Middleware for Protocol-based Coordination in Dynamic Networks

Kurt Schelfthout, Danny Weyns, Tom Holvoet
K.U.Leuven - Department of Computer Science - Distrinet,
Celestijnenlaan 200A
3001 Leuven, Belgium
{kurt.schelfthout, danny.weyns, tom.holvoet}@cs.kuleuven.be

ABSTRACT

Pervasive and ad hoc computing applications are frequently deployed in dynamic networks. Due to mobility of the computing nodes, their unreliability, or a limited communication range, at any time a node may enter or leave an interaction between a group of application components. Middleware approaches have been proposed to deal with these dynamics, by supporting the dissemination (or gathering) of information in dynamic networks. In our experience however, applications frequently need to execute a complete protocol to coordinate. Existing middleware can then be used as a discovery mechanism, but offers no support for handling the protocol itself. This paper presents a middleware model that enables an easier implementation of distributed protocols that need to take into account the continuously changing context in the dynamic network. It uses roles as a first order abstraction, handles the distributed instantiation of roles in an interaction session, and maintains the session as nodes in the mobile network move. We describe our experience with applying the middleware in a case study on a system of automatic guided vehicles.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*; C.2.4 [**Computer-Communication networks**]: Distributed Systems—*Distributed Applications*

Keywords

middleware, mobile networks, roles, coordination

1. INTRODUCTION

From the call for papers of MPAC'05 [1]: "Pervasive and ad-hoc computing environments are characterized by the need of applications to be informed about changes in their operating context in order to adapt their operation." The changes in operating context we consider in this paper are those that are caused by the dynamics in the network. Since application components are deployed on computing nodes in a dynamic network, they are exposed to the context changes caused by the network dynamics, and need to be able to deal with them effectively.

A network may be dynamic for many reasons. The nodes may be mobile, and application components are then typically coordinating with a changing subset of connected nodes that catches their interest. For example, a robot avoiding collisions is only interested in coordinating with robots that are close to it. Furthermore, due to the unreliability of the nodes themselves, a node may virtually disappear at any time, e.g. when its battery is down. Another cause of network dynamics is the unreliability or the constraints imposed by the communication infrastructure. For example, in an ad hoc network, senders have a limited communication range, and may move in and out of each other's range at any time.

Helping the application developer deal with these dynamics is the task of an appropriate *coordination middleware*. Existing coordination middleware for mobile networks can be roughly divided in two families: publish/subscribe middleware [12] and tuplespace-based middleware [14] [18].

Both publish/subscribe and tuplespace-based coordination families share two important properties: *(1)* Communication is anonymous. Message sending and receiving is based on declarative mechanisms, rather than the senders' or receivers' address. Typically receivers of a message are determined based on the content of the message and interest expressed by the receiver. This relieves application components of the bookkeeping involved in maintaining an "acquaintance list", which is difficult to maintain especially in a dynamic network. *(2)* Communication is 1 to n. A message sent once, can be received by any number of receivers. Senders can make abstraction of the number of receivers, and vice versa.

Together, these two properties enable application components to abstract from many low level details (concerning e.g. mobility and routing), and generally allow application components to be designed as loosely coupled entities, enhancing their reusability and maintainability.

Typically, both families of coordination middleware are applied in scenarios where information gathering is a key problem, e.g. finding a service such as the closest printer. In such scenarios, coordination among application components is confined to spreading information (e.g. printers regularly publish their current state), or gathering information (e.g. searching the network for available printers).Such application scenarios are a good match for the aforementioned middleware, and much research discusses efficient, reliable and flexible algorithms to support these middleware approaches [5] [17].

[1] `http://www.smartlab.cis.strath.ac.uk/MPAC/`

MPAC '05 November 28- December 2, 2005 Grenoble, France

In contrast, this paper considers a broader class of applications that consist of components that are distributed over the network, and work together as a team to achieve a coherent overall behavior. Coordination is then concerned with maintaining necessary behavioral relationships between application components. This relation needs to be maintained by the exchange of multiple, related messages, i.e. a protocol.

To support protocol-based communication, information spreading and gathering is a necessary, but not sufficient condition. For example, consider collision avoidance between between automatic vehicles. Collision avoidance needs complex coordination mechanisms: it is a mutual exclusion problem, for which a distributed protocol is needed. More generally, examples include unmanned vehicle control, traffic management and control (e.g. cooperating traffic lights, or vehicle collision warning), cooperative applications (e.g. cooperative work applications in an ad hoc community where various forms of consistency or access control are needed) and active network management (e.g. the distributed deployment of a compressor and decompressor in an adaptable network stack).

In such problems, current coordination middleware can basically "only" be used as a discovery mechanism, after which the protocol itself is handled outside the middleware using lower level communication primitives. This has as an unfortunate effect that all support from the middleware for dealing with the dynamics in the network is lost at the protocol level.

This paper argues that coordination middleware can and should provide more support for tackling the class of problems where a protocol is needed to coordinate application components. In Sect. 2, we review existing coordination middleware approaches, and discuss where their support for protocol-based coordination in dynamic networks can be extended. In Sect. 3, we discuss a novel middleware model that supports a first order abstraction of roles, allowing the middleware to set up and maintain separate interaction sessions between application components in a mobile network. After highlighting implementation issues in Sect. 4, we discuss our experience with applying the middleware to a real world problem, automatic guided vehicle control in Sect. 5. Finally, the paper ends with conclusions.

2. STATE OF THE ART

Existing coordination middleware for mobile networks can be roughly divided in two families: publish/subscribe middleware [12] and tuplespace-based middleware [14] [9] [18].

In the publish/subscribe (P/S) family, 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 notifications through subscriptions. Typically, subscriptions are defined by a filter on the notifications' contents. Research has discussed extensions of the principle of publish/subscribe for mobile networks, and for dealing with unreliable connections. In the Java Messaging Service [21], dealing with disconnections is done through durable subscriptions , which are stored and can be activated by re-subscribing. JEDI [4] deals with disconnections and mobility by defining an explicit moveIn and moveOut operator. A fixed dispatching server then buffers relevant subscriptions and forwards them upon reconnection. This is of course problematic in the case where clients do not know whether they are going to move or not. Mobility extensions of Siena [8] follow a similar approach. Elvin [22] offers a proxy server that maintains a connection with publishers on behalf of a mobile client. Clients then need to reconnect to this proxy server in order to receive buffered events. [2] describes that, due to the appearance and reappearance of hosts, there is a phase immediately after reconnection in which a mobile subscriber needs to subscribe to events and wait until some events get fired until it can reassess the current state of the network. Buffers and virtual clients are introduced to subscribe to past events and events in future locations respectively.

In [5] location-dependent subscriptions are introduced to exploit the event-based paradigm in mobile applications. Location-dependency refines a subscription to accept only events related to a mobile user's current location. STEAM [12] also deals with mobility by adding the possibility to filter notifications on the publisher's location, in addition to the contents.

Scopes [6] are an extension for publish/subscribe systems that structures publishers and subscribers by reducing visibility. Only subscribers in the same scope as a publisher can see events from that publisher. A scope can also be dynamic by marking all events in a certain session with a tag - this can be viewed as tagging all the events in an interaction. This work thus acknowledges the problem that interaction protocols are poorly supported in standard publish/subscribe like middleware, and attempts to remedy part of the problem by automatically adding information to events that occur in a certain session. The middleware proposed in this paper takes a significant step forward by allowing the application developer to describe the interaction explicitly in terms of its constituting roles, so that interaction session are naturally separated. Scopes have also been applied to sensor networks, [20], as a generic abstraction for a group of nodes.

In the tuplespace (TS) family, application components manipulate a shared collection of data objects, called tuples, to communicate. Typical operations on a tuplespace are putting, taking and reading tuples. The latter two operations also use a filter (in TS, called a template) on the tuples' contents to determine which tuples to read or take. In a mobile network, a single shared space does not exist, so it is distributed over the nodes. Extensions of TS for mobile networks support additional mechanisms to gather tuples from tuplespaces on remote nodes. For example, LIME [14] makes the tuplespaces on any given node accessible to components on connected nodes. EgoSpaces [9] allows application components to specify exactly from which nodes tuples must be gathered using "views". A view declaratively describes a set of tuplespaces on remote nodes, by specifying a constraint on the location of the nodes. An example is "tuplespaces from all nodes that are less than 20 meters away". An application component can then manipulate the tuplespaces in the view as if it was one shared tuplespace. For example, a component can execute a "take" on a view defined as "tuplespaces from all nodes that are less than 20 meters away". The middleware then transparently searches the network for tuplespaces on nodes matching the view definition, executes the take operation on each of those, and finally returns the result to the application component. ObjectPlaces [18] supports a similar notion of views, but maintains it actively as the topology of the network or as connected tuplespaces change. TOTA [11] takes a different approach: instead of gathering tuples through views, an application component can inject a tuple in its local tuplespace, together with some propagation rules. The TOTA middleware then transparently propagates the tuples to neighboring nodes, and maintains the distributed tuple as the topology of the network changes.

Discussion

These existing middleware approaches, each in their own way, solve the problem of context-awareness in the presence of network dynamics very well. It becomes easy for an application component to remain up to date with respect to its context in the network. As mentioned in the introduction, this is because the middleware *(1)* offers anonymous communication (messages are delivered based

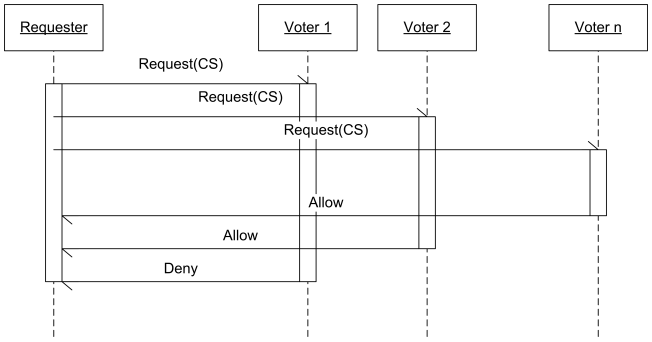

Figure 1: Schematic representation of a mutual exclusion protocol based on voting (in UML).

on their content and the location of the sender or receiver), and *(2)* offers 1 to n communication (one sender can reach any number of receivers, and vice versa). Together, these properties ensure that application components can remain loosely coupled, and for a large part rely on the middleware to deal with the low level issues involved in getting a message delivered to the intended receivers.

However, it is exactly these same properties that entail that current middleware approaches provide little support when dealing with protocols. For illustration, consider a simplified, well known mutual exclusion protocol based on voting [15]. Roughly said, in the protocol a component that wishes to enter the critical section sends a *request* message to all members of a given group. A member replies *allow* or *deny* depending on whether it determines if is safe to enter the critical section or not. The requesting component may enter the critical section if all group members reply *allow*. The protocol is illustrated in the form of an interaction diagram in Fig. 1.

We see three reasons why such a protocol is hard to implement on top of existing middleware for dynamic networks:

1. Protocols often rely on some kind of identification, or at least aggregate properties of their interaction partners. Consider the requesting component: it needs to know which voter already replied, or at least how many voters replied. Publish/subscribe systems for example hide this information, simply raising an event based on the receivers interest without revealing its publisher.
2. Communication is not always 1 to n. For example, when voters reply, this reply is only meant for the requester. Simulating 1-1 communication in middleware that supports 1-n communication leads to artificial constructions, such as the inclusion of a sender id in messages.
3. Protocols are structured as multiple, related messages, and the middleware provides no support for maintaining this structure. For example, if a requester asks to enter several critical sections at once, the requester also has to include a session id with the request message, and voters have to include this session id in their vote. Since the middleware ignores that these messages are related per session, the bookkeeping shifts to the application components.

Although the reappearance of all sorts of ids at the application level may not seem like a big problem, it becomes complex when taking into account mobility. The reason that the middleware is trying to hide the identity of components is exactly to make dealing with mobility easier. The reappearance of the ids is more than a technicality: it shows that the burden of dealing with mobility moves largely back to the application level.

The end result is that, where an application level protocol is desired, existing middleware can serve as an intelligent discovery layer - for the mutual exclusion protocol described above, a publish/subscribe system could be used for a requester to discover dynamically which other nodes it should add to the mutual exclusion group. For executing a protocol however, the application developer cannot use the middleware anymore and must handle the protocol using low level communication primitives. While this can certainly be efficient, much tedious management tasks (such as management of interaction partner ids and session ids) are again shifted to the application layer. Also, there is no middleware support during execution of the protocol to deal with the dynamics in the network.

In conclusion, for complex interaction mechanisms, many low level details the middleware should hide, resurface at the application level. There is thus an opportunity for an extension of the current middleware models, that provides explicit support for protocol-based interaction. We discuss an approach for such an extension in the following section.

3. A ROLE-BASED MIDDLEWARE

Our middleware model is based on the explicit representation of interaction between application components, by supporting the *roles* an application component plays in the various interactions as a first order abstraction.

This is inspired by ongoing work in object-oriented software engineering, where roles are used to model object collaborations. This has lead to usages of roles in framework design [16], implementation support [24], patterns [7] and languages [23].

Roles are also used in protocol-based interaction in multi-agent systems [10] [19], allowing the description of inter-agent interaction separately from agent behavior.

This paper does not focus on the specification or design of roles per se. Rather, the presented middleware focusses on the setup of an interaction session between several roles, played by application components on different nodes, and the active maintenance of the session in the presence of mobility and general dynamics in the network.

First the abstractions role and role instance are introduced. How different role instances interact with each other and with application components on the same node is discussed next.

3.1 Roles and role instances

Role. A role specifies the behavior of one class of interaction partners in a particular interaction. A role describes three aspects of this behavior:

1. The messages a role sends, in particular when the role sends these messages, and to which other role in the interaction each message is sent.
2. The messages a role expects to receive, in particular when the role expects these messages, and from which role the message is expected.
3. The influence of the role on the behavior of the application component that the role represents in the interaction, and vice versa, the influence of the application component on the behavior of the role.

An interaction can be understood in terms of the behaviors of its roles. For example, in the mutual exclusion protocol, two roles can be identified: a requester role and a voter role.

Role instance. While a role describes the behavior of an interaction partner in general terms, a role instance is a runtime entity that

represents a specific component in a particular interaction session. For example, to request a critical section, a requester role instance is interacting with several voter role instances.

Initiator and participant. The middleware supports the application by setting up and maintaining an interaction session between role instances located on various nodes. In order to do this, for every interaction there is one role that is responsible for the lifetime of the interaction session. Such a role we call an *initiator role*. An initiator role is instantiated by the application when a new interaction session is needed. After the interaction session has reached its goal, the initiator role instance can stop the session.

The dual type of role, the *participant* role, is reactive. The middleware instantiates a participant role on a node when an initiator instance indicates that it wants to start a session with a participant role on the node. The participant role instance stays instantiated until the initiator breaks up the session (or the participant goes out of the interaction's zone, see next paragraph).

Zones. An initiator instance (we omit the "role" for brevity from now on) can specify on which nodes participants must be instantiated by declaring (1) the type of participant role, identified by a name, that the initiator expects to interact with; and (2) a constraint on the properties of the nodes on which a participant role should be instantiated, called a *zone*. In general, these properties can change in the course of the interaction session, i.e. they are dynamic properties. For example, the constraint "all nodes whose position is within 20 meters of the local node's position" delineates a zone: position is a dynamic property of a node, and "within 20 meters of the local node's position" is the constraint. Another example of a property is the state of a vehicle (e.g. loaded, charging, ...). The middleware instantiates a participant role on a node if and only if the node deploys the required participant role, and the node lies within the zone.

Dealing with network dynamics. While the session is in progress, the middleware monitors the network, and maintains the appropriate instantiation of the participants as the properties of the nodes change. In particular, if a node enters the zone (i.e. its properties change such that it complies with the constraint of the zone), a participant instance is created on the node (if the node deploys the participant role). Vice versa, if a node leaves the zone, the participant instance is notified that it is outside the zone, and subsequently removed. Both of arrival and departure of participant instances in the zone, the initiator instance is notified as well.

For example, a zone may describe a constraint involving the position of the nodes, by declaring an area in which a participant should be. If a new node enters this area, a participant role is instantiated on the new node, and the initiator instance is notified of the new participant instance. If a node leaves the area, the participant instance on that node is notified that it is outside the zone of the interaction and removed. Subsequently, the initiator instance is notified that the participant instance was removed. The process of automatic instantiation of roles is illustrated in Fig. 2.

Once a participant instance is created, the middleware allows the initiator to send messages to the various participant instances, and vice versa. This allows them to execute the interaction. The middleware does not guarantee that communication between role instances is reliable; either this must be guaranteed by the lower level communication infrastructure, or the interaction protocol is assumed to take unreliability into account.

Cleanup. If the middleware detects that a participant instance is not able to communicate with its initiator instance anymore (i.e. a communication timeout has passed), the participant instance is notified that it is outside the zone and is removed. If the middleware detects that an initiator instance cannot reach one of its participants anymore, the initiator instance is notified that the participant instance is out of the zone.

When the initiator instance signals the end of the interaction session, the middleware notifies all participant instances that the interaction has finished, and subsequently removes the initiator instance and all participant instances on the various nodes safely. If some participants cannot be reached, the middleware on the node at the participant side eventually detects that the participant instance's communication no longer succeeds, and in response cleans up the participant instance as well, as above.

A node can be involved in many interaction sessions at once, both as participant as initiator. Many initiator role instances can exist at the same time, once for each interaction session. For example, a requester role can be instantiated for each new request for a critical section. How these role instances can coordinate among each other, is discussed in the next section.

Relation to groups. As noted by a reviewer, a zone can be seen as the declaration of a distributed communication group consisting of initiator and participant instances. Groups have been studied extensively in "fixed" distributed systems [3], focussing on providing strong guarantees (total ordering, all-or-nothing semantics of messages delivery). In dynamic networks such as mobile ad hoc networks providing such guarantees is difficult, so the emphasis shifts to providing scalable multicast primitives; [13] provides an overview. A zone is a declarative construct on top of a multicast group communication primitive - it relieves the application programmer of having to deal with explicitly joining and leaving groups, and also provides meaningful application level semantics for declaring the group (e.g. based on position instead of a multicast address). The middleware here presented can make good use of group communication primitives, as is discussed further in Sect. 4 concerning implementation.

3.2 Tuplespaces

Since many role instances can be instantiated on a node at the same time, an additional coordination mechanism is needed to coordinate between components and role instances, and between various role instances on the same node. For example, a voter role on a node can not vote "allowed" on a request if it has locked and entered the critical section itself. The voter is dependant on the requester role, and vice versa. Similarly, components involved in the interaction may need to influence a session while it is in progress, and role instances need to influence the behavior of application components. For example, once the mutual exclusion protocol is finished, the requester role returns the result of the protocol to the component wishing to enter the critical section.

To this end, application components share state with role instances in one or more shared tuplespaces. The tuplespace is used by the application components to share state relevant for the interaction with role instances (e.g. a driver puts a requested hull projection in the tuplespace), and vice versa, the tuplespace is used by the role instance to give feedback regarding the outcome of the interaction to the application components (e.g. a requester role instance puts a locked hull projection in the shared tuplespace). Also, a tuplespace can be used to exchange information between role instances of different interaction sessions, since they may be influencing each other. Typically, related roles and components interact in one or more tuplespaces separated from other components and roles that deal with different interactions.

The choice of using a tuplespace is justified by 3 properties:

1. A tuplespace hides the identity of communication partners from each other. This facilitates the dynamic addition of role instances, without "bothering" the application components.

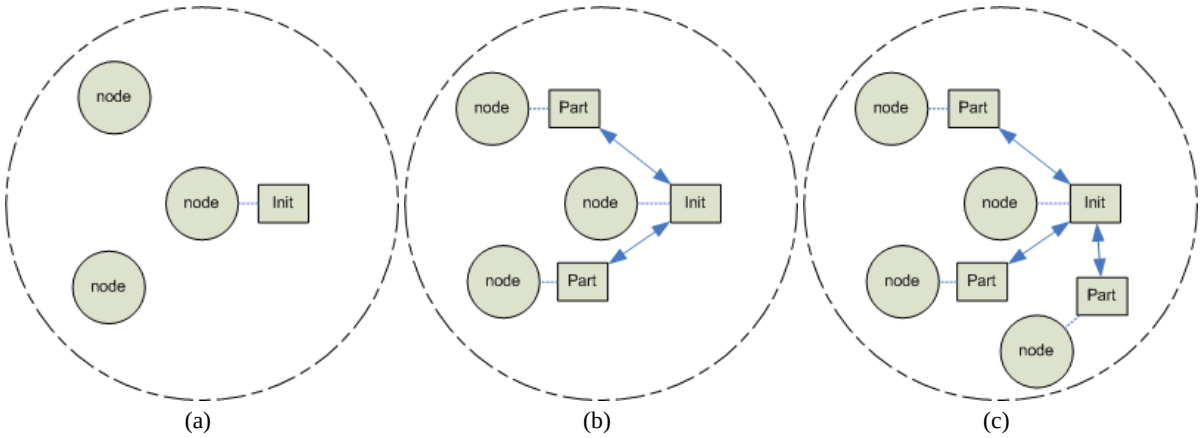

Figure 2: (a) An initiator instance is activated. The circle denotes the zone in which participant roles for this particular interaction should be instantiated. (b) The middleware instantiated the necessary participant roles. The protocol can begin executing between the roles. (c) A new node enters the zone, so a new participant role is instantiated on that node, the initiator is notified of this new participant, and the new participant starts partaking in the protocol as well.

2. A tuplespace provides 1-n communication. Typically, one application component is participating in several interactions at once, so several role instances need to observe the same state.
3. A tuplespace acts as a shared state repository. Since roles can be instantiated at any time, they must be able to assess the current state of the application component immediately after instantiation.

An alternative to a tuplespace approach would be to allow application components to publish events, that roles can subscribe too. This coordination mechanism would have properties 1 and 2, but not property 3: a newly instantiated participant role would have to wait for an event from the application component to be able to assess its current state.

Alternatively, the designer might use his or her own state container to allow roles and components to interact. This is possible, but the tuplespace implementation offers immediate support for e.g thread safety, template matching, ..., which might avoid implementation effort and unnecessary bugs. In any case, the designer must ensure that the necessary state is exposed so that application component and roles can coordinate fruitfully.

4. IMPLEMENTATION

This section discusses how, based on a zone definition, the consistent instantiation of participant roles can be maintained in a mobile network. First a solution is presented that uses an underlying publish/subscribe middleware. Then a specific solution for mobile ad hoc networks is described.

The problem is stated as follows. Given a (generally, arbitrary) constraint, called a *zone definition*, over a set of possible dynamic properties of a node, called *locations*, find the set of nodes with locations that satisfy the zone definition. On those nodes participant roles must be instantiated, and a communication channel between the participants and the node on which the initiator role was instantiated must be set up. We call these nodes resp. *participant nodes* and *initiator node*. The type of a location is called a *dimension*. In general, a constraint is a function over at most 2 locations, viz. the location of the initiator and the location of the participant (e.g. a distance measure).

For example, (2000, 7000) is a location of an AGV in the dimension "physical position in milimeters". An example of a zone definition over this dimension is: the distance between initiator and participant is smaller than 20 meters. Locations need not be related to a position of some sort. Another example of location is the status of an AGV such as loaded, unloaded, charging, etc (unfortunately this is stretching the term "location"). In general, locations should be chosen to support the application as flexibly as possible. Application requirements determine which node properties are important for the selection of interaction partners.

4.1 Using publish/subscribe

Consider a generic content based publish/subscribe system. In such a system, publishers can send notifications, containing an arbitrary content, to a set of subscribers. Subscribers express their interest based on a constraint over the content of notifications, called a filter. In some systems, publishers may advertise the notifications they can publish, allowing more efficient routing of notifications. Properties and implementation strategies for this kind of systems are well known, both for general networks [1], for mobile networks [5] as for mobile ad hoc networks [25].

A publish/subscribe system can be used to solve our problem as follows. Each node regularly publishes its communication address and updated locations in the form of notifications. Nodes that deploy initiator roles with a zone definition that uses a specific dimension, subscribe to notifications of locations in that dimension. Each node maintains a table of (location, communicationAddress) pairs for every dimension in which initiators on the node are interested. Zone definitions are resolved by searching the appropriate table of locations. If a location satisfies the zone constraint, the corresponding node is contacted to instantiate a participant role for the new session. Since each node publishes location updates, each session's participants can be maintained as the locations of nodes change.

The solution assumes that point to point routing is possible in the network. In mobile networks with infrastructure this is generally not a problem. In ad hoc networks, routing protocols are known but the extra overhead on top of a content-based routing protocol is inefficient. We return to this issue in Sect. 4.2.

The approach can essentially be reduced to a multicast approach:

all nodes interested in a specific dimension form one multicast group, and updates in locations of that dimension are multicast in that group. Instead of multicasting the locations, so that each node maintains its own table, an alternative is that one node or dedicated server is appointed for maintaining the (location, address) lists. Setting up a session is then slower, since first the server has to be contacted, but the nodes themselves need less storage space.

4.2 Mobile ad hoc networks

For mobile ad hoc networks, the instantiation of participant instances and communication between initiator and participants can be supported by existing protocols that build a shortest path routing tree or mesh in a mobile ad hoc network. Examples of these are the protocols used for building a view over tuplespaces [9] or [18]. These protocols build and maintain a shortest path tree, according to an application specific distance metric, starting from any node in a mobile ad hoc network. The constraint expressed in the zone then consists of such a distance metric, and a bound on the distance.

Building a view over tuplespaces is similar to instantiation of participant roles based on a zone definition: a view finds tuplespaces on nodes satisfying a constraint, and sends results of operations executed on those tuplespaces back to the node building the view. This is similar to finding nodes, instantiate participant roles on them, and allowing communication between initiator and participants.

The guarantees that can be given in a mobile ad hoc network are much less than in a more reliable network, usually the only guarantee that can reasonably be given is “best-effort”. We refer to [9] and [18] for more detail.

5. CASE STUDY: AGV APPLICATION

In a research project in cooperation with an industrial partner, Egemin, the feasibility of a decentralized approach for automatic guided vehicle (AGV) control is explored. AGVs are unmanned vehicles that are custom made to transport various kinds of loads through a warehouse. An AGV can move by following a physical path on the factory floor, typically marked by magnets or reflectors (for laser navigation). An AGV can pick up a load at a certain location and drop it at another location. An AGV can also park at particular locations when it is idle and charge its battery at a charging station. The main functionality the AGV system should perform is handling *transports*, i.e. moving loads from one place to another. Transports are typically generated by logistic management programs, other logistic machines or operators. More information and demonstration movies of the prototype implementation can be found at `http://emc2.egemin.com`.

Addressing the general problem of AGV control is not the purpose of this paper. Rather a subset of the problem (collision avoidance) is used to evaluate the usefulness of our middleware. So as not to overload the discussion, our exposition of a possible solution is reduced to the essentials. We highlight the connection of collision avoidance with other concerns briefly at the end of the first subsection. The second subsection describes how the middleware is put to good use in this application.

5.1 Collision avoidance

To avoid collision, AGVs exchange information about where they are going to drive to, detect conflicts using this information, and solve the conflict among themselves using a protocol similar to the mutual exclusion protocol described earlier.

In order to detect possible collisions beforehand, AGVs exchange *hull projections*. A hull represents the physical area an AGV occupies, and a hull projection projects a hull over a part of the path the AGV intends to drive on. To be able to calculate the hull projection, the AGV has access to a representation of the physical paths available in the warehouse, called the *layout*. Since the layout is divided into segments (ca. 2-3 meters in length), the length of a hull projection is one segment. When an AGV is near the end of a segment, it projects its hull over the following segment. The set of hull projections of an AGV then marks the physical area the AGV is going to drive on, see Fig. 3 (the circles in the figure are explained shortly). Hull projections are a flexible and precise way to represent the path an AGV is going to follow. A description of that path as a physical area is necessary since collisions can not only occur at intersections but also at parallel paths close to each other. In such situations, it is possible that two “small” AGVs can pass each other, while two “big” AGVs can not.

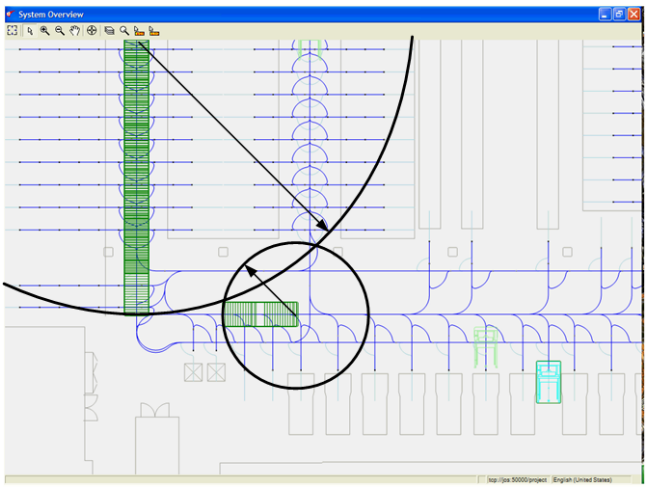

Figure 3: Two AGVs projecting hulls.

The collision avoidance protocol we use is a voting based protocol. Each AGV first requests nearby AGVs whether it may drive over a hull projection, and locks the hull projection if all the nearby AGVs have given their consent (we will define what “nearby” means shortly). An AGV may only drive over a path represented by a hull projection after the hull projection is locked.

At any point in time, an AGV has a set of *requested* hull projections, that it intends to drive over but cannot yet, and a set of *locked* hull projections, that are safe to drive over. The protocol must ensure that the AGVs’ locked hull projections do not overlap.

When an AGV wants to lock a new hull projection, it sends a *request message* containing the requested hull projection to all AGVs nearby. These AGVs then decide whether they give permission to the requesting AGV to drive over the path defined by the requested hull projection. If they give permission, they vote “allowed”, if not, they vote “denied”. An AGV locks the requested hull projection once it has received “allowed” votes from all AGVs nearby. If the request is denied, the AGV waits a random amount of time, and then requests again.

Every requested hull projection contains a priority, and priorities of all hull projections are totally ordered. Now, an AGV that receives a request message (called the *voting AGV*) sends an “allowed” vote to the *requesting AGV* in the following cases:

- The requested hull projection does not overlap with any of the voting AGV’s requested or locked hull projections.
- The requested hull projection overlaps with one or more requested hull projections of the voting AGV and the requesting AGV’s requested hull projection has the highest priority.

The voting AGV votes "denied" in all other cases.

Now, what does "all nearby AGVs" mean? It is desirable to make the set of nearby AGVs as small as possible, since it is not scalable to interact with every AGV in the system. On the other hand, the set must include all AGVs with which the requesting AGV might collide: safety must be guaranteed.

A solution to this problem is shown in Fig. 3. Requesting AGVs interact with other AGVs whose *hull projection circle* overlaps with the hull projection circle of the requesting AGV. The hull projection circle is defined by a center point, which is the position of the AGV itself, and a radius, which is equal to the distance from the AGV to the furthest point on its hull projection. If two such circles overlap, this indicates (to a first approximation) that the two AGVs might collide. This approximation has the benefit that it narrows down the possible candidates for interaction significantly, while each AGV only needs limited knowledge about all other AGVs to determine interaction partners (i.e. position and hull projection circle radius).

We say that all AGVs that satisfy the above constraint are within *collision range* of the requesting AGV. Requested hull projections are sent to all AGVs within collision range. If, during the time in which a request is pending (i.e. request has been sent out, but not all votes have arrived yet), a new AGV comes within collision range, the requester detects this and sends a request to the new AGV. Vice versa, if an AGV that was already requested, leaves the collision range, its vote can be disregarded.

Collision avoidance, as a concern in the AGV control application, depends on and is interwoven with several other concerns, such as routing of the AGVs, deadlock detection and avoidance, and starvation. These are not discussed further.

Note also that several optimizations or extensions are possible for this protocol. For example, as one reviewer pointed out, an AGV may send an explicit "release" message, indicating that it no longer requires a lock on an area - this would relieve another interested AGV from having to do periodic retries.

5.2 Using the middleware

We evaluate the usefulness of the proposed middleware by applying it to the collision avoidance example.

A natural design for our conceptual solution described above uses three components: (1) a driver component that controls the physical movement of the AGV, (2) a requester role that locks hull projections for the driver component by negotiating with nearby AGVs, (3) a voter role that votes on incoming requests. Because roles are a the first-order abstraction in the middleware, this design can be directly supported.

The driver application components' behavior should be coordinated with other driver components so that AGVs do not collide. This coordination is done by the interaction between requester and voter, the two roles in the collision avoidance protocol. Drivers, requesters and voters on the same AGV communicate through a shared tuplespace.

Requester is an initiator role. The driver component instantiates a requester instance for each requested hull projection that needs to be locked, also passing the requester instance the tuplespace in which all locked and requested hull projections for the AGV are kept. The requester instance can now set up an interaction session for the requested hull projection. To this end, it asks the middleware to activate voters on nodes within collision range. As a result, the requester gets a notification from the middleware containing a unique identifier of every activated voter in the session, and sends the requested hull projection to these voters. Each voter, upon receiving the requested hull projection, determines its vote, by comparing the requested hull projection with the locked and requested hulls of its local tuplespace. It sends the vote back to the requester.

If, during the course of this interaction, a new AGV enters collision range, the middleware detects this, instantiates a voter instance on the node, and notifies the requester of the arrival of the new voter. The requester then sends the requested hull projection to the new voter as well. If all votes from the voters currently in the zone have come in, the requester decides whether or not to put a locked hull projection in the tuplespace. If not, it stops the session, waits a random amount of time, then starts the whole process again.

Once all votes are "allowed", the requester stops the session, puts a locked hull projection in the tuplespace, and stops. The driver component sees the locked hull projection, and starts to drive on the path indicated by the locked hull projection.

5.3 Discussion

As can be seen from application to the collision avoidance example, no management of ids at the application level is necessary. The middleware takes care of this by instantiating participant and initiator instances on a per session basis. The initiator can distinguish the different participants by their unique id.

In the example, we did not even need that id. Newly arrived voters are sent a new requested hull message, and an internal counter, that contains the total number of votes is increased by one. If the requester is notified that a voter leaves, is decreased by one. When the number of received votes equals the number of votes sent, the requester can determine whether it has permission to continue.

For this to work, we rely on the fact that the middleware notifies the initiator of every new participant after the participant is instantiated, and that the initiator is notified right after the voter is removed. This makes dealing with mobility easier: nodes can be divided in two classes, those that are of interest and those that are not, and this division is actively maintained by the middleware. This allows the application to abstract from specific dynamic properties of the nodes, such as position.

The presented middleware provides the following application-level support:

1. The designer can define statically what roles should be available on each node: if a node does not deploy a needed participant role for a certain interaction, it never participates in the interaction. Furthermore, interaction partners can be selected on the basis of dynamic properties by the definition of zones.
2. Dealing with mobility is simplified: zones are actively monitored to detect new participants in an interaction session. A protocol can thus be aware of its context and an appropriate protocol step can be made.
3. Bookkeeping of ids by the application is avoided as much as possible, but unique identifiers per session and per interaction partners are generated if needed.
4. Both 1-n communication, as 1-1 communication is possible. Initiators communicate with participants (1-n), while participants can communicate with the initiator (n-1).
5. Interaction sessions are carefully separated. This maintains the structure of the interaction at the implementation level, simplifying interaction implementation.

On the other hand, although the middleware can detect changes in the network, it offers little support to actually deal with this change on the protocol level. For example, in the above protocol safety is guaranteed in the presence of mobility by letting each AGV act as a requester and by taking a safety distance into account for declaring the zone. Thus race conditions, such as an AGV entering a zone just after the protocol is closed, are taken care of at the application level. An opportunity for future work is the addition of

stronger guarantees (e.g. transactional semantics) to make it easier for the application developer to build a correct protocol.

6. CONCLUSION

We believe that, as pervasive and ad hoc computing becomes more mature, there is a growing need for applications that need to use advanced coordination mechanisms. In such cases, of which the AGV case is an example, the only feasible solution is to use distributed protocols to coordinate the nodes. Due to the dynamics inherent in pervasive computing, these protocols need to be aware of their context and able to cope with contextual changes. A middleware that supports the application developer with these tasks is then sorely needed, otherwise he or she is forced to use lower level communication primitives, complicating application design.

This paper presented a novel coordination middleware model, that supports the distributed instantiation of application specific roles to handle a clearly separated interaction session. We showed that it is able to support the design of protocol-based interaction significantly better than existing middleware for mobile computing. Furthermore, we showed the usefulness of the model by applying it in a real world case.

Acknowledgements. This research is supported by the Flemish Institute for Advancement of Research in Industry. Many thanks to Egemin, in particular to Tom Lefever and Jan Wielemans, Jan Vercammen, Jan Peirsman, Wim Van Betsbrugge, Rudi Vanhoutte and Walter De Feyter. We also thank the anonymous reviewers for their comments.

7. REFERENCES

[1] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. on Computer Systems*, 19(3):332–383, 2001.

[2] M. Cilia, L. Fiege, C. Haul, A. Zeidler, and A. P. Buchmann. Looking into the past: enhancing mobile publish/subscribe middleware. In *Proceedings of the 2nd international workshop on Distributed event-based systems*, 2003.

[3] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems Concepts and Design.* Addison-Wesley Publication Corporation, third edition, 2001.

[4] G. Cugola and H.-A. Jacobsen. Using publish/subscribe middleware for mobile systems. *ACM SIGMOBILE Mobile Computing and Communications Rev.*, 6(4):25 – 33, 2002.

[5] L. Fiege, F. C. Gärtner, O. Kasten, and A. Zeidler. Supporting mobility in content-based publish/subscribe middleware. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, 2003.

[6] L. Fiege, M. Mezini, G. Mühl, and A. P. Buchmann. Engineering event-based systems with scopes. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2002.

[7] M. Fowler. Dealing with roles. supplemental information to Analysis Patterns, Addison- Wesley, 1997.

[8] P. Inverardi, M. Caporuscio, and P. Pelliccione. Formal analysis of clients mobility in the siena publish/subscribe middleware. Technical report, Department of Computer Science, University of L'Aquila, 2002.

[9] C. Julien and G.-C. Roman. Supporting context-aware interaction in dynamic multi-agent systems. In *Environments for Multi-Agent Systems, First International Workshop, Revised Selected Papers, LNAI 3374*, 2004.

[10] E. Kendall. Role modeling for agent system analysis, design, and implementation. *IEEE Concurrency, Agents and Multi-Agent Systems*, 8(2):34–41, 2000.

[11] M. Mamei and F. Zambonelli. Self-maintained distributed tuples for field-based coordination in dynamic networks. In *The 19th Symposium on Applied Computing (SAC 04)*, 2004.

[12] R. Meier and V. Cahill. Exploiting proximity in event-based middleware for collaborative mobile applications. In *Proceedings of the 4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'03)*. Springer-Verlag Heidelberg, Germany, 2003.

[13] P. Mohapatra, C. Gui, and J. Li. Group communications in mobile ad hoc networks. *Computer*, 37(2):52–59, 2004.

[14] A. Murphy, G. P. Picco, and G.-C. Roman. Lime: a middleware for physical and logical mobility. In *Proc. of the 21th International Conference on Distributed Computing Systems (ICDCS-21)*, May 2001.

[15] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1):9–17, 1981.

[16] D. Riehle and T. Gross. Role model based framework design and integration. In *Proceedings of the 1998 Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1998.

[17] G.-C. Roman, C. Julien, and Q. Huang. Network abstractions for context-aware mobile computing. In *Proceedings of 24th International Conference on Software Engineering*, pages 363–373, 2002.

[18] K. Schelfthout and T. Holvoet. Views: Customizable abstractions for context-aware applications in MANETs. In *Workshop on Software Engineering for Large-Scale Multi-Agent Systems*, 2005.

[19] E. Steegmans, D. Weyns, T. Holvoet, and Y. Berbers. A design process for adaptive behavior of situated agents. In *Agent-Oriented Software Engineering V*. Springer-Verlag, 2005.

[20] J. Steffan, L. Fiege, M. Cilia, and A. Buchmann. Scoping in wireless sensor networks: A position paper. In *Proceedings of Workshop on Middleware for Pervasive and Ad Hoc Computing*, 2004.

[21] Sun Microsystems, Inc. Java message service spec. 1.1, 2002.

[22] P. Sutton, R. Arkins, and B. Segall. Supporting disconnectedness - transparent information delivery for mobile and invisible computing. In *Proc. of CCGrid*, 2001.

[23] T. Tamai, N. Ubayashi, and R. Ichiyama. An adaptive object model with dynamic role binding. In *Proceedings of International Conference on Software Engineering*, 2005.

[24] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 359–369, 1996.

[25] H. Zhou and S. Singh. Content based multicast (cbm) in ad hoc networks. In *Proceedings of the 1st ACM international symposium on Mobile ad hoc networking & computing*, pages 51–60. IEEE Press, 2000.