get(0).setExecuter((Volunteer)node);
                                tasks.get(0).setAssigner((Volunteer)getNode(source));
                                //actually now haven't in process....now is just get it and
put it in charge
                                tasks.get(0).setStatus("inprocess");

        tasks.get(0).setDispatchLevel(tasks.get(0).getDispatchLevel()+1);
                                exeT.add(tasks.get(0));
                                this.loadRange.add(tasks.get(0).getId());
                                this.taskRange.add(tasks.get(0).getId());
                        }
                        for(int i = 0; tasks!=null&&i<tasks.size()-1;i++){

        tasks.get(i+1).setDispatchLevel(tasks.get(i+1).getDispatchLevel()+1);
                                disT.add(tasks.get(i+1));
                                this.taskRange.add(tasks.get(i+1).getId());
                        }
                        ProjectManager newPm = new ProjectManager((Volunteer)node, 0,
(Volunteer)getNode(source),
```

```
                        data.getLevel(), disT, exeT, -1);
                this.pms.add(newPm);
        }else {
                LinkedList<Task> haveDisTasks = pm.getDispatchedTasks();
                LinkedList<Task> haveExeTasks = pm.getExecuteTasks();
                String ids = "";
                if(haveDisTasks!=null&&haveDisTasks.size() > 0){
                        for(Task task: haveDisTasks){
                        ids += ","+task.getId()+",";
                }
                }
                if(haveExeTasks!=null&&haveExeTasks.size() > 0){
                        for(Task task: haveExeTasks){
                        ids += ","+task.getId()+",";
                }
                }
                for(Task task: tasks){
                        if(ids.contains(","+task.getId()+",")){
                                System.err.println("For volunteer "+node.getID()+",
in its ProjectManager for Project "+proId+" , has already existed task "+task.getId()+"!!!!");
                                return;
                        }
                }
                int index = 0;
                if(haveExeTasks!=null&&!isBusy(haveExeTasks)){
                        tasks.get(0).setExecuter((Volunteer)node);
                        tasks.get(0).setAssigner((Volunteer)getNode(source));
                        haveExeTasks.add(tasks.get(0));
                        index = 1;
                }
                //check whether there is tasks expect the one kept within own that
need to be dispatched out!
                if(tasks.size()<2)return;
                for(;index<tasks.size();index++){
                        haveDisTasks.add(tasks.get(index));
                }
        }
        //build up upper dht with its dispatcher's lower dht
        this.upperDHT.join(node, this.getUpChordID(),
BigInteger.valueOf(tasks.get(tasks.size()-1).getId()),
                        getNode(msg.getSource()),
BigInteger.valueOf(this.firstTaskNum),BigInteger.valueOf(this.taskNum), proId);
```

```
                //continue to dispatch out tasks if this node has more than one available
tasks
                int linkableID = FastConfig.getLinkable(pid);
        Linkable linkable = (Linkable) node.getProtocol(linkableID);
                int degree = linkable.degree();
                //check whether all my neighbors are alive
                List<Node> aliveNeighbors = new ArrayList<Node>();
                for (int a = 0;a<degree;a++){
                        Node neighbor = linkable.getNeighbor(a);
                        if(neighbor!=null&&neighbor.isUp()==true){
                                aliveNeighbors.add(neighbor);
                        }
                }
                degree = aliveNeighbors.size();
                LinkedList<Task> outtasks = this.containsPro(pms,
proId).getDispatchedTasks();
                int num = degree<1?0:degree-1;
                int count = 0;
                if(outtasks==null)count = 0;
                else if(outtasks.size()<num)
                        count = 1;
                else count =
num==0?0:(outtasks.size()%num==0?outtasks.size()/num:outtasks.size()/num+1);
                int m = 0;
                if(count>0&&degree>=2){
                        this.lowerDHT.chordId =
BigInteger.valueOf(this.containsPro(pms, proId).getExecuteTasks().getLast().getId());
                        this.lowerDHT.setFirstChordId(this.lowerDHT.chordId);

            this.lowerDHT.setLastChordId(BigInteger.valueOf(this.containsPro(pms,
proId).getDispatchedTasks().getLast().getId()));
                }
        for (int j = 0;degree > 0&&j<degree;j++){
            Node neighbor = aliveNeighbors.get(j);
                if(neighbor.getID()==source){
                        continue;
                }
                        long latency =
((Transport)node.getProtocol(tid)).getLatency(node,neighbor);
                        Requirement req = new Requirement("Micorsoft Windows 7
Professional", "One-processor InteL64 Family 6 Model", 0.03, 23, 100, -1);
```

```
                        List<Integer> taskIds = new ArrayList<Integer>();

                        for(int s = 0;m < outtasks.size()&&s<count; m++,s++){
                                taskIds.add((int)outtasks.get(m).getId());
                        }
                        VCMatching neighborVC =
(VCMatching)neighbor.getProtocol(pid);
                        neighborVC.setTaskNum(outtasks.size());
                        neighborVC.setFirstTaskNum((int)outtasks.get(0).getId()-1);
                        msg = new Message(thisNode, new TaskRequestMessage(req,0,
taskIds));
                        EDSimulator.add(latency,msg,neighbor,pid);
                        System.out.println("MSG ENQUEUE: TaskRequestMessage sent
by volunteer "+ node.getID()+" to volunteer "+neighbor.getID());
                        if(thisNode == 1&&neighbor.getID() == 5){
                                System.out.println("in cvm");
                        }

        neighborVC.setTaskNum(this.lowerDHT.getLastChordId().intValue());

        neighborVC.setFirstTaskNum(this.lowerDHT.getFirstChordId().intValue());
                        if(m == outtasks.size())break;
    }

    //if no worker and has untaken tasks then move those tasks to execution tasks and
process them
    pm = this.containsPro(pms, proId);
    LinkedList<Task> exeTasks = pm==null?new
LinkedList<Task>():pm.getExecuteTasks();

if((lowerDHT.predecessor==null||(BigInteger.ZERO.compareTo(lowerDHT.chordId)==0
                &&lowerDHT.predecessor.getID()==thisNode))&&degree<2){
        LinkedList<Task> dispatchedTasks = pm==null?new
LinkedList<Task>():pm.getDispatchedTasks();
        for(Task task:dispatchedTasks){
                task.setExecuter((Volunteer)node);
                exeTasks.add(task);
        }
        pm.setDispatchedTasks(new LinkedList<Task>());
    }
        }
```

```
        /**
    * periodically triggered to check the master's status and the computing process of
own tasks of a project
    *
    * @param proId: project ID
    * @param pid: the ID of this protocol
    *
    */
        public void computeSubmit(int proId, int pid){
                ProjectManager pm = this.containsPro(pms, proId);
      LinkedList<Task> exeTasks = pm==null?new
LinkedList<Task>():pm.getExecuteTasks();
      Node node = getNode(thisNode);
      boolean needCount = false;
      for(Task task:exeTasks){

        if(!task.getStatus().equals(Task.PAR_SUBMIT)&&!task.getStatus().equals(Task.P
AR_COMPLETE)){
                needCount = true;
                break;
        }
      }
      //give the value of this counting words flag
      if(needCount == false){
        return;
      }

      long proID = pm.getProId();

      //control node failures
      Node targetNode = pm.getMyDispatcher();

      //check if the project master is gone,then:
        if(pm.isReconnect()==false&&(targetNode.isUp()==false||targetNode == null)){
            pm.setReconnect(true);
                if(this.parentSiblingID==null||this.parentSiblingID.size()<1)return;
                Iterator<Long> parentSiblings = this.parentSiblingID.iterator();
                this.siblingRedundancy = this.parentSiblingID.size();
                for(Long id:this.parentSiblingID){
                        if(getNode(id)!=null&&getNode(id).isUp()==true){
                                this.aliveSiblingRedundancy++;
                        }
```

```
			}
			long aliveSibling = -1L;
			do{
				aliveSibling = parentSiblings.next();

		}while(parentSiblings.hasNext()&&(getNode(aliveSibling)==null||getNode(alive
Sibling).isUp()==false));
			if(aliveSibling>-1){
				//try to contact this master's sibling
				long failedMasterId = targetNode.getID();
				Message newMsg = new Message(thisNode, new
MasterFailedNotifySibling(proID,failedMasterId, this.upperDHT.chordId,
this.firstTaskNum, this.taskNum, pm.getDistrLevel()));
				targetNode = getNode(aliveSibling);
				if(targetNode == null)return;
			long latency = ((Transport)node.getProtocol(tid)).getLatency(node,
targetNode);
			EDSimulator.add(latency,newMsg,targetNode,pid);
			System.out.println("MSG ENQUEUE: Master Failed so Notify known
Master's Sibling("+this.parentSiblingID+") Message sent by volunteer "
								+ node.getID()+" to volunteer "+targetNode.getID());
			this.reconnectTriger++;
			}
			return;
		}

		countWords(exeTasks, pm.getDistrLevel());
		if(pm.getDistrLevel()==0){
			for(Task task:exeTasks){
				this.reduceWords(task.getResult(), this.finalResult);
				this.reduceMonitor += task.getId()+",";
			}
		}
	if(this.taskRange != null&&this.taskRange.size()==pm.getExecuteTasks().size()){
		HashMap<Long, String> results = new HashMap<Long, String>();
		for(Task task: exeTasks){
		results.put(task.getId(), task.getResult());
		}
		Message newMsg = new Message(thisNode, new TaskSubmitMessage(proID,
results, 0, null));
		long latency = ((Transport)node.getProtocol(tid)).getLatency(node, targetNode);
		EDSimulator.add(latency,newMsg,targetNode,pid);
```

```
                System.out.println("MSG ENQUEUE: Task Results Submission Message 
sent by volunteer "
                                + node.getID()+" to volunteer "+targetNode.getID());
                if(this.endTime <= 0L)
                        this.endTime = System.currentTimeMillis();
    }
        }

        /**
   * dealing with the Master Failed Notify Sibling Message got from a worker whose 
master has failed
   * and own contact information was distributed by this failed master to that worker.
   *
   * @param msg: Master Failed Notify Sibling Message
   * @param pid: the ID of this protocol
   *
   */
        public void lookupNewHandlers(Message msg, int pid){
                MasterFailedNotifySibling data = 
(MasterFailedNotifySibling)msg.getData();
                long proId = data.getProId();
                BigInteger chordId = data.getChorldId();
                ProjectManager pm = this.containsPro(pms, 
Long.valueOf(proId).intValue());
                if(pm == null)return;
                Node sender = getNode(thisNode);
                Node source = getNode(msg.getSource());
                int firstTaskId = data.getFirstTaskId();
                int lastTaskId =  data.getLastTaskId();
                long failedMasterId = data.getFailedMasterId();
                int dispatchLevel = data.getDispatchLevel();
                if(pm.getDistrLevel()+1==dispatchLevel){
                        //the replacer of the disappeared master is in thisNode's upper 
chord layer
                        ((Transport)sender.getProtocol(tid)).send(sender, sender, new 
LookUpMessage(sender, chordId, source, 1, Long.valueOf(proId).intValue(), firstTaskId, 
lastTaskId, failedMasterId), this.getUpChordID());
                        this.reconnectLookups++;
                }else if(pm.getDistrLevel()+2==dispatchLevel){
                        //the replacer of the disappeared master is in thisNode's lower 
chord layer
```

```
				((Transport)sender.getProtocol(tid)).send(sender, sender, new
LookUpMessage(sender, chordId, source, 1,  Long.valueOf(proId).intValue(), firstTaskId,
lastTaskId, failedMasterId), this.getLowChordID());
				this.reconnectLookups++;
			}else{
				return;
			}
		}

		/**
	 * the current volunteer as a new master, after get contact of all workers of the failed
master,
	 * will send Update Message to each of those workers to notify them to update the
relevant
	 * information in their Project Manager with this new master.
	 *
	 *
	 * @param proId: the project ID
	 * @param sender: one worker of the failed master
	 * @param pid: the ID of this protocol
	 *
	 */
		public void updatePMSReq(int proId, Node sender, int pid){
			ProjectManager pm = this.containsPro(pms, proId);
			if(pm == null)return;
			if(pm.getDispatchedTasks()!=null&&pm.getDispatchedTasks().size()>0){
				for(Task task: pm.getDispatchedTasks()){
					if(task.getExecuter().getID() == sender.getID()){
						return;
					}
				}
			}
			Node node = getNode(thisNode);
			Message newMsg = new Message(thisNode, new NewMasterPM(proId));
	    long latency = ((Transport)node.getProtocol(tid)).getLatency(node, sender);
	    EDSimulator.add(latency,newMsg,sender,pid);
			System.out.println("MSG ENQUEUE: New Master PM Request Message
sent by volunteer "
					+ node.getID()+" to volunteer "+sender.getID());
			this.reconnectUpdates++;
		}
```

```
	/**
    * dealing with the New Master PM message got from the new master for a project.
    * this volunteer as a worker of the failed master should update the relevant
information in its
    * project manager and submits results to the new master or send some related
information of its own tasks to the new master.
    *
    *
    * @param msg: New Master PM message
    * @param pid: the ID of this protocol
    *
    */
	public void updatePMSRse(Message msg, int pid){
		NewMasterPM data = (NewMasterPM)msg.getData();
		int proId = Long.valueOf(data.getProId()).intValue();
		Node target = getNode(msg.getSource());
		ProjectManager pm = this.containsPro(pms, proId);
		if(pm == null)return;
	Node node = getNode(thisNode);
	Node newMaster = getNode(msg.getSource());
	if(newMaster==null||newMaster.isUp()==false)return;
		if(pm.getMyDispatcher().getID()!=msg.getSource()){
			pm.setMyDispatcher((Volunteer)newMaster);
			VCMatching vcm = (VCMatching)newMaster.getProtocol(pid);
			ProjectManager masterPm = vcm.containsPro(vcm.pms, proId);
			pm.setDistrLevel(masterPm.getDistrLevel()+1);
		}
		LinkedList<Task> allTasks = new LinkedList<Task>();
		LinkedList<Task> exeTasks = pm.getExecuteTasks();
		LinkedList<Task> disTasks = pm.getDispatchedTasks();
		//check if all tasks in this pm have been finished by now, if it is, then
submit to its new master
		//submiting
		boolean completed = true;
		if(exeTasks!=null&&exeTasks.size()>0){
			for(Task task:exeTasks){
				if(task==null)continue;

	if(!(task.getStatus().equals(Task.PAR_SUBMIT)||(task.getStatus().equals(Task.P
AR_COMPLETE)&&task.getCompleteLevel()<pm.getDistrLevel()))){
					completed = false;
					break;
```

```
                    }
                }
            }
            if(completed&&disTasks!=null&&disTasks.size()>0){
                for(Task task:disTasks){

    if(!task.getStatus().equals(Task.PAR_COMPLETE)||(task.getStatus().equals(Task
.PAR_COMPLETE)&&task.getCompleteLevel()>pm.getDistrLevel())){
                        completed = false;
                        break;
                    }
                }
            }
    if(completed){
        LinkedList<Task> tasks = new LinkedList<Task>();
        HashMap<Long, String> midResults = new HashMap<Long, String>();
        for(Task task: exeTasks){
         midResults.put(task.getId(), task.getResult());
         tasks.add(task);
        }
        for(Task task: disTasks){
         midResults.put(task.getId(), task.getResult());
         tasks.add(task);
        }
        Message newMsg = new Message(thisNode, new
TaskSubmitMessage(Long.valueOf(proId), midResults, 1, tasks));
        long latency = ((Transport)node.getProtocol(tid)).getLatency(node, newMaster);
        EDSimulator.add(latency,newMsg,newMaster,pid);
            System.out.println("MSG ENQUEUE: Task Results Submission Message
sent by volunteer "
                    + node.getID()+" to volunteer "+newMaster.getID());
            if(this.endTime <= 0L)
                this.endTime = System.currentTimeMillis();
    }else{
        if(exeTasks!=null&&exeTasks.size()>0){
                for(Task task : exeTasks){
                    if(task == null)continue;
                    allTasks.add(task);
                }
            }
            if(disTasks!=null&&disTasks.size()>0){
                for(Task task : disTasks){
```

```
                    if(task == null)continue;
                    allTasks.add(task);
                }
            }
            Message newMsg = new Message(thisNode, new
NewMasterPMRes(proId, allTasks));
        long latency = ((Transport)node.getProtocol(tid)).getLatency(node, target);
        EDSimulator.add(latency,newMsg,target,pid);
            System.out.println("MSG ENQUEUE: New Master PM Response Message
sent by volunteer "
                    + node.getID()+" to volunteer "+target.getID());
            }
      this.reconnectUpdates++;
        }

        /**
    * dealing with the New Master PM Response message got from a worker of the failed
master for a project.
    * this volunteer as the new master of the failed master should update relevant task
information in its project manage.
    * and finally send join notify message to each worker of the failed master.
    *
    *
    * @param msg: New Master PM Response message
    * @param pid: the ID of this protocol
    *
    */
        public void updatePM(Message msg, int pid){
            NewMasterPMRes data = (NewMasterPMRes)msg.getData();
            int proId = Long.valueOf(data.getProId()).intValue();
            LinkedList<Task> tasks = data.getTasks();
            ProjectManager pm = this.containsPro(pms, proId);
            if(pm==null)return;
            LinkedList<Task> disTasks = pm.getDispatchedTasks();
            if(disTasks!=null&&tasks!=null&&tasks.size()>0){
                for(Task task : tasks){
                    task.setAssigner((Volunteer)getNode(thisNode));
                    disTasks.add(task);
                    this.taskRange.add(task.getId());
                }
                Collections.sort(taskRange);
                pm.setDispatchedTasks(disTasks);
```

```
			if(taskRange!=null&&taskRange.size()>0){

	this.lowerDHT.setFirstChordId(BigInteger.valueOf(taskRange.get(0)));
			}
		}
		//if this volunteer has finished, reset its finish time
		if(this.endTime>0){
			this.endTime = 0;
		}
		//check if this node is at the first time to have a worker

	if(this.lowerDHT.successorList!=null&&this.lowerDHT.successorList.length>0&
&(this.lowerDHT.successorList[0]==null||this.lowerDHT.successorList[0].getID()==this
Node)){
			this.lowerDHT.chordId =
BigInteger.valueOf(pm.getExecuteTasks().get(0).getId());
		}
		Node node = getNode(thisNode);
		Node target = getNode(msg.getSource());
		BigInteger firstTaskId = this.lowerDHT.getFirstChordId();
		BigInteger lastTaskId = this.upperDHT.chordId;
		Message newMsg = new Message(thisNode, new JoinNotify(proId,
firstTaskId, lastTaskId));
        long latency = ((Transport)node.getProtocol(tid)).getLatency(node, target);
        EDSimulator.add(latency,newMsg,target,pid);
		System.out.println("MSG ENQUEUE: join notify Message sent by
volunteer "
				+ node.getID()+" to volunteer "+target.getID());
		this.reconnectUpdates++;
	}

	/**
     * dealing with the Join Notify message got from the new master for a project.
     * this volunteer as a worker of the failed master now got this Join Notify message
from the new master,
     * so it can join the new master's Lower Chord ring, to become a new worker of the
new master.
     *
     *
     * @param msg: Join Notify message
     * @param pid: the ID of this protocol
     *
```

```
    */
        public void joinToReconnect(Message msg){
                Node newMaster = getNode(msg.getSource());
                JoinNotify data = (JoinNotify)msg.getData();
                this.upperDHT.varSuccList = 0;
                this.upperDHT.fingerTable = new Node[this.upperDHT.m];
                this.upperDHT.successorList = new Node[this.upperDHT.succLSize];
                this.upperDHT.predecessor = null;
                this.upperDHT.setNext(0);
                for(int a= 0; a<this.upperDHT.succLSize; a++){
                        this.upperDHT.successorList[0] = getNode(thisNode);
                }
                this.upperDHT.join(getNode(thisNode), this.getUpChordID(),
this.upperDHT.chordId,
                                newMaster, data.getFirstTaskId(), data.getLastTaskId(),
Long.valueOf(data.getProId()).intValue());
                this.reconnectJoins++;
        }


        /**
    * dealing with the Task Submit message got from a worker for a project.
    * this volunteer as the master of the sender will reduce the results contained in the
message.
    *
    *
    * @param msg: Task Submit message
    * @param pid: the ID of this protocol
    *
    */
        public void reduceResults(Message msg,int pid){
                TaskSubmitMessage data = (TaskSubmitMessage)msg.getData();
                long proId = data.getProId();
                HashMap<Long, String> results = data.getResults();

                int mode = data.getMode();
                ProjectManager pm = this.containsPro(pms, (int)proId);
                if(pm == null||results == null)return;
                LinkedList<Task> dispatchedTasks = pm.getDispatchedTasks();
                if(dispatchedTasks == null || dispatchedTasks.size()<=0)return;
                if(mode==1){
                        LinkedList<Task> tasks = data.getDisTasks();
```

```
			if(tasks == null||tasks.size()<=0)return;
			for(Task task:tasks){
				dispatchedTasks.add(task);
			}
		}
		//if my sibling is dead, and now one of its workers is submitting task
results to me since i'm the new master,
		//and since this worker has also replaced one of its sibling but this
happened after this worker reconnected to me,
		//namely, this worker may submit some tasks that are not registered in
my dispatch out task list.
		HashMap<Long, String> unRegisterTasks = (HashMap<Long,
String>)results.clone();
		LinkedList<Task> executeTasks = pm.getExecuteTasks();
		Iterator<Long> it = results.keySet().iterator();
		while(it.hasNext()){
			long taskId = it.next();
			String result = results.get(taskId);
			for(int i = 0; i < dispatchedTasks.size(); i++){
				Task task = dispatchedTasks.get(i);
				if(task == null)continue;
				if(task.getId() == taskId){
					task.setResult(result);
					task.setStatus(Task.PAR_COMPLETE);
					task.setCompleteLevel(pm.getDistrLevel());
					//remove this task since it can be found in my
dispatch list
					unRegisterTasks.remove(taskId);
					//if this node is the root node, then it will final
reduce the results
					if(pm.getDistrLevel() == 0){
						reduceWords(result, this.finalResult);
					this.reduceMonitor += task.getId()+",";
					}
				}
			}
		}
		//check if there are some non-registered tasks in my dispatch list
		if(unRegisterTasks!=null&&unRegisterTasks.size()>0){
			Vector<Long> v = new Vector<Long>(unRegisterTasks.keySet());
			Collections.sort(v);
			Iterator<Long> unRT = v.iterator();
```

```
					Node node = getNode(thisNode);
					Node targetNode = getNode(msg.getSource());
					Message newMsg = new Message(thisNode, new
RegisterTaskReq(proId, unRT));
		long latency = ((Transport)node.getProtocol(tid)).getLatency(node, targetNode);

		EDSimulator.add(latency,newMsg,targetNode,pid);
				System.out.println("MSG ENQUEUE: Task Register Request Message
sent by volunteer "
						+ node.getID()+" to volunteer "+targetNode.getID());
					return;
				}
				//check if all tasks in this pm have been finished by now, if it is, then
submit to its parent
				//submitting:
				//at meanwhile, check if some of its workers have been failed, so their
load tasks will be computed by this node
				HashMap<Long, LinkedList<Task>> missTasks = new HashMap<Long,
LinkedList<Task>>();
				boolean completed = true;
				for(Task task:executeTasks){

		if(!(task.getStatus().equals(Task.PAR_SUBMIT)||(task.getStatus().equals(Task.P
AR_COMPLETE)&&task.getCompleteLevel()<pm.getDistrLevel()))){
						completed = false;
						break;
					}
				}
				if(completed){
					for(Task task:dispatchedTasks){

		if(!task.getStatus().equals(Task.PAR_COMPLETE)||(task.getStatus().equals(Task
.PAR_COMPLETE)&&task.getCompleteLevel()>pm.getDistrLevel())){
							completed = false;

		if((task.getExecuter()==null||task.getExecuter().isUp()==false)&&missTasks.cont
ainsKey(task.getExecuter().getID())){
								LinkedList<Task> taskList =
missTasks.get(task.getExecuter().getID());
								taskList.add(task);
								missTasks.put(task.getExecuter().getID(),
taskList);
```

```
                                        }else
if((task.getExecuter()==null||task.getExecuter().isUp()==false)&&!missTasks.containsKe
y(task.getExecuter().getID())){
                                                LinkedList<Task> taskList = new
LinkedList<Task>();
                                                taskList.add(task);
                                                missTasks.put(task.getExecuter().getID(),
taskList);
                                        }
                                }
                        }
                }
                if(!completed&&missTasks.size()>0){
                        Iterator<Long> keyset = missTasks.keySet().iterator();
                        HashMap<Long, LinkedList<Task>> tmp = new HashMap<Long,
LinkedList<Task>>();
                        while(keyset.hasNext()){
                                Long executer = keyset.next();
                                LinkedList<Task> tasks = missTasks.get(executer);
                                if(tasks==null||tasks.size()<=0)continue;
                                if(tasks.size() == 1){
                                        if(tmp.containsKey(-10L)){
                                                LinkedList<Task> ts = tmp.get(-10L);
                                                ts.add(tasks.get(0));
                                                tmp.put(-10L, ts);
                                        }else{
                                                LinkedList<Task> ts = new
LinkedList<Task>();
                                                ts.add(tasks.get(0));
                                                tmp.put(-10L, ts);
                                        }
                                }else{
                                        //lunch query to find the current key space keeper
of the unfinished set of tasks
                                        Node node = getNode(thisNode);
                                        ((Transport)node.getProtocol(tid)).send(node, node,
                                                        new LookUpMessage(node,
BigInteger.valueOf(tasks.get(0).getId()), null, 3, Long.valueOf(proId).intValue(), -1, -1,
executer),
                                                        this.lowChordID);
                                }
                        }
```

```
				if(tmp.containsKey(-10L)&&tmp.get(-10L)!=null&&tmp.get(-
10L).size()==missTasks.size()){
					LinkedList<Task> ls = tmp.get(-10L);
					this.countWords(ls, pm.getDistrLevel());
					for(Task each: ls){
						each.setStatus(Task.PAR_COMPLETE);
						each.setExecuter((Volunteer)getNode(thisNode));
						each.setCompleteLevel(pm.getDistrLevel());
						if(pm.getDistrLevel() == 0){
							reduceWords(each.getResult(),
this.finalResult);
						this.reduceMonitor += each.getId()+",";
						}
					}
					completed = true;
					for(Task task:dispatchedTasks){

	if(!task.getStatus().equals(Task.PAR_COMPLETE)||(task.getStatus().equals(Task
.PAR_COMPLETE)&&task.getCompleteLevel()>pm.getDistrLevel())){
							completed = false;
							break;
						}
					}
				}
			}
	if(completed){
	    if(pm.getDistrLevel() == 0){
			String finalFile = "results\\Result_Pro.txt";
			try{
				File result = new File(finalFile);
				RandomAccessFile outt = new RandomAccessFile(result, "rws");
					String lineSeparator = (String)
java.security.AccessController.doPrivileged(
							new
sun.security.action.GetPropertyAction("line.separator"));
					for(int j = 97;j<=122;j++){
						char c = (char)j;
						String key = String.valueOf(c);
						if(this.finalResult.containsKey(key)){

	outt.writeBytes(key+":"+this.finalResult.get(key));
							outt.writeBytes(lineSeparator);
```

```
                    }
                }
                outt.close();
                if(this.endTime <= 0L)
                    this.endTime = System.currentTimeMillis();
            }catch (FileNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
    }else{
        Node node = getNode(thisNode);
    Node targetNode = pm.getMyDispatcher();
     long proID = pm.getProId();
     HashMap<Long, String> midResults = new HashMap<Long, String>();
     for(Task task: executeTasks){
        if(task.getCompleteLevel()>-
1&&task.getCompleteLevel()<pm.getDistrLevel()){
            continue;
        }
        midResults.put(task.getId(), task.getResult());
     }
     for(Task task: dispatchedTasks){
        if(task.getCompleteLevel() == pm.getDistrLevel()){
            midResults.put(task.getId(), task.getResult());
        }
     }
     Message newMsg = new Message(thisNode, new TaskSubmitMessage(proID,
midResults, 0, null));

     long latency = ((Transport)node.getProtocol(tid)).getLatency(node, targetNode);

     EDSimulator.add(latency,newMsg,targetNode,pid);
        System.out.println("MSG ENQUEUE: Task Results Submission Message
sent by volunteer "
                + node.getID()+" to volunteer "+targetNode.getID());
        if(this.endTime <= 0L)
            this.endTime = System.currentTimeMillis();
    }
  }
    }
```

```
	public void registerTaskNotify(Message msg,int pid){
		RegisterTaskReq data = (RegisterTaskReq)msg.getData();
		int proId = Long.valueOf(data.getProId()).intValue();
		Iterator<Long> unRT = data.getUnRT();
		if(unRT==null||!unRT.hasNext())return;
		ProjectManager pm = this.containsPro(pms, proId);
		if(pm == null)return;
		List<Task> rts = new ArrayList<Task>();
		LinkedList<Task> disTasks = pm.getDispatchedTasks();
		if(disTasks==null||disTasks.size()<1)return;
		TaskComparator tc = new TaskComparator();
		Collections.sort(disTasks, tc);
		int i = 0;
		while(unRT.hasNext()){
			long taskId = unRT.next();
			for(;i<disTasks.size();i++){
				if(taskId == disTasks.get(i).getId()){
					rts.add(disTasks.get(i));
					break;
				}
			}
		}
		Node node = getNode(thisNode);
		Node target = getNode(msg.getSource());
		msg = new Message(thisNode, new RegisterTaskRes(proId, rts));
	long latency = ((Transport)node.getProtocol(tid)).getLatency(node, target);
	EDSimulator.add(latency,msg,target,pid);
		System.out.println("MSG ENQUEUE: Register Task Response Message
sent by volunteer "
				+ node.getID()+" to volunteer "+target.getID());
	}

	/**
   * dealing with the Register Task Request message got from the master for a project.
   *
   *
   * @param msg: Register Task Request message
   * @param pid: the ID of this protocol
   *
   */
	public void registerTasks(Message msg, int pid){
```

```
                RegisterTaskRes data = (RegisterTaskRes)msg.getData();
                int proId = data.getProId();
                List<Task> rts = data.getRT();
                if(rts==null||rts.size()<1){return;}
                ProjectManager pm = this.containsPro(pms, proId);
                if(pm==null)return;
                LinkedList<Task> disTasks = pm.getDispatchedTasks();
                HashMap<Long, String> midResults = new HashMap<Long, String>();
                if(disTasks==null||disTasks.size()<1)return;
                for(Task task:rts){
                        if(task==null)continue;
                        disTasks.add(task);
                        midResults.put(task.getId(), task.getResult());
                }
                TaskComparator tc = new TaskComparator();
                Collections.sort(disTasks, tc);
  //reduce
  msg = new Message(thisNode, new TaskSubmitMessage(proId, midResults, 0, null));
  this.reduceResults(msg, pid);
     }

     /**
* dealing with the Reassign Request message got from the master for a project.
*
*
* @param msg: Reassign Request Request message
* @param pid: the ID of this protocol
*
*/
     public void reassign(Message msg,int pid){
                RessignReq data = (RessignReq)msg.getData();
                int proId = Long.valueOf(data.getProId()).intValue();
                long executer = data.getExecuter();
                ProjectManager pm = this.containsPro(pms, proId);
                if(pm==null||getNode(executer)!=null)return;
                LinkedList<Task> missedTasks = new LinkedList<Task>();
                LinkedList<Task> disTasks = pm.getDispatchedTasks();
                if(disTasks==null||disTasks.size()<1)return;
                for(Task task:disTasks){
                        if(task.getExecuter().getID() == executer){
                                missedTasks.add(task);
                        }
```

```
			}
			countWords(missedTasks, pm.getDistrLevel());
			for(Task task:missedTasks){
				task.setCompleteLevel(pm.getDistrLevel());
				task.setExecuter((Volunteer)getNode(thisNode));
				task.setStatus(Task.PAR_COMPLETE);
				if(pm.getDistrLevel()==0){
					this.reduceWords(task.getResult(), this.finalResult);
				this.reduceMonitor += task.getId()+",";
				}
			}
			HashMap<Long, String> results = new HashMap<Long, String>();
    msg = new Message(thisNode, new TaskSubmitMessage(0, results, 0, null));
	this.reduceResults(msg,pid);
	}

	/**
   * periodically triggered with the stabilization procedure of this volunteer's Chord
rings(both Upper and Lower).
   * this volunteers will distribute the contact information of volunteers in its Upper
Chord DHT
   * to the volunteers in its Lower Chord DHT
   *
   *
   * @param pid: the ID of this protocol
   * @param proId: the project ID
   *
   */
	public void distributeUpperToLower(int pid, int proId){

	if(this.upperDHT.predecessor==null||this.upperDHT.successorList[0].getID() ==
thisNode

	||(BigInteger.ZERO.equals(this.upperDHT.chordId)&&this.upperDHT.predeces
sor.getID()==thisNode))return;

	if(this.lowerDHT.predecessor==null||this.lowerDHT.successorList[0].getID() ==
thisNode

	||(BigInteger.ZERO.equals(this.lowerDHT.chordId)&&this.lowerDHT.predecess
or.getID()==thisNode))return;
```

```
		if(upperDHT.fingerTable==null||upperDHT.fingerTable.length<=0)return;
			if(lowerDHT.fingerTable==null||lowerDHT.fingerTable.length<=0)return;
			distributePolicyOne(pid, proId);
//			distributePolicyTwo(pid, proId);
		}

		/**
	 * Distribution policy one
	 *
	 *
	 * @param pid: the ID of this protocol
	 * @param proId: the project ID
	 *
	 */
		public void distributePolicyOne(int pid, int proId){
			List<Long> dist = new ArrayList<Long>();
			String upperFingerIds = ",";
			for(Node node:upperDHT.fingerTable){

		if(node==null||node.isUp()==false||upperFingerIds.contains(","+node.getID()+","
))continue;
				upperFingerIds += node.getID()+",";
				dist.add(node.getID());
			}
			int size = 0;
			Node sourceNode = getNode(thisNode);
			Node targetNode = null;
			String fingersId = ",";
			HashMap<Node,LinkedList<Long>> l = new
HashMap<Node,LinkedList<Long>>();
			for(Node node:lowerDHT.fingerTable){
				if(node==null||node.isUp()==false)continue;
				if(!fingersId.contains(","+node.getID()+",")){
					fingersId += node.getID()+",";
					l.put(node, new LinkedList<Long>());
					size++;
				}
			}
			if(lowerDHT.successorList!=null&&lowerDHT.successorList.length>0){
				for(Node node:lowerDHT.successorList){
					if(node==null||node.isUp()==false)continue;
```

```
                    if(!fingersId.contains(","+node.getID()+",")){
                        fingersId += node.getID()+",";
                        l.put(node, new LinkedList<Long>());
                        size++;
                    }
                }
            }
            if(fingersId.length()>1){
                fingersId = fingersId.substring(1, fingersId.length()-1);
            }
            String fingers[] = fingersId.length()==1?new String[]{"-
1"}:(fingersId.length()>1?fingersId.split(","):null);
            int length =
upperDHT.successorList==null||upperDHT.successorList.length<0?0:upperDHT.succes
sorList.length;
            for(int s = 0; s<length;s++){

    if(upperDHT.successorList[s]!=null&&upperDHT.successorList[s].isUp()==true&
&!upperFingerIds.contains(","+upperDHT.successorList[s].getID()+",")){
                    dist.add(upperDHT.successorList[s].getID());
                    upperFingerIds += upperDHT.successorList[s].getID()+",";
                }
            }
            for(int i = 0,j=0; i < dist.size(); i++){
                if(j>=fingers.length){
                    j = 0;
                }
                targetNode = getNode(Long.valueOf(fingers[j]));
                if(targetNode==null){continue;}
                if(l.containsKey(targetNode)){
                    l.get(targetNode).add(dist.get(i));
                    j++;
                }
            }
            Iterator<Node> it = l.keySet().iterator();
            if(it==null)return;
            while(it.hasNext()){
                Node dest = it.next();

    if(dest!=null&&dest.isUp()==true&&dest.getID()!=thisNode&&l.get(dest)!=null&
&l.get(dest).size()>0){
```

```
				Message msg = new Message(thisNode, new
ParentSiblingNotify(l.get(dest), proId));
				long latency =
((Transport)sourceNode.getProtocol(tid)).getLatency(sourceNode, dest);
		EDSimulator.add(latency,msg,dest,pid);
			System.out.println("MSG ENQUEUE: Parent Sibling
("+l.get(dest).toString()+") Notify Message sent by volunteer "
					+ thisNode +" to volunteer "+dest.getID());
			}
		}
	}

	/**
   * Distribution policy two
   *
   *
   * @param pid: the ID of this protocol
   * @param proId: the project ID
   *
   */
	public void distributePolicyTwo(int pid, int proId){
		List<Long> dist = new ArrayList<Long>();
		String upperFingerIds = ",";
		for(Node node:upperDHT.fingerTable){

	if(node==null||node.isUp()==false||upperFingerIds.contains(","+node.getID()+","
))continue;
			upperFingerIds += node.getID()+",";
			dist.add(node.getID());
		}
		int size = 0;
		Node sourceNode = getNode(thisNode);
		Node targetNode = null;
		String fingersId = ",";
		HashMap<Node,LinkedList<Long>> l = new
HashMap<Node,LinkedList<Long>>();
		for(Node node:lowerDHT.fingerTable){
			if(node==null||node.isUp()==false)continue;
			if(!fingersId.contains(","+node.getID()+",")){
				fingersId += node.getID()+",";
				l.put(node, new LinkedList<Long>());
				size++;
```

```
				}
			}
			if(lowerDHT.successorList!=null&&lowerDHT.successorList.length>0){
				for(Node node:lowerDHT.successorList){
					if(node==null||node.isUp()==false)continue;
					if(!fingersId.contains(","+node.getID()+",")){
						fingersId += node.getID()+",";
						l.put(node, new LinkedList<Long>());
						size++;
					}
				}
			}
			if(fingersId.length()>1){
				fingersId = fingersId.substring(1, fingersId.length()-1);
			}
			String fingers[] = fingersId.length()==1?new String[]{"-
1"}:(fingersId.length()>1?fingersId.split(","):null);
			int length =
upperDHT.successorList==null||upperDHT.successorList.length<0?0:upperDHT.succes
sorList.length;
			for(int s = 0; s<length;s++){

		if(upperDHT.successorList[s]!=null&&upperDHT.successorList[s].isUp()==true&
&!upperFingerIds.contains(","+upperDHT.successorList[s].getID()+",")){
					dist.add(upperDHT.successorList[s].getID());
					upperFingerIds += upperDHT.successorList[s].getID()+",";
				}
			}
			for(int i = 0; i < fingers.length; i++){
				targetNode = getNode(Long.valueOf(fingers[i]));
				if(targetNode==null){continue;}
				if(l.containsKey(targetNode)){
					for(Long id:dist){
						l.get(targetNode).add(id);
					}
				}
			}
			Iterator<Node> it = l.keySet().iterator();
			if(it==null)return;
			while(it.hasNext()){
				Node dest = it.next();
```

```
	if(dest!=null&&dest.isUp()==true&&dest.getID()!=thisNode&&l.get(dest)!=null&
&l.get(dest).size()>0){
				Message msg = new Message(thisNode, new
ParentSiblingNotify(l.get(dest), proId));
				long latency =
((Transport)sourceNode.getProtocol(tid)).getLatency(sourceNode, dest);
		EDSimulator.add(latency,msg,dest,pid);
			System.out.println("MSG ENQUEUE: Parent Sibling
("+l.get(dest).toString()+") Notify Message sent by volunteer "
					+ thisNode +" to volunteer "+dest.getID());
			}
		}
	}

	/**
   * dealing with the Parent Sibling Notify message got from the master for a project.
   * the current volunteer as a worker will record the contact information of the master's
siblings
   * for fault-tolerance in case that this master fails silently.
   *
   *
   * @param msg: Parent Sibling Notify message
   *
   */
	public void setParentSibling(Message msg){
		ParentSiblingNotify data = (ParentSiblingNotify)msg.getData();
		int proId = data.getProId();
		LinkedList<Long> parentSiblingIds = data.getParentSiblingID();
		ProjectManager pm = this.containsPro(pms, proId);
		if(pm!=null&&pm.getMyDispatcher()!=null&&msg.getSource() ==
pm.getMyDispatcher().getID()){
			this.parentSiblingID = parentSiblingIds;
		}else{
			System.out.println("the parent sibling notify msg sent by node " +
msg.getSource()
					+ " arrived at node " + thisNode + ", who is not a
child of the sender!!!");
		}
	}

	public void reduceWords(String file, HashMap<String, Integer> result){
```

```
		File inFile = new File(file);
		RandomAccessFile inn;
		try {
			inn = new RandomAccessFile(inFile, "r");
			String line = "";
			while(inn!=null&&(line = inn.readLine())!= null){
				line.trim();
				String[] pair = line.split(":");
				counting(pair[0], result, Integer.valueOf(pair[1]));
			}
			inn.close();
		}catch (FileNotFoundException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}

	public void countWords(LinkedList<Task> tasks, int dispatchLevel){
		if(tasks.size()<=0)return;
		for(int i = 0; i < tasks.size(); i++){
			tasks.get(i).setStatus(Task.PAR_PROCESS);
			String file = tasks.get(i).getFile();
			String resultFile = "results\\result_"+file;
			File inFile = new File(file);
			RandomAccessFile inn;
			try {
				inn = new RandomAccessFile(inFile, "r");
				HashMap<String, Integer> words = new HashMap<String,
Integer>();
				String line = "";
				while(inn!=null&&(line = inn.readLine())!= null){
					line.trim();
					counting(line, words, 1);
				}
				inn.close();
				File result = new File(resultFile);
				RandomAccessFile outt = new RandomAccessFile(result,
"rws");
				String lineSeparator = (String)
java.security.AccessController.doPrivileged(
```

```
                                                    new
sun.security.action.GetPropertyAction("line.separator"));
                                        for(int j = 97;j<=122;j++){
                                                char c = (char)j;
                                                String key = String.valueOf(c);
                                                if(words.containsKey(key)){
                                                        outt.writeBytes(key+":"+words.get(key));
                                                        outt.writeBytes(lineSeparator);
                                                }
                                        }
                                        outt.close();
                                        tasks.get(i).setResult(resultFile);
                                        tasks.get(i).setStatus(Task.PAR_SUBMIT);
                                }catch (FileNotFoundException e) {
                                        e.printStackTrace();
                                } catch (IOException e) {
                                        e.printStackTrace();
                                }
                        }
                }

        public void counting(String line, HashMap<String, Integer> words, int
increment){
                byte[] letters = line.getBytes();
                if(letters.length <= 0)return;
                for(int i = 0; i < letters.length; i++){
                        int c = (int)letters[i];
                        switch(c){
                        case 65:
                        case 97:
                                countEachLetter("a", words, increment);
                                break;
                        case 66:
                        case 98:
                                countEachLetter("b", words, increment);
                                break;
                        case 67:
                        case 99:
                                countEachLetter("c", words, increment);
                                break;
                        case 68:
                        case 100:
```

```
        countEachLetter("d", words, increment);
        break;
case 69:
case 101:
        countEachLetter("e", words, increment);
        break;
case 70:
case 102:
        countEachLetter("f", words, increment);
        break;
case 71:
case 103:
        countEachLetter("g", words, increment);
        break;
case 72:
case 104:
        countEachLetter("h", words, increment);
        break;
case 73:
case 105:
        countEachLetter("i", words, increment);
        break;
case 74:
case 106:
        countEachLetter("j", words, increment);
        break;
case 75:
case 107:
        countEachLetter("k", words, increment);
        break;
case 76:
case 108:
        countEachLetter("l", words, increment);
        break;
case 77:
case 109:
        countEachLetter("m", words, increment);
        break;
case 78:
case 110:
        countEachLetter("n", words, increment);
        break;
```

```
case 79:
case 111:
        countEachLetter("o", words, increment);
        break;
case 80:
case 112:
        countEachLetter("p", words, increment);
        break;
case 81:
case 113:
        countEachLetter("q", words, increment);
        break;
case 82:
case 114:
        countEachLetter("r", words, increment);
        break;
case 83:
case 115:
        countEachLetter("s", words, increment);
        break;
case 84:
case 116:
        countEachLetter("t", words, increment);
        break;
case 85:
case 117:
        countEachLetter("u", words, increment);
        break;
case 86:
case 118:
        countEachLetter("v", words, increment);
        break;
case 87:
case 119:
        countEachLetter("w", words, increment);
        break;
case 88:
case 120:
        countEachLetter("x", words, increment);
        break;
case 89:
case 121:
```

```
                        countEachLetter("y", words, increment);
                        break;
                    case 90:
                    case 122:
                        countEachLetter("z", words, increment);
                        break;
                    default:
                        System.out.println("the letter is not we are counting for!");
                    }
            }
    }

    public void countEachLetter(String letter, HashMap<String,Integer> words, int
increment){
            if(words.containsKey(letter)){
                    words.put(letter, words.get(letter)+increment);
            }else{
                    words.put(letter, increment);
            }
    }

  public ProjectManager containsPro(LinkedList<ProjectManager> pms, int proId){
        ProjectManager result = null;
        if(pms == null) return null;
        for(ProjectManager pm:pms){
                if(pm != null&&(int)pm.getProId()==proId){
                        return result = pm;
                }
        }
        return result;
  }

  public ProjectManager containsTasks(LinkedList<ProjectManager> pms, int proId,
List<Integer> taskIds){
        ProjectManager pm = containsPro(pms, proId);
        LinkedList<Task> disTasks = pm==null?new
LinkedList<Task>():pm.getDispatchedTasks();
        LinkedList<Task> execTasks = pm==null?new
LinkedList<Task>():pm.getExecuteTasks();
        String haveTaskIds = "";
        for(Task task:disTasks){
                if(task!=null)haveTaskIds += ","+task.getId()+",";
```

```
        }
        for(Task task:execTasks){
                if(task!=null)haveTaskIds += ","+task.getId()+",";
        }
        for(int id:taskIds){
                return haveTaskIds.contains(","+id+",")?pm:null;
        }
        return null;
    }

    public boolean isBusy(LinkedList<Task> tasks){
        if(tasks ==null||tasks.size()<1)return false;
        for(Task task:tasks){
                if("complete".equalsIgnoreCase(task.getStatus()))return false;
        }
        return true;
    }

    public Node getNode(long id){
                for(int i = 0; i<Network.size();i++){
                        if(Network.get(i)==null)continue;
                        if(Network.get(i).getID() == id)return Network.get(i);
                }
                return null;
        }
}
```

ChordProtocol.class

```
/*
 * @author: Hao Ning
 * 2010 Master Thesis:
 * Robust Overlay Networks for Volunteer Computing
 */

package example.chord;

import peersim.cdsim.CDProtocol;
import peersim.config.Configuration;
import peersim.config.FastConfig;
import peersim.core.Linkable;
```

```
import peersim.core.Network;
import peersim.core.Node;
import peersim.edsim.EDProtocol;
import peersim.edsim.EDSimulator;
import peersim.transport.Transport;
import java.math.*;
import java.util.HashMap;
import java.util.LinkedList;

import example.vc.messages.Message;
import example.vc.ProjectManager;
import example.vc.Task;
import example.vc.VCMatching;
import example.vc.messages.RessignReq;


public class ChordProtocol implements CDProtocol, EDProtocol {

	// ------------------------------------------------------------------------
   // Parameters
   // ------------------------------------------------------------------------
	/**
	 *	The Transport protocol used by the the protocol.
	 */
	private static final String PAR_TRANSPORT = "transport";

	/**
	 * the top layer protocol(model) referred in this protocol
	 */
	private static final String PAR_VC = "vcmatching";

	// ------------------------------------------------------------------------
   // Fields
   // ------------------------------------------------------------------------
	//the Parameters class contains the IDs of two above protocols referred in this
protocol
	private Parameters p;

	//the current node
	private Node thisNode;

	private int[] lookupMessage;
```

```
public int index = 0;

public Node predecessor;

public Node[] fingerTable;

public Node[] successorList;

public BigInteger chordId;

public int m;

public int succLSize;

public String prefix;

private int next = 0;

private int reunionLookups;
private int reunionRangeCollect;

private int currentNode = 0;

public int varSuccList = 0;

public int stabilizations = 0;

public int fails = 0;
BigInteger firstChordId;
BigInteger lastChordId;

public BigInteger getFirstChordId() {
        return firstChordId;
}

public void setFirstChordId(BigInteger firstChordId) {
        this.firstChordId = firstChordId;
}

public BigInteger getLastChordId() {
        return lastChordId;
```

```
}

public void setLastChordId(BigInteger lastChordId) {
        this.lastChordId = lastChordId;
}

public Node getThisNode() {
        return thisNode;
}

public void setThisNode(Node thisNode) {
        this.thisNode = thisNode;
}


public int getReunionLookups() {
        return reunionLookups;
}

public void setReunionLookups(int reunionLookups) {
        this.reunionLookups = reunionLookups;
}

public int getReunionRangeCollect() {
        return reunionRangeCollect;
}

public void setReunionRangeCollect(int reunionRangeCollect) {
        this.reunionRangeCollect = reunionRangeCollect;
}

/**
 * @param next: the next to set
 */
public void setNext(int next) {
        this.next = next;
}

public int count = 0;

/**
 * constructor
```

```
	 */
	public ChordProtocol(String prefix) {
		this.prefix = prefix;
		lookupMessage = new int[1];
		lookupMessage[0] = 0;
		p = new Parameters();
		p.tid = Configuration.getPid(prefix + "." + PAR_TRANSPORT);
		p.vcid = Configuration.getPid(prefix + "." + PAR_VC);
      this.firstChordId = BigInteger.ZERO;
      this.lastChordId = BigInteger.ZERO;
      this.reunionLookups = 0;
      this.reunionRangeCollect = 0;
	}



	@Override
	public void nextCycle(Node node, int protocolID) {
		count++;
		ChordProtocol cp = (ChordProtocol)node.getProtocol(protocolID);
		cp.p.pid = protocolID;
		//if this chord ring only has one node, need not to update
		if(node.getID() ==
cp.successorList[0].getID()&&cp.predecessor==null&&cp.fingerTable[0]==null){
			return;
		}
		cp.stabilizations++;
		cp.stabilize(node);
		cp.fixFingers();
		cp.checkPredecessor();
		cp.updateChordIdRange(node);
		VCMatching vc = (VCMatching) node.getProtocol(this.p.vcid);
		vc.distributeUpperToLower(this.p.vcid, 0);
	}

	/**
	 * message switcher
	 * dealing with Lookups, joins and reunions
	 *
	 * @param node: current node
	 * @param pid: this protocol ID
	 * @param event: the incoming message
```

```
     *
     */
    public void processEvent(Node node, int pid, Object event) {
        p.pid = pid;
        currentNode = node.getIndex();
        if (event.getClass() == LookUpMessage.class) {
            LookUpMessage message = (LookUpMessage) event;
            message.increaseHopCounter();
            BigInteger target = message.getTarget();
            Transport t = (Transport) node.getProtocol(p.tid);
            Node n = message.getSender();
            ChordProtocol cp = (ChordProtocol) node.getProtocol(pid);
            if (target.intValue() <= cp.chordId.intValue()

    ||(cp.predecessor!=null&&cp.predecessor.isUp()==true&&target.intValue() !=
cp.chordId.intValue()
                                    &&this.idInab(target,
((ChordProtocol)cp.predecessor.getProtocol(p.pid)).chordId, cp.chordId))) {
                //continue to route this lookup message to the predecessor
of the current node

    if(cp.predecessor!=null&&cp.predecessor.isUp()==true&&target.intValue() <
cp.chordId.intValue()
                                &&!this.idInab(target,
((ChordProtocol)cp.predecessor.getProtocol(p.pid)).chordId, cp.chordId)){
                    this.reunionLookups++;
                    t.send(message.getSender(), cp.predecessor,
message, pid);
                    return;
                }
                int mode = message.getMode();
                if(mode == 1){
                    Node contactor = message.getContactor();
                    int firstTaskId = message.getFirstTaskId();
                    int lastTaskId = message.getLastTaskId();
                    int proId = message.getProId();
                    VCMatching vcm =
(VCMatching)contactor.getProtocol(this.p.vcid);
                    VCMatching myVcm =
(VCMatching)thisNode.getProtocol(this.p.vcid);
                    if(vcm == null)return;
```

```
//****************************************************************
                    //If the contactor's dispatch level of this project is
not next to the replacer(thisNode)'s dispatch level:
                    //try thisnode's predecessor to be the replacer-
change the position of its predecessor in its lower chord ring
                    ProjectManager contactorPM =
vcm.containsPro(vcm.getPms(), proId);
                    ProjectManager myPM =
vcm.containsPro(myVcm.getPms(), proId);
                    if(contactorPM == null||myPM == null)return;

    if(contactorPM.getDistrLevel()>myPM.getDistrLevel()+1){

    if(predecessor!=null&&predecessor.isUp()==true&&predecessor.getID()!=thisNo
de.getID()){
                              BigInteger predecessorChordId =
((ChordProtocol)predecessor.getProtocol(p.pid)).chordId;

    if(this.idInab(BigInteger.valueOf(lastTaskId), predecessorChordId, this.chordId)){

    ((ChordProtocol)predecessor.getProtocol(p.pid)).chordId =
BigInteger.valueOf(lastTaskId);

    ((ChordProtocol)predecessor.getProtocol(p.pid)).lastChordId =
BigInteger.valueOf(lastTaskId);

    ((ChordProtocol)predecessor.getProtocol(p.pid)).stabilizations++;

    ((ChordProtocol)predecessor.getProtocol(p.pid)).stabilize(predecessor);

    ((ChordProtocol)predecessor.getProtocol(p.pid)).fixFingers();
                                    this.reunionLookups++;
                                    t.send(message.getSender(),
predecessor, message, p.pid);
                              }
                         }
                         return;
                    }

                    long failedMasterId = message.getFailedMasterId();
```

```
			if(myVcm.getReunionRecord()==null)myVcm.setReunionRecord(new
HashMap<Long, String>());

			if(!myVcm.getReunionRecord().containsKey(failedMasterId)){

			myVcm.getReunionRecord().put(failedMasterId, ",");
					}else{
						return;
					}

			//**************************************************************
					this.reunionLookups++;
					t.send(thisNode, contactor, new
LookUpMessage(thisNode, BigInteger.valueOf(firstTaskId), null, 2, proId, 0, 0,
failedMasterId), vcm.getUpChordID());
					this.reunionLookups++;
					t.send(thisNode, contactor, new
LookUpMessage(thisNode, BigInteger.valueOf(lastTaskId), null, 2, proId, 0, 0,
failedMasterId), vcm.getUpChordID());

			if(myVcm.getReplceRecord()==null)myVcm.setReplceRecord(new
HashMap<BigInteger, Node>());

			myVcm.getReplceRecord().put(BigInteger.valueOf(firstTaskId), null);

			myVcm.getReplceRecord().put(BigInteger.valueOf(lastTaskId), null);
					return;
				}else if(mode == 2){
					int proId = message.getProId();
					Node newMaster = message.getSender();
					VCMatching vcm =
(VCMatching)thisNode.getProtocol(this.p.vcid);
					ProjectManager pm =
vcm.containsPro(vcm.getPms(), proId);
					if(pm == null)return;
					Node dispatcher = pm.getMyDispatcher();

			if(dispatcher.getID()!=message.getFailedMasterId())return;
					int pos = 0;
					if(target.intValue() == this.firstChordId.intValue()){
						pos = 0;
```

```
							}else if(target.intValue() ==
this.lastChordId.intValue()){
								pos = 1;
							}else{
								pos = -1;
							}
							this.reunionRangeCollect++;
							t.send(thisNode, newMaster, new
ReunionRes(proId, dispatcher.getID(), pos, this.successorList[0], thisNode, this.chordId),
this.p.pid);
							return;
						}else if(mode == 3){
							int proId = message.getProId();
							Node targetNode = message.getSender();
							VCMatching vcm =
(VCMatching)thisNode.getProtocol(this.p.vcid);
							ProjectManager pm =
vcm.containsPro(vcm.getPms(), proId);
							if(pm == null)return;
							LinkedList<Task> exeTasks = pm.getExecuteTasks();
							if(exeTasks==null||exeTasks.size()<1)return;

	if(exeTasks.get(0).getStatus().equals(Task.PAR_COMPLETE)&&pm.getDistrLevel
()-1 == exeTasks.get(0).getCompleteLevel()&&targetNode.isUp()==true){
								vcm.setEndTime(0L);
								Message newMsg = new
Message(thisNode.getID(), new RessignReq(proId, message.getFailedMasterId()));
								VCMatching targetVC =
(VCMatching)targetNode.getProtocol(this.p.vcid);
								targetVC.reassign(newMsg, this.p.vcid);
							}else{
								return;
							}
						}
						t.send(node, n, new FinalMessage(node,
message.getHopCounter()), pid);
					}
					if (target.intValue() > cp.chordId.intValue()) {
						// funzione lookup sulla fingertabable
						Node dest = find_successor(target);
						if (dest.isUp() == false) {
							do {
```

```
                                        varSuccList = 0;
                                        stabilize(node);
                                        stabilizations++;
                                        fixFingers();
                                        dest = find_successor(target);
                                } while (dest.isUp() == false);
                        }
                        if (dest.getID() == thisNode.getID()) {
                                fails++;
                        } else {
                                t.send(message.getSender(), dest, message, pid);
                        }
                }
        }
        if (event.getClass() == ReunionReq.class) {
                ReunionReq message = (ReunionReq)event;
                int proId = message.getProId();
                Node newMaster = message.getSender();
                VCMatching vcm =
(VCMatching)thisNode.getProtocol(this.p.vcid);
                ProjectManager pm = vcm.containsPro(vcm.getPms(), proId);
                if(pm == null)return;
                Node dispatcher = pm.getMyDispatcher();
                int pos = -1;
                Transport t = (Transport) node.getProtocol(p.tid);
                this.reunionRangeCollect++;
                t.send(thisNode, newMaster, new ReunionRes(proId,
dispatcher.getID(), pos, this.successorList[0], thisNode, this.chordId), this.p.pid);
        }
        if (event.getClass() == ReunionRes.class) {
                ReunionRes msg = (ReunionRes)event;
                int proId = msg.getProId();
                BigInteger senderChordId = msg.getChordId();
                int pos = msg.getPos();
                long failedMasterId = msg.getFailedMasterId();
                Node nextWorker = msg.getSuccessor();
                Node sender = msg.getSender();
                ChordProtocol joiner =
(ChordProtocol)sender.getProtocol(this.p.pid);
                VCMatching vcm =
(VCMatching)thisNode.getProtocol(this.p.vcid);
```

```
	if(pos==0&&vcm.getReplceRecord()!=null&&vcm.getReplceRecord().containsKey
(joiner.firstChordId)){
				vcm.getReplceRecord().put(joiner.firstChordId, sender);
			}else
if(pos==1&&vcm.getReplceRecord()!=null&&vcm.getReplceRecord().containsKey(joiner.
lastChordId)){
				vcm.getReplceRecord().put(joiner.lastChordId, sender);
			}

			Node firstNode =
vcm.getReplceRecord()!=null?vcm.getReplceRecord().get(joiner.firstChordId):null;
			Node lastNode =
vcm.getReplceRecord()!=null?vcm.getReplceRecord().get(joiner.lastChordId):null;
			//reunion recorder contains this failed master

	if(vcm.getReunionRecord()!=null&&vcm.getReunionRecord().containsKey(failed
MasterId)){
				if(pos==0||pos==1){

	if(!vcm.getReunionRecord().get(failedMasterId).contains(","+sender.getID()+",")){

	vcm.getReunionRecord().put(failedMasterId,
vcm.getReunionRecord().get(failedMasterId)+sender.getID()+",");
					}else if(sender.getID()==nextWorker.getID()){
						String[] workers =
vcm.getReunionRecord().get(failedMasterId).split(",");
						for(int i = 1; i<workers.length;i++){
							Node targetNode =
getNode(Integer.parseInt(workers[i]));
							if(targetNode == null) continue;
							VCMatching vcmTarget =
(VCMatching)targetNode.getProtocol(this.p.vcid);
							vcm.updatePMSReq(proId,
targetNode, this.p.vcid);
						}
						vcm.setReplceRecord(new
HashMap<BigInteger, Node>());
						return;
					}
				}
```

```
					//if head and tail workers are both recollected to this new
master

	if(lastNode!=null&&firstNode!=null&&vcm.getReunionRecord().get(failedMaster
Id).contains(","+firstNode.getID()+",")

	&&vcm.getReunionRecord().get(failedMasterId).contains(","+lastNode.getID()+",
")){
						//begin the procedure as range query by lookingup
for successor of each worker

	if(sender!=null&&nextWorker!=null&&sender.getID()!=nextWorker.getID()

	&&joiner.firstChordId.compareTo(senderChordId)<1&&joiner.lastChordId.comp
areTo(senderChordId)>-1){

	if(!vcm.getReunionRecord().get(failedMasterId).contains(","+nextWorker.getID()
+",")

	||(vcm.getReunionRecord().get(failedMasterId).contains(","+nextWorker.getID()
+",")&&pos==1)){

	if(!vcm.getReunionRecord().get(failedMasterId).contains(","+nextWorker.getID()
+","))

	vcm.getReunionRecord().put(failedMasterId,
vcm.getReunionRecord().get(failedMasterId)+nextWorker.getID()+",");
								Transport t = (Transport)
node.getProtocol(p.tid);
								this.reunionRangeCollect++;
								t.send(thisNode, nextWorker, new
ReunionReq(proId, thisNode), this.p.pid);
							}else{
								String[] workers =
vcm.getReunionRecord().get(failedMasterId).split(",");
								int linkableID =
FastConfig.getLinkable(this.p.vcid);
						Linkable linkable = (Linkable)
node.getProtocol(linkableID);
								for(int i = 1; i<workers.length;i++){
									Node targetNode =
getNode(Integer.parseInt(workers[i]));
```

```
                                                                                if(targetNode == null)
continue;
                                                                                //add those workers as my
new neighbors, and also for reset my lower chord DHT successor list size!

        linkable.addNeighbor(targetNode);
                                                                                vcm.updatePMSReq(proId,
targetNode, this.p.vcid);
                                                                        }
                                                                        vcm.setReplceRecord(new
HashMap<BigInteger, Node>());
                                                                }
                                                        }
                                                }
                                        }else{
                                                return;
                                        }
                                }
                                if (event.getClass() == FinalMessage.class) {
                                        FinalMessage message = (FinalMessage) event;
                                        lookupMessage = new int[index + 1];
                                        lookupMessage[index] = message.getHopCounter();
                                        index++;
                                }
                                if(event.getClass() == JoinRequest.class){
                                        JoinRequest msg = (JoinRequest)event;
                                        BigInteger joinedId = msg.getJoinedId();
                                        Node successor = this.find_successor(joinedId);
                                        int joinedVarSuccList = this.varSuccList;
                                        int linkableID = FastConfig.getLinkable(this.p.vcid);
                        Linkable linkable = (Linkable) node.getProtocol(linkableID);
                                        int degree = linkable.degree();
                                        //check whether all my neighbors are alive
                                        int num = 0;
                                        for (int a = 0;a<degree;a++){
                                                Node neighbor = linkable.getNeighbor(a);
                                                if(neighbor!=null&&neighbor.isUp()==true){
                                                        num++;
                                                }
                                        }
                                        degree = num;
```

```
			int lowerSuccList =
degree>2?(degree%2==0?degree/2:degree/2+1):1;
			int joinedSuccLSize = lowerSuccList;
			Node sender = msg.getSender();
			int proId = msg.getProId();
			VCMatching vcm =
(VCMatching)thisNode.getProtocol(this.p.vcid);
			ProjectManager pm = vcm.containsPro(vcm.getPms(), proId);
			long lowerChordId = -1L;
			if(pm !=
null&&pm.getExecuteTasks()!=null&&pm.getExecuteTasks().size()>0&&pm.getExecuteT
asks().size()==vcm.getTaskRange().size()){
				lowerChordId =
pm.getExecuteTasks().get(pm.getExecuteTasks().size()-1).getId();
				this.firstChordId =
BigInteger.valueOf(pm.getExecuteTasks().get(0).getId());
			}
			if(lowerChordId>-1&&this.chordId.intValue() != lowerChordId){
				this.chordId = BigInteger.valueOf(lowerChordId);
				this.lastChordId = chordId;
			}
			long latency =
((Transport)node.getProtocol(p.tid)).getLatency(node,sender);
			EDSimulator.add(latency,new
JoinResponse(successor,joinedVarSuccList, joinedSuccLSize),sender,p.pid);
		}
		if(event.getClass() == JoinResponse.class){
			JoinResponse msg = (JoinResponse)event;
			Node successor = msg.getSuccessor();
			this.fails = 0;
			this.stabilizations = 0;
			this.succLSize = msg.getSuccLSize();
			this.varSuccList = msg.getVarSuccList();
			this.successorList = new Node[this.succLSize];
			for(int a= 0; a<this.succLSize; a++){
				this.successorList[a] = successor;
			}
			this.fingerTable = new Node[this.m];
			//long succId = 0;
			this.stabilizations++;
			this.stabilize(thisNode);
			this.fixFingers();
```

```
		}
	}

	public Object clone() {
		ChordProtocol cp = new ChordProtocol(prefix);
		String val = BigInteger.ZERO.toString();
		cp.chordId = new BigInteger(val);
		cp.fingerTable = new Node[m];
		cp.successorList = new Node[succLSize];
		cp.currentNode = 0;
		cp.firstChordId = BigInteger.ZERO;
		cp.lastChordId = BigInteger.ZERO;
		cp.reunionLookups = 0;
		cp.reunionRangeCollect = 0;
		return cp;
	}

	public int[] getLookupMessage() {
		return lookupMessage;
	}

	//join
	public void join(Node myNode, int pid, BigInteger myChordId, Node
introducer,BigInteger firstChordId, BigInteger lastChordId,int proId) {
		this.predecessor = null;
		this.m = ((ChordProtocol) introducer.getProtocol(pid)).m;
		this.chordId = myChordId;
		this.lastChordId = lastChordId;
		this.firstChordId = firstChordId;
		//sending message to introducer
		long latency =
((Transport)myNode.getProtocol(p.tid)).getLatency(myNode,introducer);
		EDSimulator.add(latency,new
JoinRequest(myNode,this.chordId,proId),introducer,pid);
	}

	public void updateChordIdRange(Node myNode){
		Node successor = successorList[0];

	if(successor!=null&&successor.getID()!=myNode.getID()&&successor.isUp()==tru
e){
```

```
				ChordProtocol successorCP = (ChordProtocol)
successor.getProtocol(p.pid);
				if(successorCP.getFirstChordId().compareTo(this.firstChordId)==-
1){
					this.firstChordId = successorCP.getFirstChordId();
				}

		if(successorCP.getLastChordId().compareTo(this.lastChordId)==1){
					this.lastChordId = successorCP.getLastChordId();
				}
			}
		}

		public void stabilize(Node myNode) {
			try {
				Node node = ((ChordProtocol)
successorList[0].getProtocol(p.pid)).predecessor;
				if (node != null) {
					if (this.chordId == ((ChordProtocol)
node.getProtocol(p.pid)).chordId){
						updateSuccessorList();
						return;
					}
					BigInteger remoteID = ((ChordProtocol)
node.getProtocol(p.pid)).chordId;
					//if my old successor's predecessor is not me now, and its
chordId is between me and my old successor,
					//or now i am the first one node in this chord ring, and my
old successor is till myself,then I will
					//change my old predecessor to my old successor's
predecessor
					if (idInab(remoteID, chordId, ((ChordProtocol)
successorList[0]
							.getProtocol(p.pid)).chordId))
						successorList[0] = node;
				}
				if(successorList[0].getID() == thisNode.getID())return;
				((ChordProtocol) successorList[0].getProtocol(p.pid))
				.notify(myNode);
				updateSuccessorList();
			} catch (Exception e1) {
				e1.printStackTrace();
```

```
				updateSuccessor();
			}
		}

		public void notify(Node node) throws Exception {
			BigInteger nodeId = ((ChordProtocol) node.getProtocol(p.pid)).chordId;
			if ((predecessor == null)
					|| (idInab(nodeId, ((ChordProtocol) predecessor
							.getProtocol(p.pid)).chordId, this.chordId)))
{
				predecessor = node;
			}
		}

		private void updateSuccessorList() throws Exception {
			try {
				//try to figure the successor list
				while (successorList[0] == null || successorList[0].isUp() == false) {
					boolean update = false;
					for(int i = 0;i<successorList.length; i++){

		if(successorList[i]!=null&&successorList[i].isUp()==true){
								update = true;
								break;
							}
					}
					if(!update){
						successorList[0] = thisNode;
					}
					else{
						updateSuccessor();
					}
				}
				if(succLSize>2){
					Node succSucc[] = ((ChordProtocol)
successorList[0].getProtocol(p.pid)).successorList;

		if(successorList!=null&&succSucc!=null&&succSucc.length>succLSize){
						Node[] tempSuccList = new Node[succSucc.length];
						System.arraycopy(successorList, 0, tempSuccList, 0,
successorList.length);
						this.successorList = tempSuccList;
```

```
					succLSize = this.successorList.length;
				}
				if(succSucc!=null&&succSucc.length>0){
					String s = ",";
					for(int i = 0,j=1; i<succSucc.length-1; i++){
		
if(succSucc[i]!=null&&succSucc[i].isUp()==true){
		
if(!s.contains(","+succSucc[i].getID()+",")){
									successorList[j] = succSucc[i];
									s += succSucc[i].getID()+",";
									j++;
								}
							}
						}
					}
				}
		} catch (Exception e) {
			e.printStackTrace();
		}
}

private void updateSuccessor() {
		boolean searching = true;
		while (searching) {
			try {
				if(varSuccList>=succLSize){
					varSuccList = succLSize -1;
				}
				Node node = successorList[varSuccList];
				varSuccList++;
				successorList[0] = node;
				if (successorList[0] == null
						|| successorList[0].isUp() == false) {
					if (varSuccList >= succLSize) {
						searching = false;
						varSuccList = 0;
					} else
						updateSuccessor();
				}
				updateSuccessorList();
				searching = false;
```

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

        private boolean idInab(BigInteger id, BigInteger a, BigInteger b) {
                BigInteger b1 = BigInteger.ZERO;
                if(b.equals(BigInteger.ZERO)||a.equals(b)){
                        if(a.equals(b))return true;
                        b1 = BigInteger.valueOf(Double.doubleToLongBits(Math.pow(2L,
Integer.valueOf(this.m).longValue())));
                }else{
                        b1 = b;
                }

                if(b.compareTo(BigInteger.ZERO)!=0&&a.compareTo(b)==1){
                        if(id.compareTo(BigInteger.ZERO)==0){
                                id =
BigInteger.valueOf(Double.doubleToLongBits(Math.pow(2L,
Integer.valueOf(this.m).longValue())));
                        }
                        if(id.compareTo(a)==1||id.compareTo(b)==-1){
                                return true;
                        }
                }

                if ((a.compareTo(id) == -1) && (id.compareTo(b1) == -1)) {
                        return true;
                }
                return false;
        }

        public Node find_successor(BigInteger id) {
                try {
                        if (successorList[0] == null || successorList[0].isUp() == false) {
                                updateSuccessor();
                        }
                        BigInteger successorChordId = ((ChordProtocol)
successorList[0].getProtocol(p.pid)).chordId;
                        if(successorChordId.equals(BigInteger.ZERO)){
```

```
				successorChordId =
BigInteger.valueOf(Double.doubleToLongBits(Math.pow(2L,
Integer.valueOf(this.m).longValue())));
			}
			if (idInab(id,
this.chordId,successorChordId)||id.compareTo(successorChordId)==0) {
				return successorList[0];
			} else {
				Node tmp = closest_preceding_node(id);
				return
((ChordProtocol)tmp.getProtocol(p.pid)).find_successor(id);
			}
		} catch (Exception e) {
			e.printStackTrace();
		}
		return successorList[0];
	}

	private Node closest_preceding_node(BigInteger id) {
		for (int i = m; i > 0; i--) {
			try {
				if (fingerTable[i - 1] == null
						|| fingerTable[i - 1].isUp() == false) {
					continue;
				}
				BigInteger fingerId = ((ChordProtocol) (fingerTable[i - 1]
						.getProtocol(p.pid))).chordId;
				if (idInab(fingerId, this.chordId, id)) {
					return fingerTable[i - 1];
				}
				if (fingerId.compareTo(this.chordId) == -1) {
					if (idInab(id, fingerId, this.chordId)) {
						return fingerTable[i - 1];
					}
				}
				if ((id.compareTo(fingerId) == -1)
						&& (id.compareTo(this.chordId) == -1)) {
					if (i == 1)
						return successorList[0];
					if(fingerTable[i - 2]==null||(ChordProtocol)
fingerTable[i - 2].getProtocol(p.pid)==null){
						System.out.println("im here");
```

```
					}
					BigInteger lowId = ((ChordProtocol) fingerTable[i -
2]
							.getProtocol(p.pid)).chordId;
					if
(id.compareTo(lowId)==1&&id.compareTo(fingerId)==-1)
						return fingerTable[i - 2];
					else if (fingerId.compareTo(this.chordId) == -1)
						continue;
					else if (fingerId.compareTo(this.chordId) == 1)
						return fingerTable[i - 1];
				}
			} catch (Exception e) {
				e.printStackTrace();
			}
		}
		if (fingerTable[m - 1] == null)
			return successorList[0];
		return successorList[0];
	}

	//debug function
	private void printFingers() {
		for (int i = fingerTable.length - 1; i > 0; i--) {
			if (fingerTable[i] == null) {
				System.out.println("Finger " + i + " is null");
				continue;
			}
			if ((((ChordProtocol) fingerTable[i].getProtocol(p.pid)).chordId)
					.compareTo(this.chordId) == 0)
				break;
			System.out
					.println("Finger["
							+ i
							+ "] = "
							+ fingerTable[i].getIndex()
							+ " chordId "
							+ ((ChordProtocol) fingerTable[i]

	.getProtocol(p.pid)).chordId);
		}
	}
```

```
	public void fixFingers() {
		if(this.thisNode.getID() == 607){
			System.out.println("im here");
		}
		if (next >= m - 1)
			next = 0;
		BigInteger base;
		if (next == 0)
			base = BigInteger.ONE;
		else {
			base = BigInteger.valueOf(2);
			for (int exp = 1; exp < next; exp++) {
				base = base.multiply(BigInteger.valueOf(2));
			}
		}
		BigInteger pot = this.chordId.add(base);
		//BigInteger idFirst = firstChordId;
		BigInteger idLast = lastChordId;
		if (idLast.compareTo(BigInteger.ZERO)==1&&pot.compareTo(idLast) == 1)
{
			pot = (pot.mod(idLast));
			if (pot.compareTo(this.chordId) != -1) {
				next++;
				return;
			}
		}
		do {
			fingerTable[next] = this.find_successor(pot);
			pot = pot.subtract(BigInteger.ONE);
		} while (fingerTable[next] == null || fingerTable[next].isUp() == false);
		next++;
	}

	/**
	 */
	public void emptyLookupMessage() {
		index = 0;
		this.lookupMessage = new int[0];
	}

	public void checkPredecessor(){
```

```
		if(this.predecessor!=null&&this.predecessor.isUp() == false){
			this.predecessor = null;
		}
	}

   public Node getNode(long id){
		for(int i = 0; i<Network.size();i++){
			if(Network.get(i)==null)continue;
			if(Network.get(i).getID() == id)return Network.get(i);
		}
		return null;
	}
}
```

Bibliography

[1] LuisF.G. Sarmenta. *Volunteer Computing.* PhD Thesis. Massachusetts Institute Of Technology June 2001.

[2] David P.Anderson. *BOINC: A System for Public-Resource Computing and Storage.* 5th IEEE/ACM International Workshop on Grid Computing, pp. 365-372, Nov. 8 2004, Pittsburgh, PA.

[3] Jeffrey Dean, Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters.* Google, Inc, 2004.

[4] *Credit overview*. *BOINCSTATS.* http://boincstats.com/stats/project_graph.php?pr=bo (26 Oct, 2010).

[5] Heithem Abbes, Christophe Cérin and Mohamed Jemni. *PastryGrid: decentralisation of the execution of distributed applications in desktop grid.* Middleware Conference Proceedings of the 6th international workshop on Middleware for grid computing, Leuven, Belgium, Article No.: 4, 2008, ISBN: 978-1-60558-365-5.

[6] Ion Stoica, et al. *Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications*. IEEE/ACM Transactions on Networking (TON), Volume 11, Issue 1, pp: 17 - 32, February 2003, ISSN: 1063-6692.

[7] H. V. Jagadish, Beng Chin Ooi, Quang Hieu Vu. *BATON: A Balanced Tree Structure for Peer-to-Peer Networks*. 31st international conference on Very large data bases, Trondheim, Norway, pp: 661 - 672, 2005, ISBN: 1-59593-154-6.

[8] Karl Aberer. *P-Grid: A Self-organizing Access Structure for P2P Information Systems*. Sixth International Conference on Cooperative Information Systems (CoopIS 2001), Trento, Italy, September 5-7, 2001.

[9] Adina Crainiceanu, et al. *Querying Peer-to-Peer Networks Using P-Trees*. In Proceedings of the 7th International Workshop on the Web and Databases: colocated with ACM SIGMOD/PODS 2004, Paris, France, pp: 25 - 30, 2004.

[10] Lei Ni, Aaron Harwood. *P2P-Tuple: Towards a Robust Volunteer Computing Platform*. 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies, 8-11 Dec. 2009, pp: 217 - 223, Higashi Hiroshima, ISBN: 978-0-7695-3914-0.

[11] Zhikun Zhao, Feng Yang, Yinglei Xu. *PPVC: A P2P volunteer computing system*. 2nd IEEE International Conference on Computer Science and Information Technology, 8-11 Aug, 2009, pp: 51 - 55, Beijing, ISBN: 978-1-4244-4519-6.

[12] F. Marozzo, D. Talia, P. Trunfio. *Adapting MapReduce for Dynamic Environments Using a Peer-to-Peer Model*. Proc. of the First Workshop on Cloud Computing and its Applications (CCA 2008), Chicago, USA, October 2008.

[13] Giorgos Georgiadis, Marina Papatriantafilou. *Overlays with preferences: Approximation algorithms for matching with preference lists*. 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), 19-23 April 2010, pp: 1 - 10, Atlanta, GA, ISBN: 978-1-4244-6442-5.

[14] *PeerSim: A Peer-to-Peer Simulator. Peersim*. http://peersim.sourceforge.net/.

[15] Stephen Naicken, et al. *A Survey of Peer-to-Peer Network Simulators*. In Proceedings of the 7th Annual Postgraduate Symposium (PGNet '06) (2006).

[16] Mark Baker, Rahim Lakhoo. *Peer-to-Peer Simulators*. In Proceedings of the 13th Symposium on Operating System Principles (SOSP), 2007.

[17] *Ubigraph.* http://ubietylab.net/ubigraph/.