IP ADDRESS ASSIGNMENT IN A MOBILE AD HOC NETWORK

Mansoor Mohsin and Ravi Prakash
The University of Texas at Dallas
Richardson, TX

Abstract—**A Mobile Ad Hoc Network (MANET) consists of a set of identical mobile nodes communicating with each other via wireless links. The network's topology may change rapidly and unpredictably. Such networks may operate in a stand-alone fashion, or may be connected to the larger Internet. In traditional networks, hosts rely on centralized servers like DHCP for configuration, but this cannot be extended to MANETs because of their distributed and dynamic nature. Many schemes have been proposed to solve this problem. Some of these approaches try to extend the IPv6 stateless autoconfiguration mechanism to MANETs, some use flooding the entire network to come up with a unique IP address, and others distribute IP addresses among nodes (using binary split) so that each node can independently configure new nodes. None of these existing solutions consider network partitioning and merging. In this paper, we propose a proactive scheme for dynamic allocation of IP addresses in MANETs. Our solution also uses the concept of binary split and takes into consideration the previously unsolved issues like partitioning and merging and abrupt departure of nodes from the system. We show that our solution is scalable and does not have the limitations of earlier approaches.**

Keywords— **MANET, auto-configuration, IP address.**

I. INTRODUCTION

A *Mobile Ad Hoc Network* (MANET) consists of a set of identical mobile nodes communicating with each other via wireless links. The network's topology may change rapidly and unpredictably. Such a network may operate in a stand-alone fashion, or may be connected to the larger Internet.

Since there is no built-in infra-structure required for the network to form, all the nodes have the capability to maintain all the resources of the network in a distributed fashion. Hence the nodes rely on each other to keep the network connected.

One of the most important resources is the set of IP addresses that are assigned to the network. When a new node wants to join a network, it has to be assigned an IP address as part of its initialization.

Address autoconfiguration in MANETs is still an unresolved issue. The IETF Zeroconf working group deals with autoconfiguration issues. One simple solution is proposed by [1] in which each node randomly configures itself and then performs duplicate address detection to resolve conflicts. This solution is not scalable as it floods the entire network while performing duplicate address detection. Another approach is based on IPv6 stateless autoconfiguration [2] [3]. This solution is limited in the sense that it only works with IPv6 and is dependent on the underlying routing protocol used. In general, we can categorize the IP assignment solutions to be either reactive or proactive. Reactive protocols require a consensus among all the nodes of the network on the new IP address that is to be assigned, whereas in the proactive approach, each node can independently assign a new IP address without asking permission from any other node in the network. The former scheme is described by [4] and the latter scheme is described by [5] [6]. In this paper, we will present a proactive approach based on the *binary split* idea of [5] [6].

This work is supported in part by the National Science Foundation CAREER grant CCR-0093411.

II. SYSTEM MODEL

We consider an autonomous Mobile Ad Hoc Network working on its own. It has no gateway or connection to the external world. The network is formed by a group of nodes coming together. The nodes can join and leave the network any time and are free to move around. Hence the size and topology of the network is dynamic and unpredictable in nature.

There is no single DHCP server. Instead, all nodes collectively perform the functionality of a DHCP server. Each node has the capability of configuring a new node by providing it with an IP address. All the nodes of the network synchronize from time to time to keep the latest information about the network.

A. Requirements

A protocol for assigning IP addresses should meet the following requirements:

- There should be no conflict in IP address assignment, i.e., at any given instant of time there should not be two or more nodes with the same IP address.
- An IP address is assigned only for the duration the node stays in the network. When the node departs the network, its IP address should become available for assignment to other nodes.
- A node should be denied an IP address only when the whole network has run out of its available IP addresses. In other words, if any of the nodes has a free IP address, this address should be assigned to the requesting node.
- The protocol should handle network partitioning and merging. When two different partitions merge, there is a possibility that two or more nodes have the same IP address. Such duplicate addresses should be detected and resolved.
- The protocol should make sure that only authorized nodes are configured and granted access to network resources.

B. Key Terms and Definitions

IP Address Block: A consecutive range of IP addresses. Usually the size of an an IP address block is a power of two.
IP Address Leak: A situation when the union of all the IP address blocks associated with all the nodes is less than the IP addresses for the the whole network. In other words there are IP addresses that are neither associated with any node nor available with any node as free IP addresses.
IP Address Conflict: A situation when two or more nodes in a network are assigned the same IP address. Detection of IP address leaks is explained in section V.
Partitioning: The division of a network into two or more disjoint sub-networks. In the simplest case, a partition may consist of only one node. Partitioning leads to IP address leak (Section VIII).

Merging: The combination of two or more networks into one bigger network. Merging results when two or more partitions come in the range of each other. Merging has the potential of causing IP address conflict (Section VIII).

III. The Buddy System for IP Address Assignment

Buddy systems [7] are a type of segregated lists that support an efficient kind of splitting and coalescing. Binary buddies are the simplest and best-known kind of buddy system [8]. In this scheme, all buddy sizes are a power of two, and each size is divided into two equal parts. All the buddies are aligned on a power-of-two boundary offset from the beginning of the heap area. Each bit in the offset of a block represents one level in the buddy system's hierarchical splitting of the heap area – if the bit is 0, it is the first in a pair of buddies, and if the bit is 1, it is the second in the pair. These operations can be implemented efficiently with bitwise logical operations.

In a Buddy System for IP Assignment every node has a disjoint set of IP addresses that it can assign to a new node without consulting any other node in the network. The basic idea is as follows:

1. In the beginning, there is only one node in the network that has the entire pool of IP addresses.
2. When an un-configured node, A, wishes to join a network, it requests the nearest configured neighbor node, B, for an IP address. Node B assigns the requesting node A an IP address from its pool of IP addresses. It also divides the set of IP addresses into two and gives one half to the requesting node A (keeping the other half with itself).
3. A node can leave the network either gracefully or abruptly. When node A leaves a network gracefully, it gives its pool of IP addresses to any node B nearby. Node B has the responsibility of handling this set of IP addresses — it can either keep this block of IP addresses with itself or it can find the buddy of node A (i.e. the node whose IP address block is a buddy of node A's IP address block) and forward this block of IP addresses to the buddy. On the other hand, when node A leaves the network abruptly it leads to IP address leak (because there is some IP address that is neither assigned to any node nor available for assignment to an un-configured node). This situation is handled later on when the nodes synchronize and exchange information about the latest status of the network.
4. Nodes synchronize from time to time to keep track of the IP addresses assigned and detect any leaks in the available pool of IP addresses. Every node keeps a record of all the IP address blocks in the network by maintaining a table similar to the one given below:

TABLE I
IP Address Table

NodeID	IPAddresses
1	0 - 31
5	32 - 63
3	64 - 127
...	...

This table is sorted with respect to the IP addresses. *This table can be implemented as a binary tree. A particular block of IP addresses can be searched in the table in $O(\log n)$ time.*

IV. Assigning a New IP Address

Let the node requesting an IP address be a *client node* and the node that actually assigns the IP address be a *server node*. In this way we can view the scheme of assigning an IP address as a handshaking protocol between the server and the client.

1. The client periodically sends broadcast messages using its hardware address.
2. When a server receives this broadcast message, it responds by sending a reply message using its IP address and the client's hardware address. It is possible that two or more servers reply to the same broadcast message.
3. The client sends an acknowledgment message back to the server. If the client has received more than one reply messages from different servers, it sends the acknowledgment message only to the first server and ignores the rest.
4. When the server receives this message, it realizes that it is ready to assign a new IP address to the requesting client node. If the server has multiple blocks of IP addresses (as a result of departures of neighbors), it assigns one of these blocks to the client node. Otherwise, it divides its set of available IP addresses into two disjoint subsets. It then sends one subset to the client node and keeps the other subset with itself. The server also sends its latest version of IP address table to the client.
5. When the client receives this set of available IP addresses, it assigns itself the first IP address from this set and keeps the rest as its available set of IP addresses. It then sends a confirm message to the server indicating a successful configuration.
6. When the server receives this confirm message, it terminates the IP assignment process.

It is possible that a node has allocated all the IP addresses from its IP address block but some other nodes in the network still have some available IP addresses. If this is the case and a new request for IP address assignment arrives, then we can employ the following three solutions:

1. The server node searches its IP address table for a neighbor node that still has some available IP addresses. The server then requests that neighbor node for some available IP addresses. The neighbor node divides its block of IP addresses into two and gives one half to the server (keeping the second with itself). The server then continues with the configuration of the client using this acquired block. If none of the neighbors has free IP addresses available, then the server checks for nodes which are at a distance of two hops, and so on.
2. The server sends a multicast message to all its neighbors for a block of available IP addresses. The server accepts the reply of the neighbor that replies first. The rest is the same as case 1. If none of the neighbors has free IP addresses available, then the server sends a multicast message to the nodes that are at a distance of two hops from the server, and so on.
3. The server node searches its IP address table for the node that has the biggest block of IP addresses. If two or more nodes have the same number of free IP addresses available, then it simply selects the nearest one. The rest is the same as case 1.

The first two solutions are very similar. The only difference is that in the first solution the server searches the IP addresses table whereas in the second solution the server sends multicast messages. The first one is efficient in terms of the number of messages it sends whereas the second one is efficient in terms of time required to discover a node that has free IP addresses available. In both the solutions, the server node increases the radius by one hop and searches for a node that has free IP addresses available until it finds one.

The third solution is different. The server node targets the node that has the most number of free IP addresses available. If all the IP addresses in the network have been assigned, the server does not need to exchange any messages to determine that. The advantage of this scheme is that it leads to less fragmentation as available IP address blocks get evenly distributed over the network whenever a node runs out of its free IP addresses. This is extremely useful when the network is getting concentrated at a given location and the nodes in that location run out of free IP addresses.

In terms of the amount of state information maintained, the second solution requires the least amount of state information because it does not need to know about the IP address block of any node in the network, whereas the third solution requires the most amount of state information as it needs to know about the IP address block of all the nodes in the network. The first solution, however, requires an intermediate amount of state information because the server only needs to know about the IP address blocks of it's neighbors.

In this paper, we will focus our attention on the third solution.

A. *Format of Messages*

The following messages are used in the assignment of a new IP address:

- **request** This is the broadcast message that the client sends periodically to start its configuration process. The source field of this message is the client's hardware address and destination field contains the broadcast address.
- **reply** This is the message that a potential server sends to the client when it receives a request message. This message lets the client identify the server it will be dealing with.
- **ack** This is the acknowledgment message that the client sends to the server in response to the reply message to permit the server to start the configuration process.
- **addressBlock** The server sends this message to the client. It contains the block of IP addresses that the client can use for the configuration of other requesting nodes. The first address of this block is used by the client as its own IP address.
- **confirm** This is the confirmation message by the client to the server indicating that it has been configured properly. When the server receives this message, it terminates the IP assignment process.
- **borrow** This is the message sent by the server to any other node asking for a free block of IP addresses. The server uses this message when it runs out of available IP addresses to assign to a new requesting node.
- **deny** The message is used by a server to a requesting client indicating that it has no free IP address available. When the client receives this message it realizes that there is no IP address available in the entire network and it cannot get configured.

B. *Timers*

The following timers are used in the assignment of a new IP address. Note that the name of the timer depend upon the message it is waiting for:

- **replyTimer** The client starts this timer after sending the request message and stops it when it receives the reply message from the server. When the timer expires, it sends the request message again and resets the timer.
- **addressBlockTimer** The client starts this timer after sending the ack message to the server and stops it with it receives the addressBlock message from the server. When the timer expires, the client assumes that the server has left the network before completing its configuration process, so it starts the IP assignment process from scratch again.
- **confirmTimer** The server starts this timer after sending the addressBlock message to the client and stops it when it receives the confirm message from the server. When the timer expires, the server assumes that the client has left the network before completing it configuration process and does nothing, otherwise it updates its IP address table (described in the next section) and make a new entry for the client.

V. Node Synchronization

Nodes of a MANET synchronize from time to time to keep record of IP address assignment in the entire network and detect any IP address leaks. The synchronization process involves every node broadcasting its pool of IP addresses. This broadcast is used by every other node to update its IP address table.

A. *Detecting IP Address Leaks*

If a node leaves the network abruptly, or if the network is partitioned into two or more networks, there are IP addresses that are assigned to nodes which are no longer part of the network. We need to devise a mechanism to detect such IP address leaks, and then take corrective actions to reclaim those IP addresses.

The detection of IP address leaks is simple. Let there be nodes i and j such that i and j are buddy nodes of each other. In order to detect IP address leak every node scans the IP address table (section III) for its buddy's IP address block, i.e., node i will scan the IP address table for node j and vice versa. If node i discovers that the IP address block corresponding to node j is missing from the table, it concludes that node j is missing and the network has been unable to reclaim j's IP address block. Node i then merges node j's IP address block with its own. The advantage of this scheme is that every node is responsible for finding out if its buddy node is missing and acquiring that buddy's IP address, as compared to flooding the entire network to find out IP address leaks.

VI. Returning a Pool of IP Addresses

We will adopt the same convention of *client node* and *server node* for this section. The node that has to return the pool of IP addresses is the client node and the node that accepts the pool of IP addresses is the server node. The protocol is designed in such a way to minimize work done by the departing node as far as possible. This is because the departing node might not have enough time to complete the whole process of giving its block of IP addresses to the node with its buddy block. The departing node just initiates the process before leaving the network and it is up to the network to make this a graceful departure.

1. When a client node decides to leave the network, it informs of its departure to any of the neighboring nodes (server node).
2. When the server node receives this message, it marks the IP addresses of the client as being available and sends a message back to the client signaling that it may now leave the network.
3. Upon receiving this message, the client node sends a bye() message to the server and leaves the network. This message reconfirms the client's intention to leave the network. If the client does not send this message it means that it has changed its mind and no longer wishes to leave the network.
4. When the server receives this bye() message, it is sure that the node has left the network. It then finds the buddy of the client (by looking up its IP address table) and requests it to acquire the client's block of IP addresses, or it can keep this IP address block with itself for configuring other nodes.
5. The buddy node acquires the block of IP addresses and sends a confirmation message back to the server.
6. On receiving this message, the server is sure that the block has been acquired successfully, so it terminates the process.

Note that if anything goes wrong in the above process, for example, if the client leaves the network without sending the bye() message, it will be treated an address leak and will be reclaimed when the nodes synchronize.

A. Format of Messages

The following messages are used in a graceful departure of a node:

• **depart** The client sends this message to the server when it wants to leave the network. The server can be any of the neighboring nodes.

• **ok** The server sends this message to the client after marking its entry in the IP address table. When the client receives this message, it is sure that the server will take care of its IP address block, so it can leave the network.

• **bye** The client send this message to the server indicating that it has left the network and the server can now send its IP address block to its buddy node.

• **acquire** The server sends this message to the buddy of the departing node asking it to acquire the IP address block of the client.

• **done** The buddy of the departing node sends this message to the server indicating that it has acquired the IP addresses of the client. When the server receives this message it deletes the departing client's entry from the IP address table and relaxes as it no longer needs to worry about taking care of the client's IP address block.

B. Timers

The following timers are used in the graceful departure of a node:

• **okTimer** The client starts this timer after sending depart message to the server and stops it when it receives the ok message from the the server. When the timer expires, the client can either start the departure process again or it does nothing and just leaves the network assuming that it is no longer a part of the network. Note that the latter case leads to IP address leaks and it is up to the network to resolve this situation.

• **byeTimer** The server starts this timer after sending the ok message to the client and stops it when it receives the bye message from the client. When the timer expires, the server pings the client node. If it hears nothing from the client, it assumes that the client has left, so it starts the process of assigning the client node's IP address block it its buddy.

VII. The Algorithm

The algorithm for the client using the Abstract Protocol Notation (APN) is given below:

```
const threshold;
bool begin := true;
bool configured := false;
int count := 0;
bool firstReply := true;

begin = true ∧ count < threshold →
    broadcast request;
    start replyTimer;
    begin := false;

count = threshold →
    self-configure();
    configured := true;

receive reply from server ∧ firstReply →
    stop replyTimer;
    send ack to server;
    start addressBlockTimer;
    firstReply := false;

receive addressBlock from server →
    stop addressBlockTimer;
    configured := true;

configured = true →
    send depart to server;
    start okTimer;

receive ok from server →
    stop okTimer;
    send bye to server;
    releaseIPAddress();
    configured := false;
    count := 0;

timeout(replyTimer) →
    begin := true;
    count := count + 1;

timeout(addressBlockTimer) →
    begin := true;
    count := count + 1;

timeout(okTimer) →
    configured := false;
```

The corresponding server algorithm for the IP address assignment is given below:

```
bool availableBlock := true;

receive request from client →
    send reply to client;
    start ackTimer;
```

```
receive ack from client →
    stop ackTimer;
    if availableBlock = true then
        divideBlock();
        send addressBlock to client;
        if free IP address is available then
            availableBlock := true;
        else
            availableBlock := false;
        fi
    else
        checkIPAddressTable();
        if no node has a new IP address then
            send deny to client;
        else
            send borrow to newServer;
            start addressBlockTimer;
        fi
    fi
receive confirm from client →
    updateIPAddressTable();
receive addressBlock from newServer →
    stop addressBlockTimer;
    newAddressBlock = divideBlock(addressBlock);
    send newAddressBlock to client;
    if free IP address is available then
        availableBlock := true;
    else
        availableBlock := false;
    fi
receive depart from client →
    send ok to client;
    start okTimer;
receive bye from client →
    stop byeTimer;
    findClientBuddy();
    send acquire to clientBuddy;
    start doneTimer;
receive done from clientBuddy →
    stop doneTimer;
    updateIPAddressTable();
timeout(confirmTimer) →
    pollClient();
    if client is present then
        updateIPAddressTable();
    else
        skip;
    fi
timeout(byeTimer) →
    pollClient();
    if client is not present then
        findClientBuddy();
        send acquire to clientBuddy;
        start doneTimer;
    else
        skip;
    fi
```

Most of the functions given in the above algorithm can be implemented in a number of ways. Given below is a brief description of the different possibilities:

The function `self-configure()` is called when a node discovers that it is the only node in the network and needs to configure itself. This node has the entire block of IP addresses and assigns itself the first IP address of that block. This makes the division of IP address blocks very easy, as a node can keep the first half and give the second half to a requesting node. `configure()` too works on the same principle. It also has a block of IP addresses (obtained from a server node) and assigns the first IP address from that block to itself. Both these functions also update their copy of IP address table. Whenever a node gets configured, it sends a broadcast message to the network informing of its arrival. This helps in updating the routing and IP address tables. The function `releaseIPAddress()` is very simple. All it does is assign null values to the IP address block, it own IP address and the IP address table. It is worth mentioning here that if we are dealing with gateways and subnets then the above three functions should also deal with this situation.

`updateIPAddressTable()` adds or deletes an entry in the IP address table to reflect the latest state of the network. Adding a new entry means arrival of a new node and splitting of an IP address block into two smaller blocks. Similarly, deleting and entry means departure of node and merging of two IP address blocks into one bigger block. When a node updates its IP address table it does not immediately informs this to all other nodes in the network. This is because as long as a node has a pool of IP addresses, it is autonomous in configuring client nodes. Other nodes get to know of this only when needed, e.g. when a node has exhausted its IP address block and it asks for a block from some other node. Even if this does not happen then the time to time node synchronization will keep the information updated throughout the network.

The function `checkIPAddress()` scans the IP address table for the node that has the largest IP address block. This function is used by a server node that has run out of its IP address block wants to borrow a block from some other node. `findClientBuddy()` is also a similar function. It also scans the IP address table but it takes the IP address of the client as a parameter and returns the IP address of the node that is the buddy of the given client node.

`divideBlock()` divides an IP address block into two blocks. It returns the second half of the original block encapsulated as the addressBlock message that can be sent to the client.

The function `pollClient()` simply pings the client just to find out whether it is alive or not.

VIII. Network Partitioning and Merging

One main advantage of the Buddy System for IP address assignment is that it works very well with network partitioning and merging. In order to differentiate two or more partitions, each partition is associated with a PartitionID which is a universally unique identifier, UUID. Hence, it is guaranteed that every partition will have a unique PartitionID.

There can be two cases when a new PartitionID is generated. First, when a new network is formed, i.e. when a node discovers that it is the only node in the network, it generates a new PartitionID and assigns to itself. Later, whenever a new node wishes to join the network, this PartitionID is passed to the new node as a part of the configuration process. The second case is when an existing network is partitioned into two or more partitions. When a node discovers that it has been partitioned from the main network, then each node in that new partition is assigned a new PartitionID. The protocol to detect network partitioning is described in [4].

Now let us consider what happens when a network gets partitioned and the different partitions merge after some time. Let **A** and **B** be two partitions after a network is partitioned into two. As long as **A** and **B** have some IP address available, they do not assign themselves a new PartitionID. The Buddy System for IP address assignment will continue to work under such conditions. If the two partitions **A** and **B** merger before they run out of IP addresses, then we do not need to do anything. In other words, there is no problem if disjoint partitions merge.

On the other hand, consider the situation when **A** has used all its IP address and a new request comes in. In this case, partition **A** generates a new PartitionID and acquires the entire IP address block of partition **B**. Now if **A** and **B** merge then their intersection is not null and there are IP address conflicts. In order to resolve such conflicts, one of the conflicting nodes in either **A** or **B** has to give up its IP address block. Our algorithm requires the node with the larger IP address block to give up its IP address and acquire a new IP address. In this way the number of changes required to resolve the conflicts is minimum. The details of the algorithm are given below:

```
find the node in other partition with same IP address
if there is no such node then
    adjust PartitionID, if necessary
else
    let A = node in other partition with same IP address
    if blockSize = A's blockSize then
        give up IP address
        request for an IP address from A
    else if blockSize < A's blockSize then
        divide block into two
        give the second half to A
        (A will be requesting for an IP address)
        adjust PartitionIDs
    else
        request for an IP address from A
    end if
end if
```

Figure 1 illustrates how IP address conflicts are resolved using the above algorithm. Nodes A and B have the same IP address. Node A gives up its IP address since its block size is greater than node B's block size. Node B then divides its IP address block into two and gives the second half to A.

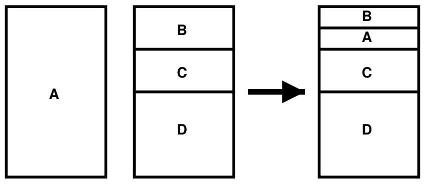

Fig. 1. Block sizes before and partition merging

Referring to the figure, several improvements can be made to the above algorithm. Instead of dividing block B into two and giving one half to A, we can divide the largest block, in this case D, into two and give one half to A. In this way, IP addresses will be much more uniformly distributed among the nodes. Also if we have some information about the amount of network a node is using, then we can decide which node to give up its IP address. For example, suppose that A has a lot of telnet and ftp sessions going while B is idle, then it's a good idea to give B the second block while A keep the first block.

IX. Performance Measures

Assigning a new IP address is very efficient, as only the client and server nodes are involved in this process and there are no overheads. There are a finite number of messages needed for this protocol.

Synchronization of nodes involves broadcast messages as every node periodically broadcasts its information about its block of IP addresses. So, in a network with n nodes, we will have n broadcast messages during a finite interval of time. We can adopt a lazy synchronization mechanism here in which the nodes synchronize only when needed. This significantly reduces the number of messages in a given amount of time.

When a node departs a network, again we have a finite number of message exchanges. This is possible because the server node can search for the node with the buddy block from the table and it does not need to send any broadcast messages.

As far as memory requirement is concerned, it depends on the size of the table every node has to maintain. In the worst case, all IP addresses has been assigned and each block of IP addresses consists of just one IP address. In this case, the number of entries in the table are equal to the size of the network. Note that the buddy scheme not only helps minimize the size of the table but also helps minimize fragmentation of IP address blocks.

X. Conclusion and Future Work

We presented a simple solution for automatic IP address assignment of nodes in a MANET. The solution employs the buddy system, a well known method for memory management. The solution has low overheads, and is capable of handling node arrivals, departures, and network partitioning.

A major issue that has been ignored in this paper is *security*. We assumes that each node trusts every other node, but if this is not the case then the following situations can arise:

- A node requests IP addresses for nodes that do not exist. In this way a node can acquire all the IP addresses denying others to participate in the network.
- A node assigns IP addresses to other nodes without following the given protocol. This can lead to IP address conflicts which might be difficult to resolve.
- A node selectively gives wrong information to other nodes.

The synchronization process in our protocol depends on reliable broadcast. Since no such broadcast exists in a mobile distributed environment, one can question the robustness of the protocol. There is need for further research on security and reliable multicast/broadcast communication in MANETs.

References

[1] C. E. Perkins, E. M. Royer, and S. R. Das: *IP Address Autoconfiguration for Ad Hoc Networks*. Technical Report draft-ietf-manet-autoconf-00.txt, Internet Engineering Task Force, MANET Working Group, July 2000.
[2] S. Thomson, and T. Narten: *IPv6 Stateless Autoconfiguration*, RFC 2462, December 1998.
[3] Kilian Weniger, and Martina Zitterbart: *IPv6 Autoconfiguration in Large Scale Mobile Ad Hoc Networks*, University of Karlsruhe, Germany.
[4] Sanket Nesargi and Ravi Prakash: *MANETconf: Configuration of Hosts in a Mobile Ad Hoc Network*, Proceedings of INFOCOM 2002.
[5] A. J. McAulley, and K. Manousakis: *Self-Configuring Networks*, MILCOM 2000, 21st Century Military Communications Conference Proceedings, Volume 1, Pages 315-319.
[6] A. Misra, S. Das, A. McAulley, and S. K. Das: *Autoconfiguration, Registration, and Mobility Management for Pervasive Computing*, IEEE Personal Communications, Volume 8, Issue 4, August 2001, Pages 24-31.
[7] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles: *Dynamic Storage Allocation: A Survey and Critical Review*, International Workshop on Memeory Management, September 1995.
[8] Kenneth C. Knowlton: *A Fast Storage Allocator*, Communications of the ACM, Volume 8, Number 10, October 1965.