Relying on Safe Distance to Ensure Consistent Group Membership in Ad Hoc Networks

Qingfeng Huang, Christine Julien, Gruia-Catalin Roman, Ali Hazemi
Department of Computer Science
Washington University
St. Louis, MO 63130
{qingfeng,julien,roman,hazemi}@cs.wustl.edu

ABSTRACT
The design of ad hoc mobile applications often requires the availability of a consistent view of the application state among the participating hosts. Such views are important because they simplify both the programming and verification tasks. Essential to constructing a consistent view is the ability to know what hosts are within proximity of each other, i.e., form a group in support of the particular application. In this paper we propose a protocol that allows hosts within communication range to maintain a consistent view of the group membership despite movement and frequent disconnections. The novel features of this protocol are its reliance on location information and a conservative notion of logical connectivity that creates the illusion of announced disconnection. Movement patterns and delays are factored in the policy that determines which physical connections are susceptible to disconnection. An implementation of the protocol in Java is available for testing.

Keywords
Mobility, ad hoc network, group membership, consistency.

1 INTRODUCTION

Ad hoc mobile networks define a new computing environment that consists of hosts traveling through physical space and communicating in an opportunistic manner via wireless links. In the absence of a fixed network infrastructure, the mobile hosts must discover each other's presence and establish communication patterns dynamically. In some cases the hosts share a single broadcast medium in a limited region of space while in other situations they may act as routers for each other thus extending the range of communication beyond the direct reach of the wireless transmitters. Almost always, communication is assumed to be symmetric even though physically it is easy to envision circumstances in which some hosts may be able to reach much farther than some others.

The absence of a fixed network infrastructure, frequent and unpredictable disconnections, bandwidth limitations, and power considerations render the development of ad hoc mobile applications a very challenging undertaking. Yet, ad hoc networks are emerging as an important platform for new applications of practical importance. Some applications, such as emergency response to a major disaster, must function in situations characterized by a total collapse of the wired infrastructure while others, such as mine surveying, benefit from the flexibility of the ad hoc network structure [4]. Even when the wired infrastructure is available, as the number of devices present in a single room grows into the tens or even hundreds, it becomes infeasible to provide them all with wired connectivity.

Efforts are underway to construct the infrastructure required to support such applications. Standards such as IEEE 802.11, IEEE 802.15, and Bluetooth [2], are being developed to make basic wireless connectivity possible. Protocols are being adapted or redesigned to accommodate the characteristics of wireless communication, to facilitate interoperability with the wired networks, to provide ad hoc routing capabilities [3, 14, 19], and to offer broadcast and multicast functionality. More recently, middleware is being developed for coordination and communication in mobile systems, in an effort to make mobility totally transparent to the application programmer. Typical examples include IBM's TSpaces [11], Sun's Jini and JavaSpace [12], and LIME [15]. TSpaces provides tuple space access and event notification capabilities in a mostly wired environment. Similar reliance on the client-server architecture is found in Jini, which offers support for service registration and discovery. Finally, LIME assumes a coordination perspective that enables application programmers to reduce the effects of mobility to atomic changes in the contents of a virtual global data structure. Its content is dynamically determined by connectivity among hosts and is presented to the application as a tuple space resulting from the union of the tuple spaces residing on hosts in proximity to each other. All three examples use the same tuple space communication strategy as in the Linda model [9]. This is not at all surprising given Linda's focus on content-based accessing and spatial and temporal decoupling among processes.

LIME provided the initial impetus for this research. To con-

struct transparently a global data structure that is dynamically restructured based on the arrival and departure of neighboring hosts, one must be able to determine who is around at any point in time. This is not an entirely new problem. Discovery protocols are routinely used in establishing connectivity among wireless devices. The new element that we bring to this problem is the concern with consistency. From an application perspective, it is desirable in some cases that the data structures presented to the application appear to be the same on all participating hosts, i.e., all operations on the data structures are serializable whether they originate with the application or they are induced by the movement of hosts. A precondition to accomplishing this is the need to maintain a consistent view of the group membership across all the hosts in the group. Ricciardi and Birman [17] introduced a strong group membership protocol to deal with crash failures. It relies on a scheme focused on the majority partition. When a majority partition cannot be constituted, their approach treats all partitions as minority ones and effectively halts system processing. Our approach differs in two important respects. We focus on link failures, and we never halt the system.

The fact that distributed consensus is impossible in the presence of arbitrary link failures [7, 5, 10], a frequent event in ad hoc networks, makes our job particularly difficult. In fixed networks, one way of dealing with link failures and network partitions is to assume that they are benign and short lived and that the system has enough resources to resolve any data discrepancies after the respective failure or partition disappears [6, 1]. In the extreme case of the mobile ad hoc network, connections are transient, partitioned networks may never come together again, and, if they do, there may not be sufficient memory to keep the historical data needed to achieve consistency. Furthermore, lots of ad hoc mobile systems are real-time systems, for instance, a group of vehicles coordinating their route on the highway. In such systems, any intermiadte inconsistency could be very costly. Therefore the retransmission and reconcilation approach for group data consistency is no longer a favorable choice in this ad hoc mobile scenario. The only reasonable option is to hide the mobility-induced link failures. The basic idea is to provide a service that makes mobility totally transparent to the application and renders link failures unobservable to the processes using the service.

In this paper we discuss an algorithm for accomplishing this task in an ad hoc environment. We rely on location information to decide when a host within communication range is admitted to or eliminated from a group. The policy is conservative in nature in order to ensure that all communication activities within a group and the changes in group membership appear to be atomic, i.e., are serializable transactions. The algorithm accommodates both the merging of groups and the partition of one group into multiple disjoint groups. Our membership service assumes a low-level mechanism that monitors the movement and location of the participating processes and prevents any harmful link failure by mandating an announced disconnection (from the group) before a link failure affecting the group could happen. In this way, processes using the membership service in the ad hoc environment can implement consistent, fault-tolerant algorithms under reasonable assumptions about host failures and continuity of movement.

The remainder of the paper is organized as follows. Section 2 provides a formal definition of the problem. Section 3 introduces the concept of safe distance and presents our solution strategy for the group membership problem. Section 4 describes an implementation of our protocol. Section 5 analyzes the relationships between safe distance, network delay and mobile host speed. Discussions and conclusions appear in Sections 6 and 7, respectively.

2 PROBLEM DEFINITION

Our ultimate goal is to provide application developers with the ability to maintain a consistent global data structure in a setting in which mobile hosts come and go as they please and engage in transient collaborative activities. Applications that require this level of consistency are not common today, but with the advent of wireless communication, the situation is expected to change dramatically. Any situation that demands (for legal or technical reasons) the presence of two or more specific entities to carry out a task may impose the need for a consistent membership view. One can envision the futuristic notion of an electronic witness to a contractual transaction or the circumstance in which routine maintenance of commercial aircraft requires secure logging in the presence of an FAA inspector carrying an authorized electronic badge.

In this context, the group membership problem reduces to knowing who is present, willing to cooperate on a specific task, and able to do so reliably. We assume that hosts and links do not fail, i.e., mechanisms exist to overcome the effects of transient failures. The only threat to maintaining a consistent group membership view comes from the mobility of hosts traveling in and out of each other's transmission range. Motion is assumed to be continuous, random, and subject to a known maximum speed limitation. Hosts may shut down intentionally, but, if this happens, they will declare their intention to do so before powering down the transmitter. In the remainder of this section we provide a formal characterization of the problem.

We model a mobile ad hoc network as a graph $C_0 = G(V, E_0)$, where V is the set of mobile hosts and E_0 is a set of bi-directional communication links among the hosts. Graph C_0 changes over time. The presence of an edge (u, v) indicates that host u is within transmission range of host v, and vice versa. We refer to this graph as the physical connectivity graph. In practice, each host can make itself known to its neighbors by generating a beacon at regular intervals and by listening to signals from other hosts around. When

a beacon ceases to be heard, a node is considered to be no longer within transmission range. The frequency of the beacon transmissions determines the accuracy of the information available at each host.

Since any attempt to maintain an accurate picture of the physical connectivity graph C_0 in the presence of unexpected disconnections is infeasible, we introduce the notion of a logical connectivity graph, call it $C = G(V, E)$. The latter is a subgraph of the former. The two share the same set of vertices (hosts) but the logical graph is missing some of the edges (links). The choice of edges to include in the logical graph is determined by some group management policy designed to overcome the difficulties caused by unannounced disconnections taking place in the physical system. The choice of policy is not relevant at this point in the definition but it is critical when it comes to demonstrating the feasibility of the algorithm used to solve the group membership maintenance problem. Before proposing our formulation of the problem we need to define one more concept, the notion of a group. A group G is connected subgraph of the logical connectivity graph C. Since each host u is always a member of some group, we use $G(u)$ to denote the group that includes u, and we extend the notation to $V(u)$ and $E(u)$ to refer to the vertices and edges of the group. Clearly, the logical connectivity graph C is always partitioned into a set of disconnected groups. As the underlying physical connectivity graph changes, so does the logical one. The group management policy is assumed to add a new edge to the logical graph after it appears at the physical level and satisfies certain properties, and to remove it from the logical connectivity graph before it is likely that it might disappear from the physical graph or no longer satisfies certain properties.

The group membership maintenance problem is defined as the requirement for each host in the logical connectivity graph to have knowledge of what other hosts are members of its group and for such knowledge to be consistent across the entire group at all times. Any feasible solution to this problem requires one to make some reasonable assumptions about host movement and network delays, to define a group membership policy, and to develop a protocol that serializes all configuration changes. In this paper, we assume a random mobility model in which a node, at any point in time, can move in any direction at any speed subject to some upper bound V_{max}. We purposely choose this extreme case in order to explore the limits imposed on the membership problem by ad hoc mobility. We also assume that ad hoc routing exists and the network has bounded delay, i.e., a message is always delivered within time t_d if a physical path exists between the origin and destination of the message. Under these assumptions, the requirement of our membership service has the following two specific goals: (1) No message between group members can be lost. (2) Messages are sent and received in the same configuration. We achieve the first goal by employing a policy in which admission to the group is based on location and the second goal by creating a synchronization barrier between successive configuration changes. The basic ideas behind our solution strategy are explained in the next section.

3 SOLUTION STRATEGY

Our ultimate goal is to assist software developers in their efforts to design and build mobile applications over ad hoc networks. The key to our strategy is to provide, at the application level, the appearance of stability in a domain that is characterized by high degrees of dynamic restructuring, caused mostly by frequent disconnections. Even though many different factors can contribute to communication failures, we assume that short-lived transient failures are masked by the communication layer thus relegating all disconnections to the mobility of hosts. In such a setting, the application programmer perceives the configuration (i.e., group membership) to be stable if changes to it are atomic, i.e., cannot affect any operation already in progress. This is why the problem definition requires all messages to be sent and received in the same configuration. Operations that are at a level of granularity above that of message passing would work the same way. One can simply think of two types of transactions that must be serialized: operations issued by the application and configuration changes triggered by mobility. To accomplish this, a system needs the ability to determine the constellation of groups within communication range and under what conditions it is prudent to redefine the way mobile hosts are placed together into groups. A strategy that is too conservative may impede application progress by keeping apart hosts that need to and can work together while one that is too aggressive may make it impossible to preserve the appearance that configurations are more or less stable. A group discovery protocol is used to determine who is around. A reconfiguration protocol enforces the atomicity of configuration changes which include merging groups that are in contact and splitting groups that are being threatened by the possibility of unexpected disconnections. Key to this protocol is the notion of *safe distance* among hosts and groups, i.e., the idea that if hosts are “close enough”, disconnection is not possible and that if they are ”just far enough” there is plenty of time to carry out a configuration change before disconnection occurs. In the remainder of this section we explain the safe-distance concept and present the discovery and reconfiguration protocols.

Key Concept: Safe Distance

Given two mobile hosts equipped with compatible wireless transmitters of equal range R, we state that the distance between them is a *safe distance* if it does not exceed a threshold $r(v, t, t')$, defined as the maximum distance at which one can guarantee that any communication task that takes at most t time units can be completed with certainty, assuming the two hosts move randomly at a speed that does not exceed v, and the upper bound for a single atomic configuration change is t'. Clearly, safe distance cannot be defined

in absolute terms but must be considered relative to a context having certain mobility and application characteristics. For example, in Figure 1(a) mobile hosts a and b are within communication range (R), i.e., they are able to talk to each other directly. They may want to be in the same group and start coordination or resource sharing immediately. Yet, they cannot do so at this point if they wish to guarantee message delivery within the group. This is because a and b can move out of each other's range immediately after acknowledging membership in the same group, with the result being that messages between them could not be delivered successfully. The problem arises from the mobile nature of the hosts and the asynchronous nature of message passing. Our solution is to require a and b to agree on membership in the same group only when they are "close enough", i.e., they are at a distance

$$r \leq R - 2v * (t + t') \quad (1)$$

In this context, t is the upper bound for network latency (because the consistency requirement is reliable message delivery) and t' is the time needed for a group level operation (merge or split) already in progress to complete. The factor $2v$ accounts for the worst case movement pattern, i.e., a situation in which a and b are moving in opposite directions at maximum speed. One can readily see that with this restriction, the reliable message delivery between group members is guaranteed because it takes more than $t + t'$ time for the two group members to become physically disconnected, no matter how they move. Within this time any message delivery completes even if a configuration change is in progress.

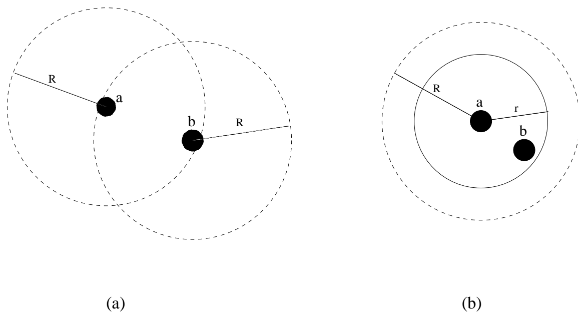

Figure 1: An example of safe distance

We call a group safe if any two members of the group are connected via a path along which all consecutive hosts are at a safe distance. We extend the notion of safe distance from pairs of hosts to pairs of (safe) groups by requiring that at least two hosts, one in each of the two groups, are at a safe distance. While this definition seems to assume that the safe distance is independent of group size, this assumption is generally not true because both simple message delivery and reconfiguration actually depend on the number of hops that messages must traverse en route. Because the time bound on message-passing depends on the group size, our approach works only when the group size is limited by the nature of the application or is constrained by the reconfiguration protocol.

The concept of safe distance is used to determine when two groups can be merged and when a group must be split in order to maintain the requirements for group membership. To find out whether two groups are within safe distance, one has to know the location of all hosts in the region. Since it is too expensive for everyone to keep track of the location of others all the time, we designate a leader for each group to do the job. All hosts in a group constantly report their location to the leader, and the leader keeps the map (Π) of the group, checking constantly to see if the group members are still within safe distance of each other and whether new hosts are present in the region. The map of a group records the spatial location of group members.

Group Discovery Protocol

For a mobile host to join a group, or for a group to merge with another group, it must be able to find out which other groups are present in its vicinity. The discovery protocol carries out this function and serves as a supporting layer for the group membership maintenance protocol, i.e., the reconfiguration protocol. In our discovery protocol, hosts in each group use safe distance as a criteria for finding out who is close enough to be a merge candidate and they report any positive discoveries to their respective leader. Without loss of generality, we assume that every mobile host has a unique identifier (id), belongs to precisely one group and keeps a group member list (π). As mentioned in the previous section, every group has an assigned leader. We choose the identifier (id) of the leader to serve as the identifier for its group (gid) as well. By definition, an isolated host is a group containing itself as a single member. Our discovery mechanism requires every host to periodically broadcast a hello message which contains its location information (xy) and its group identifier (gid). When two groups move close, several members of one group may receive hello messages from members of the other group. When a host u receives a hello message, it checks the sender's group identifier and location. If u finds the sender, say v, to be a member of another group located within safe distance, u passes on the information to its group leader that, in turn, will use it for merge related operations. As all group members are involved in discovery, it is possible for the group leader to receive multiple copies of the same notification regarding the appearance of one host. Duplicates are discarded.

There are several things one can do to reduce discovery costs. First, each host may attach discovery information to its periodic location updates to the leader rather than send them separately. This pushes the discovery information towards the leader almost for free, since the location and new neighbor information represent only a few bytes that fit easily in a single packet. The cost associated with this piggy-backing method is the need for each host to keep a short-term memory (ξ) of newly discovered neighbors. Second, by utilizing neighborhood information already available at the MAC layer, a host may send neighbor greetings only when the

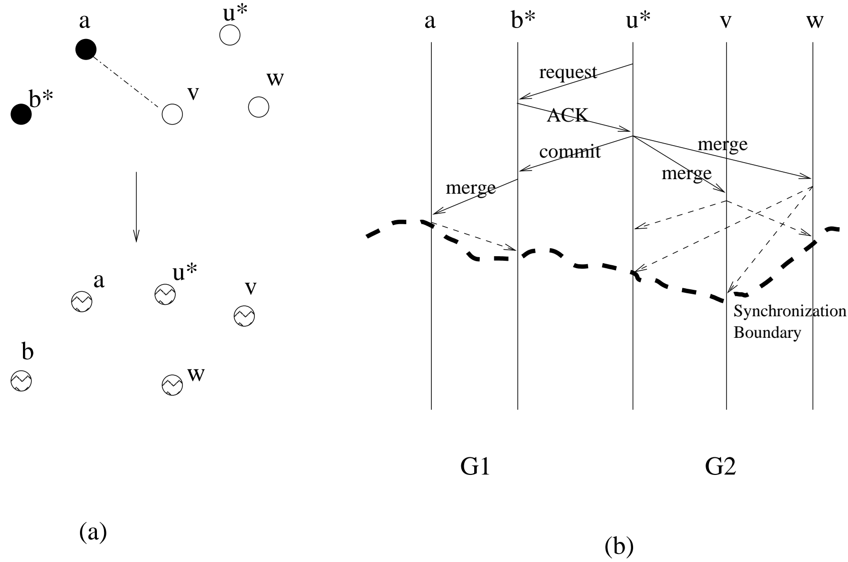

Figure 2: Merging Process

MAC layer discovers a new neighbor. This reduces the discovery cost significantly in the case when the network topology changes infrequently. The drawback for this method is its dependence on the implementation of the MAC layer on the specific host supporting the application. We chose not to do so in our prototype. The group discovery protocol allows the group leader to maintain a list of groups which are close enough to be considered for merging. We present the group merging protocol in the next section.

Reconfiguration Protocol

The reconfiguration protocol is the key layer in our group membership service. It seeks to merge groups in contact and to split groups that can no longer stay together. From the information collected in group discovery, a leader may find that there are one or more potential candidate groups in its vicinity suitable f or merging. If so, it will initiate merging negotiations with the set (Θ) of candidates. Once an agreement is reached regarding who is to participate and who is responsible for coordinating the merger, all affected hosts receive a formal notification about the configuration change from the coordinator. After that, in order to prevent messages sent in one configuration from being processed in a different configuration, all participants must perform a barrier synchronization. One way of accomplishing this is to flush the messages in transit before doing anything in the new configuaration. In addition, the participants need to delay the processing of messages arriving from "future" configurations until the synchronization is completed. Message delaying can be accomplished by tagging each message with a configuration sequence number (τ). Flushing requires the participating hosts to send extra messages whose arrivals guarantee that no more messages originating from a prior configuration are in transit. The result is an atomic configuration change. Another way of creating the synchronization boundary is by using a time-out delay, if message delivery has a time bound. Partitioning works in the same way but without any negotiation because it involves only one group at a time. Next, we use several simple examples to illustrate the merging and partition processes.

An example of merging

Figure 2 depicts the merging process between two groups, G1 and G2. G1 contains hosts a and b, the latter being the leader. G2 contains hosts u, v, w, and has u as its leader. Assume u finds G1 to be in its vicinity through the discovery data sent in by v, and G1 is safe for merging. Next, u initiates the merger by sending a *merge-request* message to b, the leader of G1. If willing to participate in the merger, b sends back an acknowledgment (ACK) along with its group membership list and its configuration sequence number; otherwise, it sends back a disagreement message (NACK), which forces u to abort the merger with G1. If u gets back an ACK, as in Figure 2, it generates a new configuration number by adding one to the larger of the current configuration numbers of the two groups. Next, it sends a *merge-commit* to b and a *merge-order* to its own members. Both the *merge-commit* and *merge-order* messages contain the new group membership list, the new configuration number, and the new leader identity. Upon receiving the *merge-commit* message, b sends a *merge-order* to its own members. A host enters the flushing phase after it gets a *merge-order* message. It sends a *flush-message* to all other members of its original group and stops sending any other messages until it has received all the expected flush-message(s) from its group members in the old configuration. The dashed arrows in Figure 2 and 3 represent flush messages. After receiving all the expected flush-message(s), a host enters the new configuration and all new

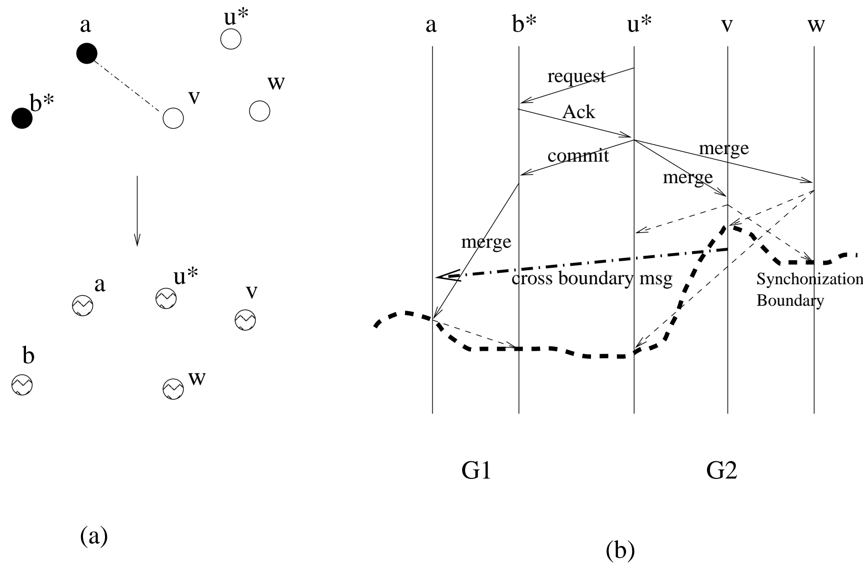

Figure 3: Synchronization and the Configuration Number

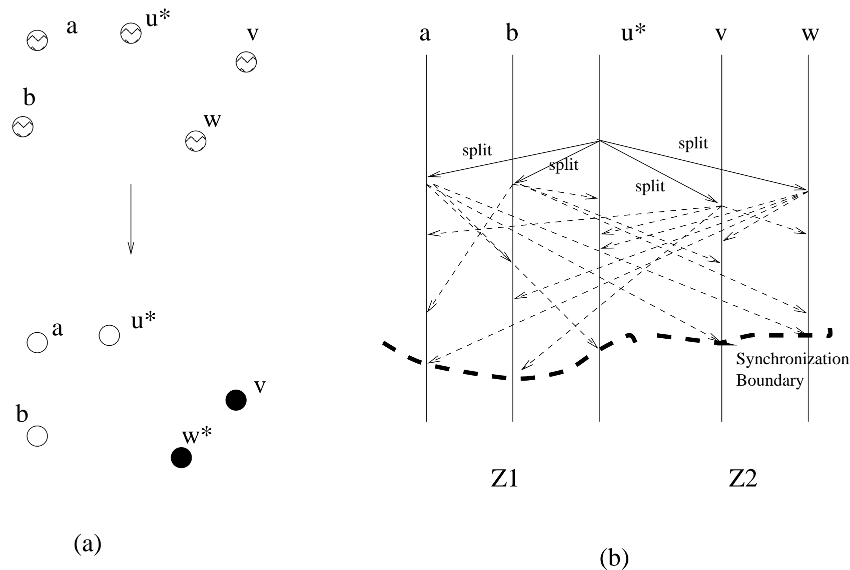

Figure 4: The Partition Process

messages it sends will have the new configuration number in their headers.[1] Clearly, hosts may enter the new configuration at different times. It is possible for a host that is still in the old configuration to receive a message from a host that has already reached the new configuration, as shown in Figure 3. In such cases, the recipient must postpone the processing of this "future" message until the new configuration is established, thus pretending that the message is "received" in the new configuration. Otherwise, the consistency requirement that messages must be sent and received in the same view would be violated. Obviously, implementation of this requires a host to check each received message for the configuration in which it was sent before it is processed.

It is possible for u and b to initiate the merging at the same time. In this case, a tie-breaking mechanism decides who is to coordinate the merger. We use the id as the tie-breaker. The host with the lower id aborts its merger request when the collision happens. Additional complications may appear when more than two groups are involved. For example, u might have entered a merging process with other groups

[1] While the flush mechanism is a straightforward way to achieve the desired synchronization, it is expensive for large groups. In such cases we can replace it with a time-out mechanism, i.e., every host stops sending messages for the duration of one round trip in the network.

when it receives b's *merge-request* message, or it may no longer be a leader because of a merge with other groups or due to a partition process. In all such cases, u replies with a NACK.

An example of partitioning
Figure 4 illustrates the partition process. Assume that two subgroups of the group G, Z1 and Z2, are moving away from each other. By constantly checking the locations of its group members, the leader u is able to identify if its group is in a safe spatial configuration, given predefined distance-based safety criteria. Once the leader u deems the configuration to be no longer safe, it immediately issues a *split-order* message to all the group members. A *split-order* contains three pieces of information: (1) the new leader (gid) for the recipient, (2) the new group membership list for the recipient, and (3) the new configuration number, which is the old configuration number incremented by one. The new leader for each subgroup is determined arbitrarily by u, the old leader. Upon receiving a *split-order* message, a host enters a message flushing phase, similar to the third phase in the merge process. Each host waits until it is sure that all messages sent to it by group members in the previous configuration are received, either by receiving all the expected flush messages or by employing a time-out delay. Each newly assigned leader assumes its leadership role after the synchronization.

The group leader must check its group configuration frequently enough in order to discover any unsafe situation in a timely fashion. As we will see in the analysis section, the threshold for safe distance does depend on the checking frequency, in addition to factors discussed earlier. The example above shows a process in which a group partitions itself into two other groups. In general, a leader might find it necessary to split its group into more than two subgroups in order to preserve the safe distance property. The partition process is the same. Next we explain how the leader determines when the group configuration is not safe and how to split it into safe sub groups.

The split algorithm
To determine if its group configuration is safe, the leader maintains a logical connectivity graph. In the logical connectivity graph, two nodes have an edge of weight one between them if the physical distance between them is less than a partition safe distance (d_p) and no edge between them otherwise. Whenever a new location is reported, the graph is updated by recomputing all the edges to the reporting node. This takes $O(N)$ steps per update, where N is the number of nodes in the group. Given the logical connectivity graph, the depth first search (DFS) algorithm can be used by the leader to determine connected clusters in $O(N)$ steps. So the total time complexity for our splitting algorithm is linear.

Figure 5 shows the support functions used in the protocol presentation that follows. A brief description of each function is included. Figure 6 summarizes the state variables a node needs to keep for the execution of the protocol.

The protocol is presented in Figure 7. The table lists each action taken by host u, the action's precondition, and the action's effect, given the satisfaction of the precondition. There are two types of actions in the system. The first column of the figure shows actions that are triggered by a change in the local state at host u. The second column lists actions that are triggered by the arrival of the message at host u. Each of the actions in the latter group have the form GET MESSAGE. For each of these, there is a corresponding SEND MESSAGE. For example, GET NEIGHBORHELLO at host u is coupled with a SEND NEIGHBORHELLO at another host. The figure shows only the protocol executed at a single host, u, in the system. Each host in the network has its own instance of the actions shown.

Our implementation of the group membership maintenance protocol is discussed in the next section.

4 IMPLEMENTATION

The implementation of the protocol is written entirely in Java. The package's main component is the `GroupMember` object, which contains several threads that control communication between the hosts in the network. Each different type of communication is handled by a different Java thread. These threads coordinate with each other through their owner object, the `GroupMember`. As required by the algorithm, this communication includes beaconing (using a multicast) a hello message, listening for other hosts' hello messages, forwarding discovery information to the group leader, responding to merging and partitioning instructions, and updating the group leader with current location information. Group leaders carry the additional responsibilities of listening to their group members, communicating with other nearby group leaders, and periodically calculating the group's safety.

The group membership package presupposes ad hoc routing

State Variables

id	:node identifier
gid	:group identifier
xy	:node location constantly updated by some external mechanism
τ	:group transaction sequence number
π	:group member list
ξ	:the set of newly discovered leaders
Π	:group map containing all members' locations (empty except for the leader)
Θ	:the set of merging contacts, all of which are leaders of other groups (empty except for the leader)
$Timer$	:monitors the periodic location update

Figure 6: State Variables

Support Functions	
update(ξ, gid);	add gid to the list of newly discovered leaders.
update(Θ, ξ');	update merge contact list Θ with newly discovered leaders from a member's report.
update(Π, v, xy');	update group map Π with group member v's new location, xy'.
MergeSafe(Π, Π', P);	verify that the merger of Π with Π' is safe, according to policy P. P includes safe distance information and the merging status of this group member. For example, if a host is in the process of merging, it is not safe to start a merger.
ClearOldChannels();	clear all group communication channels.
GeneratePartitions(Π, P);	generate partitions for Π, subject to policy P. This function generates a set of triples of the form $\langle \Pi_{new}, \pi_{new}, gid_{new} \rangle$.

Figure 5: Support Functions

with multicast support to be running on every host participating in the network. Therefore many of the messages discussed above are routed through other hosts in the network. As such, the leader of a group need not be directly connected to every member of the group.

The interface to the group membership protocol builds on the `EventObject` and `EventListener` classes in the Java language. An application running on a host that uses the group membership package to participate in groups in the network simply creates a `GroupMember` object. It then registers as a listener to `GroupChangedEvents` generated by its `GroupMember` object. When a new group configuration arises, the group membership package generates a `GroupChangedEvent` that is passed to all registered listeners. The application can take further actions, based on the implementation of this listener.

The `GroupMember` interface allows the user to specify the parameters needed for safe distance calculation. For example, the creator of the `GroupMember` can specify the host's maximum speed and its communication range. In addition to parameters for safe distance, the `GroupMember` creator also specifies the frequency of the hello beacon and the frequency of the group update messages to be sent to the leader.

While the implementation of the algorithm was a straightforward exercise in the use of Java threads and socket programming, some differences worth noting cause the implementation to vary from the examples presented in the previous sections. As presented, the protocol assumes that application level messages and the group membership protocol messages are sent on the same channel. As indicated in the discussion on merging and partitioning, ensuring that messages are received in a FIFO order and that application messages are sent and received in the same group configuration requires some additional work. The example presented in the previous section used flush messages and configuration numbers to accomplish this. The implementation, however, attempts to separate as much as possible the group discovery and maintenance from the application level and therefore leaves the flush messages and configuration numbers presented as part of the example protocol to the particular application. This separation allows each application to choose its own mechanism for ensuring atomicity. Applications with weak consistency requirements may use the group membership package without any atomicity guarantees.

Another concern addressed in the design was the clean separation between the group membership package and the application. By building on a model already integral to the Java language, the simple interface requires only that the application programmer understand the Java event model to successfully use the package. The simple interface composed of a single type of listener and a single type of event provides the desired ease of understanding. Figure 8 shows the public interface of the `GroupMember` object. The constructor accepts parameters for the safe distance calculation. With a handle to the `GroupMember` object, the programmer can start, stop, pause, and resume the `GroupMember` object. These methods affect the running of the threads that the `GroupMember` object uses for communication. The programmer can also add and remove a `GroupChangedListener`. The two final methods are not used often by the application programmer as they are used by other packages necessary for the group membership protocol to function properly. The first method allows a location generating package (e.g. a GPS monitor) to update the physical location of the host. The second method allows the `GroupMember` to respond to beacon events that are generated by a separate beaconing package. These beacons are the multicasted hello messages discussed previously. Figure 9 shows an example usage of the group membership package.

As indicated in the previous sections, this protocol was developed because the LIME middleware requires the ability to transparently reconstruct global virtual data structures to reflect the physical mobility inherent in ad hoc networks. The LIME middleware as released requires a mobile agent or host to explicitly announce its intention to engage or disengage from a group. The integration of this protocol with the LIME middleware transforms the processes of engagement and disengagement into transparent reconciliations of LIME information when agents or hosts move in the network

Actions triggered by changes in the local state

NEIGHBORGREETINGS(u)
  Precondition:
    a new neighbor, v, is detected;
  Effect:
    SEND NEIGHBORHELLO(u, gid) to v;

LOCATIONUPDATE(u)
  Precondition:
    $Timer$ expires or ξ changes;
  Effect:
    reset $Timer$;
    SEND INFORMLEADER(u,xy,ξ) to gid;

MERGE(u)
  Precondition:
    u is the leader;
    Θ contains merging contacts;
  Effect:
    add u's members to a new map ($\Pi_{new} = \Pi$);
    add u's members to a new list ($\pi_{new} = \pi$);
    for each v in Θ
      SEND MERGINGREQUEST(u, Π) to v
      if GET MERGINGACK(v, τ_v, Π_v)
        update group map ($\Pi_{new} = \Pi_{new} + \Pi_v$);
        update group member list ($\pi_{new} = \pi_{new} \cup \{\Pi_v\}$);
        store τ_v;
      if GET MERGINGNACK(v)
        v will not participate in merger;
        remove v from Θ;
    if Θ is not empty
      set τ to the max of all τ_v received;
      for each v in Θ
        SEND MCOMMIT($\pi_{new}, gid, \tau_v, \tau_{new}$) to v;
      for each w in π
        SEND MERGE($\pi_{new}, gid, \tau_{new}$) to w;
    empty Θ;
    update group member list ($\pi = \pi_{new}$);
    update group map ($\Pi = \Pi_{new}$);

PARTITION(u)
  Precondition:
    u is the leader;
    partition predicted based on location updates;
  Effect:
    $\Psi = GeneratePartitions(\Pi, P)$;
    for each $\langle \Pi_{new}, \pi_{new}, gid_{new} \rangle$ in Ψ
      for each w in π_{new}
        SEND PARTITION($\Pi_{new}, \pi_{new}, gid_{new}, \tau$) to w;
    empty Ψ;

Actions triggered by the arrival of a message

GET NEIGHBORHELLO(v, gid)
  Precondition:
    true;
  Effect:
    $update(\xi, gid)$;

GET INFORMLEADER(v, xy', ξ')
  Precondition:
    u is the leader;
  Effect:
    $update(\Theta, \xi')$;
    $update(\Pi, v, xy')$;

GET MERGINGREQUEST(v, Π')
  Precondition:
    true;
  Effect:
    if $MergeSafe(\Pi, \Pi', P)$
      SEND MERGINGACK(u, τ, Π) to v
      empty Θ;
      update safety condition P;
    else
      SEND MERGINGNACK(u) to v;

GET MCOMMIT($\pi_{new}, gid_{new}, \tau_u, \tau_{new}$)
  Precondition:
    u is the leader;
    transaction numbers match ($\tau == \tau_u$);
  Effect:
    ClearOldChannels();
    for each w in π
      SEND MERGE($\pi_{new}, gid_{new}, \tau_{new}$) to w;
    update group id ($gid = gid'$);
    update transaction sequence ($\tau = \tau_{new}$);
    update group member list ($\pi = \pi_{new}$);
    empty Π;

GET MERGE($\pi_{new}, gid_{new}, \tau_{new}$)
  Precondition:
    true
  Effect:
    ClearOldChannels();
    update group id ($gid = gid_{new}$);
    update transaction sequence ($\tau = \tau_{new}$);
    update group member list ($\pi = \pi_{new}$);

GET PARTITION($\Pi_{new}, \pi_{new}, gid_{new}, \tau_{new}$)
  Precondition:
    true;
  Effect:
    ClearOldChannels();
    update group id ($gid = gid_{new}$);
    update transaction sequence ($\tau = \tau_{new}$);
    update group list ($\pi = \pi_{new}$);
    if $u == gid$
      update group map ($\Pi = \Pi_{new}$);

Figure 7: Protocol specification for host u

```
public class GroupMember implements GroupBeaconListener {
     public GroupMember(InetAddress leaderAdd, Location loc,
                        int period, int range, int maxSpeed,
                        int updatePeriod, int networkDelay);
     public void start();
     public void stop();
     public void pause();
     public void resume();
     public synchronized void addGroupChangedListener(GroupChangedListener gcl);
     public synchronized void removeGroupChangedListener(GroupChangedListener gcl);
     public void setLocation(Location newLocation);
     public void newGroupBeacon(GroupBeaconEvent gbe);
}
```

Figure 8: The Public Interface of the Group Membership Package

```
// The test class monitors the changes to a particular group member's group
// An instance of this class runs on each participating host
public class Test implements GroupChangedListener {
     // keep a handle to the group member object
     private GroupMember g;
     // integer count of the number of changes that have occurred
     private int changes = 0;
     public Test(GroupMember g) {
          this.g = g;
          // make this object a listener for events generated by the package
          g.addGroupChangedListener(this);
     }
     // this method is required by the GroupChangedListener interface
     // it is called when a new GroupChangedEvent occurs
     public void groupChanged(GroupChangedEvent gce) {
          // log the receipt of the change
          changes++;
          System.out.println(``Change:  `` + changes);
     }
     public static void main(String[] args) {
          // create a new GroupMember object for this host
          GroupMember g = new GroupMember(InetAddress.getLocalHost(),
                                          new Location(0,0), 1000, 3, 0, 100, 0);
          // create an instance of the Test class to monitor the GroupMember
          Test t = new Test(g);
          // start the GroupMember
          g.start();
     }
}
```

Figure 9: An Example Use of the Group Membembersip Package

thereby changing their status with respect to the protocol's safety requirements.

Because the group membership package is completely independent of LIME or any other application that may use it, changes to the package do not affect LIME, as long as the changes to the package do not affect its interface. This allows for future 'pluggable' versions of the group membership package to replace the current version. One can envision an implementation in which the safe distance is based on something more complex than physical location.

5 SAFE DISTANCE ANALYSIS

The key feature of our algorithm is the use of location information and safe distance in the group membership management. The leader of a group checks frequently the members' locations to make sure that only those that are guaranteed to stay connected with the group for at least $t+t'$ more units of time remain in the group, where t is the time specified by the application layer and t' is the time bound for configuration changes. The combination of t and t' determines the safe distance for a specific operation, which could be the merging operation, the partitioning operation, or any other group operations specified by the application. Let's assume that t_d is the maximum delay between the time a control message is issued and the time it is received and processed, i.e., the sum of the maximum network delay and the maximum process queuing delays both at the sender and the receiver. For convenience, we refer to t_d as the network delay. In the case

of splitting, the maximum time it takes for a group to be partitioned successfully is twice the network delay.

If the leader continuously monitors the group configuration and all member locations are up to date, then mobility-induced unannounced disconnection can be caught in advance and dealt with successfully by requiring $t' > 2 * t_d$. Yet, the leader's information about members' locations is always a little bit out of date. If members sample and report their locations every t_u units of time, the location information the leader has about a member could be outdated by time $t_u + t_d$. Taking this into consideration, the reserved time T must be greater than $t_u + t_d + 2 * t_d = t_u + 3t_d$. Whether or not we can use $d_r = R - 2V_{max}(t_u + 3t_d)$ as the safe distance for partitioning depends on the requirement for merging. Because we do not allow a merging process to be aborted once committed, the computation of safe distance for partitioning also needs to account for the time associated with the merging process. Consider the following scenario: right before a commit in a merging process, the group configuration is safe using safe distance d_r; right after the commit, a leader might discover that its group is no longer safe, and a partion process needs to be carried out immediately. However, the merging process hasn't finished. This is not acceptable. Taking into account that the two-phase merging process needs at most an execution time of $4t_d$ (4 messages), and the configuration needs to be safe right after merging, the total reserved time for both merging and partitioning needs to be $t_u + 3t_d + 4t_d = t_u + 7t_d$. In other words, the safe distance for both merging and partitioning is

$$d_s = R - 2V_{max}(t_u + 7t_d) \quad (2)$$

Using the same distance for merging and partitioning introduces the problem of 'shuttle nodes', i.e., if a node is moving in and out the safe boundary, merges and partitions occur repeatedly. To avoid this, one can further tighten the safe distance for merging, creating a 'buffer zone', and thus reducing the probability of shuttling.

Our algorithm also requires V_{max} to be no greater than V_{adm}, i.e., the maximum admissible speed for the specific wireless network system the mobile hosts are using. Most wireless network systems (e.g. DECT, GSM, PCS, ETACS) have a maximum admissible speed. When a mobile node is moving too fast, it simply becomes invisible to the network. For GSM and PCS, V_{adm} is about $50m/s$; for DECT microcellular system, V_{adm} is about $11m/s$. Without the condition for $V_{max} \leq V_{adm}$, a speed change from $V \leq V_{adm}$ to $V > V_{adm}$ creates an unannounced disconnection. Speed monitoring would be needed to prevent this kind of unannounced disconnection from happening.

Figure 10 illustrates the relation between the safe distance r and the maximum admissible network delay t_d with reasonable values of $R = 150m$, $V_{max} = 10m/s$ and location reporting frequency of 1 Hz ($t_u = 1s$). It shows that as the delay bound increases, the safe distance decreases.

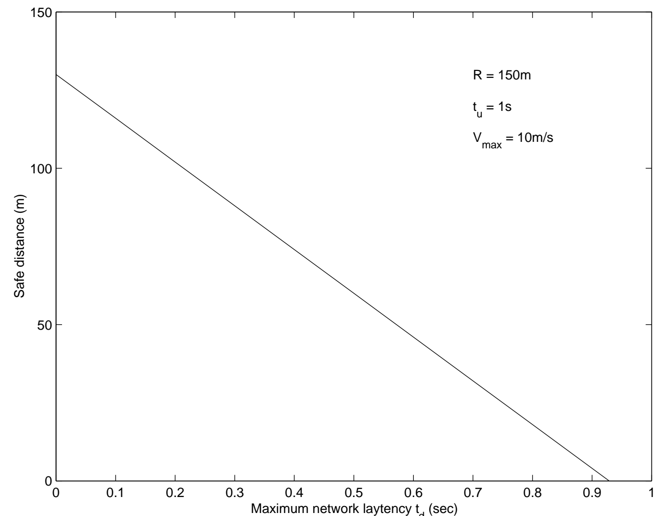

Figure 10: Safe distance vs. network delay

Figure 11 shows the relation between the safe distance threshold, the upper bound on speed, and the network delay bound. The region above the top curve corresponds to $d_s < 0$. In this parameter space we cannot provide any consistency guarantees for a group containing more than one member. On the other hand, if a mobile system's network delay bound and maximum speed bound fall into the region below the ($d_s = 90m$) curve, we could provide the group view consistency guarantee by using $90m$ as maximum safe distance.

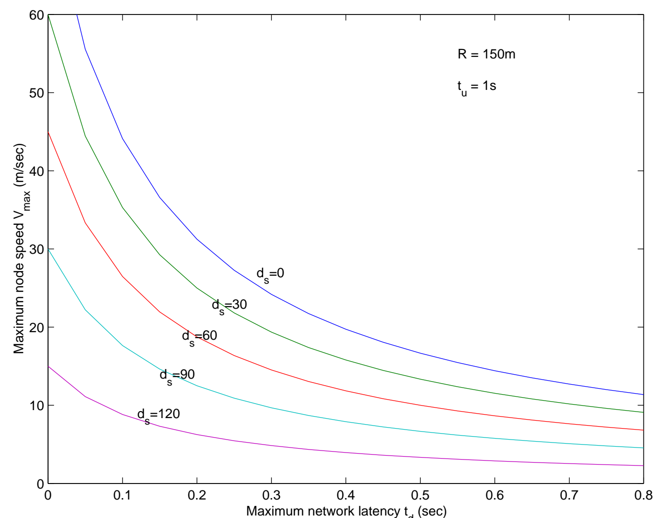

Figure 11: Relation between safe distance, speed bound and delay bound

The correctness of our algorithm relies on the assumption that the network has a delay bound. At this moment, we are not aware of any ad hoc routing protocols which can provide a good delay bound. Yet, it is conceivable that a routing protocol with good delay bound for prioritized group control messages is possible by restricting group size and using location information.

6 DISCUSSION

Maintaining a consistent view of the global state in a distributed network is difficult in general and essentially impossible in the presence of unannounced disconnections. In ad hoc mobile systems, mobility-induced unannounced disconnection occurs frequently, as part of the normal operation of the network. In this paper, we have presented an algorithm that maintains a consistent group membership view in an ad hoc network. The novel feature of this algorithm is its ability to create the illusion of announced disconnection. By using location information about the mobile hosts in the region, the membership service is able to guarantee to the application layer results that are not affected by mobility-induced unannounced disconnection, in the absence of node failures.

Group membership services were traditionally studied [8] for distributed applications running on top of a reliable, usually fixed, wire-line network in which link failures and network partitions are rare. The Transis project [6, 20] has dealt with membership services and group communication in environments in which the network itself may be partitioned due to node and link failures, and nodes may operate for extended periods of time in disconnected mode. The idea of using location information in ad hoc networks is not entirely new. Ko and Vaidya [13], for instance, used location information to improve the efficiency of ad hoc routing. Prakash and Baldoni [16] used location information in the determination of group membership when the network stays connected.

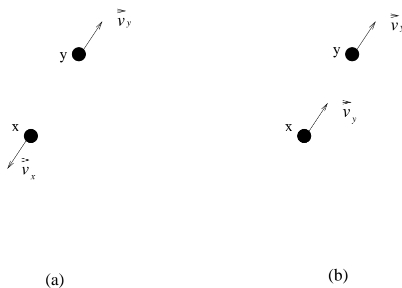

Figure 12: Contribution of velocity information

In addition to being applicable to ad hoc networks, our basic strategy is very general. At present, our algorithm makes the assumption that all mobile nodes in the system have a known maximum speed. Unbounded speed is another possible source of unannounced disconnection. Low speed is a requirement for most wireless networks. In systems involving mobile nodes that can control their own velocity, e.g., cars and planes, a safe relative velocity threshold can be used in the decision of merging and splitting. Of course, in such cases we would have to assume a maximum acceleration for the mobile nodes in order to make disconnection predictions possible.

The quality of our membership service can be improved when velocity information about each mobile host is available. For example, let's consider cases (a) and (b) of Figure 12. In case (a), hosts x and y are moving away from each other, while in (b), they are moving in the same direction. Clearly, x and y are less likely to disconnect in the latter case than in the former. Translating this into the language of safe distance, the maximum safe distance between x and y is greater in case (b) than in case (a). In the current algorithm, as we assume the velocity information is not available, we cannot differentiate cases (a) and (b), so we have to consider the worst case movement scenario for each pair of hosts, i.e., they may be moving away from each other at the maximum relative speed at any point of time. When velocity information is available, as in Figure 12, the safe distance threshold between hosts x and y can change dynamically according to the formula

$$R - |\vec{v}_x - \vec{v}_y| \cdot t - |\vec{a}_{max} \cdot t^2| \tag{3}$$

where $\vec{a}_{max}$ is the maximum acceleration for all hosts and t is the time needed for a group operation already in progress to finish. Simple changes to our algorithm allow us to use the velocity information: (1) each host includes its velocity information in hello messages and location-update messages, and (2) the safe distance is computed using Equation (3) with $t = t_u + 7t_d$. The rest of the algorithm remains unchanged.

Although only safe physical distance is used in our protocol to avoid unannounced disconnection, other physical attributes can also be used to determine safety. For instance, if link failure is predictable through monitoring the bandwidth change between two nodes, a similar group membership protocol can be built by exploiting a similar concept of 'safe bandwidth'.

7 CONCLUSION

The motivation for this work rests with our desire to provide data consistency in applications that execute over ad hoc networks. The idea is to allow software on each host to view and modify data that is distributed across mobile hosts as if it were accessible through a single shared global virtual data structure while the hosts are within proximity of each other. The first step in this direction was to develop the ability to maintain a consistent view of the list of participants in the application, i.e., a group. Despite the presence of disconnections, we were able to accomplish this by drawing a sharp distinction between physical and logical connectivity, i.e., even when connectivity is actually available, group membership is controlled by the kind of guarantees the application demands. This approach represents a new direction in networking, one that factors into protocols information about

mobility and space. This work also provides a practical solution to masking mobility induced unannounced disconnections in ad hoc mobile systems.

ACKNOWLEDGEMENTS
This research was supported in part by the National Science Foundation under Grant No. CCR-9970939. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

The authors wish to thank Amy Murphy for helpful discussions during the development of these algorithms.

A preliminary report on this work appeared in *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*, May 2001 [18].

REFERENCES

[1] O. Babaoglu, R. Davoli, and A. Montresor. Group Communication in Partitionable Systems: Specification and Algorithms. *IEEE Transactions on Software Engineering*, 27(4):308–336, April 2001.

[2] Bluetooth. SIG. http://www.bluetooth.com.

[3] J. Broch, D.B. Johnson, and D.A. Maltz. The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks. Internet Draft, October 1999. IETF Mobile Ad Hoc Networking Working Group.

[4] R. Brooks and J. McLurkin. Using Cooperative Robots for Explosive Ordnance Disposal. http://www.ai.mit.edu/projects/microrobots/. MIT Artificial Intelligence Laboratory.

[5] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Asynchronous Systems. *Journal of ACM*, 43(2):225–267, 1996.

[6] D. Dolev, D. Malki, and R. Strong. A Framework for Partitionable Membership Service. Technical Report CS95-4. The Hebrew University of Jerusalem.

[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of ACM*, 32(2):374–382, 1985.

[8] R. Friedman and R. van Renesse. Strong and Weak Virtual Synchrony in Horus. Tr95-1537, Cornell University, Department of Computer Science, August 1995.

[9] David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[10] J. Gray. Notes on Database Operating Systems. In *Lecture Notes in Computer Science*, volume 60, pages 393–481. Spring-Verlag, 1978.

[11] IBM. TSpaces. http://www.almaden.ibm.com/cs/TSpaces.

[12] JavaSpaces. The JavaSpaces Specification web page. http://www.sun.com/jini/specs/js-spec.html, 1999.

[13] Y.B. Ko and N.H. Vaidya. Location-Aided Routing (LAR) in Mobile Ad Hoc Networks. In *Proc. ACM/IEEE MOBICOM '98*, October 1998.

[14] C.E. Perkins, E.M. Royer, and S.R. Das. Ad Hoc On Demand Distance Vector (AODV) Routing. Internet Draft, October 1999. IETF Mobile Ad Hoc Networking Working Group.

[15] G.P. Picco, A.L. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In *Proc. of the 21st Int. Conf. on Software Engineering*, pages 368–377, May 1999.

[16] R. Prakash and R. Baldoni. Architecture for Group Communication in Mobile Systems. In *Proc. of the IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 235–242, October 1998.

[17] A. Ricciardi and K. Birman. Process Membership in Asynchronous Environments. Tr93-1328, Cornell University, Department of Computer Science, February 1993.

[18] G.-C. Roman, Q. Huang, and A. Hazemi. Consistent Group Membership in Ad Hoc Networks. In *Proc. of the 23rd Int. Conf. on Software Engineering (ICSE)*, May 2001.

[19] E. Royer and C. Toh. A Review of Current Routing Protocols for Ad-Hoc Mobile Wireless Networks. *IEEE Personal Communications*, pages 46–55, April 1999.

[20] R. van Renesse, K.P. Birman, and S. Maffeis. Horus, a flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996.