Published in Lecture notes in computer science 2346, 153-168, 2002
which should be used for any reference to this work

Optimisation of Distributed Communities Using Cooperative Strategies

Saiho Yuen[1], Peter Kropf[1], and Gilbert Babin[2]

[1] Informatique et recherche opérationnelle
Université de Montréal, Montréal, Québec, Canada
{yuensaih,kropf}@iro.umontreal.ca
[2] Technologies de l'information
HEC – Montréal, Montréal, Québec, Canada
Gilbert.Babin@hec.ca

Abstract. The complex structure of the Web requires decentralised, adaptive mechanisms efficiently providing access to local and global capacities. To facilitate the development of such mechanisms, it seems reasonable to build clusters of machines with similar structures and interests. In such a manner, communities of machines can be built. In a community, every machine contributes to the overall success through a division of management work and a respective collaboration. This article presents and analyses experimental results for algorithms optimising service response times in a community. It extends previously published results on the *Wanderer* optimisation algorithm; we describe variations of the *Wanderer* and present simulation results of these variations.

1 Introduction

The Internet represents a large pool of resources. However, these resources are difficult to access. Furthermore, the sheer size of the Internet makes it difficult, if not outright impossible to keep track of all these resources. One promising solution approach is to manage the information about resources using self-organizing and adaptive information bases [5].

There are currently many projects underway which use this approach [1, 2, 4, 3]. One such project is the Web Operating System (WOS) [3, 6] which is built to support communities of client and server machines. These machines do not only share a common communication context, but also sets of similar parameters and interests. The WOS is an open middleware solution allowing for software services to be distributed over the Internet. The WOS infrastructure provides the tools to search for and prepare all the necessary resources that fulfil the desired characteristics for a service request (e.g., performance, storage, etc.).

In the WOS context, a community, or WOSnet, is a set of WOS nodes requesting or providing a specific service. This implies that there exists a dichotomy: within a community, nodes are either servers or clients. By client, we mean nodes requesting the service offered in a community. By server, we mean nodes providing the service offered in a community. Needless to say, WOS nodes may participate in many community and may therefore be both server and client.

A WOSnet is dynamically formed; nodes may dynamically join and leave a community. The WOSnet evolves through the location and execution activities performed by the different WOS nodes. The knowledge about the WOSnet, accumulated through these activities, is stored by the nodes in *warehouses*. These warehouses are the node's knowledge center. For example, a service location request will leave its result in the warehouses of the nodes visited. In general, service location requests are processed using message chains, transmitted in parallel [7].

In [8] we presented and analysed experimental results for two algorithms optimising service response times in a community, namely the *Whip* and *Wanderer* algorithms. This paper extend this previous work; it presents and analyzes variations of the *Wanderer* algorithm. The simulation environment is briefly described in Section 2. We also define the notion of network community optimisation and we describe the *Wanderer* algorithm, on which the results presented herein are based. We also explain in that section why variations are necessary. In Section 3, we introduce the different variations developed and analyse them, based on simulation results. Finally, Section 4 concludes the paper with a discussion of the approach.

2 Simulating Communities

We developed a tool to simulate the behavior of a community [8]. The service provided by the community is a simple data transfer service. Basically, client nodes request a data transfer and server nodes provide the requested data. The simulation tool represents a WOSnet as a randomly generated graph of nodes placed in a 2D grid. Each node has a warehouse containing the list of server nodes it knows, along with a measure of the quality of that node. The quality of the service is measured by the response time for a service request of client c to server s, denoted $t(c, s)$. For simplicity, we assume $t(c, s) = 1/b(c, s)$, where $b(c, s)$ is the bandwidth between client c and server s. We estimate $b(c, s)$ by using the euclidian distance between client c and server s in the simulation plane.

A simulation is divided in cycles. Within each cycle, the number of requests made by a client increases linearly. As the number of requests increases, servers have more difficulty in fulfilling all the requests received, and may therefore reject requests.

At any point during the simulation, a client may become unsatisfied with the response time of its current server or may even see his request rejected. When this occurs, the client will seek a better server to fulfil his requests. This is what we call a *community optimisation*. The goal of the optimisation is to minimise the response time for each client. To achieve this goal, the optimisation process reorganises the community by selecting a more suitable server for that client. It does so by recursively searching for server nodes the client does not know yet and by inspecting the warehouse entries at each server visited. During this process the warehouses searched are updated, thus dynamically restructuring the virtual network or community.

Many different parameters can be controlled by the simulation tool:

- the proportion of nodes acting as servers,
- the maximum number of entries in the warehouse (i.e., the length of the list of servers),

– the number of cycles in a simulation,
– the duration of a cycle, calculated in *units of time* (ut).

2.1 Evaluation of Optimisation Algorithms

Different algorithms may be defined to perform the optimisation process. In order to compare these algorithms, we use different measures.

– The effectiveness of the algorithm (E) is the ratio

$$E = r_{\text{avail}}/r_{\text{min}} * 100,$$

where r_{avail} is the average response time of requests, if all the clients of the network launch exactly one request simultaneously and the clients are using the server with the largest bandwidth in the network, and r_{min} is the average response time of requests, if all the clients of the network launch exactly one request simultaneously and the clients are using the server with the largest bandwidth that the algorithm was able to find. The effectiveness measures the distance between the configuration obtained with the algorithm and the optimal attainable configuration.
– The convergence time of the algorithm (t_c) is the time required for reaching r_{min}. It is measured in ut.

2.2 The Wanderer Algorithm

The *Wanderer* algorithm [6] is based on the transmission of a message, named a wanderer, from node to node. A wanderer w is a tuple (c, k, l, h), where c is the list of clients for which the optimisation is performed, k is the list of servers visited (k for knowledge), l is the identifier of the node on which the wanderer currently is located (l for location), and h is the hopcount of that message[1]. Given that S is the set of server node, C is the set of client nodes, and $\perp$ is an undefined identifier, we can formally define the set of all possible wanderers W as

$$W \subseteq \mathcal{P}(C) \times \mathcal{P}(S) \times (S \cup \perp) \times \mathbb{N}.$$

Therefore, $c \in \mathcal{P}(C)$, $k \in \mathcal{P}(S)$, $l \in S \cup \perp$, and $h \in \mathbb{N}$.

The *Wanderer* algorithm is launched by a client every time it wishes to find a better server. It proceeds in three distinct stages: initialisation, search, and update stages. At the *initialisation stage*, the wanderer is initialised by the client: c contains the client launching the wanderer, k contains the list of servers known by the client, l contains the identifier of the first server to visit, and h is set to 0. At the *search stage*, the wanderer is sent to the node identified by l. Once there, k is updated with new information about servers, found on the current node. The value of l is set to the identifier of the next node to visit, if any. The value of h is incremented. It then proceeds with the search stage, until all nodes in k are visited. At the *update stage*, the client selects the node in k with the shortest response time. Formally, we have:

[1] The hopcount h is defined for completeness but is not used herein.

Let c_0: client launching the wanderer.
Let c_w: the client list c of wanderer w.
Let k_w: the knowledge list k of wanderer w.
Let l_w: the location l of wanderer w.
Let h_w: the hopcount h of wanderer w.
Let k.*append*(k'): appends knowledge k' to knowledge k.
Let n.*next*(k): based on the information available in the warehouse of node $n \in S \cup C$ and on k, returns the next node to visit, if any; returns $\perp$ otherwise.
Let c.*update*(k): update the warehouse of client c using knowledge k.

A-Initialisation stage

Let $c_w \leftarrow \{c_0\}$
Let $k_w \leftarrow \emptyset$
Let $l_w \leftarrow c_0$.*next*(k_w)
Let $h_w \leftarrow 0$

B-Search stage

While $l_w \neq \perp$
    "Send" wanderer w to node l_w
    $k_w \leftarrow k_w$.*append*(content of the local warehouse)
    $l_w \leftarrow l_w$.*next*(k_w)
    $h_w \leftarrow h_w + 1$
End-While

C-Update stage

For each $c \in c_w$
    c.*update*(k_w)
End-For

A wanderer has been defined as a tuple or data structure upon which the above algorithm is executed. Since the algorithm includes communication, i.e. sending the tuple to another node where the same algorithm is executed, one might also define the tuple together with the algorithm as an entity that migrates from node to node. At each node, the tuple is updated using that nodes local information. Therefore, the wanderer tuple could, together with the code of the algorithm, be implemented as mobile agent. In the following sections, we use sometimes rather the notion of agent for reasons of simplicity of explanation.

3 Variations of the Wanderer Algorithm

From previous simulations [8], we observed that the *Wanderer* algorithm is very effective, even when varying all the simulation parameters. It turns out, however, that its convergence time changes greatly depending on these same parameters. The algorithm is also very demanding in terms of computational resources; since a wanderer is

sent from node to node, it requires resources on every node it visits. The quantity of resources required is proportional to the number of wanderers currently in the network. In a larger network, that number can be extremely high. As a consequence, the *Wanderer* algorithm may create network congestion.

The main difference between the *Wanderer* algorithm and its variations is cooperation. The main goal of the cooperation is to eliminate the problems created by the original *Wanderer* algorithm described in the previous section, such as the computational resources required, network congestion, etc. We introduced two main strategies to alleviate these effects: sharing and merging.

Sharing. We define *sharing* as an exchange of knowledge in k. The process of sharing happens during the search stage (at the end of the loop) when there is more than one wanderer on the same node. Whenever two wanderers meet at a node, they will exchange information. Each wanderer will exchange at most once on each node visited. Formally, we can define the process of sharing as follows:

Let w.*sharing*(k,k'): the sharing function of wanderer w.

If $(l_w = l_{w'})$
$k_w \leftarrow w.sharing(k_w, k_{w'})$
$k_{w'} \leftarrow w'.sharing(k_w, k_{w'})$
End-If

As shown in Figure 1, wanderers W1 and W8, coming from different nodes, meet on node S4. Before choosing the next node to visit, they share their information. Before sharing, k_{W1}={S3, S4} and k_{W8}={S2, S4}. After sharing, both wanderers have exactly the same contents, that is k_{W1}=k_{W8}={S2, S3, S4}, and both wanderers continue their search process with that knowledge.

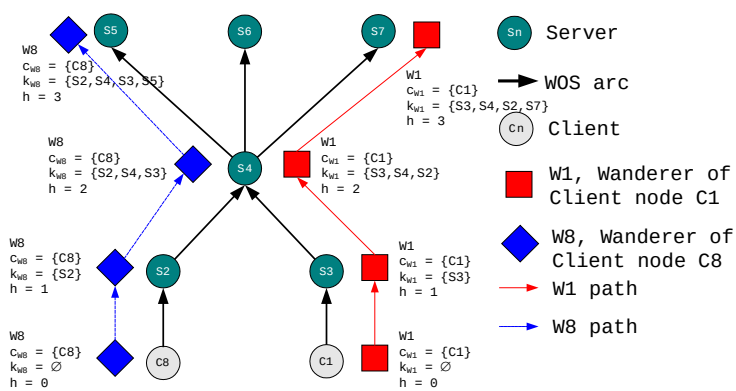

Fig. 1. Wanderers Sharing Information

Merging We define *merging* as an operation where one wanderer takes all the knowledge of another wanderer before destroying that wanderer. As is the case with sharing, merging also happens at the end of the search stage loop, when there is more than one wanderer on the same node. Furthermore, merging is not automatic. Different criteria may be used to determine whether merging should occur or not. Each wanderer will merge at most once on each visited node. Formally, we can define the process of merging as follows:

Let w.*merging*(): the merging function of wanderer w.
Let n.*criterion*(w, w'): a boolean function indicating if wanderers w and w' can merge on node n.

If $[(l_w = l_{w'}) \wedge l_w.criterion(w, w')]$
$k_w \leftarrow w.merging(k_w, k_{w'})$
$h_w \leftarrow \max(h_w, h_{w'})$
$W \leftarrow W \setminus w'$
End-If

As shown in Figure 2, wanderers W1 and W8, coming from different nodes, meet on node S4. Before choosing the next node to visit, they verify if it is possible to merge. Wanderer W1 initiates the negotiation and both wanderers agree to merge. This corresponds to the evaluation of the merge condition described above. Thus W8 is merged with W1. Before merging, k_{W1}={S3, S4} and c_{W1}={C1}, while k_{W8}={S2, S4} and c_{W8} = {C8}. After merging, wanderer W1 remains, with k_{W1} = {S2, S3, S4} and c_{W1} = {C1, C8}, and wanderer W8 is destroyed.

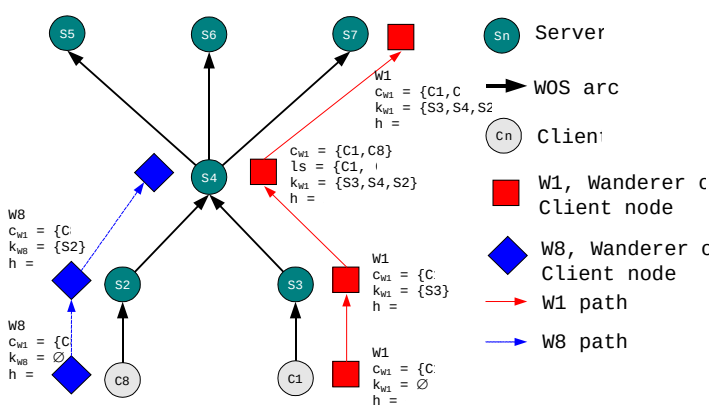

Fig. 2. Wanderers Merging

Based on those two strategies, different variations of the *Wanderer* algorithm are defined. These variations can be broken into three categories:

1. the share category, where wanderers share their knowledge,

2. the merge category, where wanderers merge with the other wanderers,
3. the mixed category, where wanderers either share their knowledge or merge with other wanderers depending on different parameters.

We tested each variation under the same conditions and network configuration as the original *Wanderer* algorithm: the network comprises 2,000 nodes, of which 2 % are servers. We ran from 5 to 20 simulations for each experimental parameter. The results from the different variations are compared to the results obtained from the original *Wanderer* algorithm, allowing us to assess the effect of sharing and merging.

3.1 Share category

There is only one variation in this category: the *Wanderer-Share*. In this algorithm, the wanderer uses the sharing strategy, as described in Sect. 3, i.e., at the end of the search stage loop, it tries to share its knowledge with other wanderers present on the current node. The results from the simulation are presented in Table 1 and can be summarized as follows:

Table 1. Comparison of the Wanderer and the Wanderer-Share Algorithms (2000 nodes; 2 % of server nodes)

	E (%)	t_c (ut)	t_l (ut)	K_g (%)	K_e (%)	N_s	K_s
Wanderer	99.81	440	82.11	100	0	–	–
W-share	99.61	380	38.46	27.67	72.33	3.17	18.25

- The efficiency (E) of the *Wanderer-Share* is almost identical to that of the *Wanderer* algorithm. This may be explained by the fact that both algorithms do a complete search through the network. Furthermore, the *Wanderer-Share* does not try to reduce the number of wanderers in the network.
- The convergence time (t_c) of the *Wanderer-Share* algorithm (380 ut) is 14 % lower than that of the *Wanderer* algorithm (440 ut). Recall that the client node performs its warehouse update only once the wanderer has finished its network walk-through. By sharing information, the search takes less time since the wanderers do not need to visit the whole network.
- Most of the information about the servers is exchanged, rather than being gathered. An average of 72 % of the wanderer's knowledge is exchanged (K_e), while only 28 % is gathered (K_g).
- As a direct consequence of sharing, the average life span of the wanderers (t_l) has decreased by 53 %, from 82.11 ut to 38.46 ut.
- A wanderer will share information 3.17 times on average (N_s). Each time it shares information it obtains knowledge about 18.25 nodes (K_s).

Sharing creates an unexpected problem that we call the *search path convergence* problem. When analyzing the path used by the wanderers, we realize that each time

two wanderers share data, they tend to visit the same node after the sharing. The reason for this convergence is that all wanderers use the same selection process: they use the data gathered to decide which node to visit next. After sharing their knowledge, the internal data of both wanderers is exactly the same. Thus, with the same selection process and the same input, they end up making the same decision. As a result, there is a high probability of network congestion. Network congestion occurs earlier with a large network, because the number of wanderers increases proportionally with the number of client nodes, which in turn increases the number of shares.

3.2 Merge category

We have developed three variations in the Merge category, the *Wanderer-Merge-Random*, *Wanderer-Merge-Select*, and *Wanderer-Merge-Node* algorithms. In these algorithms, the wanderer uses the merging strategy, as described in Sect. 3, These variations differ in the way wanderers decide to merge with other wanderers, i.e., the $n.criterion(w, w')$ function. In the *Wanderer-Merge-Random* algorithm, the wanderer agent decides to merge randomly, using a binomial probability. In the *Wanderer-Merge-Select* algorithm, the wanderer's decision to merge is made according to the similarity with other wanderers. The similarity refers to number of nodes that two wanderers have in common; if two wanderers have visited the same nodes, their similarity would be 100 %. In the *Wanderer-Merge-Node* algorithm, the wanderer agent decides to merge based on the number of wanderer agents on the current node. In this case, merging occurs only if there is more than a certain number of wanderer agents on a node.

Wanderer-Merge-Random Algorithm. For this variation, we looked at how the algorithm's behavior changes when changing the probability to merge. We tested with probabilities varying from 25 % to 100 %. Table 2 presents the results, which can be summarized as follows:

Table 2. Comparison of the Wanderer and the Wanderer-Merge-Random Algorithms (2000 nodes; 2 % of server nodes)

	E (%)	t_c (ut)	t_l (ut)	N_w	K_g (%)	K_e (%)	N_m	K_m
Wanderer	99.81	440	82.11	27.90	100	0	–	–
W-Merge-R (25 %)	99.84	440	57.86	17.62	74.88	25.12	4.80	20.09
W-Merge-R (50 %)	99.92	420	54.67	10.97	75.22	24.78	17.91	19.82
W-Merge-R (75 %)	99.96	400	56.27	5.81	73.07	26.93	29.05	21.54
W-Merge-R (100 %)	99.91	400	59.03	5.44	70.78	29.22	36.99	23.38

- The efficiency (E) of the *Wanderer-Merge-Random* algorithm does not differ from *Wanderer* algorithm. This result follows from the complete search of the network.
- The convergence time (t_c) of the *Wanderer-Merge-Random* algorithm is about 20 to 40 ut faster than the *Wanderer*. The increase seems to depend on the probability to merge.

- The number of wanderers in the network (N_w) for the *Wanderer-Merge-Random* is much smaller than for the *Wanderer*. Furthermore, it decreases significantly as the probability to merge increases. This is a logical outcome of merging.
- When varying the probability to merge, we observe that the average life of wanderers (t_l) decreases until the probability reaches 50 % and then increases again.

This last observation may be explained by the fact that when the probability is less than 50 %, there is a higher probability that the knowledge obtained through merging is not yet known, while that probability decreases when the probability to merge increases. This is confirmed by the knowledge exchanged when merging (K_e). Furthermore, as more merges occur, more time is spent updating client nodes in c_w.

Finally, if two wanderers have already visited the same nodes, merging does not accelerate the search process. This effect is shown in Table 2 where we can see that the number of merges (N_m) increases significantly with the probability to merge, but the information exchanged at each merge (K_m) remains relatively stable.

Wanderer-Merge-Select Algorithm For this variation, we looked at how the algorithm's behavior changes when changing the degree of similarity required to merge. We tested with similarity degrees varying from 25 % to 100 %. We obtained the following results (Table 3):

Table 3. Comparison of the Wanderer and the Wanderer-Merge-Select Algorithms (2000 nodes; 2 % of server nodes)

	E (%)	t_c (ut)	t_l (ut)	N_w	K_g (%)	K_e (%)	N_m	K_m
Wanderer	99.81	440	82.11	27.90	100	0	–	–
W-Merge-S (25 %)	99.01	440	60.81	4.95	78.05	21.95	41.14	17.56
W-Merge-S (50 %)	98.80	420	57.75	6.87	84.44	15.56	23.50	12.45
W-Merge-S (75 %)	99.65	420	57.85	16.67	84.26	15.74	7.41	12.59
W-Merge-S (100 %)	99.66	420	80.28	27.03	91.66	8.34	0.57	3.80

- The efficiency (E) of the *Wanderer-Merge-Select* algorithm is similar to the *Wanderer* algorithm.
- The convergence time (t_c) is constant but slightly faster (20 ut) than the *Wanderer*.
- As we increase the degree of similarity required to merge, the number of wanderers in the network (N_w) increases significantly, while the number of merges (N_m) decreases.
- We also observe that the amount of data exchanged (K_e) decreases as the degree of similarity increases. Indeed, when two wanderers merge and the degree of similarity required is 50 % or more, the knowledge gained by merging is necessarily lower than 50 %.
- The higher the required degree of similarity, the more difficult it is to merge.
- The life span of the wanderers (t_l) in this algorithm follows a pattern similar to that of the *Wanderer-Merge-Random* algorithm.

When the required degree of similarity is high, merging occurs in two situations:

1. Merging occurs at a very early stage of the search, when wanderers still have very little data. It is then easier to find other wanderers with similar content.
2. Merging occurs much later during the search stage. Wanderers need to visit more nodes before merging, otherwise they do not reach the required similarity degree.

However, neither situation helps in accelerating the search, because the knowledge exchanged by merging is not significant enough to have an impact on the convergence time. In the extreme case where the similarity is very high (around 100 %), the wanderers cannot even cooperate. Therefore, in order to have reasonable results with the algorithm, we need for the required similarity degree to be low. In addition, the time to evaluate similarity increases when the required similarity increases.

Wanderer-Merge-Node Algorithm For this variation, we looked at how the algorithm's behavior changes when changing the number of wanderers required before merging can occur. We tested with the number of wanderers needed varying from 5 to 80, which is from 10 % to 160 % of the "maximum" number of wanderers in the network. Since the distribution of wanderers follows a normal distribution, there is no maximum value. In most cases however (99 % of the time; see Fig. 3), this number is less than 50. For simplicity, we fix that maximum to 50 wanderers.

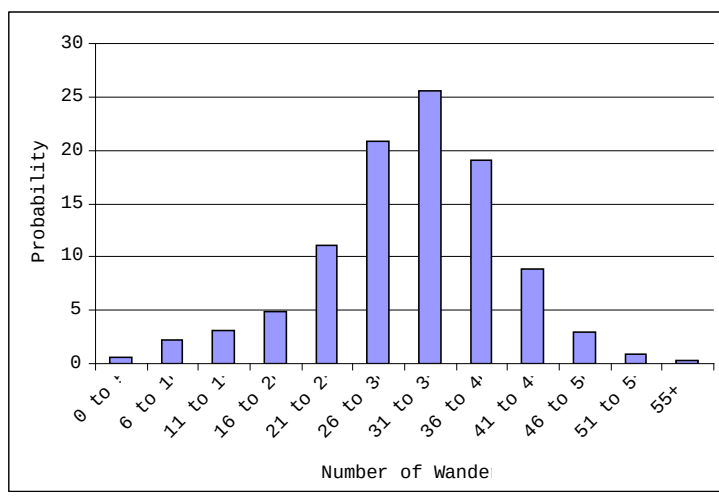

Fig. 3. Probability Distribution of the Number of Wanderers in the Network (2000 nodes)

Results are illustrated in Table 4 and can be summarized as follows:

– The efficiency (E) of the algorithm is again very high.
– As the two other variations of this category, this algorithm has a significant effect on the population of wanderers in the network (N_w).

Table 4. Comparison of the Wanderer and the Wanderer-Merge-Node Algorithms (2000 nodes; 2 % of server nodes)

	E (%)	t_c (ut)	t_l (ut)	N_w	K_g (%)	K_e (%)	N_m	K_m
Wanderer	99.81	440	82.11	27.90	100	0	–	–
W-Merge-N (5)	99.67	460	53.63	5.65	68.73	31.27	25.42	25.02
W-Merge-N (10)	99.23	400	49.58	7.06	67.94	32.06	12.88	25.65
W-Merge-N (20)	97.34	380	52.59	11.99	65.52	34.48	4.32	27.59
W-Merge-N (30)	99.60	440	59.39	17.39	62.99	37.01	1.97	29.61
W-Merge-N (40)	99.76	420	70.24	23.02	70.37	29.63	0.75	23.70
W-Merge-N (50)	99.87	400	79.50	26.71	91.66	8.34	0.19	6.67
W-Merge-N (60)	99.89	420	80.54	27.34	99.25	0.75	0.02	0.60
W-Merge-N (80)	99.87	400	81.43	27.62	100	0	0	0

When the number of wanderers required to merge reaches 100 %, there is almost no chance that merging will occur. Thus, without merging, the algorithm behaves exactly like the original *Wanderer* algorithm. Therefore, the required number of wanderers to merge should always be less than 100 % of the maximum number of wanderers in the network.

The most significant effect of using the number of wanderers on the node as a parameter of merging is that the number of wanderers (N_w) is efficiently controlled when the required number of nodes for merging is small. However, the reduction is not efficient when that number is too small. For instance, when the required number of wanderers is 5, the reduction is about the same as when that required number is 10. Furthermore, when that required number is too small, the average life span (t_l) and the convergence time (t_c) also increase. The reason is that since merges happen too often, the time saved by merging is not sufficient to compensate the extra time spent at updating the starting nodes.

We also would like to point out that when the required number of nodes for merging is too small, merging becomes unrealistic for the server node and the network. Since the quantity of wanderers produced by client nodes is so large, it cannot be reduced even when a large number of merges occur. In addition, the merging process consumes a lot of computational resources on the server node. Therefore, when we have a large number of merges, the server node may spend too much resources for merging, instead of answering clients' requests.

The performance of the algorithm strongly depends on the required number of wanderers to merge. This number should be between 30 % and 50 % of the maximum number of wanderers in the network. These levels should yield good performance. A value smaller than 30 % or higher than 50 % would yield a degradation of performance.

3.3 Mixed category

The goal of the mixed algorithms is to combine the advantages of sharing and merging. Whenever there is more than one wanderer on a node, either merging or sharing will occur, based on a selection criterion. Formally, algorithms of the mixed category work as follows:

If $[(l_w = l_{w'}) \wedge l_w.criterion(w, w')]$
$k_w \leftarrow w.merging(k_w, k_{w'})$
$h_w \leftarrow \max(h_w, h_{w'})$
$W \leftarrow W \setminus w'$
Else-If $(l_w = l_{w'})$
$k_w \leftarrow w.sharing(k_w, k_{w'})$
$k_{w'} \leftarrow w'.sharing(k_w, k_{w'})$
End-If

We have developed three different algorithms in this category, which are the cross-product of the sharing and merging variations: the *Wanderer-Mixed-Random* algorithm, the *Wanderer-Mixed-Select* algorithm, and the *Wanderer-Mixed-Node* algorithm. As is the case with algorithms of the merge category, these algorithms are differentiated by the way they decide which strategy (sharing or merging) to apply (i.e., evaluating function $n.criterion(w, w')$).

The *Wanderer-Mixed-Random* algorithm selects sharing or merging using a binomial random variable. The *Wanderer-Mixed-Select* algorithm will choose to share or merge based on the similarity of wanderers. Finally, The *Wanderer-Merge-Node* algorithm makes that decision based on the number of wanderers currently on the node.

Wanderer-Mixed-Random Algorithm For this combination, we looked how the algorithm's behavior changes when changing the ratio of the strategy chosen. The algorithm applies merging with probability p and sharing with probability $1 - p$. The tests used probabilities p ranging from 20 % to 80 %. The results obtained are presented in Table 5 and can be summarized as follows:

- The efficiency (E) of the *Wanderer-Mixed-Random Algorithm* is similar to the previous variations of the *Wanderer* algorithm.
- The convergence time (t_c) is comparable to the one obtained by the *Wanderer-Share* algorithm. It thus appears that the sharing effect, where wanderers do not need to visit all the nodes, dominates with regard to the convergence time.
- Increasing the merge probability leaves the amount of data obtained through merging (K_m) fairly constant. On the other hand, the information gained by sharing (K_s) increases with increasing merge probability p. This is due to the increased elimination of redundant information by merging.
- The number of wanderers in the network (N_w) is governed by the merging component of the strategy and therefore lower than for the *Wanderer* and *Wanderer-share* algorithms.

With this algorithm, we observe that the combination of the two strategies yields good results since both strategies are focused on distinct area. Sharing focuses on the exchange of knowledge and will maximize knowledge exchanged (K_e). Merging focuses on the control of the population (N_w); it will minimize the risk of network congestion and reduce computation resources needed by the algorithm.

Table 5. Comparison of the Wanderer and the Wanderer-Mixed-Random Algorithms (2000 nodes; 2 % of server nodes)

	E (%)	t_c (ut)	t_l (ut)	N_w	K_g (%)	K_e (%)	N_m	K_m	N_s	K_s
Wanderer	99.81	440	82.11	27.90	100	0	–	–	–	–
W-Share	99.61	380	38.46	22.67	27.68	72.31	–	–	3.17	18.25
W-Merge-R (50 %)	99.92	420	54.67	10.97	75.22	24.78	17.91	19.82	–	–
W-Mixed-R (20 %)	99.78	400	49.56	7.76	85.60	14.40	7.41	12.50	0.43	3.55
W-Mixed-R (40 %)	99.25	400	45.38	10.70	79.13	20.87	3.81	13.32	0.89	6.79
W-Mixed-R (50 %)	99.10	380	44.56	12.52	73.93	26.07	2.82	13.96	1.15	8.43
W-Mixed-R (60 %)	99.21	400	42.56	14.70	66.91	33.09	1.89	14.68	1.47	10.02
W-Mixed-R (80 %)	99.38	400	40.14	18.95	46.19	53.81	0.67	11.08	2.17	13.74

The Wanderer-Mixed-Select Algorithm For this combination, we looked at how the algorithm's behavior changes when changing the ratio of merging and sharing. The degree of similarity to decide upon merging varies from 25 % to 100 %. If there is no merging, the sharing strategy applies. The results presented in Table 6 are summarized as follows:

- The efficiency (E) is of the same order as for the other strategies.
- The time to live of wanderers (t_l) is closer to the results obtained with the *Wanderer-Share* algorithm and thus better than in the case of the *Wanderer-Merge-Select* algorithm.
- Sharing increases the knowledge exchanged (K_e) and the life time of the wanderers (t_l) decreases accordingly.

Table 6. Comparison of the Wanderer and the Wanderer-Mixed-Select Algorithms (2000 nodes; 2 % of server nodes)

	E (%)	t_c (ut)	t_l (ut)	N_w	K_g (%)	K_e (%)	N_m	K_m	N_s	K_s
Wanderer	99.81	440	82.11	27.90	100	0	–	–	–	–
W-Share	99.61	380	38.46	22.67	27.68	72.31	–	–	3.17	18.25
W-Mixed-S (25 %)	99.60	400	46.37	9.21	77.05	22.95	4.90	11.29	0.84	8.41
W-Mixed-S (50 %)	98.03	420	42.66	15.35	50.32	49.68	1.89	9.93	1.82	16.38
W-Mixed-S (75 %)	99.66	400	40.83	19.58	42.05	57.95	0.94	7.90	2.24	17.18
W-Mixed-S (100 %)	99.84	380	39.13	22.88	38.33	61.67	0.17	0.77	2.80	17.35

This algorithm deals with the "late merge problem" of the *Wanderer-Merge-Select* algorithm, where it becomes more difficult to merge as the similarity degree increases. When two wanderers meet on the same node and do not have sufficient similarity, they share their data in the case of the mixed strategy. After exchanging information, they have the same knowledge and will thus tend to choose the same next node. In such a case, however, these two wanderers meet again on the next node, but this time their knowledge is identical. Therefore a merge occurs. This means that wanderers always end up merging. After the first sharing, the wanderer's knowledge is nearly the same,

so a merge occurs sooner or later. Therefore, the threshold must be high in order to achive good results. However, it should not be too high, otherwise the algorithm will be reduced to the *Wanderer-Share* algorithm. In some way, this algorithm also corrects the search path convergence problem: since only the wanderers who have a high degree of similarity will merge, the algorithm controls the wanderer population (N_w) by eliminating the "useless doubles."

The Wanderer-Mixed-Node Algorithm For this case of combining merging and sharing, we varied the number of wanderers required for merging. Again, this number was varied from 5 to 80 wanderers, or from 10 % to 160 % of the "maximum" number of wanderers on the network. If the number of the wanderers on a node is higher than the number required, then wanderers will merge, otherwise they will share knowledge. Table 7 summarizes the results obtained in this simulation.

Table 7. Comparison of the Wanderer and the Wanderer-Mixed-Node Algorithms (2000 nodes; 2 % of server nodes)

	E (%)	t_c (ut)	t_l (ut)	N_w	K_g (%)	K_e (%)	N_m	K_m	N_s	K_s
Wanderer	99.81	440	82.11	27.90	100	0	–	–	–	–
W-Share	99.61	380	38.46	22.67	27.68	72.31	–	–	3.17	18.25
W-Mixed-N (10)	99.21	380	44.70	9.38	72.43	27.57	3.52	14.98	1.03	6.87
W-Mixed-N (20)	99.57	360	39.48	15.27	46.00	54.00	1.04	17.86	2.00	12.67
W-Mixed-N (30)	99.49	400	37.70	19.43	38.70	61.30	0.32	6.31	2.70	15.28
W-Mixed-N (40)	99.73	400	38.41	22.02	29.53	70.47	0.06	1.20	3.05	18.09
W-Mixed-N (50)	99.77	400	38.30	22.91	27.48	72.52	0.01	0.12	3.16	18.32
W-Mixed-N (60)	99.69	400	38.30	22.91	27.45	72.55	<0.01	<0.01	3.18	18.25
W-Mixed-N (70)	99.83	400	38.29	23.20	26.80	73.20	0	0	3.15	18.59

- The efficiency (E) is similar to all the other algorithm variations.
- The number of wanderers (N_w) is slightly lower than for the *Wanderer-Share* algorithm.
- The same observations with regard to the choice of the number of wanderers required for merging (compared to the "maximum number" of nodes) yield as for the *Wanderer-Merge-Node* algorithm. If that number is too high, no merge occurs and the algorithm behaves like the *Wanderer-Share*.

The application of the *Wanderer-Merge-Node* and *Wanderer-Mixed-Node* algorithm requires the "maximum number" of nodes in the network to be known in order to determine the correct number of nodes for taking the merging decision. However, in a completely decentralized system as the WOS, this information, the "maximum number" of nodes, is not available. This suggests to introduce a mechanism for guessing or approximating that number locally at each node, based on the local information gathered over time.

4 Discussion

This work is related to the optimisation of communities. In previous experiments on the *Wanderer* algorithm [8], we showed that although this algorithm could easily adapt to changes in the network and showed high and constant efficiency, it required large amounts of computational and communication resources. Furthermore, the life span of wanderer agents is very large. In this paper, we presented variations of the *Wanderer* algorithm and analyzed whether these variations resolved the problems observed with the original *Wanderer*. In order to address these limitations, we opted for cooperation among wanderer agents. We have focused on two strategies of cooperation: sharing and merging. Both strategies have their advantages and disadvantages. Sharing increases the gain of knowledge, but may create network congestion because of the path convergence problem. Merging decreases the population of wanderers of the network, but only shows small increases of performances; in some cases, performances may even decrease. From our observations, we conclude that the most appropriate solution is a combination of both strategies with a careful selection of thresholds between sharing and merging. The results obtained with the mixed *Wanderer* algorithms indicate that the resource, the performance and congestion problems can be satisfactorily resolved.

Aknowledgements

We are grateful to Dr. Habil. Herwig Unger, from University of Rostock (Germany), for is involvement in this project.

References

1. L.N. Foner. YENTA - A Multi-Agent Referral System for Matchmaking. In *First International Conference on Autonomous Agents (Agents '97)*, Marina del Rey, CA, 1997.
2. K. Kramer, N. Minar, and P. Maes. Mobile software agents for dynamic routing. *Mobile Computing and Communications Review*, 3(2), 1999.
3. P. Kropf, H. Unger, and G. Babin. WOS: an Internet Computing Environment. In *Workshop on Ubiquitous Computing*, PACT 2000 (IEEE International Conference on Parallel Architectures and Compilation Techniques), pages 14–22, Philadelphia, PA, 2000.
4. V. Menkov, D.J. Neu, and Q. Shi. Ant world: A collaborative web search tool. In P. Kropf et al., editor, *Distributed Communities on the Web (DCW 2000)*, LNCS 1830, pages 13–22, Quebec, Canada, 2000. Springer.
5. D. Milojicic. Operating systems - now and in the future. *IEEE Concurrency*, 7(1):12–21, 1999.
6. H. Unger. Distributed Resource Location Management in the Web Operating System. In SCS A. Tentner, editor, *High Performance Computing 2000 (ASTC)*, pages 213–218, Washington, DC, 2000.
7. Herwig Unger, Peter Kropf, Gilbert Babin, and Thomas Böhme. Simulation of search and distribution methods for jobs in a Web operating system (WOS^{TM}). In A. Tentner, editor, *High Performance Computing Symposium '98*, Boston, MA, USA, April 1998. SCS International.
8. Herwig Unger, Saiho Yuen, Peter Kropf, and Gilbert Babin. Simulation of communication of nodes in a wide area distributed system. In *Eurosim 2001 Congress*, Delft, The Netherlands, June 2001. Published on CD-ROM.