# An Optimization of the Buddy Model for Securing Mobile Agents

Yueh-Hua Lee and Hyunyoung Lee
Department of Computer Science
University of Denver
Denver, CO 80208, U.S.A.
{ylee7, hlee}@cs.du.edu

**Abstract**

This paper proposes an improvement over a multi-agent system (MAS) security model, known as the "buddy model". The buddy model partitions the mobile agents in a system into several security groups. In a group, an agent is protected by its neighbors, which are its "buddies". The model achieves the security requirement by periodically exchanging tokens within a group. We propose "Join", "Leave", and "Accept" algorithms to collect nearby mobile agents into a group; moreover, our "Merge" and "Split" algorithms maintain a reasonable size and diameter of a group where the diameter is the distance between two farthest mobile agents in a group.

## 1 Introduction

This paper proposes a scheme to improve a security model, namely the "buddy model", for an agent community. Mobile Agents are autonomous programs that can migrate from hosts to hosts [2, 7]. A mobile agent can decide when and where to move. When migrating, a mobile agent brings the implementation, data, and execution state with it. Mobile agent technology has continuously developed in recent years. The concept of agent community results from the maturity of mobile agent technology. The agent community is a MAS with a specific community goal [8]. A community integrates different mobile agents that perform different functions in order to achieve one common goal. An agent community is a dynamic entity: Mobile agents join and leave their community frequently. Furthermore, an agent community may consist of several sub-communities with different secondary community goals; several communities may form a super-community that has a primary community goal. It is natural that an agent community is designed in a hierarchical way; however, hierarchical implementation brings vulnerabilities. Attackers can easily target the mobile agents in the top level of the hierarchy or create malicious mobile agents that assume the identities of administrative mobile agents in order to ruin the whole community.

The "buddy" model is proposed by Page et al. [8, 9]. The term "buddies" refers to the neighbors of a mobile agent in a pre-assigned group. In the model, the mobile agents within a group generate tokens and send the tokens to their buddies periodically. By using the tokens, they protect buddies and monitor the health condition of buddies. Page et al. argue that the model avoids the vulnerabilities of a hierarchical scheme for mobile agent security: Because each mobile agent performs an identical role in the security function, it is hard for an attacker to find and attack a central coordinator.

This paper explores an improvement of the "buddy" model. The *optimization issue* should be taken into account when the model is applied to a large-scale scenario. If the mobile agents move far away from one another, the network suffers from their frequent token delivery. We try to reduce the network traffic by grouping nearby agents together dynamically. When a mobile agent migrates to a far away host, the agent leaves the old group and joins a new nearby group. We also note that the size of a group dynamically changes in time. Our improvement scheme provides means to merge two small groups into a larger group and to split a large group into two or several smaller groups. The scheme can also be applied to other MAS if the system can be divided into groups and requires frequent communication between group members. A system gains better performance because the group members that need to communicate are geographically near to one another.

There are five algorithms in our optimization approach. The "Join", "Leave", and "Accept" algorithms collect nearby mobile agents into a group. Our "Merge" and "Split" algorithms maintain reasonable size and diameter of a group by using a temporary central coordinator. The following sections are organized as follows: Section 2 describes related works. Section 3 explains our system model. Section 4 describes the five algorithms of our optimization scheme. Section 5 contains the analysis of the algorithms. Section 6 concludes the paper.

## 2 Related Works

Communication is a prime factor of MAS performance. There have been a great deal of research trying to optimize the communication cost for MAS in different scenarios. Zhang et al. [13] examine the proactive communication in multi-agent teamwork scenario and introduce a dynamic decision-theoretic approach. The concept of multi-agent teamwork is similar to that of an agent community. They focus on analyzing information production and support of proactive communication. The agents adopt different strategies, which are Silence, ActiveAsk, and Wait, to optimize the communication utility. Goldman et al. [3] observe the communication of decentralized cooperative MAS. They develop a theoretical model and an approximation technique of the decentralized control of the communication optimization problem. The model helps to analyze the trade-off between the cost of information exchange and the value of the information. Helin [4] discusses the communication of mobile agents in wireless networks. There are two policies of communication, which are migrating or sending request to the host. Helin proposes a model to find the optimal decision among them. Jim [6] studies a MAS in which all agents communicate simultaneously. He shows that such MAS is equivalent to a Mealy machine with finite state while the states are determined by the concatenation of the strings in the agent communication language. In addition, increasing the language improves the performance of the MAS. Yang et al. [12] analyze four factors that affect the performance of MAS: communication mode, semantics, frequency, and agent migration sequence. They propose a mathematical model to describe the communication tasks and provide means to optimize the cost.

The mobile agent security issue is receiving more and more attention in recent years. To migrate and operate in an untrustworthy network is the nature of mobile agents. A general security infrastructure for all potential threats has not been found because of the heterogeneity of the execution environments such as the Internet. Several researches try to find specific solutions for specific security threats; others try to classify security threats and integrate different countermeasures. Jansen classifies the security threats into four categories: an agent attacking an agent platform, an agent platform attacking an agent, an agent attacking another agent, and other entities attacking the agent system [5]. Similarly, Bierman classifies the malicious host threats into four categories namely integrity attacks, availability refusal, confidentiality attacks, and authentication risks [1]. They propose systematical classification and definition of threats in order to help developing a general security scheme. However, a perfect solution has not been achieved.

## 3 System Model

We assume an asynchronous network with arbitrary topology. A security group is represented by an undirected graph $S(V, E)$. Each vertex represents a mobile agent and each edge represents buddy relationship. In other words, if two mobile agents are buddies, there is an edge that connects them. We assume that the mobile agents have ability to compute the distance between any two mobile agents. The distance is measured by the number of hops between them. We define $D$ as the maximum diameter of a group. The size of a group is between $L$ and $U$ where $L$ is the lower bound and $U$ is the upper bound. We choose $L < U/2$ to avoid iterative merge and split.

When a mobile agent broadcasts a request, it waits for a threshold $T$ of time. $T < 2\delta$ where $\delta$ is the maximum message delay for a communication channel whose length is $D$. If a reply message cannot arrive

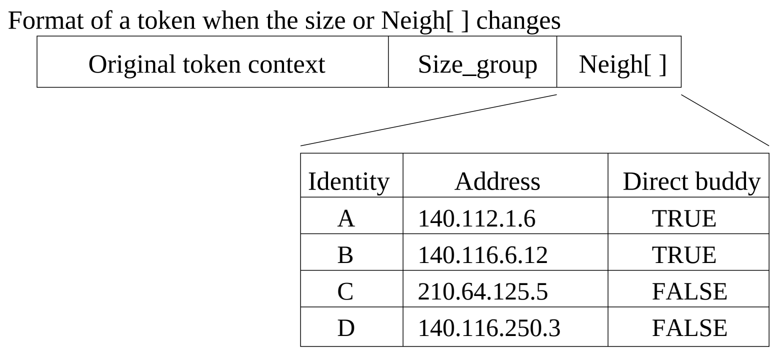
$\boldsymbol{S(V, E)}$: a graph that represents a security group.
$\boldsymbol{p, p_i, q, q_i, u, u_i, v, v_i, w, w_i}$, where $i = 1, 2, \ldots$: mobile agent.
$\boldsymbol{G, G_i, H, H_i}$, where $i = 1, 2, \ldots$: security groups.
$\boldsymbol{C}$: temporary coordinator.
$\boldsymbol{n, m, m_i}$: the number of mobile agents (size) of the agent community or in a security group.
$\boldsymbol{d}$: the number of hops between two mobile agents.
$\boldsymbol{r}$: the diameter of a security group. The distance between two farthest mobile agents in a group.
$\boldsymbol{D}$: (system parameter) the maximum allowable diameter.
$\boldsymbol{T}$: (system parameter) the time threshold for an agent to wait.
$\boldsymbol{T'}$: the number of local clock ticks that can represent $T$.
$\boldsymbol{U, L}$: (system parameter) the upper bound and lower bound of the size of a group.
$\boldsymbol{\delta}$: the maximum message delay for a communication channel of length $D$.

Figure 1: Notations

within $T$ time period, we assume that the source of the reply message is unable to respond or is far away from the mobile agent that sends the broadcast message. However, we have to synchronize mobile agents that dock at different machines. The threshold $T$ is measured by clock ticks. When a mobile agent moves to a new host, it sends a synchronization request to its home platform (the home platform is the origin of the agent). The home platform will send two replies, which are "start" and "end", and the time between the two replies is $T$. The mobile agent counts the number of local clock ticks, which is $T'$, between receiving "start" and "end". Using this approach, $T$ is evaluated by $T'$ and mobile agents on different hosts are synchronized. The system parameters $T$, $D$, $L$, and $U$ are known to every agent.

In the "Merge" and "Split" algorithms, we use a temporary coordinator to lead the merge and split processes. We assume that a randomized leader election algorithm can choose a unique leader at random. Therefore, there is still no way for an attacker to find and attack the temporary coordinator. Ramanathan et al. propose a randomized leader election algorithm that works in arbitrary network topology and requires only $O(n)$ messages where $n$ is the number of processes [10]. The algorithm is further discussed in Section 4.

Each mobile agent knows the addresses and identities of its direct buddies and buddies of buddies. The information is stored in an array, *Neigh[]*. Each entry in *Neigh[]* has three fields. The first field records the identity; the second field records the address; the third field indicates the agent is a direct buddy or buddy of a buddy. If a mobile agent senses that the size of the group or *Neigh[]* changes, it appends the information to its token. Upon receiving the token, the buddies modify their local information according to the token.


Format of a token when the size or Neigh[ ] changes

| | | |
|---|---|---|
| Original token context | Size_group | Neigh[ ] |

| Identity | Address | Direct buddy |
|---|---|---|
| A | 140.112.1.6 | TRUE |
| B | 140.116.6.12 | TRUE |
| C | 210.64.125.5 | FALSE |
| D | 140.116.250.3 | FALSE |

Figure 2: Illustration of a token

**community_identifier, agent_identifier, group_identifier**: Unique ID issued by home platform.
**size_group, q_size_group**: The number of mobile agents of a group.
**my_location, q_location**: The address of a mobile agent.
**Neigh[], q_Neigh[]**: Array that stores identities, addresses, and other information of the buddies.
**All_Address[]**: Array that stores the addresses of all mobile agents in a group.
**parent, child**: The identities of parent and child nodes.
**flag_split_candidate**: If a mobile agent is a candidate for split, the flag is set to TRUE; otherwise, it is FALSE. *Flag_split_candidate* is TRUE if *size_subtree* is greater than or equal to $L$.
**num_split_candidate**: The number of candidates (or TRUE bits of *flag_split_candidate*) in the subtree.
**size_subtree**: The size of the subtree rooted at a particular node.
**split_factor**: The parameter of split.
$\langle$**tok**$\rangle$: The token of the group.

Figure 3: Variables

## 4 The Algorithms

Our scheme consists of the five algorithms: "Join", "Accept", "Leave", "Merge", and "Split". If two variables have the same name, the term "this" refers to the local variable of a mobile agent.

**Join, Accept, and Leave.** When a mobile agent $p$ moves far away from its buddies and decides to join a new group, $p$ broadcasts a "join request" and waits for $T$ time to receive "accept messages" form $q$. The "accept message" contains the size of $q$'s group and the distance between $p$ and $q$. When the timeout occurs, $p$ chooses the nearest agent $q$. If there are two or more agents that have the same distance and the distance is the smallest, $p$ chooses the agent that resides in a smaller group. If the sizes of the groups are also the same, $p$ chooses one of them arbitrarily. Then, $p$ sends a confirmation, to the nearest agent $q$ and receives *Neigh[]* of $q$. After all, $p$ joins the group, which $q$ resides in, and leaves the old group. If no accept message is received, $p$ ceases its work and returns to the home platform.

In the Accept algorithm, an agent $p$ computes the distance from $p$ to $q$ when it receives a join request from $q$. If the distance is smaller than $D$, $p$ sends an accept message to $q$. When $p$ receives a confirmation, $q$ becomes a buddy of $p$. At the same time, $p$ appends *size_group* and *Neigh[]* in the token. When $p$'s buddies receive the token, they update their *size_group* and *Neigh[]* accordingly.

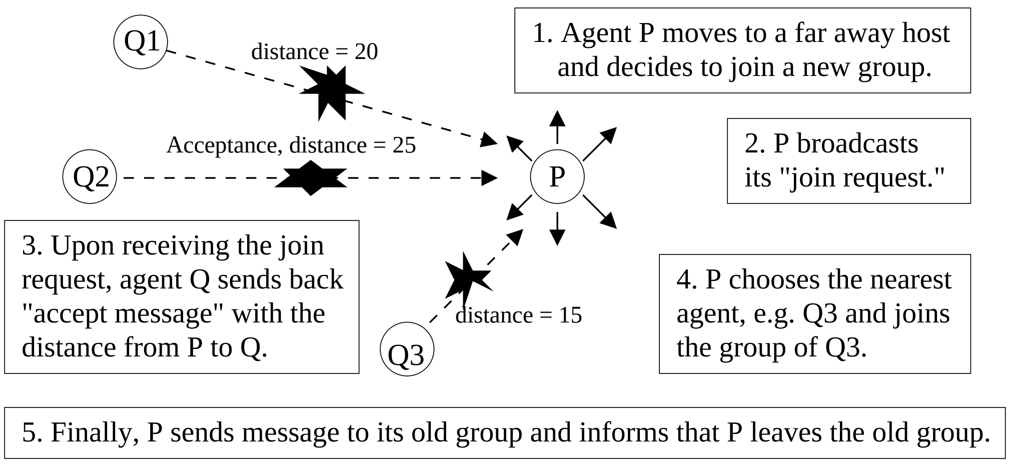
**Merge.** When a group becomes smaller than $L$, it should merge with a nearby group. In order to have efficient communication among the buddies, we want to minimize the total path lengths of the new group. There are three phases in the Merge algorithm. In phase one, the mobile agents elect a temporary coordinator $C$ at random. In phase two, each mobile agent $p$ in the group $H$ finds an agent $u$ that belongs to another group $G_i$ and is nearest to $p$. We use the same approach in the Join algorithm to find such agent $u$: $p$ broadcasts a message and waits for $T$ time. Other agents who receive the message report the distance and *size_group*. If there are two or more mobile agents that report the same distance and the distance is the smallest, the one with smaller *size_group* is chosen. If *size_group* are also the same, $p$ chooses arbitrarily. Then, $p$ finds the agent $w$, which is the farthest agent of $p$ in $G_i$. Agent $p$ queries all agents in $G_i$ to find such $w$. Then, $p$ computes the distance $d$ between $p$ and $w$. We call $d$ the "longest distance", which indicates the diameter of the new group after merging. Next, $p$ reports $d$, $G_i$ and the size of $G_i$ to the coordinator $C$.

In phase three, $C$ chooses the group $G$ among $G_i$ according to $d$ and leads the merge process. Again, if there are more than two groups that have the same $d$ and such $d$ is the smallest, $C$ compare the sizes of the groups and chooses the one with smaller size. If the tie situation happens again, $C$ chooses arbitrarily. The merge process is to gather addresses of all agents in $H$ and $G$, assign buddies for all agent, and update local variables of all agents. Application developers decide how to assign buddies. The subroutine Gather_Address

**Algorithm 1** Join, Accept and Leave Algorithms (for agent $p$)

```
Algorithm Join
(The distance between any buddy $> D$)
    $distance$, $size$ := infinity; $location$ := NULL
    Broadcast "join request" $\langle$ $my\_location$, $community\_identifier$ $\rangle$
    Set timer (timer := $T$)
    Receive "accept message" $\langle$ $q\_distance$, $q\_location$, $q\_size\_group$ $\rangle$
    If ($q\_distance < distance$) OR ($q\_distance = distance$ AND $q\_size\_group < size$)
        then $location$ := $q\_location$; $size$ := $q\_size\_group$
    /* Timeout occurs */
    If $p$ receives no accept message then return to home platform
    Send confirmation $\langle$ $my\_location$, $agent\_identifier$ $\rangle$ to the nearest agent $q$
    Receive $\langle$ $q\_Neigh[]$ $\rangle$ from $q$
    Send "leave message" $\langle$ $Neigh[]$ $\rangle$ to the old buddies
    Update $Neigh[]$, $size\_group$
Algorithm Accept
(receive join request $\langle$ $q\_location$, $community\_identifier$ $\rangle$ from $q$ )
    if ($community\_identifier = this.community\_identifier$) AND (distance between $p$ and $q \leq D$)
        then send "accept message" $\langle$ $distance$, $my\_location$, $size\_group$ $\rangle$ to $q$
    receive confirmation $\langle$ $q\_location$, $identifier$ $\rangle$ from $q$ )
    send $\langle$ $Neigh[]$ $\rangle$ to $q$
    Update $Neigh[]$, $size\_group$
Algorithm Leave
(receive leave message from $q$)
    Update $Neigh[]$, $size\_group$
```

**Illustration of the Join and Accept**


Q1
distance = 20
1. Agent P moves to a far away host and decides to join a new group.
Acceptance, distance = 25
Q2
P
2. P broadcasts its "join request."
3. Upon receiving the join request, agent Q sends back "accept message" with the distance from P to Q.
distance = 15
Q3
4. P chooses the nearest agent, e.g. Q3 and joins the group of Q3.
5. Finally, P sends message to its old group and informs that P leaves the old group.

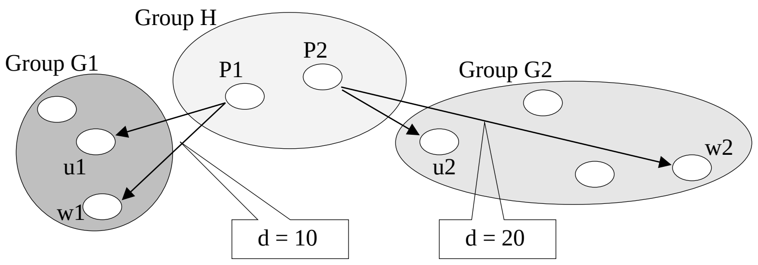


Figure 4: Illustration of the Merge. $L = 3$ and $U = 8$. The group $H$ runs the Merge algorithm. Agent $u1(u2)$, which is not in $H$, is the nearest agent of $p1(p2)$. Agent $w1(w2)$ is the farthest agent of $p1(p2)$ in $G1(G2)$. Because the distance between $p1$ and $w1$ is smaller than that of $p2$ and $w2$, Group $H$ decides to merge with $G1$.

uses a depth-first search to gather addresses of all mobile agents. When gathering addresses, the subroutine uses an array, *All_Address[]*. Each entry in *All_Address[]* contains two fields: The first field is the identity of mobile agents and the second field is the address.

**Algorithm 2** Merge Algorithm (for agent $p$)

```
Merge Algorithm
($size\_group < L$)
    Phase 1: Vote for a temporary coordinator $C$ at random
    Phase 2:
        Find the nearest mobile agent $u$ that is in a different group $G_i$
        Find the farthest node $w$ in $G_i$
        Compute the distance $d$ between $p$ and $w$
        Report $d$, $G_i$, and size of $G_i$ to $C$
    Phase 3:
        If $p = C$ then
            Integer $i := 1$ // initialize All_Address[]
            Create array All_Address[]
            All_Address[0].identity := this.agent_identifier
            All_Address[0].identity := this.my_location
            Choose the group with the shortest $d$
            Gather_Address(Neigh[])
            Request Neigh[] from $w$ denote as it w_Neigh[]
            Gather_Address(w_Neigh[])
            Assign buddies for each agent
            Command all agents to update their local variables
```

**Split.** The Split algorithm consists of three phases. In phase one, we randomly elect a temporary coordinator $C$ and build a spanning tree over the mobile agents where $C$ is the root. Each node represents a mobile agent and has one parent variable and several child variables. Parent/child stores the identity of parent/child node respectively. The subroutine Build_Tree is a breadth-first algorithm. Every mobile agent sends message ⟨*your_parent, identity*⟩ to its direct buddies. The buddies reply message ⟨*your_son, identity*⟩ or ⟨*not_your_son*⟩ to indicate whether the buddy is a child of the mobile agent that sends ⟨*your_parent, identity*⟩.

In phase two, we compute *size_subtree*, *split_factor*, *flag_split_candidate*, and *num_split_candidate*. Computing *split_factor* and *flag_split_candidate* requires no external messages. On the contrary, computing

**Algorithm 3** Subroutine Gather_Address

```
Gather_Address(input_array[]) // depth-first search
    If all entries in input_array[] is in All_Address[] then return
    For each entry in input_array[]
        If the entry is not in All_Address[]
            then copy the entry to All_Address[i]; $i := i + 1$
                Request Neigh[] from All_Address[i].address denoted as q_Neigh[]
                Gather_Address (q_Neigh[])
```

*size_subtree* and *num_split_candidate* at an agent $p$ requires the *num_split_candidate* and *size_subtree* of all children of $p$. The approach is straightforward. If the node is a leaf node, the *size_subtree* is one; otherwise, the *size_subtree* is the sum of *size_subtree* of all children plus one. If the node is not a candidate, the *num_split_candidate* is zero; otherwise, the *num_split_candidate* is the sum of *num_split_candidate* of all children. Of course, each node other than the root should report *size_subtree* and *num_split_candidate* to the parents.

A split candidate (or candidate) is a node whose size of subtree is greater than or equal to $L$. Obviously, the coordinator $C$ must be a candidate. The algorithm splits the group by cutting edges of the tree. If there is only one candidate, which is the coordinator $C$, we preserve the edge to the child whose *size_subtree* is the smallest. If a tie situation occurs, $C$ chooses arbitrarily. All other edges between $C$ and its children are cut. If the new groups are too small, they run the Merge algorithm. If there are two candidates, we cut the edge between them. If there are three or more candidates, the algorithm chooses according to *split_factor*, where $split\_factor = |size\_subtree - (size\_group/2)|$. It makes the sizes of two new groups roughly half of the original group. In phase three, $C$ gathers addresses, assigns buddies for all mobile agents and update their local variables.

**Failure and Malicious Agents Handling.** A mobile agent stops responding when it crashes or behaves abnormally when it is compromised by a malicious entity. In [8], a mobile agent returns to the home platform when any of the buddies fails. In [9], a mobile agent reports the failure situation to the home platform. The home platform calculates the "confidence factor" of the group. If the confidence factor becomes smaller than a pre-determined threshold, the home platform will call back all mobile agents or take other evasive actions. Since the buddy model deals with the failed or malicious agents periodically, we assume that the number of failed or malicious agents is $f$ and $\frac{f}{size\_group} < \epsilon$. The system developer can control the proportion $\epsilon$ using the threshold of the confidence factor.

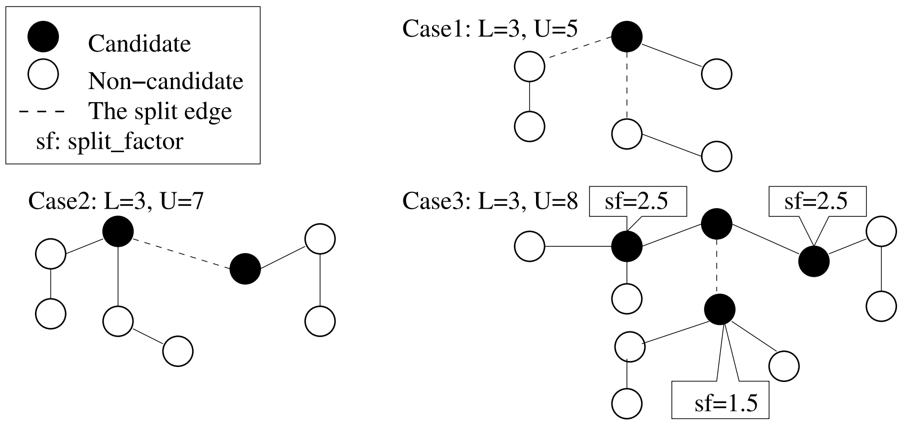
In our scheme, an agent waits for $T$ time after sending a request. The agent ignores all replies after the timeout occurs. In the Merge and Split algorithm, we need to gather all addresses using subroutine, Gather_Address. Because we maintain the information of direct buddies and buddies of buddies in *Neigh[]*, Gather_Address fails only when all direct buddies and buddies of buddies fail. The redundant information makes the algorithm robust.

The buddy model avoids hierarchical vulnerabilities because all mobile agents act an identical role with respect to security function. It is undeniable that the temporary coordinator in the Merge and Split algorithms is more important than other mobile agents. We exploit randomized leader election algorithms to hide the coordinator from the outsiders. However, a malicious mobile agent inside a group can blindly claims itself as the winner of the election and become the coordinator. We employ a variation of Ramanathan's algorithm [10] to solve the problem: in the competition, each contender must act for another randomly-chosen contender. When a contender wins the election, the leader is the mobile agent that the winner represents. Under this condition, a malicious mobile agent becomes the coordinator only if the mobile agent that acts for it wins the election and the malicious mobile agent has no control of its representer. Thus, the algorithm

**Algorithm 4** Split Algorithm (for agent $p$) and Subroutine Build_Tree

```
Split Algorithm
($size\_group > U$)
Phase 1: Vote for a temporary coordinator $C$ (at random)
            integer $parent, child := -1$
            Build_Tree()
            If $p = C$ then $parent := this.agent\_identifier$; Create array $All\_Address[]$
Phase 2: Compute $size\_subtree$
            If $size\_subtree < L$ then $flag\_split\_candidate$ := FALSE else $flag\_split\_candidate$ := TRUE
            Compute $split\_factor, num\_split\_candidate$
            If $p = C$ then
                If $num\_split\_candidate = 1$
                    then preserve the edge between $C$ and the child whose $size\_subtree$ is the smallest
                        cut edges between $p$ other children
                else if $num\_split\_candidate = 2$
                    then cut the edge between the nodes whose flags are TRUE
                else
                    find the node $q$ with smallest $split\_factor$
                    cut the edge between $q$ and the parent of $q$
Phase 3: If $p = C$ then Gather_Address($Neigh[]$); Assign buddies to each agent;
                        Command all nodes to update their local variables
Build_Tree()
($parent \neq -1$)
    for each entry in $this.Neigh[]$ send message $\langle$ your_parent, $this.agent\_identifier$ $\rangle$ to $this.Neigh[].address$
(Receive message $\langle$ your_parent, $parent\_identity$ $\rangle$ )
    If $parent = -1$
        then $parent := parent\_identity$; Send message $\langle$ your_son, $this.agent\_identifier$ $\rangle$ back
        else Send message $\langle$ not_your_son $\rangle$ back // The address of the parent can be found in $Neigh[]$
(Receive message $\langle$ your_son, $child\_identity$ $\rangle$ )
    If $child\_identity$ does not equal to any existing integer variable $child$
        then generate new integer variable child; the new variable child := $child\_identity$
```

**Illustration of the Split**



can resist the malicious mobile agent.

## 5 Analysis

We evaluate the algorithms by the message complexity and the communication overhead. The message complexity is the total number of messages that are needed to complete the algorithm. The communication overhead is the total length of edges in the graph $S(V, E)$ where $V$ represents agents and $E$ represents buddy relationship. As mentioned in Section 3, if two mobile agents are buddies, there exist an edge that connects them. The length of the edge is the number of hops along the shortest route between the two end points. Of course, the routing algorithm chooses the shortest path to transfer tokens. Therefore, we regard the edges as the communication channel to exchange tokens in the group $S(V, E)$ and use the total length of edges to represent the communication overhead for token exchange. Obviously, the topology has significant impact on the communication overhead. We consider it to be an implementation issue and leave the flexibility to choose the topology to the application developers. This section evaluates the communication overhead in the worst-case topology, which is a clique. Thus, $|E| = O(|V|^2)$.

**Theorem 1** *Let $n$ be the number of mobile agents in an agent community. Let $G$ be a group of $m-1$ agents, that has diameter $r$. Suppose that agent $p$ joins the group $G$. Let $x$ be the distance between $p$ and its nearest agent $q$ in $G$. The message complexity of the Join algorithm is $O(n)$ and the communication overhead is $O(m^2(x + r))$.*

**Proof.** When $p$ decides to join $G$, it broadcasts a "join request", receives at most $n$ "accept messages", sends one "confirmation message" and one "leave message". Therefore, the Join algorithm requires one broadcast message and $O(n)$ point-to-point messages.

The new diameter is at most $x+r$. The communication overhead is $O(m^2(x+r))$. We choose the nearest group to minimize $x$. ■

**Theorem 2** *Let $n$ be the number of mobile agents in an agent community. The message complexity of the Merge algorithm of two groups of size $m_1$ and $m_2$ is $O(nm_1 + 3m_2)$ and the communication overhead of the merged group is $O(rm^2)$, where $m = m_1 + m_2$.*

**Proof.** Suppose that a group $H$ runs the Merge algorithm. Each agent $p_i$ in $H$ executes the second phase of the Merge algorithm to find group $G_i$. Finally, the coordinator $C$ chooses the group $G$ among $G_i$. Let $r$ be the diameter of the merged group. The communication overhead is $O(rm^2)$. The algorithm chooses the smallest $r$ in the second phase.

Let $m_1$ and $m_2$ be the number of mobile agents of $H$ and $G$ respectively. Phase one of the Merge algorithm involves a leader election problem. Reference [10] shows that the leader election requires $O(m_1)$ messages. In phase two, each agent $p_i$ in $H$ broadcasts one message and receives at most $n$ replies to find its nearest neighbor $u_i$ that resides in group $G_i$. Then, $p_i$ sends at most $U$ queries and receives at most $U$ replies to find the farthest agent $w_i$ in $G_i$. After that, $p_i$ reports to the coordinator $C$ and $C$ chooses the group $G$. Because there are $m_1$ mobile agents in $H$, this phase requires $m_1$ broadcast messages and $O(m_1(n + 2U))$ point-to-point messages.

In phase 3, the subroutine Gather_Address requests *Neigh[]* from all agents in the merged group. It requires $2m$ messages. After that, $C$ assigns buddies for all agents in the new group by sending update information to them, which requires $m - 1$ messages. The total number of point-to-point messages is $O(m_1 + m_1(n + 2U) + 3m - 1)$. The number of broadcast messages is $m_1$. Because $m_1 < L < m_2 < U \ll n$, $O(m_1 + m_1(n+2U) + 3m-1) = O(m_1 + m_1n + 3m_1 + 3m_2 - 1) = O((n+4)m_1 + 3m_2 - 1) = O(nm_1 + 3m_2)$. ■

**Theorem 3** *Suppose that a group $G$ of $m$ mobile agents decides to split into two smaller groups, $G_1$ and $G_2$. Let $m_1$ and $m_2$ be the number of mobile agents in $G_1$ and $G_2$. Let $r$ be the diameter of $G$. The message* *complexity of the Split algorithm is $O(m^2)$ and the total communication overhead of the two new groups is $O(r({m_1}^2 + {m_2}^2))$.*

**Proof.** Phase one builds a spanning tree and elects a leader. For the subroutine Build_Tree, each agent sends messages $\langle$*your_parent*$\rangle$ to at most $m$ mobile agents in *Neigh[]*. After that, it receives either $\langle$*your_parent*$\rangle$ or $\langle$*not_your_son*$\rangle$. Therefore, the message complexity for each agent is $O(m)$ and the message complexity for all agents in $G$ is $O(m^2)$. Besides, we need $O(m)$ messages to elect a leader.

In phase two, each node reports *size_subtree* and *num_split_candidate* to its parent. This step requires $O(2(m-1))$ messages because there are $m-1$ edges in a tree of size $m$. Note that the number of edges in $G$ does not equal to the number of edges in the spanning tree.

In phase three, the coordinator $C$ collects addresses, assigns buddies and updates local variables of all mobile agents of $G$. The subroutine Gather_Address requires at most $2m$ messages. To assign and update local variables, the coordinator sends $m-1$ messages. The total number of point-to-point messages required in the Split algorithm is $O((m^2) + 2(m-1) + (3m-1)) = O(m^2)$. The total communication overhead of $G_1$ and $G_2$ is $O(r({m_1}^2 + {m_2}^2))$. ■

## 6 Conclusions

We proposed a means to improve the communication overhead of the buddy model by collecting nearby mobile agents together in a dynamic way. Obviously, both size and diameter of a security group affect the performance of the mobile agents. The main advantage of our approach is that the system is easy to maintain by keeping a reasonable size and diameter of the security groups.

## References

[1] E. Bierman, T. Pretoria, and E. Cloete. "Classification of Malicious Host Threats in Mobile Agent Computing". *Proc. of South Africa Institute of Computer Scientist and Information Technology* (SAICSIT) pages 141–148, 2002.

[2] B. Brewington, R. Gray, K. Moizumi, D. Kotz, G. Cybenko, and D. Rus. "Mobile Agents in Distributed Information Retrieval". *Intelligent Information Agents*, chapter 15, pages 355–395, Springer-Verlag, 1999.

[3] C. Goldman and S. Ziberstein. "Optimizing information exchange in cooperative multi-agent systems". *Proc. of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 137–144, 2003.

[4] H. Helin, H. Laamanen, and K. Raatikainen. "Mobile Agent Communication in Wireless Networks". *European Wireless'99/ITG'99*, pages 211–216, 1999.

[5] W. Jansen and T. Karygiannis. "Mobile Agent Security". NIST Special Publication 800-19, 1999.

[6] K. Jim, and C. Giles. "How Communication Can Improve the Performance of Multi-Agent Systems". *Proc. of the Fifth International Conference on Autonomous Agents*, pages 584–591, 2001.

[7] D. Kotz and R. Gray. "Mobile Agent and the Future of the Internet". *ACM Operating Systems Review* 33(3), pages 7–13, 1999.

[8] J. Page, A. Zaslavsky, and M. Indrawan. "A buddy model of security for mobile agent communities operating in pervasive scenarios". *Proc. of the Second Australasian Information Security Workshop* (AISW2004), 32:17–25, 2004.

[9] J. Page, A. Zaslavsky, and M. Indrawan. "Countering security vulnerabilities using a shared security buddy model schema in mobile agent communities". *Proc. of the First International Workshop on Safety and Security in Multi-Agent Systems* (SASEMAS 2004), pages 85–101, 2004.

[10] M. Ramanathan, R. Ferreira, S. Jagannathan, and A. Grama. "Randomized Leader Election". Purdue University Technical Report, http://www.cs.purdue.edu/homes/rmk/pubs/leader1.pdf, 2004.

[11] G. Tel. *Introduction to Distributed Algorithms*, 2nd edition, chapter 7, 2000.

[12] B. Yang, D. Liu, and K. Yang. "Communication Performance Optimization for Mobile Agent System". *Proc. of the First International Conference on Machine Learning and Cybernetics*, 2002.

[13] Y. Zhang, R. Volz, T. Ioerger, and J. Yen. "A Decision-Theoretic Approach for Designing Proactive Communication in Multi-Agent Teamwork". *ACM Symposium on Applied Computing*, pages 64–71, 2004.