Views: Customizable Abstractions for Context-Aware Applications in MANETs

Kurt Schellhout
DistriNet - K.U.Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium
kurts@cs.kuleuven.be

Tom Holvoet
DistriNet - K.U.Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium
tom@cs.kuleuven.be

Yolande Berbers
DistriNet - K.U.Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium
yolande@cs.kuleuven.be

ABSTRACT

Programming applications for highly dynamic environments such as mobile ad hoc networks (MANETs) is complex, since the working context of applications changes continuously. This paper presents "views" as abstractions for representing and maintaining context information, tailored to applications in MANETs. An application agent can define a view by declaratively describing the context information it is interested in. A supporting middleware platform, called ObjectPlaces, ensures that the information represented by a view continuously reflects the agent's context information, despite the dynamic situation in a MANET. We elaborate on the distributed protocol that ObjectPlaces uses to maintain the information of views, and give a thorough evaluation.

1. INTRODUCTION

Given the increasing pervasiveness of networks due to the advent of wireless communication, the next generation of distributed systems presumes little network infrastructure, and is comprised of computing devices (nodes) that can be carried around, or are placed on moving vehicles. Mobile computing nodes connected in such an ad hoc way form a mobile ad hoc network (MANET).

An application developed for MANETs necessarily consists of distributed agents that need to communicate among each other to achieve a common goal. Given the dynamics of a MANET, this communication between application agents is complex: the nodes on which these agents live are mobile, and can disappear from a conversation at any time. Dealing with this complexity at the application level is hard, so an application developer benefits greatly from a coordination middleware that offers high level mechanisms to manage communication among agents.

ObjectPlaces is a coordination middleware that supports a first order abstraction of an agent’s context. For our purposes, an agent’s context is the aggregate of all available information on currently reachable nodes in the MANET. An agent’s context thus changes because (1) information on a reachable node is changed, or (2) the set of reachable nodes changes. In ObjectPlaces, an agent can gather context by defining a view. A view is built by the middleware based on a declarative specification that describes "how far" over the network the view reaches, and in what information the viewing agent is interested in.

The most important property of a view is that it is an actively maintained structure, that changes as the agent’s context changes. The agent can listen for new context events, e.g. the arrival of a new node carrying interesting information. A motivating example for such an active notion of context occurred in a real world application, wherein we are currently applying the ObjectPlaces middleware. In this application, automatic guided vehicles (AGVs) are used to avoid collisions (among other things); an AGV thus needs to be notified actively when another vehicle enters in a possible collision range. Other examples are rescue workers that want to be notified of incoming and leaving ambulances, battlefield scenarios where soldiers want to be notified of incoming transports, etc. If the middleware does not support an active context representation, the application programmer is forced to program such a representation using polling.

In the following section, we describe the ObjectPlaces middleware in more detail, as well its relation to existing coordination middleware. In Sect. 3 we describe a protocol to maintain a view in a MANET. Then, the ObjectPlaces middleware and specifically the notion of a view is evaluated analytically and empirically in Sect. 4. Finally, we conclude.

2. VIEWS ON OBJECTPLACES

In ObjectPlaces, each agent can maintain "viewable data" in a local collection of objects called an objectplace\(^1\). The objects in an objectplace represent information that is of interest to other agents, and so contribute to the overall context information available. Other agents can gather copies of objects from objectplaces on nodes in their neighborhood using a view. In other words, context information is made available by putting it in a local objectplace, and can be gathered by building a view. Agents can coordinate by influencing each other’s context, much like what happens in everyday traffic: when a car driver is going to take a right turn, he or she turns on the turning indicator. Other drivers can see this "context change", and react appropriately.

First we describe how an agent can manipulate its local objectplace, then views are described in more detail. The section is concluded with a discussion of objectplaces in relation to existing work.

\(^1\)"ObjectPlaces" - the middleware - is written in plural and with two capitals, "objectplace" - the software entity - is written without capitals.
2.1 Manipulating an objectplace

An objectplace by itself is basically a tuplespace variant: an objectplace is a set of objects that can be manipulated by operations such as put, read and take. Contrary to existing tuplespace approaches however, an objectplace has a fundamentally asynchronous interface: operations return control to the client immediately (an agent that uses the middleware is called a client of the middleware), and results are returned as they are available via a callback. This is important in order to allow views to be built efficiently and conveniently, see Sect. 2.3.

For clarity, we have made two simplifications. First, although an objectplace can be manipulated remotely, for this paper the reader can assume that an objectplace is only locally accessible. Views, discussed in the next section, provide a way for clients to observe the contents of remote objectplaces. Second, although a client can create objectplaces at will, we assume that the middleware offers exactly one objectplace on each node.

Similar to tupsesaces, in order to read or take objects from an objectplace, clients indicate what objects they want to read by means of a template. A template is a function from the set of objects to a boolean value. Every object for which the template function returns true matches with the template.

The operations on an objectplace are the following:

- **put(Set, Callback)** puts the given set of objects in the objectplace. Returns true to the callback if all objects were successfully added.

- **take(ObjectTmplt, Callback)** removes the objects matching with the template from the objectplace and returns them to the callback.

- **watch(ObjectTmpl, EventTmpl, Lease, Callback)** observes the content of the objectplace. Returns copies of objects matching the object template to the callback (the objectplace is not changed). The event template is used to indicate in what events on the matching object the client is interested in. Event template and lease are described in detail shortly.

A watch operation’s event template can match with three possible events: isPresent, isPut or isTaken. If the event template matches with isPresent, the objectplace returns copies of all matching objects it currently contains to the callback, and returns the empty set if there are no matches (this is important for so-called test-for-absence operations, where the client needs to know there are no matching objects). If isPresent is the only event with which the event template matches, the watch operation is finished. However, when the event template also matches with isPut or isTaken, the objectplace must return copies of matching objects that are either put in or taken from the objectplace. In order to do so, the operation is registered by the objectplace, and when an appropriate event occurs on an object matched by the object template, the client is notified. An event template that matches with isPresent and isPut for example, first returns copies of objects currently contained in the objectplace. When new matching objects are put in the objectplace, copies of these are returned in a subsequent call to the callback. Objects returned to the callback are also annotated with the event that occurred on the object, so the client can distinguish between events.

To unregister watch operations that are waiting for an event, a watch can be provided with a Lease object. A client calls the discard operation on the Lease to unregister the corresponding watch operation. The same lease can be given as an argument to more than one watch: in that case, upon discard all these operations are unregistered atomically.

2.2 Views

A view is a local collection of objects, reflecting the contents of multiple objectplaces on reachable nodes in the network based on a declarative specification. This collection is continuously updated by the middleware, both with respect to changing contents of the objectplaces in the view, as with respect to changes in the network topology. A view thus represents the context of the viewing agent.

To build a view, a client specifies:

A distance metric and a bound determines “how far” over the network the client’s view will reach. An example is a hop count metric with a bound of three: the view will span objectplaces reachable in a maximum of three hops from the node where the view is issued.

An object template constrains what objects will be included in the view.

Given these parameters, the ObjectPlaces middleware searches the network for nodes satisfying the constraints (using the protocol described in Sect. 3). On these nodes’ objectplaces, a watch operation with the given object template is executed. The watch’s event template matches with all possible events. The results of all these watch operations, which are events indicating the presence, arrival or removal of objects in an objectplace, are sent to the node issuing the view. This allows the viewing node to keep the view up to date with respect to changes in the content of the participating objectplaces. Changes in the network are handled by managing the watch registrations on the nodes in the view. When it is detected that a node moves out of the view, the view’s watch on the objectplace of that node is unregistered, and the viewing node is notified; when a node moves into the view, a watch on its objectplace is registered and consequently the viewing node is notified as well. How this detection is done is discussed in Sect. 3. A view is actively maintained in this way until it is released by the client. Any client on any node can build a view, and clients can specify as many views as they want. Each client can have its own set of views to observe the context that it is interested in.

A client can not only specify constraints on the content of the view based on its interests, the content can also be represented in a client specific way (e.g. as a sorted collection). In its raw form, a view is built out of the events generated by the watch operations on the participating nodes, as well as events generated by the middleware when a node leaves or joins the view. Users of the middleware can program their own representation using these raw events. For example, we have provided an implementation of a view as a collection, supporting a collection interface common in object-oriented API’s (i.e. java.util.Collection in Java). This allows sorting and iterating over the collection for example. The collection is updated as watch events are generated, e.g. when an isPut is received, the corresponding objects are added to the collection. Usually, a view is described as such a collection in this paper, since it is the most common usage pattern. Other representations of a view can be an accumulation to a single value, e.g. the average of an attribute of the objects in the view, or translating objects in the view to objects the agent understands, to deal with heterogeneous agents. Both the contents and the representation of a view can thus be customized.

As an example, suppose a car arriving at an intersection wants to know the positions of other cars, since the car has to give way from the right. It builds a view on all cars within 50 meters (distance metric and bound), containing position and car id objects (object template). The car chooses a sorted list as representation, closest car first. A traffic monitor nearby also builds a view on nearby
cars, and car id objects, but accumulates the car id’s in a counter for traffic information purposes. In both cases, the content of the view is partly the same (car id’s), but the representation is different (a sorted list, vs. a count of the unique id’s).

In conclusion, a view is a powerful abstraction that can be put to good use for coordination, since it is an active representation of a agent’s context. A view takes the heterogeneous nature of a MANET into account, because both the contents and the representation of a view can be tailored to each client’s wishes.

2.3 Discussion and Related Work

The ObjectPlaces middleware can be seen as a hybrid between two well known approaches for coordination: publish/subscribe systems and tuplespaces.

In a publish/subscribe (P/S) system (e.g. [3, 14]), publishers send notifications of state changes to a list of subscribers. Subscribers are not known in advance but let the P/S middleware know of their interest in certain events through the use of subscriptions. The P/S middleware handles the delivery of notifications to the right subscribers.

ObjectPlaces can be seen as a P/S variant: a view is a subscription to events on objectplaces in the vicinity of a client. Clients “publish events” by manipulating objects in their local objectplace, which triggers watch operations of views that are observing that objectplace. The resulting events are then delivered to viewing clients.

ObjectPlaces solves a problem when applying P/S middleware in MANETs: in [4] it is described that, due to the appearance and reappearance of nodes in a MANET, there is a phase immediately after reconnection in which a mobile subscriber needs to wait until events are published so that the subscriber can assess the current state of publishers. For example in STEAM [9], a P/S middleware for MANETs, publishers are burdened with sending events periodically to account for possible newly arrived subscribers. Generally, this problem is handled by buffering previous events on reliable and fixed infrastructure, e.g. [5] [15]. Such a fixed infrastructure is however not available in MANETs. In ObjectPlaces this problem is handled naturally. Objects are put in a local objectplace; newly arrived clients can be brought up to date immediately by querying neighboring objectplaces using a view. At the same time, the view subscribes to events on the objectplace, and so keeps the viewing client up to date in the future as well. The objectplaces thus act as a buffer in which clients can store observable state.

Tuplespaces (TS) systems, with Linda [2] as a first incarnation, provide a shared collection of data, called tuples, and a set of operations (read, write, take) on the collection to manipulate the data. Throughout the years, Linda has spawned many variants, e.g. [1] [16].

Directly applying tuplespace systems in MANETs is difficult since clients need to know the location of a tuplespace to interact in beforehand, or a shared tuplespace must be discovered and agreed upon at runtime. Since there is no reliable infrastructure available in a MANET, solving this problem is not trivial. In ObjectPlaces, the solution is to let agents manipulate a local objectplace, combined with the use of a view to gather the contents of objectplaces on reachable nodes. From this viewpoint, a view is the necessary “discovery mechanism” that detects reachable objectplaces according to client-specific constraints.

As another point, objectplace operations are specifically designed to support views efficiently. Existing tuplespace approaches lack the event-based, asynchronous interface an objectplace provides. An event-based interface is necessary since a view requires active maintenance, so the viewing agent needs to be notified whenever an objectplace in its view changes. Since a tuplespace provides mostly synchronous operations (e.g. rd, rdp), a tuplespace would have to be polled regularly to keep the view up to date, which is an inefficient solution. With a watch operation, the client can query both the current contents (isPresent) and subscribe to events on the objectplace(isPut and isTaken). This ensures that the objectplace notifies the client when appropriate, thus eliminating the need for polling.

Although existing models do support event-like operations (e.g. JavaSpaces’ notify [16] or LIME’s reactions [10]), these are not sufficient. For example, JavaSpaces notify can only send notifications of incoming tuples, while a view also needs to react to objects that are removed from an objectplace in order to be up to date. Another issue is that an objectplace supports both querying the current contents and subscribing to events atomically: otherwise, a view may miss some objects. JavaSpaces can support this by providing a transaction mechanism, but we feel this is too heavyweight for application in a MANET. Also in programmable or extensible tuplespaces like TuCSON [11] and MARS [1], extending the interface with asynchronous operation is possible, but these mechanisms are too heavyweight for application in MANETs. The objectplace’s watch operation solves the problem by allowing clients to specify an appropriate event template.

Finally, we discuss the relation of ObjectPlaces to a number of coordination middleware approaches specifically for MANETs. LIME [10] is a middleware for mobile agents and mobile nodes that attaches a personal tuplespace to each agent in the application, and shares these tuplespaces transparently when two agents are on the same or connected node. The LIME middleware was extended recently in EgoSpaces [12]. In EgoSpaces, clients can build a view over the locally available network, similarly based on a distance metric. EgoSpaces’ view is always represented as a tuplespace (supporting primitives like in, inp, . . . ). In ObjectPlaces, the representation of the view can be tailored to the client’s wishes (e.g. a sorted collection), thus supporting heterogeneous agents better. Furthermore, while it is possible to register reactions on a view in EgoSpaces, these can react “only” to incoming tuples [7]. It is unclear how an agent can be notified of a tuple leaving it’s view (except by polling), a necessary precondition for building an up to date client-side representation such as in ObjectPlaces.

TOTA [8] (Tuples On The Air) is a middleware that provides applications with the notion of a self-maintaining distributed tuple. Each node in the network hosts a tuplespace. A distributed tuple is propagated to nearby nodes, and can be changed with each propagation according to an application-specific rule (e.g. counting the number of hops from the root). This tuple is then maintained by the middleware as the network changes. An important difference with ObjectPlaces is that a view is specific for every client, while a distributed tuple is the same for all observers. In other words, while in TOTA the “sender” of a message (the agent that adds a distributed tuple) determines both who it reaches and what the content is, in ObjectPlaces the “receivers” of a message (the clients that build a view) can determine both content and representation.

3. VIEW PROTOCOL

The construction of a view in a MANET requires a distributed protocol that is able to find and maintain a set of reachable nodes in the network given a bound on a distance metric. This means that the protocol should be able to: (1) notify the viewing node when a node enters or leaves the view (2) register and deregister watch operations on nodes entering and leaving the view respectively; (3) route the events generated by the watch operations to the node where the view was defined. Existing ad hoc routing protocols (for an overview, we refer to [13]) are not adequate because...
they can only provide a fixed "distance metric" (usually hop count). Since the distance metric used for building the view is application specific and determined at runtime, we need a protocol that does fit our requirements. This protocol can be viewed as an alternative to the one proposed in [12].

We assume from the underlying network layer that: (1) a single-hop broadcast is available that broadcasts a message to all nodes within communication reach, represented by the function broadcast (message); (2) a reliable single-hop unicast that sends a message to a designated node within communication reach, represented by the function unicast(message, id_receiver). We can reasonably expect these functions to be built based on data link standards such as IEEE802.11 [6].

![Algorithm 1](image)

The data structures necessary for the protocol are depicted in Table 1, while the protocol is depicted as Algorithm 1. The protocol builds and maintains a shortest path spanning tree, starting on the node with the client issuing the view, called the root. This tree determines which nodes are included in the view - these are called the participants in the view. In the text and in algorithm 1 the protocol is described as if there is only one view - this is only for expository purposes. In reality this protocol is executed for every view in which a node participates. That is why with each message a unique view id, id_view is transmitted. The data structures in table 1 are also for one single view, and are duplicated for every view in which a node participates.

The length of a path is determined by the distance metric, which is a parameter of the protocol. This allows building a shortest path based on hop count, but also on any other distance metric such as physical distance, or bandwidth. The only constraint on the distance metric is that it increases monotonically the further it gets propagated from the root, to ensure that the distance does not grow out of bounds as it gets propagated in a loop. Such a metric can always be chosen, by including a hop count with any other metric the client chooses (e.g. physical distance) [12].

The view building process starts when a client issues a view. The node on which this client is located is responsible for building and maintaining it, and is the root for that view. The root first builds a unique id for the view, consisting of its own id and a sequence number that is unique on each node. Two activities now occur in parallel: the building and maintaining of the shortest path spanning tree, and the building and maintaining of the contents of the view. The contents of the view is maintained at the root and consists of a list of participants and the objects they objectplaces currently contain.

The spanning tree. To build and maintain the spanning tree, the root and all participants regularly broadcast distance messages. This allows participants to determine what their distance is from the root (d_current), given the view’s distance metric. The root starts the tree-building process by broadcasting a distance message with the id of the new view, its own id and a distance of zero (line 2). Nodes that receive a distance message check whether they are participants in the view. To this end, each participant maintains a list of its neighbors - nodes from which it has recently received a distance message. Upon receipt of a distance message, the participant records the time the distance message was received from that neighbor (function updateTime, l. 16), and the distance that neighbor broadcasted (function updateDistance, l.17). The participant sets the alive variable to indicate it should check whether it is in a view, and starts a timer (if the participant already was in the view, these have no effect).

After every broadcast period, given by t_br, a participant recalculates its distance from the root based on its list of neighbors. First, neighbors for which the timestamp of the distance message that was last received is older than a given timeout, are removed (using removeOutdated, l.16). This timeout is given as a neighbor freshness factor f_br, that is multiplied with t_br to determine the actual timeout. The participant then determines the neighbor that is closest to the root using findParent (l.7). Based on its parent’s distance, it calculates its own distance from the root d_current using the distance metric (l.8). If it is within the bound, it broadcasts a distance message itself (l.10), and repeats the process every t_br time to account for changes in the tree. Otherwise, it goes to sleep, waiting for new distance messages to come in that might change the situation (l.14-15). The end result of this protocol is that each node regularly checks whether or not is in the view, and knows a parent in the tree which is the closest to the root of all its neighbors.

The contents of the view. Each node that is in the view, determines the objects it contributes to the view by executing a watch operation on its local objectplace. The resulting events of this watch are transported to the root using content messages. The events variable is the set of buffered events at the participant that have not been sent to the root yet (we are stretching notation in line 11, since these events are actually sent to callback cb). A node that receives such a content message from one of its children forwards it to its parent, so that it reaches the root (l.21). The root maintains a list of participants of the view. For each content message it receives, it updates the objects that participant contributes to the view (using updateView, l.14). It also records the reception timestamp of each content message (l.3), and removes those participants it hasn’t heard from for f_br t_br time, where f_br is the freshness of the participants (l.1). This means that participants must periodically send a content message to the root, even when there are no events to be returned (events = {} ) to ensure that the root does not consider them out of the view (l.12).

This protocol tolerates mobility of any node and disappearance...
of participants. New nodes are discovered by the periodic broadcasting of the distance messages. Nodes that should be removed from view are detected because they discover for themselves that they don’t have any more neighbors, or their distance has gone out of bounds. Changes in the spanning tree are similarly detected through exchange of distance messages.

Using the spanning tree to deliver events. When sending content messages, the spanning tree is actually used as a multi-hop routing tree to the root. Although the shortest path from participant to root may not always be known, at least some path exists (possibly temporarily containing cycles) and events are delivered as long as every node on the path forwards the content message to its parent.

However, in the case where a node that received an event goes down before forwarding the event, the event is lost. To compensate for this loss, one can choose to use a reliable link protocol from node to root, such as the alternating 1-bit protocol. Each event received by the root would then have to be acknowledged before the node can send another event. Although this makes the protocol reliable, it also induces additional overhead. One possible solution is to use unreliable best-effort event delivery, which may be tolerable for some applications where the view may be slightly incorrect or out of date. Regularly, the view may be “flushed” and current state from all nodes is sent again.

Propagation of other parameters. The parameters of the view, such as the distance metric, also need to be sent from the root to the participants. This was not discussed, but is done in a straightforward way: whenever a node receives a distance message with an idview it does not know, it requests the relevant view parameters from the sender of the distance message. When the node determines it is not in the view, it may delete this information - or decide to keep it for a while longer because it might become a participant in the view later. We do not elaborate further.

4. EVALUATION

The following parameters influence the correctness and the performance of the view: (1) the broadcast period \(t_{br}\), (2) the freshness for neighbors \(f_n\), and (3) the freshness for participants \(f_p\). Instead of letting the application designer choose these experimentally, we mathematically derive bounds on these parameters, supporting the designer in making the right trade-offs.

The view protocol is influenced by uncontrollable factors concerning the dynamics of MANETs. In order to keep the presentation clear, we focus on the dynamics of the network only - in other words we assume for the rest of this section that the contents of the objectplaces stays the same. We focus on this problem because this is where any problems and bad assumptions will be revealed. Specifically, we take into account the number of nodes on a given area and the speed of these nodes. The speed influences how busy the protocol will be updating changes, and the concentration determines connectivity, or the number of nodes in the view.

We study how accurately the protocol can represent the perfect view, which is obtained by “stopping time” and comparing the view the protocol built at that time with how the view should look, given the current position and connectivity if the node is \(v\). There are two kinds of errors: false exclusions, objects that are not in the view but should be; and false inclusions, objects that should not be in the view but are.

4.1 False exclusions

A first way how false exclusions occur is when a node enters the view, while the view didn’t notice the new arrival yet. The critical parameter to minimize this kind of false exclusion is the broadcast period. When this period is short enough, a new node entering the view receives a distance message from a neighbor early, and the view updates fast. Suppose nodes have communication range \(r\), and relative speed \(v\), and we want to detect a node when it has traveled at most distance \(l\) into communication range of any participant. The worst case scenario occurs when an undetected node moves straight at another in the view. In this case, the maximum broadcast period to ensure that a node is detected is: \(t_{br} \leq \frac{l}{v}\), with no transmission delay and no message loss.

Supposing a message can get lost with probability \(p_{loss}\), then the minimum broadcast period to ensure that a node is detected at a distance \(l\) with minimum probability \(1 - p_{loss}\) is: \(t_{br} \leq \frac{l}{v(1-p_{loss})}\), with \(n\) the number of resends until a probability of receiving a message \(p_{detect}\) is reached. From the inequality \(p_{detect} \geq 1 - p_{loss}\), it follows that \(n \geq \frac{\log(1-p_{loss})}{\log(p_{loss})}\).

To incorporate message delays, these bounds should be tighter: one should subtract the message delay from a node to its neighbor and subtract the delay to send the contents from the new participant to the root.

The second way a false exclusion occurs is when the root removes a participant while it should not, because a content message did not reach it in time. We write the time it takes to send a message \(i\) as \(t_{d,i}\) (from the start of sending to the end of receiving), the time of reception of message \(i\) as \(t_{rec,i}\) and the freshness of a participant as \(f_p\). For each two consecutive content messages 1 and 2 it should be true at the root that \(t_{rec,2} \leq t_{rec,1} + f_p - t_{br}\), otherwise a participant is removed in error. After some calculation, this becomes: \(t_{d,2} - t_{d,1} \leq (f_p - 1) t_{br}\). This inequality gives a lower bound for both the freshness and the broadcast period. The inequality shows that the bound is dependent on the difference between two delays only. This means that if the delay increases fairly slowly, the protocol adapts to this increase. Only when the delay suddenly increases with a value of \((f_p - 1) t_{br}\) does a false exclusion result. In other words, the broadcast period and the freshness determine the robustness of the protocol to message delays and congestion.

The upper and lower bound described above show the tradeoff between accuracy and performance. The smaller the broadcast period, the more accurate and responsive the view becomes. However, sending more messages affects the performance, and causes congestion. Choosing a small broadcast period and freshness also causes errors in the view by decreasing the protocol’s tolerance for

<table>
<thead>
<tr>
<th>Root</th>
<th>Participant</th>
<th>Distance msg</th>
<th>Content msg</th>
</tr>
</thead>
<tbody>
<tr>
<td>idview</td>
<td>{idview, timestamp, {object} }</td>
<td>idview</td>
<td>idview</td>
</tr>
<tr>
<td>distance metric, bound</td>
<td>idneighbor, timestamp, distance</td>
<td>idsender, distance</td>
<td>idsender</td>
</tr>
<tr>
<td>object template</td>
<td>idparent, dcurrent</td>
<td>{event}</td>
<td>{event}</td>
</tr>
</tbody>
</table>

Table 1: Contents of data structures and communication messages.
First, the influence of the speed and concentration of the nodes on the number of errors in the view is studied, fixing \( d = 3 \). In figure 1(a) we see the total number of seconds a view is wrong versus the number of nodes on the given area, for different speeds, due to false exclusions and false inclusions. The speed does not influence the error significantly, but the concentration does. This is as expected, because the broadcast period was set to 500ms, which gives the view plenty of time to update given the range of speeds we are looking at. However, we see that the number of views that are wrong due to false exclusions is very high with higher concentration of nodes. The reason for this is congestion: too many nodes in the view mean that content messages are not reaching the root in time, which causes the root to remove participants in error. This is due to the fact that \( f_p = 1 \), which is the most unforgiving value possible (see Sect. 4.1).

To find a good trade-off, we fixed \( v = 0.5 \frac{m}{s} \), \( d = 50 \) and \( N = 25 \), a scenario that gave quite some errors in the previous tests. We also set \( f_p = 2 \), \( f_n \), because as the content messages are multi-hop, they are more susceptible to delay. As can be seen in figure 1(c), the combination of freshness and broadcast period influences the correctness of the view greatly. Good values for this particular scenario seem to be \( t_{br} = 2000 \text{ms} \) and \( f_n = 1 \), or alternatively \( t_{br} = 1000 \text{ms} \) and \( f_n = 2.5 \). As was shown in the analysis, a shorter broadcast period and smaller freshness do not increase the accuracy of the view: on the contrary, they cause congestion which does more harm than good. The improvement obtained by choosing a higher broadcast period alone is remarkable, and stresses the importance of this parameter.

So far, we have looked at the total number of seconds a view is wrong, and have been able to reduce this time from 100% to about 16% of the total duration. Although these numbers don’t look very promising, the criteria for marking a view as wrong are harsh: if only one object is missing, or should be missing but is included, the whole view is “wrong”. However, such a view may still be accurate enough to be used for practical purposes.

In order to know how wrong the view actually is, we measured the total duration a node is falsely in- or excluded, and the average size of the view measured in the number of nodes included per second, for the same runs as the previous experiment. We found that the number of nodes in the view was on average 8 nodes (averaged over time). The number of falsely excluded nodes per second is 3.50. The protocol manages to build the perfect view 84% of the time, and gathers contents from 70% of the content it should gather over time. The number of falsely excluded nodes per second is 3.50. The protocol manages to build the perfect view 84% of the time, and gathers contents from 70% of the content it should gather over time. The number of falsely excluded nodes per second is 3.50. The protocol manages to build the perfect view 84% of the time, and gathers contents from 70% of the content it should gather over time. The number of falsely excluded nodes per second is 3.50. The protocol manages to build the perfect view 84% of the time, and gathers contents from 70% of the content it should gather over time. The number of falsely excluded nodes per second is 3.50. The protocol manages to build the perfect view 84% of the time, and gathers contents from 70% of the content it should gather over time. The number of falsely excluded nodes per second is 3.50. The protocol manages to build the perfect view 84% of the time, and gathers contents from 70% of the content it should gather over time.

The results are workable for applications where network infrastructure is not available (e.g. search and rescue scenarios) or where a best effort approach is tolerable (e.g. calling a cab via a PDA). However, the described protocol is but one way of supporting a view; in the automatic guided vehicle application discussed in the introduction, we are actually using a wireless network with access points in order to improve the reliability of communication. Collision avoidance is an application in which a best effort approach is not adequate. However, the concepts ObjectPlaces offers remain the same and retain their strengths for mobile applications; only the underlying implementation differs.

We do not discuss message overhead, an area in which improvements are possible (especially in the presence of multiple views). We leave this for future work.

5. CONCLUSION

This paper discussed a middleware system for MANETs that provides a powerful abstraction of context to the application. In ObjectPlaces, agents coordinate by building a view on information made available by remote agents in the network. The view is actively maintained, and is client-specific both in contents and in representation. We presented a distributed protocol that maintains such a view in a MANET, and showed acceptable performance re-

### Table 2: Simulation parameters.

<table>
<thead>
<tr>
<th>( l.w )</th>
<th>size of area</th>
<th>100.100m²</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_{sim} )</td>
<td>duration of the simulation</td>
<td>30 minutes</td>
</tr>
<tr>
<td>( r )</td>
<td>transmission range of the nodes</td>
<td>20m</td>
</tr>
<tr>
<td>( N )</td>
<td>number of nodes</td>
<td>variable</td>
</tr>
<tr>
<td>( v )</td>
<td>node speed</td>
<td>variable</td>
</tr>
<tr>
<td>( d )</td>
<td>range of the view in hops</td>
<td>variable</td>
</tr>
<tr>
<td>( t\text{send} - t\text{recv} )</td>
<td>duration of send and receive</td>
<td>50ms</td>
</tr>
</tbody>
</table>

deals. The bounds given accurately characterize the trade-off to make and so support the designer in his or her decisions.
results, mainly focussing on the accuracy of the view. While for the construction of a view we necessarily take a best-effort approach, it was shown that good insight in the working of the protocol helps performance.

An interesting direction for future work is to provide to the application using the view an indication of how good the view represents reality at this point in time - in other words, estimate the accuracy of the view and show this to the application. This estimate can for example be based on a node’s location and a known probability distribution of the nodes in space, or can be learned through experience. This is useful where the accuracy of the view is important. If nodes know the view is probably wrong, they can move more slowly or change the broadcast frequency in order to increase the accuracy.

6. REFERENCES