2010 2nd International Conference on Information and Multimedia Technology (ICIMT 2010)
IPCSIT vol. 42 (2012) © (2012) IACSIT Press, Singapore
DOI: 10.7763/IPCSIT.2012.V42.1

Software Deployment Based on Peer-to-Peer

Xuehan Xu, Mingfa Zhu, Limin Xiao, Li Ruan

State Key Laboratory of Software Development Environment
School of Computer Science and Engineering, Beihang University
Beijing, China
xxhdx1985126@gmail.com, {zhumf, xiaolm, ruanli}@buaa.edu.cn

Abstract—In the past few years, along with cluster systems being broadly deployed in many application fields, large-scale clusters have become the mainstream of the high-end computing, while the bandwidth of interconnection networks is relatively low. The bottleneck of the deployment server, brought about by the low bandwidth of the interconnection networks in Cluster Systems, is making it more and more difficult to deploy software in a large-scale cluster. This paper proposes a cluster software deployment system ---- PDeploy ---- which achieves the goal of rapidly deploying software in very large-scale cluster systems with only one initial deployment server by combining DHT and the improved P2P swarming mechanism. Based on the result of our experiment, PDeploy is presenting a better performance than other well-known resource distribution systems both when deploying largely requested software and little requested software with highly loaded servers.

Keywords-peer; deployment server; cluster; fast deployment; software resources

1. Introduction

Along with the development of high speed interconnection networks and the upgrading of the application requirement in terms of processing speed and storage space, cluster systems, with their advantages such as high cost performance and being easy to implement, are becoming more and more popular in various kinds of commercial and scientific computing fields. Almost all the cloud computing platforms presented today are cluster systems; and meanwhile, cluster systems are also dominating in the field of high-end scientific computing. 424 of all the 500 machines, about 84.8%, participating the 35th TOP500 rating in June 2010, are cluster systems[1]. In addition, as the clusters being more and more used, the scale of a single cluster is also growing, from hundreds of nodes in the early time to tens of thousands of nodes today. At present, the Google Cluster can combine more than 15,000 PCs[2]. On the other hand, the scale of clusters used for high-end scientific computing has also experienced a great expansion. The Dawning 5000A manufactured in 2008 has 1920 nodes; the Dawning Nebulae, 4640 nodes; and the world's fastest machine, Jaguar, 18688 nodes. According to article[3], this number will continue to grow in the future.

With the scale being so large, it is very hard and unnecessary to combine all the nodes to run a single task. In fact, these nodes are usually distributed into several groups, each running a different task. On the other hand, the tasks running on a cluster can vary greatly over time, to accomplish these tasks efficiently, the nodes groups has to adjust their scale accordingly, which makes the fast execution environment construction on the nodes, the core of which is the fast deployment of the software required to run the task, very important to the working efficiency of the cluster. However, as the bandwidth of the interconnect network is relatively low to the cluster scale, the efficiency of software deployment is gravely interfered by the bandwidth bottleneck of deployment servers, which makes the fast software deployment in large-scale clusters a urgent-to-solve problem. Let's assume that there are 5000 nodes waiting for an OS installation. With a 1GB/s bandwidth on

the deployment server side and a 3GB OS installation image, both of which are common cases in today's cluster systems, we can easily come to a conclusion that we need 33 hours to accomplish the OS deployment, which means we need a day and a half to merely install the OS. This is just within an ideal environment. Considering the actual situation can be totally different and the load on the interconnection network can vary continuously, the actual deployment time could probably grow to an unacceptable end. Although many clusters solve the problem by increasing the number of deployment server, this can only alleviate the pressure temporarily and can be very inefficient for these servers will become idle when there is no need to deploy software in cluster.

However, we can see that although the bandwidth on the deployment server side is very limited, there are still a lot of network resources remaining unutilized during the traditional software deployment procedure, which can be much faster if we can use these idle resources. As a result, in this paper, we developed a software deployment system based on the P2P technology ---- PDeploy, which can easily achieve the goal to realize fast software deployment with only one additional deployment server by utilizing idle network resources in the cluster system.

The design goals for PDeploy are the following:

- Highly efficient: PDeploy should be able to efficiently deploy multiple software in parallel.
- Scalable: PDeploy should be able to fit into very large-scale clusters.
- Reliable: PDeploy should be able to guarantee the availability of software at any time.

To realize the fast software deployment, PDeploy combines DHT and the swarming mechanism[15] like used in BitTorrent. The base strategy that PDeploy adopted to provide deployment service is to select some amount of nodes which has a relatively low load to other nodes in the cluster to run as deployment servers, each holding a portion of the software images distributed by the initial deployment server using DHT. And then, when deploying software, all the requesting nodes download the software in a BitTorrent-like manner which utilize the swarming mechanism to accelerate the deployment procedure. One thing to emphasize is that every software image can have multiple copies spread over the servers. The reason for doing so is that the swarming mechanism is mainly design for large, popular files, while as to unpopular files, the effect of acceleration is not very distinctive. What's worse, when deploying multiple software to a massive amount of nodes, the deployment of unpopular software can be even slower for the bandwidth for deploying these software is compressed. So, to provide acceleration to deploying both of these kinds of software, we need to set up multiple servers holding some redundant of the software images, which can increase the downloading sources of unpopular software. Although most system implementing BitTorrent protocol also support multiple resource servers, they cannot insure the load balance among them. So we adopted DHT to make up this shortcoming. On the other hand, storing multiple copies over the servers can also improve the reliability of the system.

In addition, when using BitTorrent systems to deploy software, all the nodes will ask the server for resources that are not yet been downloaded by these nodes, which is acceptable on internet, as the time of the emerging of download requests is random. However, in cluster systems, the emerge of download requests is controlled by a center server. When there is necessity to reconstruct some nodes' the software environment, the center server will send the reconstruct command, and the nodes will send out the requests following the instruction, which makes the deployment strategy mentioned above lead to a server crash due to the massive concurrent requests. So, to avoid this situation, PDeploy proposed a back off strategy.

The rest of this paper is organized as follows: in section II, we introduce related work about software deployment and P2P technology. In section III, we present the overlay network structure of PDeploy. In section IV, we describe the resource deployment strategy of PDeploy. We show our experiment result in section V, and the future work in section VI.

2. Related Work

To avoid the bottleneck on the server side, many researchers have adopted multicast technique which could be used on the network layer[4] [5] or the application layer[6]. The main drawback of using IP multicast is that it usually needs the support of network switch devices. In addition, adopting IP multicast needs to maintain multiple multicast group, which could quickly run out the resources on routers and make the switch

device the bottleneck of the software distribution. Application layer multicast performs well when the nodes averagely remain in the overlay network for a relatively long time. But the performance could be degraded rapidly when the nodes join and leave the overlay frequently, which may be a common case for the cluster system in order to reduce the impact of deployment system to the performance of the whole cluster.

As mentioned previously, some clusters set up multiple deployment servers to alleviate the bandwidth pressure[7]. The drawbacks are obvious, bad scalability and wasting resources.

Another strategy[8] which, similar to us, also picks working nodes within the cluster to run as deployment servers works in a different way. Unlike PDeploy, it first deploy a portion of the requesting nodes, then set them up as servers of the software they requested, providing software deployment in the same way as the initial server, which forms a tree during the deployment. This strategy can, to some extent, utilize the unused network resources within the cluster to provide an acceleration to deployment, but drawbacks remain. At first, nodes can only obtain the requested software from its parent node, which can't still fully utilize the idle resource in the cluster. In addition, obtaining software only from the parent node makes the system very vulnerable to the crash of the internal node of the deployment tree.

Recently, P2P technology has made considerable progress and has been applied to many tasks, including software deployment. Article[9] put forward a strategy that use BitTorrent to synchronize images among deployment servers. But the downloading is based on the traditional C/S model which is still inefficient.

The work closest to PDeploy is BitTorrent[10], [11]. For systems implementing BitTorrent protocol, a server called "tracker" will be required to help the clients discover each other. In addition, a random mesh is constructed for the clients to notify others about their new downloaded resources. The lack of back off mechanism and the setting of a central tracker makes the system very vulnerable to massive concurrent requests when used in large-scale clusters. Besides, BitTorrent lack the load balancing strategy for the "seeds", which means that when used for software deployment in cluster systems, its performance can be degraded due to the imbalance among servers.

3. System Architecure

PDeploy is totally based on the overlay network it constructed in the cluster which is comprised of two Chord[12] rings, one for distributing software resources among the deployment servers in a load balancing way, the other for all the nodes in the cluster to request the software images. Therefore the former contains only the deployment servers, while the latter contains all the nodes in the cluster. With this mechanism a node can quickly locate a specific image and then capture it following the deployment strategy that will be described in section VI.

3.1 Overlay Network Structure of Deployment Servers

It is required that an initial deployment server already existed when PDeploy start running which will capture load information reported by the nodes in the cluster, comprised of the current load, including CPU and disk usage and network bandwidth per connection, and the total capacity of the reporting node, and then determine which nodes to select to set up as deployment servers. All the nodes ask the initial deployment server for the number of servers it picked which is crucial for the lookup of the software images at the first time they request software images. Here, we just ignore the dynamism of the cluster scale which may lead to dynamism of the number of deployment servers, and leave it for the future work.

Deployment Server Selection Algorithm

INPUT:
$X_i = [x_1, x_2, \dots, x_n]^T$: load of the *n* bottleneck resources on node *i*.
$Y_i = [y_1, y_2, \dots, y_n]^T$: capacity of the *n* bottleneck resources on node *i*.
$Z_i = Y_i - X_i$: bottleneck resources left on node *i*.
$W=(Z_1, Z_2, \dots Z_m)$: matrix of the bottleneck resources left in the cluster.
$U = [u_1, u_2, \dots, u_n]$: bottleneck resources required to run the deployment.
n: number of bottleneck resources.
m: number of nodes in the cluster.
OUTPUT:
N: the number of deployment servers
DS: the set of deployment servers
ALGORITHM:
1: $N \leftarrow 0$
2: $DS \leftarrow$ NULL
3: *nodes* ← All the nodes in the cluster
4: **for** $i \leftarrow$ 1 to n do
5: norm (i) ← (n_j: the weight)
6: sort the vectors in W by the corresponding value in vector "norm".
7: $sum \leftarrow 0$
8: **for** $i \leftarrow$ 1 to n do
9: **for** $j \leftarrow$ 1 to m do
10: $sum \leftarrow sum + W_{ij}$
11: **if** ($sum + W_{i(j+1)} > U_i$)
12: goto 15;
13: **endfor**
14: **endfor**
15: $N \leftarrow i$
16: $DS \leftarrow \{nodes(1), nodes(2), \dots, nodes(N)\}$

Algorithm 1. N: the number of deployment servers; DS: the set of deployment servers

The pseudo-code of the algorithm is shown above. At first, we seek the norm of all the vector in the matrix W, $||Z_i|| = \Sigma n_j W_{ij}$, where $n_j \geq 0$ is the weight of the component W_{ij}, which is used to demonstrate the importance of the resource to the deployment procedure. Then, we sort the vectors in W by the corresponding value in vector norm. At last, For every component u_i in U, accumulate the value of the corresponding item in vectors of W until the requirement u_i is satisfied. N is the maximum value of all these sums and DS is the first N nodes corresponding to the vector in W.

As we can see, the selection will succeed as long as the U is known. The algorithm above runs in $O(n^2)$ time.

The following shows the method for determining U. Here, we assume that the resources required to run the deployment service is determined by the resources required to deploy software of the largest size to the whole cluster. Let *t* be the maximum time the deployment task can tolerate, *Q* be total number of nodes requesting the software, then:

$$u_j = k_j \frac{Q}{t} \tag{1}$$

k_j is the coefficient which can be determined by experience. For instance, for network bandwidth, k_j can be set to:

$$k_j = \frac{\text{size of the largest software}}{\theta} \tag{2}$$

where θ stands for the ratio of the bandwidth required on the server side after utilizing swarming to that required before using swarming.

PDeploy put the deployment servers into a Chord ring. A software image is split into several blocks and distributed to the deployment server using the Consistent Hash[13]. Note that every block have several copies also saved in the Chord ring to improve the reliability of PDeploy. Every block is further split into pieces which is the base unit of the deployment procedure. The reasons for adopting Chord is that Chord can insure the static load balancing among servers easily. We leave the dynamic load balancing on structured P2P network for the future work. The metadata of the software images is also saved and spread in the Chord ring, including the name of the software, addresses of the servers holding blocks of the software, all the nodes requesting the software, etc. The content of metadata is created by the initial deployment server. The structure of the Chord ring for deployment servers is shown in Figure 1. Note that the metadata of a specific software has only one copy in the ring, while the software images can have several copies.

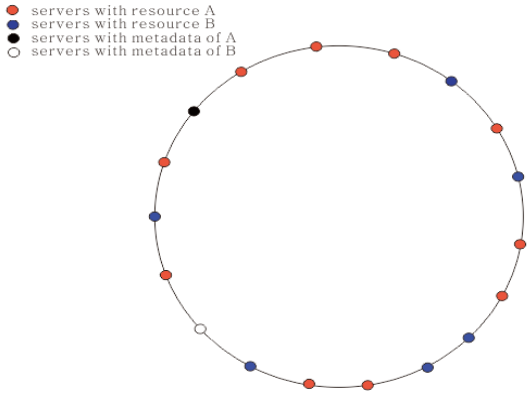

Figure 1. The Chord ring of the deployment servers. The red nodes represent servers holding software A, the blue ones represent servers holding software B, the black ones represent the servers holding metadata of A, and the white ones represent servers holding metadata of B

3.2 Overlay Network Structure of the Nodes

To accomplish the deployment, PDeploy organize all the nodes, including deployment servers, into another Chord ring in which servers take a server name set obeying a uniform naming rule, a string adding a random number(In this paper, we use string "server" adding some random number within the scope from 1 to the number of servers to comprise a server name like "server07"), as the input of the hash function. All other nodes ask the initial deployment server for the string when the first time they request software, and add it with a random number within the same scope as prescribed by the naming rule before they start their lookup every time they query some specific software. Through this method, we can insure that there is always a server corresponding to the server name. In PDeploy, nodes acknowledge the addresses of the servers holding the specific software through the metadata. To locate it, nodes has to make a lookup as following:

1. Create a server name randomly in the way mentioned above, and locate the server through consistent hashing.
2. Ask the server to lookup the metadata of the specific software on the node's behalf in the Chord ring of servers.

After capturing the metadata, the node pick some other nodes recorded in the metadata randomly, to establish a TCP connection called neighbor connection to exchange the RDI, the *Resource Downloading Information*. Each node has a upper bound η for neighbor connections which is determined according to the available bandwidth through a lightweight bandwidth estimate algorithm as described in article[14]. This algorithm runs every 400ms, and returns *underutilized*, *at-capacity*, and *throttle-back*. we increase the η by 1 on the *underutilized* returned.

During the deployment of a specific software, each node broadcast their RDI through the neighbor connections containing information every time they get a specific number of new pieces. Nodes receiving this information forward it through the rest neighbor connections, which finally makes all the nodes in network acknowledge the RDI. Nodes determine where to get the pieces they want according to RDIs.

The RDI takes the form <IP address, port number, list of updated pieces>. PDeploy uses bit sequence to represent the pieces a node downloaded the length of which represents the total number of pieces a software

has got. The *i*th bit corresponds to the *i*th pieces of the software, and is set to 1 if the pieces has been downloaded to the node or 0 if it has not. Every node maintains a RDI tree as shown in Figure 2. The leaf nodes are the RDIs of the nodes in the cluster. The internal nodes' bit sequences are the OR of their children's. It's easy to see that nodes requesting the same software can easily acknowledge the downloading progress of all other nodes. With this tree, a node determining where to get some specific piece can easily find the corresponding nodes by traversing the tree in a top-down manner. Note that each node has only one copy of their RDIs, once a new RDI arrives, the old one is simply replaced.

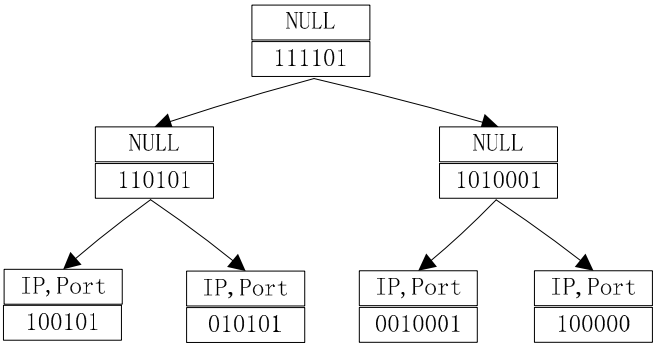

Figure 2. The RDI tree. As we cansee, all the bit sequences of internal nodes are the OR of their childrens.

4. Software Deployment Strategy

The overlay network structure described above is used for the initial server to distribute software images among servers and other nodes to look them up. In this section, we will present how the nodes in the cluster download software images from the servers after the lookup finished.

4.1 Back Off Mechanism

Before deployment of a specific software begins, only servers have the software image, which means that it's probable for the servers to go down due to the massive concurrent requests. To prevent this from happening, we designed the back off mechanism which falls into two parts, back off mechanism for requesting metadata and back off mechanism requesting software pieces.

Back Off Mechanism for Requesting Metadata

At the beginning, all nodes send requests to the metadata server with the probability p/m, the m being the number of nodes requesting the same software, and then, the nodes failed to pass this arbitration wait an interval ψ and send the requests again with the same probability. This procedure continues until all the nodes have sent requests successfully. The setting of p is crucial to the efficiency of this procedure. Being too high will lead to the high probability with which the servers can claps, while being too low will lead to an unacceptable execution time.

Back Off Mechanism for Requesting Software Pieces

Similar to the situation mentioned above, if all the nodes start their downloading from the piece No. 1 concurrently, the server may claps due to the massive concurrent connections with other servers holding blocks of the requested software being idle. To solve this problem, we designed a randomly back off mechanism which requests each nodes starts their downloading from the server with probability p/m, and chooses a random piece of the software to start with if it passed the arbitration. The rest wait an interval τ and then check the RDI tree. If some pieces has already been downloaded to some nodes, then the rest nodes download the pieces following the strategy described in section B, or, they will again start their downloading with probability p/m. If, during the deployment procedure, the server is the only machine that possesses some specific pieces, all the nodes would ask the server with probability p/m following the similar manner mentioned above. Note that the connection to the server will be closed when bandwidth estimate algorithm returns *at-capacity*.

4.2 Strategy for Downloading Pieces

To save the bandwidth used by PDeploy, nodes download pieces from the ones that they have already established downloading connections with, except the servers. When a piece is not existed on these ones, the node check the RDI tree to see if there is other nodes possessing the piece. If so, say node B is holding the piece, the node will ask B for the piece with probability ω, that is the node will ask deployment servers for the piece with probability 1-ω. Since in most cases, nodes requesting the same software is much more than deployment servers, we should set ω larger than 0.5 to make nodes more likely to download pieces from each other, reducing the load on the servers. When a piece is already existed on a node that it has established downloading connection with and it has also connected to a server, then the piece on the node will be selected.

Each node has a target number of downloading connections, and again, it is determined according to the return of bandwidth estimate algorithm in the same way as neighbor connections.

5. Experiments

In this section, we designed two experiments to compare PDeploy with other two protocols, BitTorrent and FTP, in both of which PDeploy exhibited a better performance. Due to the limitation of the laboratory environment, we only test PDeploy on the laboratory network. Leaving tests on large-scale clusters for future works.

5.1 Testing Environment

We organize 33 PCs containing one Intel® Core™ 2 Duo E4400 @2.00GHZ cpu, 1GB RAM, 100Mb Ethernet and a server with one Quad-Core AMD Opteron™ Processor 2350 cpu, 4GB RAM, 100Mb Ethernet into one LAN. The network topology is shown in Figure 3.

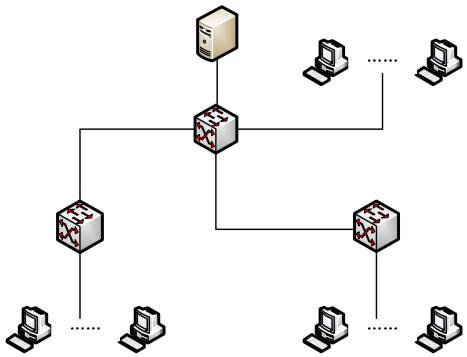

Figure 3. Testing environment

5.2 Experiment Scenario

In this paper, we designed two Scenarios:

1. Deploy the same software using PDeploy, BitTorrent and FTP separately.
2. Deploy different software simultaneously. We deploy a 4.3GB software on 22, 26, 30 PCs successively, and in the mean time, deploy a 4GB software on the rest PCs to test the effect of acceleration PDeploy can make to the unpopular software.

5.3 Experiment Result and Analysis

In the experiments, arguments are set as follows:

- As we consider the load on network, only the weight *n* corresponding to the bandwidth is set to 1, others are set to 0. k_j in Equation (1) is set to(ϴ=1.6):

$$k_j = \frac{4.3 \times 1024}{1.6} = 2752$$

- In back off mechanism, *p* in probability *p/m* is set to 3, ψ is set to 300ms. ω is set to 0.8.

Result of Scenario 1

The result of scenario 1 is shown in Figure 4:

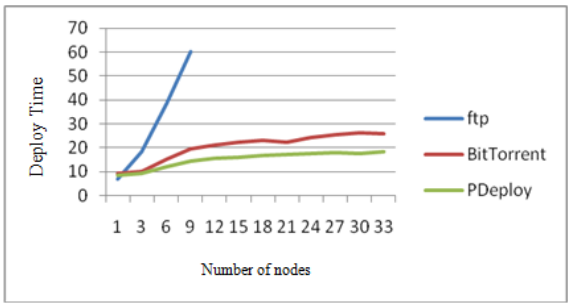

Figure 4. Result of scenario 1. The blue line represents the deploy time of ftp, the red line represents BitTorrent and the green one represents PDeploy.

From the result, we can see that along with the increasing of the node number, the deployment time of the traditional deploy strategy increases rapidly, becoming unacceptable very soon. BitTorrent can alleviate the problem well, while, as we can see, with the node number increasing, the gap between BitTorrent and PDeploy augmented rapidly. According to the result, PDeploy complete the task 30% faster than BitTorrent. There are three major causes of this phenomenon:

1. During the deployment, PDeploy constructed multiple servers, while BitTorrent has only the initial deployment server.
2. Along with the increasing of nodes number, the back off mechanism of PDeploy can reduce the load on the deployment server and utilize the network resource of the downloading nodes effectively, accelerating the deployment, while, in comparison, BitTorrent only ask the only server for the pieces that are not yet downloaded, which causes the increase of the load on the server, limiting the deployment speed.
3. The tit-for-tat mechanism, to some extent, limited the speed of deployment, while PDeploy abandoned this mechanism.

We can see that with the scale of cluster augmenting, the deployment time of PDeploy tends to be stable, from which we can infer that even for large-scale cluster, PDeploy can insure accomplishing the deployment a software within several hours.

Result of Scenario 2

The result of scenario 2 is shown in Figure 5:

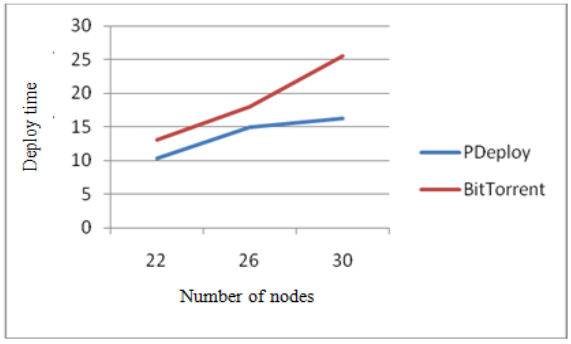

Figure 5. Result of scenario 2. The blue line represents the deploy time of PDeploy, while the red one represents BitTorrent.

Due to the unacceptable time FTP taken to accomplish the deployment, we didn't show it in our chart.

As we can see from Figure 5, as to unpopular software, PDeploy can still offer almost unchanged deployment speed. This is mainly because PDeploy adopted DHT by which software images are distributed onto deployment servers in a load balancing way, decentralizing the load; and on the other hand, every software has several copies spread over the group of deployment servers offering deployment service simultaneously, which increases the exists of the unpopular software leading to an acceleration.

In conclusion, PDeploy can provide acceleration to the deployment of the popular software and unpopular software while deploying multiple other software, which is the exact goal of the our design.

6. Future work

Although PDeploy can provide fast deployment of multiple software simultaneously, several problem besides the ones mentioned before remains:

- Dynamic load balancing among deployment servers
- PDeploy network traffic control strategy

7. Acknowledgment

This research work is supported by the National Natural Science Foundation of China under **Grant No. 60973008**, the fund of the State Key Laboratory of Software Development Environment under **Grant No. SKLSDE-2009ZX-01**, the Fundamental Research Funds for the Central Universities under **Grant No. YWF-10-02-058**.

8. References

[1] http://www.top500.org/

[2] Luiz André Barroso, Jeffrey Dean , Urs Hölzle. WEB SEARCH FOR A PLANET: THE GOOGLE CLUSTER ARCHITECTURE. In *Proceedings of IEEE micro,* 2003.

[3] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Peter Kogge, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, Katherine Yelick. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. September 28, 2008.

[4] Lin Han, Nahid Shahmehri. Secure Multicast Software Delivery. In *Proceedings of the 9th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 2000. Pages: 207 – 212.

[5] Marc Gumbold. Software Distribution by Reliable Multicast. In Proceedings of 21st Annual IEEE International Conference on Local Computer Networks (LCN'96), 1996. Page(s): 222 - 231

[6] Mojtaba Hosseini, Dewan Tanvir Ahmed, Shervin Shirmohammadi, Nicolas D. Georganas. A Survey of Application-Layer Multicast Protocols. In *IEEE Communications Surveys & Tutorials,*2007.

[7] Dillinger M, Becher R. DECENTRALIZED SOFTWARE DISTRIBUTION FOR SDR TERMINALS. In *IEEE Wireless Communications Magazine*. Vol. 9, no. 2, pp. 20-25. Apr. 2002

[8] Xue Zhenghua, Dong Xiaoshe, Hu Leijun, Wu Weiguo. Study on Transfer Model of Deployment System for High Performance Server Cluster. In *Chinese Journal of Computers*, 2008. Vol. 31 No. 11.

[9] Purvi Shah, Jehan-François Pâris, Jeffrey Morgan, John Schettino. A P2P-Based Architecture for Secure Software Delivery Using Volunteer Assistance. In *Proceedings of the 2008 Eighth International Conference on Peer-to-Peer Computing,* 2008. Pages: 131-139.

[10] http://www.bitconjurer.org/BitTorrent

[11] B. Cohen, "Incentives build robustness in bittorrent," in P2P Economics Workshop, 2003.

[12] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakr-ishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[13] David Karger,Eric Lehman,Tom Leighton, Matthew Levine,Daniel Lewin, Rina Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, El Paso, Texas, United States, 1997, Pages: 654 – 663.

[14] Rob Sherwood, Ryan Braud Bobby, Bhattacharjee. Slurpie: A Cooperative Bulk Data Transfer Protocol. In *Proceedings of the IEEE INFOCOM*, 2004.

[15] Nazanin Magharei, Reza Rejaie. PRIME: Peer-to-Peer Receiver-Driven Mesh-Based Streaming. In *IEEE/ACM TRANSACTIONS ON NETWORKING*, VOL. 17, NO. 4, AUGUST 2009