# SAMSON: SMART ADDRESS MANAGEMENT IN SELF-ORGANIZING NETWORKS

Kristóf Fodor, Dániel Krupp, Gergely Biczók, János L. Gerevich, Krisztián Sugár
*Department of Telecommunications and Media Informatics*
*Budapest University of Technology and Economics*
*Magyar Tudósok krt. 2., 1117 Budapest, Hungary*
{fodork,dkrupp,biczok}@tmit.bme.hu, {gj198,sk335}@hszk.bme.hu

**Abstract** In this paper we present the SAMSON (Smart Address Management in Self-Organizing Networks) protocol, which is a simple and effective solution for assigning unique addresses to nodes of a MANET. Our protocol reduces the configuration time of arriving nodes by optimizing the number of configuration messages and the number of hops the configuration messages have to pass through. Furthermore, SAMSON handles the merger and partitioning of networks efficiently.

**Keywords:** ad hoc network, auto address configuration, distributed algorithm

## 1. Introduction

Nodes in a mobile ad hoc network (MANET) usually use short range wireless connections for connecting to each other, thus a data packet may have to travel through several hops to reach the destination party. Topology in these so-called multihop networks is rapidly changing due to the movement of the nodes, moreover, the set of devices participating in the communication is not permanent. These MANETs are independent from any pre-installed network infrastructure and may be formed by a group of mobile nodes spontaneously. Any node can leave the network at will, sometimes without informing the other participants about its departure. The multihop packet delivery between two parties can be based on any routing protocol specially designed for networks with dynamically changing topology (e.g., DSDV [Perkins and Bhagwat, 1994], AODV [Perkins and Royer, 1999], DSR [Johnson and Maltz, 1996]).

Although routing protocols are able to find the multihop route between any pair of nodes, they all assume that every device taking part in the commu-

nication owns a unique address. Thus, neither of these solutions deals with the question how the parties should obtain the unique identifiers. In managed IP-based fix networks a widespread solution is to set up a Dynamic Host Configuration Protocol (DHCP) [Droms, 1997] server, which returns unique and unused network identifiers on request. This and similar centrally managed approaches are not applicable in MANETs due to the fact that spontaneous networks lack any pre-installed and always available infrastructure.

Several address distribution solutions for MANETs exist, however they are either not scalable enough, or rather communication resource consuming. This paper describes a distributed protocol, which is able to configure a unique address for a newly joining node by optimizing the needed resources, and above that, it is capable of handling the partitioning and merger of separate MANETs. Our solution accomplishes these tasks without overloading the network burstfully, which makes the solution very well scalable over large MANETs. An often neglected but rather important issue is the fault tolerance of the address distribution system. It is expected that the system should continue working properly in case of any kind of failure on behalf of any node. The protocol presented in this paper successfully deals with these situations and manages the available address space in a way that no (permanent) address loss may occur in case of any participant's failure.

The remainder of the paper is organized as follows. Section 2 outlines the different concepts of address distribution mechanisms for MANETs by categorizing them, moreover presents the most important existing solutions. Section 3 describes our address distribution protocol, and Section 4 analyzes the optimization of the configuration time elapsed when assigning a unique address to a newly arrived node. Finally, Section 5 concludes the paper.

## 2. Related Work

Considering the address space, the existing address distribution mechanisms can be grouped into two categories based on the length of the addresses. The first category consists of distribution mechanisms that use fixed length addresses, while the mechanisms in the second category deal with variable length addresses. By using a mechanism belonging to the former category, the length of the addresses should be chosen carefully, so that all nodes in the network can obtain a unique address. If the address space turns out to be too small, there is hardly any way to extend it in the future. However, fixed length addresses have the advantage that they can be handled easier than those of variable length.

The largely different way of treating the address space leads to different optimization goals and difficulties. In both cases the address space can be represented as a binary tree, where the leaves of the tree identify the assigned addresses. Algorithms operating on *non-fixed length* addresses (e.g., [Boleng,

2002]) try to keep the number of levels in the address tree as low as possible, thus providing the shortest possible address length supposing a given number of participants in the network. However, in the *fixed-length* case, the size of the tree (and the address space) is predefined and the goal of the algorithms is to assign a free leaf to each newly arriving node. At a first glance the latter goal seems to be easily achievable, but consider that in MANETs any node can leave the network at will without informing the others about its departure. These so-called *non-graceful departures* can lead to the *address leakage* phenomenon, when the network is unable to detect that some addresses are not engaged anymore. Of course, this may lead to significant shrinkage of the address space after a certain amount of time. In the rest of the paper we will focus on the fixed-address length approaches.

An other basis of categorizing the existing solutions is the manner they treat the state of the address space. Those approaches where none of the participants in the network maintain information about the reserved addresses are *stateless* solutions, while the ones which make an effort to store a quasi up-to-date version of the address space are called *stateful* solutions. It can be generally claimed about the stateless solutions that there is no centrally or distributedly stored knowledge of the address space. When a node is willing to join the network, it has to agree on its candidate address with all the other—already configured—parties. This is often done by flooding the network, which causes burstful load on the network each time a node joins.

Other approaches, which are called stateful address distribution mechanisms, store an up-to date state of the currently available addresses. These solutions have to take care of address space maintenance continuously, which generates a slight, but permanent traffic load. When designing such an algorithm, special care has to be taken maintaining the address space, because inconsistent states may lead to address leakage or, even worse, to *duplicate addresses* in the network.

Stateful address distribution mechanisms usually provide faster address allocation process than stateless ones, since the joining node does not have to agree on its candidate address with all the nodes in the network, instead it gets an address from a dedicated node. Very often a neighboring node—called *proxy*—carries out the address allocation on behalf of the requesters, since this neighboring node already has a valid address and adequate knowledge on the network structure.

The task of address distribution mechanisms does not end at the configuration of a newly joining node. Often a common demand is to handle the merger and the separation of different MANETs, called *network partitions*. In the former case it should be prohibited that nodes with the same address participate in the same partition, while in the latter case departed nodes should be identified and based on this information the partition should be reorganized. In order

to detect nodes with the same address, several *Duplication Address Detection* (DAD) mechanisms were invented.

Many algorithms detect address collisions based on monitoring the packets containing routing information. If collision detection used by the algorithm is performed without sending extra messages, i.e., it is only based on observation, the method is called *passive duplicate address detection* mechanism. Such solutions were presented in [Weniger, 2003] and [Weniger, 2004]. There may be a stronger requirement toward the DAD mechanisms, which allow the partition merging processes to terminate only when there are no more nodes in the merged partition with the same address. This requirement is called *strong DAD* [Vaidya, 2002]. Solutions fulfilling this requirement usually perform global collision revealing mechanisms by flooding the detection packets through the whole network.

One of the first decentralized address assigning protocols was presented in [Cheshire et al., 2004] by the IETF (Internet Engineering Task Force) Zeroconf Working Group. The solution is based on the IP addressing scheme and assumes link-local connections between all parties, which means that all link-level broadcasts have to reach all nodes in the network. Thus, this restriction makes the solution directly unapplicable in MANETs where the communicating nodes may be in multihop distance from each other. Despite of this fact, the solution became the basis of several *stateless* approaches developed in the past few years.

Among others, the solution presented in [Nesargi and Prakash, 2002] is also based on the idea of the Zeroconf Working Group. It introduces a so-called *soft state maintenance* mechanism, which makes it possible to serve multiple join requests at the same time. Each node in the network maintains an array for the allocated addresses and an other one for addresses under allocation (pending addresses). Each device wishing to join the network, randomly chooses a candidate identifier and sends it to a neighboring proxy node, which already possesses a valid address. The proxy then floods the network with the address request including the chosen random address and waits for the replies. A node in the partition can only send back a positive acknowledgement for the request if it does not consider the requested address as allocated or as being under allocation. If the proxy node has received positive acknowledgements from all parties, then it assigns the candidate address to the joining node. The solution applies message flooding each time a node joins, which causes bursts in the network load. The protocol is simple and applicable in small MANETs, though the mentioned undesirable property makes it non-scalable over larger networks.

Although, the solutions described in [Cheshire et al., 2004] and [Nesargi and Prakash, 2002] differ in terms of the amount of information the nodes have to store about the address space, both methods use only premature techniques and

explicit messages to avoid address collisions. In addition, configuration of a node takes a lot of time in both cases. The Weak Duplicate Address Detection [Vaidya, 2002] mechanism solves this latter weakness. It allows duplicate addresses in a network as long as every packet is routed to the right party. The joining nodes simply choose themselves an identifier (e.g., based on their MAC addresses) and they start using it, without checking whether the chosen address is already used by another node. As soon as an address conflict arises, the duplication of the concerned address has to be resolved immediately.

There is a group of address distribution mechanisms, where the state of the assigned addresses is registered in some way. For example, in the Self-Organising Node Address Management (SONAM) solution [Toner and O'Mahony, 2003], a special node is responsible for assigning addresses to newly arriving nodes. Whenever a node requests an address, the newly arriving node receives—in addition to the address—a list of assigned addresses and the version number of the returned list, moreover it becomes the new special node. If the actual special node disconnects from the network, the new special node will be the node with the highest list version number.

Another possible way for obtaining an address is presented in [Zhou et al., 2003]. Each already configured node in the Prophet system possesses an *f(n)* function with a seed and a state. New addresses are generated based on these. If a node knows the default seed and the default state used in the network, then it can foretell which addresses will conflict when merging its network with another one.

# 3. SAMSON

## 3.1 Basic structure

The highest priority design goals of the *Smart Address Management in Self Organizing Networks* (SAMSON) protocol are the balanced control traffic load and the short address configuration time. Thus, it is a *stateful* protocol, able to handle the joining of nodes quickly and efficiently by adapting to the high amount of address requests concentrated to certain parts of the network.

The SAMSON protocol divides the available address space of the network partition into equal disjoint ranges. Each of these ranges is assigned to a so-called *pool*. Thus, each pool represents an address range and is responsible for managing the addresses of the given address range. The pools are scattered throughout the whole partition and are usually located at those nodes where unique addresses can be assigned to newly joining nodes in the shortest time in average. The nodes where the pools are located at are called *carrier nodes*. A pool is not bound to a certain carrier node: it can move from a carrier node to another if it detects that the address requests could be served more efficiently (in terms of *total expectable address configuration time*) by being located at

another node. Thus, every node in the partition can turn into a carrier node anytime. There is also no restriction on the number of pools that can be carried by a single carrier node at the same time.

Each pool can be uniquely identified by its managed address range, however, since the address ranges are equal, it is enough to identify a pool with the first address of the address range assigned to it. Sometimes it is desirable to treat a pool like a node (e.g., in case of sending an address request to it), therefore the first address of the address range is reserved for the pool itself. This way pools become *virtual nodes* and can be addressed by any node of the partition.

The network partition is initialized by a so-called *initiator node*. This node has to define the NETID, which is the unique identifier of the network partition. This identifier may be chosen by considering the lately observed network identifiers. A new pool is to be created at partition initialization time, or upon detecting that the address space of a pool will be exhausted soon due to the large number of configuration requests. This prediction is always to be made by the pools. In addition, a new pool must be generated when a network partition has been divided into two partitions in a way that no free addresses have remained in one of the partitions.

The pools are free to move in the network. Each pool chooses its next carrier node based on the frequency of address requests coming from different parts of the network. In order to measure the frequency of address requests we a defined a metric, which is used by the pool movement algorithm.

## 3.2 Joining process of a new node

A newly joining node can obtain a unique address by asking one of its neighbors for help in the address claiming process. This neighbor will be the *proxy node* of the joining node. The selection of such a proxy node is necessary, since during the address request process the joining node does not own any legal address that can be used in the MANET, only a link-layer address (MAC address), which can only be used to communicate with nodes in one-hop distance.

At first, the joining node $a$ broadcasts a $HELLO\_REQ$ to discover its neighbors among its link-layer neighbors (Figure 1). After the reception of the replies sent in answer to the request, node $a$ chooses one of its replying neighbors to be its proxy node during the address allocation. Then node $a$ asks its chosen proxy node $p$ to request a valid address from a pool in the network. If node $a$ has a desired address it wants to get (for instance it used this address last time it joined the network), then node $a$ includes the demanded address in the message sent to the proxy node. The proxy then sends an address request to the closest pool ($O$). If the request includes a desired address, pool $O$ checks the engagement of this address at pool $C$, which is the responsible pool for

the required address. Otherwise, if there is no address desire included in the request, pool $O$ allocates its first free address for the joining node and returns it to the proxy node. The proxy node forwards the ready-to-use address to node $a$ using node $a$'s MAC address.

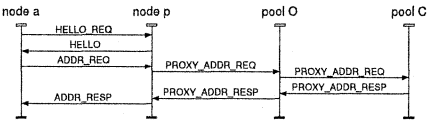


*Figure 1.* A new node joins the network

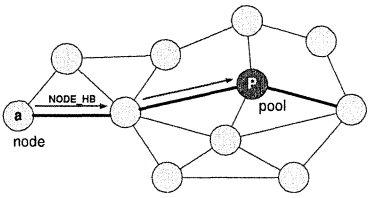


*Figure 2.* Keeping in touch with the pool

The closest pool can always be found at the special address 0. Thus, in addition to the first address of the address range assigned to a pool, each pool allocates itself a second address as well: a special address we named 0. This of course leads to address collisions, however, in the case of SAMSON it is allowed to have more of this special address in the network at the same time. These address conflicts do not have to be resolved. This way every address request packet sent to the address 0 is automatically delivered to the closest pool, irrespectively of the underlying routing protocol.

## 3.3 Wandering process of pools

Finding a good motion algorithm for pools is extremely important. If the pools are capable of moving to the areas where new addresses are requested more frequently then they can minimize in average the total time necessary for configuring addresses to arriving nodes. Finding these so-called *hotspots* is far not an easy issue, because the place of the hotspots may change from time to time with the changing of the arrival interval of new nodes at certain parts of the network. Moreover, the pools cannot have an up-to-date global knowledge about the topology of the network, except in case of certain kinds of underlying routing protocols (link-state routing protocols [Clausen and Jacquet, 2003]). Since one of our design goals was to create a protocol which may operate over any kind of ad hoc routing protocol, we designed an algorithm that doesn't assume any knowledge of the network topology.

Let us imagine a node $a$ which carries a pool $P$. Pool $P$ has to decide if it stays at node $a$ or moves to the neighboring node $b$. In order to make a decision, $P$ continuously monitors the address requests coming from different link-local neighbors of the carrier node or from the carrier node itself. Pool $P$ increases its corresponding counter when it receives an address request message from one of the carrier node's neighbors or from the carrier node directly. Based on these counters, pool $P$ periodically makes a movement decision. If pool $P$

finds that it received more address requests through the neighboring node $b$ than through its other neighbors and the carrier node in total, then pool $P$ initiates a *pool transfer*. In frame of this process the entire pool is transferred from node $a$ to node $b$. Following this rule the total expected address configuration time always decreases. The proof of the statement will be presented in Section 4.

## 3.4 Maintaining the states of the addresses

Since the address management is operating in an ad hoc environment, the protocol has to be fit for handling the sudden, *non-graceful* departures of the nodes. If a node disappears from the network, its address should be assignable again after a certain amount of time. The SAMSON protocol supports the detection of node losses at two levels. First, every pool continuously checks whether the nodes whose addresses are from the address space of the given pool are still present. Second, the pools continuously check the availability of the other pools.

After getting an address from a pool, every node has to send so-called heartbeat messages (NODE_HB) periodically to the pool from which it has obtained the address from. This procedure can be seen on Figure 2. If a pool has not received a NODE_HB message from a corresponding node for a long time, then it requests a presence signal from the node. If the addressed node fails to answer this warning, then the node is regarded as detached from the partition, so the address of this node is set as free and ready to be redistributed.

It may also happen that a carrier node leaves the network without informing the other nodes. In this case not only the address of the departed carrier node, but also the carried pools should be retrieved. As described previously, the pools continuously check the presence of others. Each pool—except the first one—sends a so-called keepalive message (POOL_HB) to the pool which is just inferior in the order of the represented address space. The first node in the order sends the keepalive message to the last pool, thus the pools form a ring to check their availability. This way the departure of a pool can be detected by its successor pool, i.e. by the pool whose address range is the continuation of the address range of the disappeared pool.

If a pool realizes that a predecessor pool has left the network, then it has to regenerate the lost pool. First, the successor pool reallocates the given pool by reserving the concerned address space. Then, it immediately freezes the address space (no address can be assigned from this address space) and waits for NODE_HB messages. Since the regenerated pool has the same address as the lost one, furthermore the new location of the pool is propagated by the routing mechanism, the nodes that have an address from the concerned address space can send their heartbeat messages to the pool without noticing the regeneration. Thus, after a while the regenerated pool will be aware of the

assigned addresses and therefore it can lift the blocking of the address space. From that time on the unused addresses of the pool can be assigned to newly joining nodes again. In this way the recovery period is directly proportional to the frequency of sending NODE_HB messages.

If two or more successive pools leave the network at the same time without informing the others, then they are recovered step-by-step: first the one which is the highest in the pool order, and finally the one which is the last in that order. Considering the fact, that neither the address reclaiming, nor the pool recovery processes are time critical, then the ratio of the period of the *POOL_HB* messages to the period of the NODE_HB messages may be chosen relatively big.

In case of losing all pools, the nodes must start to form a new partition.

## 3.5 Pool generation and deletion

As described previously, a pool can be generated in the partition initialization phase, moreover when a pool determines that its address space will be soon exhausted, or when there is no pool left in a partition after splitting up the original partition. In the first and third case the creator of the new pool is an arbitrary node, which noticed the absence of the pools by not reaching any virtual node at the special address 0. The node can simply generate the first pool in the order, i.e. the pool with the lowest addresses of the whole address space. In the second case, when there are already some pools in the network, the pool which detects the depletion of the free addresses has to send a pool request to the highest pool in the order. This so-called *chief pool* is responsible to generate a new pool and pass it to the requester. Moreover, if the new pool has addresses in a higher range than the current chief pool, then the new pool takes the role of the chief pool, and all the pools have to be informed about this. This way, two or more identical pools cannot be created at the same time.

Pools can be deleted if all the belonging addresses become free. If a pool detects that this is the case, the pool starts to delete itself. First, it sends a message to the chief pool indicating that the pool will be deleted and can be assigned to another nodes. Then, it sends an indication message to the first existing successor pool (it may happen that one or more successor pools are already deleted). After that, the pool destroys itself and the successor pool periodically informs the chief pool about the free pools it knows (at least this one). From now on, the successor pool sends the *POOL_HB* messages to the predecessor pool of the deleted pool. This way, the actual chief pool is always aware of the deleted pools, so it can reuse them, moreover the pool ring is kept alive.

## 3.6 Merging two partitions

When a node detects the presence of an other network partition (e.g., it receives a HELLO message containing a different NETID than its own partition), then it has to report this event to its pool. The pool that receives this notification has to forward the message to the chief pool, which then decides whether to merge the two partitions or not. If the decision is to merge the networks, then the chief pool sets up a tunnel to one of the nodes that noticed the presence of the other network. This node is called *bridge node*. Moreover, the chief pool instructs the bridge node to establish a bridge to a randomly selected neighboring node from the other partition. After that, the concerned bridge node in the other network will inform the chief pool in it's network (via a pool) about the established bridge and a tunnel will be set up between the chief pool and the bridge node in this network as well. From now on the two chief pools can communicate with each other through the tunnel–bridge–tunnel connection. Of course, it may happen that two tunnel–bridge–tunnel connections are established in parallel. In this case the chief pools have to agree which one to use.

After establishing the connection between the two chief pools, the chief pools have to agree on a new network identifier, furthermore they have to resolve the address conflicts. This latter task is done in one or two steps. First, they try to shift the address ranges of the pools to eliminate the duplicate addresses. Then, if the former step was not successful, i.e., in total there are more pools than the length of the address space divided by the length of one address block, then the corresponding pools have to harmonize the addresses they assigned to the nodes.

As it can be seen on Figure 3 the proximity of Partition *2* was detected by node *48*, which informed the chief pool *45* through pool *9*. Therefore chief pool *45* built up a tunnel to node *48*, node *48* set up a bridge to node *76*, and then another tunnel was built up between bridge node *76* and the chief pool *80* in Partition *1* by the aid of the pool located at node *40* (for locating the chief pool).

## 3.7 Dividing a partition

Partitions may be divided at any time without any prior indication. Thus, we designed the SAMSON protocol to be capable of reorganizing a partition in case it splits up into several disjoint partitions. The reorganization function is based on the pool recovery mechanism. Whenever a pool has to be regenerated, the presence of the chief pool has to be checked. If it is not available any more, then the network may be divided into two or more partitions. Therefore the new chief pool (which is automatically restored) in each new partition generates a new network identifier and floods it in the network partition. If there is no pool

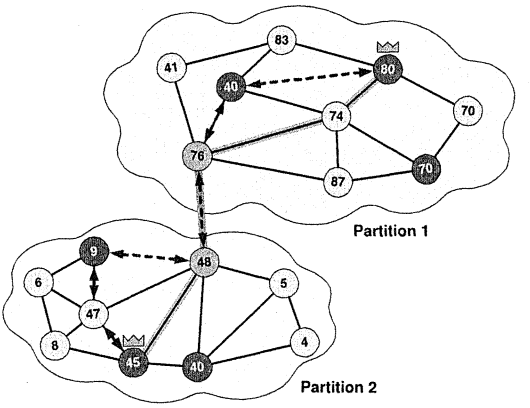


*Figure 3.* Merging of two partitions

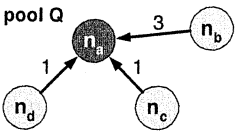


*Figure 4.* A sample situation of pool motion

remaining in a group of disconnected nodes, then the first node ($a$) that realizes the loss of the pools starts to form a new partition by creating a pool (the first one in the partition). The other nodes can then connect to this partition directly by requesting a new address or by forming a new partition and merging it to the partition set up by node $a$.

## 4. Analyzing the pool motion

Finding a good motion algorithm for pools is a relevant issue, since pools located at proper places (close to the *hotspots*) can keep the communication of the address claiming processes local, thus avoiding high traffic loads in the network. Furthermore, a good motion algorithm can reduce the average configuration time of newly arriving nodes.

To be able to examine the problem formally, let us assume the following. Let us take an $N$ set of nodes and associate an $R_i$ variate to its every $n_i \in N$ element. Among the $|N| = n$ element node set there is one $Q$ pool, which handles every address claims. $R_i$ variate indicates the number of address requests sent by node $n_i$ (which is a proxy node in such a case) to the pool in one time slot. If we make the simplification, that during a time slot none of the newly configured nodes can configure another node, we can state the following about the *expected configuration time for node* $n_i$ ($T_i$) in one time slot when the pool is located at node $n_p$:

$$T_i(n_p) = (2 \cdot T_{fw} \cdot d(n_i, n_p) + T_{serv}) \cdot \sum_{l=0}^{\infty} l \cdot P(R_i = l) \tag{1}$$

where constant $T_{fw}$ is the average forwarding time of a message from one neighbor to the other, $d(n_i, n_p)$ is the distance in hops between the proxy node

$n_i$ and pool $Q$ that is located at $n_p$. Constant $T_{serv}$ stands for the average service time of a request, and $P(R_i = l)$ indicates the probability that $l$ address requests arrived during the time slot. According to equation 1, the expected configuration time for node $n_i$ can be smaller only if the $d(n_i, n_p)$ distance is smaller.

The *total expected configuration time* ($T$) is the sum of $T_i$s over the whole network. Thus, the total expected configuration time is

$$T(n_p) = \sum_{i=1}^{n} T_i = \sum_{i=1}^{n} ((2 \cdot T_{fw} \cdot d(n_i, n_p) + T_{serv}) \cdot \sum_{l=0}^{\infty} P(R_i = l) \cdot l) \quad (2)$$

supposed that $Q$ is located at node $n_p$. So, the goal is to find the node $n_p$ (or one of them, if two or more nodes exist) that is the best carrier node for the pool $Q$ in order to minimize the total expected configuration time in the network. The total expected configuration time is the following if the pool is at an optimal location:

$$T_{min} = \min_{n_s \in N}(T(n_s)) = \min_{n_s \in N}(\sum_{i=1}^{n} \sum_{l=0}^{\infty} (2 \cdot T_{fw} \cdot d(n_i, n_s) + T_{serv}) \cdot P(R_i = l) \cdot l) \quad (3)$$

Let us call $n_{opt}$ the node where $T(n_{opt}) = T_{min}$, i.e. which is at the optimal position. Finding this $n_{opt}$ node is possible if the whole connectivity graph of the network is known. In this case, the optimal position can be calculated with an $O(n^3)$ algorithm, since the problem can be transformed to a similar problem, which is about finding the centrum of a graph. However, knowing the total connectivity graph of a network without sending explicit messages is possible only if the routing mechanism is a link-state protocol. Since it was among the design goals of the SAMSON protocol that the mechanism should work over any routing protocol, we supposed that pool $Q$ has only local knowledge of the connectivity graph (e.g., it knows its neighbors).

In order to examine how the pool motion rule presented in Section 3.3 can decrease the total expected address configuration time, let us take a look at Figure 4. Pool $Q$ is located at node $n_a$ and it has to decide whether it should move to an other neighboring node. It can be seen that two address requests have arrived in total from nodes $n_c$ and $n_d$, and three requests from node $n_b$ during the examined time period. In this case pool $Q$ can be placed at a more optimal position by moving it from node $n_a$ to node $n_b$. This movement is done according to the motion rule of the SAMSON protocol.

We can draw a general conclusion from the observation presented in the previous paragraph.

**Theorem 1 (The SAMSON moving rule is optimal).** Let us name with $N(n_a)$ the set of the neighboring nodes of node $n_a$. Furthermore let us depict with $K(n_i, n_a)$ the expected number of arriving requests to node $n_a$ through the link between nodes $n_i$ and $n_a$, and with $E(n_a) = \sum_{l=0}^{\infty} l \cdot P(R_a = l)$ the expected number of requests directly arriving to node $n_a$. In addition, let us suppose that

node $n_a$ is the carrier node of pool $Q$. If there is a node $n_b \in N(n_a)$ for which the

$$K(n_b, n_a) > E(n_a) + \sum_{n_i \in N(n_a), n_i \neq n_b} K(n_i, n_a) \tag{4}$$

inequality stands, then the

$$T(n_a) > T(n_b) \tag{5}$$

inequality stands as well. This means that in this case the total *expected configuration time* of the joining nodes over the whole network can be minimized by moving the pool $Q$ from node $n_a$ to node $n_b$. By replacing its location to node $n_b$, the total expected address configuration time of the joining nodes over the whole network will be decreased.

*Proof.* Let us depict with $B$ the set of nodes which are sending their requests to $n_a$ through node $n_b$ (including node $n_b$). Then:

$$\begin{aligned} T(n_a) = & \sum_{n_j \in N \setminus (B \cup \{n_a\})} (2 \cdot T_{fw} \cdot d(n_j, n_a) + T_{serv}) \cdot E(n_j) \\ & + \sum_{n_j \in B} 2 \cdot T_{fw} \cdot (d(n_j, n_b) + 1) \cdot E(n_j) + \sum_{n_j \in B} T_{serv} \cdot E(n_j) + E(n_a) \cdot T_{serv} \end{aligned} \tag{6}$$

$$\begin{aligned} T(n_b) \leq & \sum_{n_j \in N \setminus (B \cup \{n_a\})} 2 \cdot T_{fw} \cdot (d(n_j, n_a) + 1) \cdot E(n_j) + \sum_{n_j \in N \setminus (B \cup \{n_a\})} T_{serv} \cdot E(n_j) \\ & + \sum_{n_j \in B} (2 \cdot T_{fw} \cdot d(n_j, n_b) + T_{serv}) \cdot E(n_j) + E(n_a) \cdot (2\, T_{fw} + T_{serv}) \end{aligned} \tag{7}$$

The inequality 7 is true since there may be some nodes that are connected directly to nodes $n_a$ and $n_b$ as well. Thus, it is enough to see the following:

$$\begin{aligned} T(n_b) \leq & \sum_{n_j \in N \setminus (B \cup \{n_a\})} 2 \cdot T_{fw} \cdot (d(n_j, n_a) + 1) \cdot E(n_j) + \sum_{n_j \in N \setminus (B \cup \{n_a\})} T_{serv} \cdot E(n_j) \\ & + \sum_{n_j \in B} (2 \cdot T_{fw} \cdot d(n_j, n_b) + T_{serv}) \cdot E(n_j) + E(n_a) \cdot (2 \cdot T_{fw} + T_{serv}) \\ < & \sum_{n_j \in N \setminus (B \cup \{n_a\})} (2 \cdot T_{fw} \cdot d(n_j, n_a) + T_{serv}) \cdot E(n_j) \\ & + \sum_{n_j \in B} (2 \cdot T_{fw} \cdot (d(n_j, n_b) + 1) + T_{serv}) \cdot E(n_j) + E(n_a) \cdot T_{serv} = T(n_a). \end{aligned} \tag{8}$$

From which, after simplifications we can get

$$\sum_{n_j \in N \setminus (B \cup \{n_a\})} E(n_j) + E(n_a) < \sum_{n_j \in B} E(n_j). \tag{9}$$

And this is just another form of inequality 4, since the left side of the inequality expresses the expected number of configuration requests that are sent to $n_a$ but not through $n_b$, while the right side is the number of requests arriving at $n_a$ through node $n_b$ including the address requests addressed directly to $n_b$. So, taking into account that

$$\sum_{n_j \in B} E(n_j) = K(n_b, n_a), \text{ and } \sum_{n_j \in N \setminus (B \cup \{n_a\})} E(n_j) = \sum_{n_i \in N(n_a), n_i \neq n_b} K(n_i, n_a) \tag{10}$$

we get the inequality 4, which was the starting point. Thus, the theorem is proven, i.e., the total expected address configuration time always decreases when the pool moves based on the proposed moving algorithm. □

## 5. Conclusion

In this paper we have presented the SAMSON address distribution protocol which is able to assign unique identifiers to nodes in a self-organizing network. We have highlighted some limitations of existing address distribution approaches, and we incorporated the lessons learnt to our novel mechanism. The proposed solution is highly distributed and is able to handle network partitioning and merger. The presented protocol can cope with multiple joins at the same time and tolerates message losses and link failures. Furthermore, the described method comes up with a unique feature among ad hoc address distribution protocols: it can adapt to the node arrival intensity distributed in space. Thus, it is able to provide low configuration time for newly arriving nodes, even in case of a large number of participants in the network.

## References

Boleng, Jeff (2002). Efficient Network Layer Addressing for Mobile Ad Hoc Networks. In *Proc. of International Conference on Wireless Networks (ICWN02)*.

Cheshire, Stuart, Aboba, Bernard, and Guttman, Erik (2004). Dynamic Configuration of IPv4 Link-Local Addresses. IETF Internet Draft.

Clausen, Thomas and Jacquet, Philippe (2003). Optimized link state routing protocol. IETF Internet Draft.

Droms, Ralph (1997). Dynamic Host Configuration Protocol. RFC 2131.

Johnson, David B and Maltz, David A (1996). Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, volume 353. Kluwer Academic Publishers.

Nesargi, Sanket and Prakash, Ravi (2002). MANETconf: Configuration of Hosts in a Mobile Ad Hoc Network. In *Proc. of INFOCOM 2002*.

Perkins, Charles and Bhagwat, P. (1994). Routing over Multihop Wireless Network of Mobile Computers. In *SIGCOMM '94: Computer Communications Review,*, pages 234–244.

Perkins, Charles and Royer, Elizabeth (1999). Ad Hoc On-Demand Distance Vector Routing. In *Proc. of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100.

Toner, Stephen and O'Mahony, Donal (2003). Self-Organising Node Address Management in Ad Hoc Networks. In *Personal Wireless Communications, IFIP-TC6 8th International Conference*, volume 2775, pages 476–483.

Vaidya, Nitin H. (2002). Weak Duplicate Address Detection in Mobile Ad hoc Networks. In *Proc. of the 3rd ACM international symposium on Mobile ad hoc networking and computing (MobiHoc '02)*, pages 206–216. ACM Press.

Weniger, Kilian (2003). Passive Duplicate Address Detection in Mobile Ad hoc Networks. In *Proc. of IEEE Wireless Communications and Networking Conference (WCNC) 2003*.

Weniger, Kilian (2004). Passive Autoconfiguration of Mobile Ad hoc Networks. Technical report.

Zhou, H., Ni, L., and Mutka, M. (2003). Prophet Address Allocation for Large Scale MANETs. In *Proc. of INFOCOM 2003*.