Mobility Impact in Initializing Ring-Based P2P Systems over MANETs

Wei Ding
University of Maine at Fort Kent
Fort Kent, Maine 04743, USA
wei.ding@maine.edu

Alban Moreau
University of Brest
Brest, France
alban.moreau@etudiant.univ-brest.fr

Abstract— **With the encouragement from success of P2P systems in real world application, recently we have seen active research on synergy of P2P systems and mobile ad hoc networks. The paper proposes a solution for mobility disturbance problem in initialization of ring-based P2P systems over ad hoc networks. It is a decentralized ring construction protocol in presence of mobility. A Mobile Ring Ad-hoc Networks (MRAN) protocol is presented. MRAN is an extension of RAN [1] under the mobile condition. Simulation result shows MRAN works well with mobility. Upper bound of maximum speed of moving nodes is investigated in simulation.**

Keywords-mobility; topology; ring construction; peer-to-peer;ad-hoc networks

I. INTRODUCTION

Decentralized computing is believed to have great potential to replace the client/server model. As leaders of decentralized computing, Mobile Ad-hoc Networks (MANETs) and Peer-to-Peer (P2P) systems are recent hot topics. They share many aspects in decentralization. However, their levels of real world application are widely divergent. Cachelogic reported that in January 2006 P2P traffic accounted for 71% of all Internet traffic. Contrarily, only few applications of MANETs have been commercialized. The synergy of P2P systems and MANETs has stimulated considerable research effort. Though most have focused on routing, initialization is indispensable.

In [1] a Ring Ad-hoc Networks (RAN) protocol was proposed, which is an initialization protocol for ring-based P2P systems over MANETs. The feasibility of RAN protocol is mathematically proved. Simulation results support the proof in static scenarios. In this paper we extend RAN to MRAN, i.e. Mobile RAN, to cover mobile scenarios. Random walk model is used. In [1], three algorithms are tested. Only distributed exhaustive pattern (RAN-DE) performs well in both efficiency and effectiveness. MRAN inherits RAN-DE. In this paper, sometimes we use RAN as synonym as RAN-DE.

MRAN applies to some other models such as random waypoint as well. It targets at walking based applications, like conference, airport, museum, hospital, etc. The simulation shows that MRAN is adaptive and flexible to mobility.

The problem from mobility is the topology disturbance. Each node in a connected component of a MANET has a virtual spanning tree called component tree, of which the root is always itself. For a static MANET, the component tree is fixed. With mobility, neighbor set of a node is not steady any more. The change in the neighbor set tree causes the change of component tree. Adjustment has to be made to remove detached branches and to accommodate new arrived branches. And nodes keep moving while we adjust to previous movement.

The rest of paper is organized as follow. Section 2 introduces related works. Section 3 gives a review of RAN. Section 4 analyzes the impact of mobility and possible counter measures. Section describes our solution, the MRAN protocol. Section 6 reports the simulation and gives the upper-bound of mobility. Section 7 concludes the paper.

II. RELATED WORKS

Initialization or bootstrapping of a structured P2P system is the initial procedure in which nodes are assigned IDs and overlay topology is constructed. After it, the system should be able to run normally. Bootstrapping could be traced back to the very beginning of structured P2P systems. In 2001, Chord [2, 3] and Pastry [4] were developed. Since then bootstrapping has been dominated by node joining approach. It was usually concealed. [2, 3] and [4] did not describe how the ring should be set up. Same are new systems like Virtual Ring Routing [5]. [6, 7, 8, 9] tried to transplant ring-based P2P systems into MANETs. They handled bootstrapping similarly, either with joining method or ignored bootstrapping.

Solutions came recently in P2P over wired networks. In [10], T-Man protocol gave a simple and elegant solution for topology construction. It found that many topologies could be expressed with a ranking function, which is often as simple as Euclidean distance. In [11] T-Man was applied directly in Chord over wired networks. Ring Network [12] suggested an intuitive algorithm for searching closest successor in node identifier space.

RAN protocol [1] is probably the first workable approach in initialization of ring-based P2P systems over MANETs. RAN benefited from T-Man and Ring Networks. It builds a ring for each connected component in a MANET. Upon this ring, ring-

based P2P systems could run immediately without lengthy stabilization.

III. Review of RAN Protocol

A. Outline

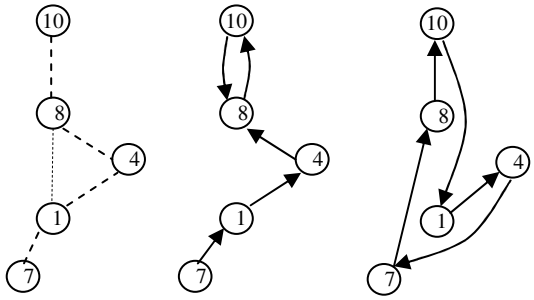

Figure 1. Left shows neighborhood relation. Middle shows original successor relation. Right is successor relation after running RAN

RAN is a decentralized message passing protocol to build a ring for each connected component in a MANET. Each node dynamically builds but does not store a component tree. It sets itself as the root of the tree. Distance from node A to node B is defined as $(ID_B - ID_A) \bmod max$. *max* is a relatively huge modulo. The ring topology is determined by the successor relation among nodes. At each step, if a node has shorter distance from root, it is selected as the new successor. The procedure repeats till the tree is traversed.

RAN integrates the overlay layer directly into Network and MAC layers. Similarly dynamic source routing is integrated into ring-based distributed hash table (DHT). It avoids direct mapping of overlay layer into lower layers. Every node goes through its component tree on the fly. No node keeps entire component tree in storage. Only some parts of the tree exist in memory when searching for next closer successor. This statelessness considerably increases flexibility and robustness.

Three patterns are tested to find optimal balance between effectiveness and efficiency. Two exhaustive patterns search the entire tree for closer successor. So the ideal ring is guaranteed. However, this exhaustion may suffer from high overhead of time and message. RAN-DE has better performance because only neighbors exchange messages and no multi-hop message is transmitted. In RAN-DE, the root sends a *getCandidate* message to each direct child. At all following levels, each receiver of this message forwards the message to its own children at the next level until leaf nodes are encountered. From leaf nodes, the closest successor of the root in the subtree is calculated and is returned to the parent node in a *candidate* message. From one level to next upper level, *candidate* messages are forwarded and integrated at parent nodes. This procedure is repeated till the root node is met. The downward execution is a limited flooding.

Three options could be applied to three patterns to improve the efficiency. Plain option means no additional operation and the search should end with a complete ring. The approximation option loosens the end condition of search. A small fraction of nodes are allowed to be left out of the final ring. In multicast option a node sends message to all direct children by multicasting.

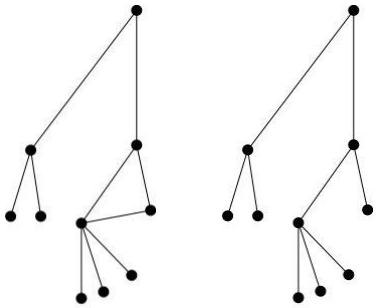

Figure 2. Convert a connected component to a component tree

B. Variables

There are two types of variables. One is collective type, which is for all component trees on this node. The other is node-based. The important *root_set* is the set of root of component trees that cover this node. For each node, these trees use common resources on this node by time sharing. If we ignore the microscopic details of sharing, we can simply think these component trees as running on this node concurrently. Other important collective variables are: the queue for incoming messages *in_queue*, the queue for outgoing messages *out_queue*, set of current one-hop neighbors Γ.

All node-based variables are referenced in the framework of a component tree. So each node has *n* copies of node-based data structures. Homonymous variables at a node are differentiated by their root. The most important node-based variable is *r*, ID of the root of component tree. It designates the component tree to node-based variables. Other important node-based variables are:

- *candidate*(*r*): closest candidate for *r*
- *candidate_distance*(*r*): ID space distance from *r* to *candidate*(*r*)
- *finished*(*r*): indicate if the search for *r*'s closest successor is finished

Messages are not normal variables. But they are similar from the view point of data structure. Three messages are defined in the distributed exhaustive pattern [1], namely, *getCandidate*, *candidate*, and *alreadyReceived*. All are transferred inside a certain component tree. All have fields *r*, *sender*, and *receiver*. *r* is the root of the component tree.

C. Algorithm of Distributed Exhaustive Pattern

The algorithm is written in AP notation. *u* represents current node.

(a1)
Protocol starts →
construct Γ
if $\Gamma_{=} \varnothing$ **then** *finished* := **true**

```
reply_received := 0
for each h ∈ Γ do send getCandidate(u, u, h) to h
[]

(a2)
receive getCandidate(r, q, u) from q →
if r ∈ root_set
then send alreadyReceived(r, u, q) to q
else
  root_set := root_set + {<r, q>}
  candidate(r) := u
  candidate_distance(u) := (u – r) MOD maximum
  for each h ∈ (Γ - {q}) do send getCandidate(r, u, h) to h
[]

(a3)
receive candidate(r, cd, q, u) from q →
if r∉ root_set then return
x := parent of r in root_set
reply_received ++
d := (cd – r) MOD maximum
if d < candidate(r)
then
  candidate(r) := cd
  candidate_distance(r) := d
if u ≠ r
then
  if reply_received = |Γ| - 1
  then send candidate(r, candidate(r), u, x) to x
else
  if reply_received = |Γ|
  then finished := true
[]

(a4)
receive alreadyReceived(r, q, u) from q →
if r∉ root_set then return
x := parent of r in root_set
reply_received ++
already_received_neighbor ++
if u ≠ r
then
  if reply_received = |Γ| - 1
  then send candidate(r, candidate(r), u, x) to x
else
  if reply_received = |Γ|
  then finished := true
[]
```

IV. Impact of Mobility

A. Types of Changes

For individual nodes, mobility changes their position. It causes the fluctuation in its neighbor set, unless all its neighbors are in a same collective mobility pattern. The change of neighbor set in turn causes the change in network topology. The topology change is shown in two aspects. First is the change of connected components, which is more radical. The second is the change of the component tree within unchanged component set.

Now we elaborate on the change on the component tree. If we look at the disturbance from in the framework of single component tree, changes could be generalized into two types, that is, changes from children nodes and changes from parent nodes. Most changes from children nodes are lost branches (or lost subtree). A change from parent nodes is usually remedied by shift to *hidden parent*. The former is generic disturbance occurring in most cases.

The latter only happens when the connected component is highly connected, which means, a node may have more than one possible parent nodes. Due to the requirement of tree structure, it can only pick up one as its current parent. All others become its hidden parents. They are still parent nodes by radio range, but not used as parents in this particular root's component tree. Keeping hidden parents is very helpful in dense or well-connected component to minimize the adaptation overhead.

B. Lost Branch

In a component tree, there are three possible scenarios in lost branches caused by mobility. To distinguish these three scenarios, we define the concept of *current line of search*. In a component tree, the search for the closest candidate in MRAN is always done from root to leaves. In the protocol inherited from the distributed exhaustive pattern in RAN, the search starts from the root, simultaneously run down through all braches of root toward the leaves. At each point in time, there is only one active node on any path from root to a leave. At any point in time, current line of search consists of active nodes from all paths from the root to every leaf nodes.

There are two kinds of current line of search that are slightly different. One is seen as real time ideal line of search, which probably only exists in physical world, perceivable to researchers, and always reflects the exactly accurate connected component and corresponding component tree, and is not skewed by disturbance from mobility. One is visible to computers and embodied by data structure in storage, which is likely vulnerable to disturbance from mobility. We use the concept to clarify problem and define solution, so what we use here is the former form, which is invisible to computer.

In the first scenario, a lost branch is below the current line of search. There is almost no effect of this loss. Actually the execution of a search protocol will not notice this change.

In the second scenario, the loss occurs on the current line of search. That means, the sender has an incorrect neighbor set when it sends out the *getCandidate* message, though it has the right neighbor set very short time before the sending. If the neighbor set is updated with much higher a rate, for example, by using hello message, this scenario is impossible. Otherwise, if the neighbor set is not updated that frequently, the receiver node has left the radio range of the sender. But the sender has not noticed this. This situation is a little tricky. The receiver definitely can not tell the occurrence of this transmission. The sender would not know as well if the message passing is not

enforced with acknowledgement message. In this paper we assume all message are acknowledged by a very short message, which is processed with very high priority when received and guaranteed to be sent immediately. Under this assumption, after the sender updates to the correct neighbor set, the second scenario could be reduced to the first scenario.

In the third scenario, the lost branch is above the current line of search. According to the MRAN algorithm, everything is still same on the downward running of the protocol. Problems only arise when the execution reaches the bottom of the component tree and the candidate message is sent back to the root. Note that in every step of downward running, a return path is calculated and sent along with the *getCandidate* message. In this case, the root of the lost branch will not be able to find parent node recorded from downward running. In another word, the node finds that supposed parent is not in neighborhood, the effort of returning of candidate should be ended here. This branch should be cut off the component tree of this particular root. In (a5) of AP notation of MRAN algorithm (see Section 5), a node updates its parent set as soon as it finds that it has lost current parent nodes. However, here is a possibility of duplicated parent nodes if the component well connected, which we will discuss in next section.

C. Hidden Parents

In above third scenario for a dense or well-connected network, another possibility could be brought out by mobility. We call it hidden parents. In a well-connected network, a node of a certain component tree may have more than one nodes in its radio range, which connect itself back to higher layers of this component tree. Mobility disturbance may break links to some hidden nodes, but it is unlikely that all links to hidden nodes are broken. If a lost child node has unbroken links to connect it to the remaining part of the component tree, it should use this links to connect back to the original component tree. The advantage of shifting is: the topology change due to mobility could be minimized; so could overhead of adaptation. If there are more than one links available, we simply use the rule of first come first serve.

The hidden parent list is constructed when duplicate *getCandidate* messages are received. By the rule of first come first serve, the sender of first *getCandidate* message becomes the current parent; following senders are saved along with corresponding *getCandidate* messages. Saving of messages is essential because the path from root is included in the message. When the *getCandidate* message is too big or there are many duplicated *getCandidate* messages in incoming queue, it is advisable to just save the parent node ID (or index) and the path.

The shifting adaptation is not limited to parent change. For the current node, we also need delete the *getCandidate* message from its lost parent in its incoming message queue, and in the root set. This message should be the only messages from the lost parent. These two pieces of information — node ID and the path from root via new parent — should be used to generate a new entry in the root set. When a *candidate* message comes back from a leave, it only contains the root as the destination. The root is used to look up the root set table, and the new parent will be return as the receiver of the next *candidate* message.

Obviously, hidden parents are accompanied with hidden children. Hidden children are those nodes in neighborhood that have sent *alreadyReceived* message to the current node.

D. Rescuing Descendent Nodes of a Definitely Lost Node

In last section, if no node in the hidden parent list is connected to the current node, the current node is lost for sure. We call this kind of node *definitely lost node*. Unlike in Section 3.1 where a lost branch is simply cut off from the original tree, we choose another approach with much limited deviation from the original component tree. The definitely lost node floods a *definitelyLost* message to its descendents. In its subtree, a node do similar thing as described in Section 3.2 after receiving this *definitelyLost* message. It checks its own hidden parent list to find a working hidden parent to connect itself back to the original component tree. The *definitelyLost* message is forwarded to receiver node's children only if it finds itself lost as well. That means it could not find a hidden parent in neighborhood which could be used to replace its lost parent.

Note that a node can only send out a *definitelyLost* message after it has received and forwarded a *getCandidate* message. In another word, this children rescue procedure is not applicable to nodes below the current line of search. It is also not applicable to the root node of the component tree. The reason is: a node below the current line of search has not had the chance to compile its hidden parent list. The list could only be finalized after a node is completely above the current line of search.

V. MRAN

A. Possible Approaches

There are two radical solutions. One is to apply thorough and frequent update to reflect mobility. Another is to apply no update for mobility at all. The first approach totally abandons previous component trees and restarts everything from the very beginning. It would cause tremendous overhead in the adjustment. More importantly, when average speed of mobility is high, the rate of update of component trees may not be able to catch up mobility change. With the second approach, MRAN is reduced to the distributed exhaustive pattern of original RAN protocol. The algorithm would not stop when there is an unrecovered lost branch. The program will remain in an infinite loop since the end condition is for the root of a component tree to collect all response from its descendents. Obviously, a third course has to be found to keep MRAN workable facing mobility.

In this paper, a minimum adjustment approach is proposed to handle mobility change. This minimum adjustment is completed in rather short time so it will not be interfered by next round mobility. It is independent to searching cycle of MRAN protocol.

Similar to RAN, MRAN is based upon message passing. MRAN keeps the major part of search algorithm of RAN. In the downward flow of RAN, almost nothing is changed. The

getCandidate message is passed in the same manner exactly as in RAN. Even in an already lost branch of a component tree, the message will still be forwarded to leaves. In this largely distributed setting, only way of inter-node communication is message passing. To notify lower nodes an incorrect message, we have to send another message. However, this correction message could never catch the first wrong message, so no remedy could be implemented. The only thing we can do is letting it be.

B. Variables

MRAN inherits many variables from RAN. Following is the list: *mobility_interval*: the period of mobility; $\Gamma_l(r)$: old set of one-hop neighbors; *unreplied_children*(*r*): set of ID of *unreplied* children; *replied_children*(*r*): set of IDs of children that returned the *candidate* messages; *candidate*(*r*): current closest candidate for *r*; *candidate_distance*(*r*): ID space distance from *r*; *hidden_parent*(*r*); *hidden_children*(*r*).

Two more messages are added to RAN-DE in MRAN: *update* and *definitelyLost*. They all have fields *r*, *sender*, and *receiver*. *update* is for sending new candidate caused by disturbance of mobility. New candidate is either from the lost branches or from arrival of new neighbors. *definitelyLost* message is used for sending signals to descendents so that they can switch their parent link to other nodes if they have active hidden parents.

C. MRAN Algorithm

(a1) of MRAN is similar to (a1) of RAN-DE. Only new code is for resetting sets for replied children, unreplied children, and hidden children. Actions involving those three children sets are common in MRAN. Code in most actions has them. (a2) in MRAN is also similar to (a2) in RAN-DE. The minor difference is in MRAN, sender of late *getCandidate* will not only receive an *alreadyReceived* message, but also be added into the *hidden_parent* set.

Significant revision is made in the upward passing of the *candidate* message, which is algorithm (a3) in both RAN and MRAN. When a receiver node of the *candidate* message finds that the root of the message is missing — the node ID does not exist in *root_set* of current node, it deletes all messages with same root in its incoming queue *in_queue* and outgoing queue *out_queue*. This is usually caused by loss of parent nodes, both active parent and hidden parents. However, (a3) does not handle loss of parents. It is the mobility handler (a5) that performs the frequent regular adjustments.

(a4) handles *alreadyReceived*(*r*, *q*, *u*) message. (a4) in MRAN is similar to (a4) in RAN as well. The difference is (a4) in MRAN will add the sender to the hidden children set.

Key part of MRAN is the mobility handler (a5). It deals with lost branches, hidden parents, and new neighbors. For lost branch, it just processes nodes at direct children level and does not go any deeper. It removes lost children from *replied_children* and *unreplied_children* sets. If the closest candidate node is a lost child, new candidate needs to be selected. For lost hidden parent node, it does the same thing. For the lost children, it first checks if there currently is any hidden parent. If there is, then it shifts from the lost one to any available hidden parent. A selection may be needed when there are multiple hidden parents. New arrival neighbors are naturally treated as children nodes. A *getCandidate* message is sent to each new child.

```
(a5)
End of mobility interval →
Update neighbor set Γ
for each root g ∈ root_set do // Update root_set
  if parent(g) ∉ Γ then lost_parent(g)
for each root g ∈ root_set do  // Update lost children
  for each h ∈ (Γ_l(g) – parent(g)) do
    if h ∉ Γ
    then
      if h ∈ unreplied_children(g)
      then unreplied_children(g) := unreplied_children(g) –
           {h}
      else if h ∈ replied_children(g)
      then replied_children(g) := replied_children(g) – {h}
      else if h ∈ hidden_parent(g)
        remove all messages with h as receiver in in_queue
        remove all messages with h as sender in out_queue
      if candidate(g) = h
      then
        candidate(g) := min {(k – g) MOD maximum}
          //k ∈ candidate_set(g)
        if finished(g) = true
        then send Update(g, candidate(g), u, parent(g)) to
             parent(g)

  // Update new comer
  for each h ∈ Γ do
    if h ∉ (Γ_l(g))
    then
      unreplied_children(g) := unreplied_children(g) + {h}
      send getCandidate(g, u, h) to h
      if finished(g) = true
      then finished(g) := false
[]
```

(a6) and (a7) process *update* message and *definitelyLost* message respectively. See Section V.*B*.

VI. Simulation

The simulator is written in Visual C++.NET 2005. It uses discrete time step as general framework. The code is organized like in AP notion. Actions are triggered by events such as arrival of messages. All nodes are randomly located in a 100×100 meters square. All nodes are powered up at time 0. Neighbor relation is solely determined by radio radius and position of nodes. Connected components are determined by neighbor relation. Some code from RAN simulator is reused. The simulation assume a simple congestion recovery strategy, it may be different as in real world.

A random walk model is used to simulate mobility. All nodes move synchronously. They start a walk at same time and

finish at the same time. The direction is randomly picked up from [0, 2π), the speed is randomly picked up at [0, Speed Limit]. The Speed Limit is usually between 1 meter/second and 100 meter/second. When a node hits the boundary, it is bounced back exactly following rules of classical mechanics. The ideal end condition is the complete formation of overlay ring. However, it becomes very difficult when mobility is present. So approximation option is often used.

Unlike in RAN, primary objective here is not efficiency like time and numbers of messages. Nor is it completeness. Our purpose is to find out the upper bound of mobility, that is, the maximum speed as a scalar quantity.

In the simulation, MRAN works well with various setting of mobility. Its execution is similar to RAN in static environment. The difference is MRAN usually shows more overhead in terms of time and message complexities. The simulation result shows that the upper bound of speed is proportional to nodes in the network. The intuition behind it is the increase in node number reduces the average distance between nodes, and generates larger and denser connected component in which each node has more connection: more children, more hidden parents, and more hidden children.

VII. Conclusion

This paper proposed a Ring Ad-hoc Networks protocol, which is an initialization protocol for ring-based P2P systems over MANETs. It constructs a ring DHT at P2P layer, at same time, connects the ring in lower layer. It is an extension of distributed exhaustive pattern in RAN [1] protocol. It has solved the mobility adjustment problem in RAN. The simulation shows MRAN works effectively in the presence of mobility.

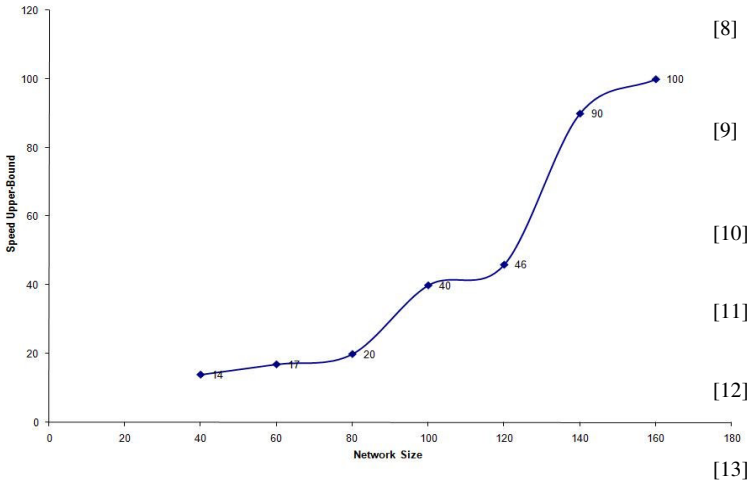

Figure 3. Speed upper bound as a function of network size

This paper analyzed in details various types of disturbances caused by mobility and gave corresponding solution in MRAN. Detailed algorithm change in MRAN is also discussed in comparison with RAN.

This is a new area of research. Many things are waiting to be done, for example, localization optimization in the ring construction, the problem of returning of lost branches, application in other synthetic individual mobility models, application in group mobility models, and utilization of flexible radio range.

References

[1] Wei Ding and S. S. Iyengar, "Bootstrapping Chord over MANETs - All Roads Lead to Rome," in Proceeding of IEEE Wireless Communications and Networking Conference 2007, Hong Kong, China, March 2007.

[2] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," In Proceeding of ACM SIGCOMM 2001, pp. 149-160, San Diego, CA, August 2001.

[3] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica, Hari Balakrishnan, "Building Peer-to-Peer Systems with Chord, a Distributed Lookup Service," In the Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), Schloss Elmau, Germany, May 2001.

[4] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," Proceeding of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), pp 329-350, Heidelberg, Germany, November 2001.

[5] Matthew Caesar, Miguel Castro, Edmund B. Nightingale, Greg O'Shea, and Antony Rowstron, "Virtual Ring Routing: Network Routing Inspired by DHTs," Proc. ACM SIGCOMM 2006.

[6] Y. C. Hu, H. Pucha, and S. M. Das, "Exploiting the Synergy between Peer-to-Peer and Mobile Ad Hoc Networks," in Proceedings of HotOS-IX: Ninth Workshop on Hot Topics in Operating Systems, Lihue, Kauai, Hawaii, May 18-21, 2003.

[7] Himabindu Pucha, Saumitra M. Das, Y. Charlie Hu, "Ekta: An Efficient DHT Substrate for Distributed Applications in Mobile Ad Hoc Networks," Sixth IEEE Workshop on Mobile Computing Systems and Applications, pp. 163-173, 2004.

[8] T. Heer, S. Gotz, S. Rieche, and K. Wehrle, "Adapting Distributed Hash Tables for Mobile Ad Hoc Networks," Proceeding of Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops, pp.173 – 178, 2006.

[9] Mei Li, Wang-Chien Lee, Anand Sivasubramaniam, "Efficient peer to peer information sharing over mobile ad hoc networks," the Second WWW Workshop on Emerging Applications for Wireless and Mobile Access (MobEA'04), New York City, NY, May 2004.

[10] M. Jelasity and O. Babaoglu, "T-Man: Gossip-based overlay topology management," In Engineering Self-Organising Applications (ESOA'05), 2005.

[11] Alberto Montresor, Márk Jelasity, Ozalp Babaoglu, "Chord on Demand," Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05), pp. 87-94, 2005.

[12] Ayman Shaker and Douglas S. Reeves, "Self-Stabilizing Structured Ring Topology P2P Systems," Proceeding of Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05), pp. 39-46, 2005.

[13] T. Camp, J. Boleng, and V. Davies, "A Survey of Mobility Models for Ad Hoc Network Research," Wireless Communication & Mobile Computing (WCMC): Special Issue on Mobile Ad Hoc Networking Research, Trends and Applications, Vol. 2, No. 5, pp. 483-502, 2002.