Load Balancing of Access Intensive Keys in Distributed Hash Table: Application to P2P Scientific Computing

Arnaud Dury

CRIM, H3A 1B9, Montréal, Québec, Canada,
Arnaud.Dury@crim.ca,

Abstract. Classical distributed computing projects generally use a specialized client/server model. Recent approaches, such as BOINC, favor instead the development of distributed computing platforms, relying on a generic client/server model. We propose a fully decentralized computing model, considering all participant as peers that can submit personalized computing tasks to any number of other peers currently offering their services, listed in a peer directory. Our model is built upon Chord, a particular Distributed Hash Table. Chord allows load balancing of the number of keys per node, but offers no way to balance the bandwidth load of a frequently accessed key, such as a peer directory. Our model extends Chord with load-balancing of those access-intensive keys. We present a modelization of the bandwidth and storage costs of our model and experimental performance results using a variable number of peers, tasks, tasks time, and a variable ratio of contributors and solicitors roles among peers.

1 Introduction

Distributed computing is a tool used in a growing number of research fields: mathematics [13][3], biology [10], radio-signal analysis [19], protein folding [21], [7], genome analysis [17], [9], meteorological previsions [14], crypto-analysis [1] and others. These works are performed in a massively distributed manner, using idle time from computers of generous contributors over Internet. But for their popularity, the number of participants is still limited by the absence of tangible rewards for the contributors, and the barriers of entry for any new project are high. We present in this paper a Peer-to-Peer model for distributed computing addressing these two issues, based on distributed directories over a Distributed Hash Table (DHT). We propose a new solution for the handling of the "hot spots"[1] such directories generate, and for which classical caching algorithms in P2P systems are not applicable. We present a theoretical modelization of the bandwidth and storage cost or our model, and we relate this modelization to experimental results. The first part of the paper gives a description of the

[1] Hot spots are keys in a DHT so frequently accessed that they introduce network congestion for the nodes responsible for them.

classic client-server approaches, as well new, more distributed approaches [4], [8]. We then introduce our model, which extends the consistent hashing used in Distributed Hash Table to balance the load of keys that are both frequently read and frequently modified. We present a study of performance of our model, by modelization and experiments, and we conclude on the possible use and future extensions.

2 Distributed Computing Approaches

Distributed computing is in growing use to tackle computationally-intensive research projects. Several models exist in this domain, and we summarize below three of them, from the most centralized to the most distributed. We then introduce our own model.

2.1 Specialized clients/server platform

The most famous of these projects is Seti@Home [19], with more than four millions users. Seti@Home uses a client/server approach: a database of "data units" is stored on a central server, to which the distributed clients, implementing the algorithm to be applied to the data, connect at their own pace. In this model, clients contribute without deriving any benefit from their participation, and participants are restricted to the most altruistic or science-inclined persons. [2]. Moreover, the barriers of entry are high: soliciting organizations must produce the client and server code, and take care of the bandwidth limitations and availability of their Internet link, so as not to miss returned results. They need to *advertise* their projects, and this is probably the biggest hurdle: potential contributors will not download clients of a project they are not aware of.

2.2 Generic client/server platform

BOINC [4], [20] is a new generic software platform for distributed computing. BOINC offers an existing base of code to re-use, thus facilitating the development of new projects. Each contributor can easily participate in several BOINC projects simultaneously, by downloading the specific code of each project inside the BOINC client. BOINC offers security via cryptographic signature: a contributor can validate the origin of any code he chooses to use on his own machine. There is no equity of contributions in the BOINC model, and bandwidth and availability constraints are still present. Furthermore, a contributor need to download manually the code of any project he wants to contribute to: advertising is still needed for any new project.

[2] Seti@Home represents only 0.6% of the 665 millions Internet users worldwide in 2002 (http://www.etforecasts.com/pr/pr1202.htm)

2.3 Generic Peer-to-Peer Platform

In [8] the authors underscore that while a lot of work has been done on Grid Computing oriented toward research organizations, few works have been oriented toward small teams of researchers, or the general public. They suggest a new approach they call "Personal Grid", using a Peer-to-Peer model to perform distributed computing. In the "Personal Grid" model, computers organize themselves in a P2P network using geographical constraints: they create clusters of machines physically close to each others. A particular machine is elected as the supernode of each cluster, and is used to route queries among the network. Queries for idle nodes are propagated using a broadcast policy on the cluster of the node making the query, and a selective broadcast policy to a set of others clusters. This request mechanism is a controlled flooding of the network, that can lead to network congestion as the authors indicate if the set of clusters to broadcast to is too large. Alternatively, if this set is too small, it is possible that a request may not be satisfied immediately, while idle nodes exist in the system.

2.4 The Units Allocation Issue

A units allocation issue arises in fully decentralized P2P model: as the requests for resources are requests for the nodes of the network itself, how to answer them while avoiding network congestion, due to massive propagation of the requests to each node of the P2P network ? Another problem is that the solicitor is assumed to be permanently connected to the network, and able to handle the load of any number of units returning results simultaneously. When a long computation is ongoing over *already distributed units*, the solicitor has no other choice but to stay connected until the last result returns. There is no guaranty that contributors would retry their submission of results, if the solicitor is not available the first time. Lost results will lead to the re-submission of units to the network, wasting time for every participant.
In the following, a *Node* describes a participating computer of the system, a *Unit* describes a pair $< Code, Data >$ representing a particular computation over a particular set of data, a *Contributor* describes a node in the system offering its services, and a *Solicitor* describes a node in the system currently soliciting others nodes for their time. Contributor and Solicitor are thus *roles* any Node can have (and switch to) over the course of its execution, and do not refer to a static description of the behavior of any one.

3 Network Efficient Nodes Discovery in a Distributed Hash Table

We build a P2P distributed computing model atop of a P2P file-sharing model. The two main concepts of our model are the indexing of idle contributors nodes and solicitors nodes in two distributed directories over a Distributed Hash Table (DHT), and the use of the same DHT to store computed results as soon as they

are produced, if their originator is not connected. We developed a distributed directory model, that offers load balancing of the bandwidth among the nodes, and provides exhaustive answers to queries while preventing the flooding of the network. We chose the DHT model over more classic approaches such as Gnutella [15], Napster [2] or FreeNet [6], because requests algorithms in these models are either not sufficiently efficient or not fully decentralized. Recent approaches such as Chord [18], Pastry [16] and Kademlia [11] introduce a new request procedure, in which query routing in the node identifier space is based on a hashing of the key name to find. These approaches, that are moreover fully decentralized, are now favored in the research literature due to their provable communication costs, provable stability under nodes join and leave and support for both Read and Write operations. We choose to use a Distributed Hash Table model such as Chord [18], a completely decentralized model offering the API of a Hash Table with $Get(key)$ and $Put(key, data)$ operations. Chord uses a hash of the key name, and map it onto an identifier space shaped as a ring, where it will be attributed to the node having the closest identifier following the computed hash on the ring.

The system can have at times an excess of contributors, and at other times a excess of solicitors. To address the units allocation issue, our model uses two distributed directories, one for solicitors and one for contributors. When a new contributor node joins the network, it checks for existing solicitors in the solicitors directory. If any are found, it will contact one of them, chosen randomly [3], to collect units to process. If none are found, the contributor will register itself to the contributor directory. The same principle applies when a solicitor joins the network. Each failed directory lookup will be memorized for a short-time by the directory itself to alleviate the race-condition of this simple algorithm. Using a DHT, we have the necessary primitive at our disposal to implement this model, but the issue of bandwidth costs remains. In a DHT, an information is stored under a key, on a certain node. The bandwidth of the node storing the directory will quickly become saturated, because it will receive all the requests made by all the nodes of the system. The load balancing present in Chord is a balancing of the number of keys among the nodes, but not a balancing of the load access to a particular popular key [4] While caching extensions to Peer-to-Peer systems (such as [12]) have been proposed to solve this problem of "hot spots", or highly accessed keys, they are not applicable to directories. Directories can be voluminous, and in a period of high activity, nodes may enter and leave them frequently. Caching is inefficient in the case of frequently modified data: the maintenance of the coherence of voluminous keys frequently accessed throughout several caches is costly. We present an alternative and more efficient method, based on an extension of consistent hashing, avoiding the cost associated with caching. We

[3] The use of metadata for a better matching of solicitors and contributors is an obvious extension that we are considering.

[4] Balancing for the number of nodes is done using the concept of Virtual Nodes: each key is assigned to a Virtual Node Identifier, and these virtual nodes identifiers are distributed to the real nodes.

built a theoretical cost model that we compare to experimental results obtained in simulation.

4 Segmentation of an access-intensive key over a consistent set of linked nodes

Chord associate to each key K_i a hash code H_i using a function $Hash(K_i) = H_i$. The node ω_i responsible for K_i is the node whose identifier is the closest to H_i when proceeding in a clockwise manner from H_i on the identifier ring. We propose in this paper to extend this hashing from $Hash(K_i) = H_i$ to $Hash(K_i) = \{H_j : H_j = \phi(Hash(K_i), j),\ 1 \leq j \leq k\}$. We construct ϕ such as $\phi(Hash(K_i), 1) = Hash(K_i)$, and in such a way that the values returned by ϕ can be computed by any node of the system, independently of the information they have on the current topology of the network. The function ϕ can not use any meta-information stored in the DHT because the node storing this meta-information would become the new saturated node. We will use the name "segmented list" for the set of nodes $\{\omega_1, \omega_2, ..., \omega_k\}$ that are responsible for the hash code $H_1, H_2, ..., H_k$, the name "segment" for each of these nodes to differentiate them from others nodes of the system, and the name "segment population" to define the number of identifiers stored in a segment. Each of the segments will be linked to the next one, in a circular manner, as we show on the figure 1. A nodes directory is thus a named segmented list, whose name serves as the key of the list. The choice of the ϕ function embodies the strategy for the key distribution. The strategy we choose tries to avoid the use of nodes already responsible for the replication of another part of the same list, in the redundant copying scheme of Chord. This strategy spreads out the data and bandwidth load as much as possible throughout the identifier ring, by allocating further segments of the list to nodes currently as far as possible from the nodes already responsible of others segments of the list (see figure 1). The function ϕ is thus only based on the key hash code (defining the location of the first segment location on the ring) and the number of segments, and can be computed independently by any node. A node willing to register randomly chooses a segment number from 1 to the estimated bound of the number of segment. *This random choice will make nodes spread the load on the directory key over the sets of segments responsible for it.* The bound will be computed using the number of nodes per segment, which is a parameter of the system, and the estimated total number of nodes in the system, which is impossible to compute precisely, but for which we can obtain a good estimation. The computation relies on the assumption of an homogeneous identifiers distribution over the identifiers space, which will generally be the case if a random choice is used for the attribution of identifiers. In a Chord ring, each node maintains an index table of successors at successive power of two from them: the first successor, the second one, the fourth one, etc. Each node ω_i can compute the angular density of nodes on the ring, using data from its index table. Assume l is the number of entries in the index table of ω_i, $idx_i[l]$ is the last entry, and $\Theta(\omega_a, \omega_b)$ is the angle from ω_a to ω_b on the ring. The

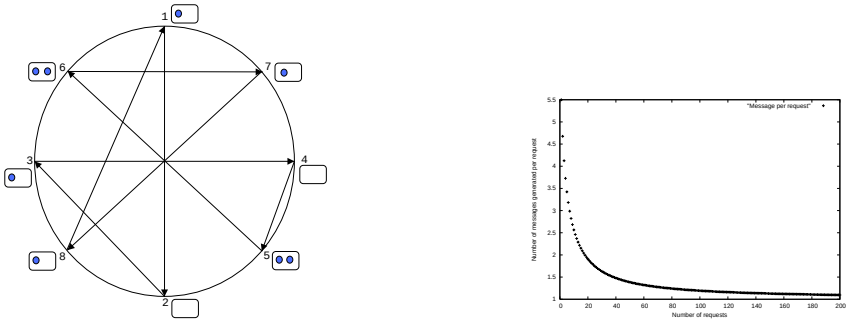

Fig. 1. Segments choice strategy (left) - Number of messages per request (right)

angular density d is calculated as $\frac{2^l}{\Theta(\omega_a, idx_a[l])}$. The total number of nodes is then estimated by $T = 360 \times d = \frac{360 \times 2^l}{\Theta(\omega_a, idx_a[l])}$. We note β the maximal segment population. The total number of segments is thus $\frac{T}{\beta}$. Experiments with randomly generated identifiers show that with 600 nodes and a maximum segment population of 50, each node compute the correct number of segments with a margin of error of $+-1$ segment. A registration request to a directory of name K_i is sent to the node responsible for the hash code $\phi(Hash(K_i), j)$ where j is chosen randomly in the $[1, \frac{T}{\beta}]$ interval. The node receiving this request will register the node if its local population is under β, and will confirm to the subscribing node that its registration is done. If the node has reached its maximal population, it will forward the request to the next node in the chain. When a node wants to discover an other node registered to a particular directory, a similar method is used: a node is selected randomly among the chain, the request is sent and propagated until a non-empty chain node is reached, or until the request has traversed all the chain nodes. The first node able to answer will do so, and stop the propagation. When a node wants to remove itself from a particular directory, it sends a removal request to the node responsible for its registration, that was stored when the registration answer was received. Removal is thus an operation with minimal communication costs.

5 Theoretical modelization

We introduce a modelization of memory and bandwidth costs associated with two models: the DHT storing the whole index in one node, and our model using a segmented list that can be accessed at any segment.

5.1 Notations

- T is the number of nodes in the system. Each node can act as a contributor or a solicitor whenever it wants.
- Δ is the average time of computation for a work unit in seconds.

- U is the total number of work units to compute.
- B is the maximal upload bandwidth consumed.
- S is the size of request and reply packet. We assume $S = 1500$ bytes.
- β is the maximal population of a segment. $1 \leq \beta \leq T$

5.2 Distributed Hash Table

Assuming that $U \gg T$, we compute the bounds of the memory and bandwidth imposed to the node storing the directory, during a period of $d = 1$ seconds. This node will store every identifier of the system. We assume these identifiers consume m bytes each. The memory consumed is thus $M = \frac{T*m}{1024}$Kb. The node responsible for the key will also answer directory look-ups. We compute the upload bandwidth (bandwidth used for replying to request) for this node: $B = d * \frac{T}{\Delta} * S$ bytes per second. Such a model can keep with at most : $T = \frac{B*\Delta}{d*S}$ nodes. Assuming a connexion with a upload bandwidth limit of 512 kbit/s (optimistic upper limit of most broadband ADSL lines), and assuming $\Delta = 60s$ on average, the maximal supported number of nodes is slightly over 2600. Using a linked list of nodes, the first ones of which would be the one directly responsible for the key, would be useless: the memory constraints for each node would be lessened to $M = \frac{T*m}{1024*\frac{T}{\beta}} = \frac{m*\beta}{1024}$Kb, but the bandwidth load would stay the same for the head of the list.

5.3 Linked list with access through any segment

The memory requirements for our directory model are the same than in the linked list case. There is thus no theoretical limit to the number of nodes that the system can accommodate, memory-wise, because in an extreme situation each node may be responsible for as many or as few identifiers as we choose. We study the new bandwidth requirements, under worst-case, best-case and average-case situation.

Worst case and Best case scenarios In the worst case scenario, new nodes willing to act as solicitors arrive constantly while all the contributors are occupied: no request can be satisfied, and each one has to go through all the nodes of the list. But in this case, each solicitor will only generate one request before subscribing itself to the solicitors list, and entering a wait state. In most conditions, the arrival of new solicitors can be assumed to be spread over time. So even in this worst case, the system would probably stay efficient due to its use of a double directory structure, avoiding too many active polling for resources. In the best case scenario, the system is in the following state: no segment of the list ever becomes empty, due to a sufficiently large number of contributors. In this situation, each contributor request is answered immediately by the first segment receiving it. The bandwidth generated by the request is minimal, and the system can expand indefinitely.

Average case scenario The system stays between the two previous extremes. A number of contributors register as idle each second, and a number of solicitors send requests each second. We model the equilibrium case, with an average equal number of registration and request per second. We compute the total consumed bandwidth, in number of messages, to answer R simultaneous requests from a segmented list containing R identifiers. We assume that the R requests will be answered before any new contributor arrives. Let α be the number of segments in the chain: $\alpha = \frac{T}{\beta}$. A solicitor requesting a contributor identifier send its request to any node of the segmented chain, and this message is forwarded until an answer is found. The total number of messages produced depends on the density of contributors identifiers available in the chain. We consider a request reaching a randomized segment. To simplify the model we restrict ourselves to the case where $R \leq \beta$. For one identifier, the chance to be absent from the first segment reached by the request is $\psi_1 = \frac{\alpha-1}{\alpha}$. The chance to be absent from the segment i of the list, knowing that the $i-1$ previous segments are empty is $\psi_i = \frac{\alpha-i}{\alpha-i+1}$. The segment i is empty if all of the R identifiers are absent. Using the assumption $R \leq \beta$, there is no dependency between the location of an identifier and the locations of the others. The probability $\Psi_i(R)$ that the first i segments are empty is thus: $\Psi_i(R) = \psi_i^R = (\frac{\alpha-i}{\alpha-i+1})^R$. We compute now the chance to discover at least one identifier in the segment i after having traversed the first $i-1$ empty segments as : $\lambda_i(R) = 1 - \Psi_i(R) = 1 - (\frac{\alpha-i}{\alpha-i+1})^R$. Now we can compute the average number of requests messages generated by one request for a contributor identifier. The request have a probability $\lambda_1(R)$ to be satisfied by the first segment asked, thus generating only one request message (the initial request itself). If unsuccessful (with a probability $1 - \lambda_1(R)$), the request then have a probability $\lambda_2(R)$ to be satisfied by the second segment, thus generating two requests messages, the first request, and its retransmission from the first segment asked to the segment one. The probability for the request to be satisfied after $s \leq \alpha$ steps is thus $Satis(s) = \lambda_s(R) \times \prod_{\delta=1}^{s-1}(1 - \lambda_\delta(R))$. We assume than the distribution of identifiers stays homogeneous after each request has been satisfied. This assumption implies that there is a redistribution of the identifiers after each request is processed, which is not the case. We are overestimating the homogeneous spread of identifiers. We compute thus M, a lower bound of the average number of messages needed to answer **one request**, while there are R identifiers left in the chain.

$$M(R) = \sum_{s=1}^{\alpha} s \times Satis(s) \tag{1}$$

$$= \sum_{s=1}^{\alpha} \left(s \times \lambda_s(R) \times \prod_{\delta=1}^{s-1}(1 - \lambda_\delta(R)) \right) \tag{2}$$

$$= \sum_{s=1}^{\alpha} \left(s \times (1 - (\frac{\alpha - s}{\alpha - s + 1})^R) \times \prod_{\delta=1}^{s-1}(\frac{\alpha - \delta}{\alpha - \delta + 1})^R \right) \tag{3}$$

The *total* number of messages produced to answer R requests is thus $\sum_{i=1}^{i=R} M(i)$. We show on the figure 1 the predicted average number of messages needed to process one request, over a variable number R of requests for a list of R identifiers present in the list of contributors of 10 segments, and a maximal segment population of 20. This scenario is our equilibrium state scenario describe previously. Our model predicts an interesting property: the higher the number of requests received in an equilibrium state, the lower the average number of messages needed to answer each one is, given a fixed number of segments and a fixed segment population. We will show how the experimental results confirm this modelization.

6 Experimental results

We use a Network Simulator we built in Java, that allows us to parametrize the connexion speed between each pair of machines. We coded our DHT model on top of this simulator. All the computations at each node are simulated (time taken and results size are parametrized). This simulator accept "Send Messages" actions from our Peer-to-Peer system, and deliver messages to nodes mailboxes when they are available. Each connexion between two nodes takes a share out of the available bandwidth available between the two corresponding machines. There is no packet loss, congestion only lead to longer delay between machines. All nodes have the same computing power, all units take the same (parametrized) time to compute and all nodes have the same bandwidth, that of a high-speed ADSL connexion[5]. We show the experimental results obtained with our model, using a variable population of nodes, units and computation time and a variable mix of contributors versus solicitors. The upper-left figure 2 shows the acceleration factor with $T = 300$ participating nodes compared to the speed of one node alone, doing the same amount of work. One node only acts also a solicitor. Unit computation time is set to $\Delta = 3000$ seconds and a variable number of units U to compute are used. The upper right figure 2 shows the acceleration factor with a variable number of nodes, from which only one acts also a solicitor, $\Delta = 3000$ seconds and $U = 3000$. The middle left figure 2 shows the acceleration factor with $T = 300$ nodes, $U = 3000$ units and a variable unit-time. These results show that the acceleration factor is near the number of nodes when the unit computation time is high, which makes sense as the P2P infrastructure introduces delays of its own, which have more impact with short computation times tasks. Acceleration factor is higher then the number of units is a multiple of the number of nodes [6].

The middle-right figure 2 shows the bandwidth costs associated with our communication model. It shows only the P2P directory infrastructure overhead, independently of the size of the units related to a particular application. We show this cost in megabytes per node. On the middle-right figure 2, we see that the

[5] Download bandwidth of 3Mbits/s, upload bandwidth of 512 Kbits/s

[6] 301 units computed with 300 nodes will take Δ more seconds than 300 units computed on the same number of nodes

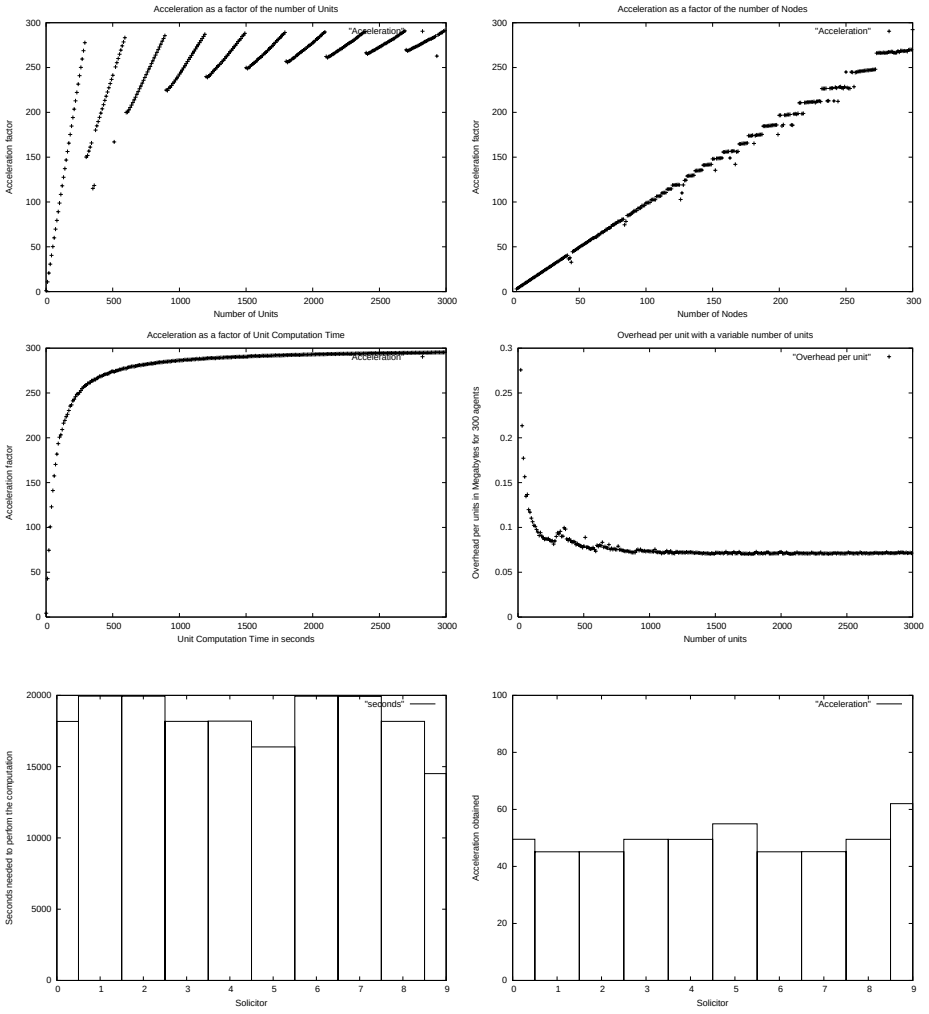

Fig. 2. Acceleration factor. Overhead of the system over a variable number of units. Fairness among solicitors.

simulation confirms our theoretical model: the higher the number of units, the lesser the communication cost per unit is, keeping the number of nodes the same. The cost decreases as long as there is more nodes than units, and then stabilizes, because solicitors stop emiting new requests and start using their own directory. We also ran experiments with a variable number of solicitors, to measure the fairness of the units allocation algorithm among solicitors, and results are shown on figure 2.

7 Conclusion and Future Works

We introduced a new model for the efficient distribution of a directory model over a DHT. We proposed a new Peer-to-Peer distributed computing model, based on our distributed directory model, providing bandwidth load-balancing for the frequently accessed directories entries. This model offers a complete equity for the participants, anyone having the possibility to be a contributor or a solicitor, and each solicitor being treated fairly. Network availability requirements are lessened, a permanent connexion is not required to collect all the returning results. While our current implementation runs on simulation, the real implementation will use the Java Virtual Machine as its security and code mobility layer. Our model is a generic platform, offering code to re-use for future distributed computing projects. Modelization indicate that the bandwidth cost per computed units is lowered when the number of units increases, which we verified in our implementation. We are now working on a real implementation of this model, offering a generic API to allow anyone to interface their code with our P2P network. We will implement research applications such as a distributed version of the Spin model-checker (see [5]), and a distributed version of a genetic algorithm library. A first possible improvement of our model is inter-segments communications. Each node will communicate with its two neighbors in two cases: node switch from empty state to not-empty state, and node switch from full state (maximal population reached) to not-full state. Propagation of information from node to node will allow them to know the closest node with free space, or with an identifier of an idle contributor. This allows quicker registration and request phases, at the cost of new communications messages between neighboring nodes. Another improvement will be to take into account meta-data for each node, such as the computing power of the node, in the request phase. This will allow each solicitor node to choose the best available contributor, and to implement a distributed scheduling algorithm. Pre-caching on the DHT of future units to distribute is also considered, when upload bandwidth is available. The bandwidth bottleneck that occurs when too many contributors return their results and ask for new units at the same time will be alleviated.

References

1. http://www.distributed.net/.
2. http://www.napster.com.

3. http://mersenne.org, 1996.
4. David P. Anderson. Public computing: Reconnecting people to science. In *Conference on Shared Knowledge and the Web*, November 2003.
5. Jiri Barnat, Lubos Brim, and Jitka Stribrna. Distributed ltl model-checking in spin. In *Lecture Notes in Computer Science*, pages 200–216, 2001.
6. I. Clarke, S. Miller, T. Hong, O. Sandberg, and B. Wiley. Protecting free expression online with freenet, 2002.
7. Vijay Pande et al. Atomistic protein folding simulations on the submillisecond timescale using worldwide distributed computing. *Biopolymers*, 2002.
8. Jaesun Han and Daeyeon Park. A lightweight personal grid using a supernode network. In IEEE, editor, *Third International Conference on Peer-to-Peer Computing*, Linköping, Sweden, 2003.
9. Stefan M. Larson, Christopher D. Snow, Michael Shirts, and Vijay S. Pande. Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology. In Richard Grant, editor, *Computational Genomics*. Horizon Press, 2002.
10. Laurence Loewe. evolution@home: Experiences with work units that span more than 7 orders of magnitude in computational complexity. In *2nd International Workshop on Global and Peer-to-Peer Computing on Large Scale Distributed Systems*, pages 425–431. IEEE Computer Society, 2002.
11. P. Maymounkov and D. Mazieres. Kademlia: A peer to peer information system based on the xor metric. In *Proceedings of IPTPS02, Cambridge, USA, March 2002*, 2002.
12. Moni Naor and Udi Wieder. Novel architecture for p2p applications: the continuous-discrete approach. In ACM, editor, *SPAA*, San Diego, June 2003.
13. Andrew Odlyzko. Zeros of the riemann zeta function: Conjectures and computations. In *Foundations of Computational Mathematics conference*, Minnesota, 2002.
14. University of Oxford, the Rutherford Appleton Lab, and The Open University. http://www.climateprediction.net, 2003.
15. OpenSource. http://www.gnutella.com.
16. Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.
17. Larson SM, Garg A, Desjarlais JR, and Pande VS. Increased detection of structural templates using alignments of designed sequences. In *Proteins: Structure, Function, and Genetics*, pages 390–396, 2003.
18. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.
19. Berkeley University of California. http://setiathome.berkeley.edu/.
20. Berkeley University of California. http://boinc.berkeley.edu.
21. Bojan Zagrovic, Christopher D. Snow, Michael R. Shirts, and Vijay S. Pande. Simulation of folding of a small alpha-helical protein in atomistic detail using worldwide distributed computing. *Journal of Molecular Biology*, 2002.
22. B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.