Chapter 5
Middleware for Distributed Multi-Agent Systems

One of the major challenges in the software development of a distributed multi-agent system is the coordination necessary to align the behavior of the agents. Since coordination determines whether agents cooperate effectively, it has a direct impact on the satisfaction of a distributed application’s functional requirements. Furthermore, since coordination is realized primarily by communication, coordination has a large impact on quality attributes such as efficiency and resource usage.

Decentralization of control implies a style of coordination in which the agents cooperate as peers with respect to each other, and no agent has global control over the system or global knowledge about the system. As a result, complex interactions are necessary to achieve consensus since there is no single agent that can make a centralized decision. In the case of mobile applications, agents have to take into account the distribution of the nodes in physical space and other properties of the environment, which add extra complexity to the realization of coordination. Since development of distributed multi-agent systems is difficult, usually middleware is used to support the application developer.

We start this chapter with introducing middleware support for distributed systems and multi-agent systems in particular. Then, we explain in detail a concrete middleware that was developed for the case study and we illustrate how this middleware supported a complex coordination problem in a mobile setting. The chapter concludes with a summary.

5.1 Middleware Support for Distributed, Decentralized Coordination

We give an overview of the role of middleware for supporting the development of distributed systems. First, we zoom in on the multiple layers of middleware in distributed software systems in general. Then, we take a closer look on middleware for multi-agent systems.
5.1.1 Middleware in Distributed Software Systems

Over the last decade, the development of software systems increasingly emphasizes the reuse of software components. There is an ongoing trend away from programming applications from scratch to integrating them by configuring and customizing reusable components and frameworks [145]. Requirements for greater reuse in developing distributed software systems motivate the use of middleware-based architectures. Middleware is software that resides between the application and the underlying operating systems, network, and hardware. Middleware shields software developers from low-level tedious and error-prone platform details. It provides software developers with a consistent set of higher level abstractions and services closer to the application requirements. Figure 5.1 shows the multiple layers of middleware in distributed software systems [145].

Host Infrastructure Middleware encapsulates communication with the operating system. Widely used examples are the Java Virtual Machine and the .NET platform. Distributed Middleware defines higher level distributed programming models with reusable APIs and components that help programming distributed applications. Examples are Java Remote Method Invocation, Common Object Request Broker Architecture (CORBA), and SOAP that provide a simple XML-based protocol allowing applications to exchange structured information on the Web. Common Middleware Services define higher level domain-independent services that support

![Fig. 5.1 Middleware layers with surrounding context [145]](image-url)
programming of application logic such as transactional behavior, security, and database access. An example technology is Enterprise Java Beans that enables software developers to link predefined services ("beans") without having to write much code from scratch. Finally, Domain-Specific Middleware Services are tailored to the requirements of a particular interest group. Examples are middleware services for telecom, electronic commerce, and grid computing. Today, domain-specific middleware services tend to be less mature partly due to the lack of common middleware standards which are needed to provide a stable basis to create domain-specific services [145].

As distributed software applications have to deal with increasing dynamics and heterogeneity, software must be dynamically composed and adapted at runtime. A major trend in middleware is to combine domain-specific middleware functionality with specific component frameworks (e.g., JEE, .NET, etc.). This approach enables the construction of applications from independently developed third-party components and integrate built-in services covering nonfunctional requirements of a distributed application such as persistency and security. A typical example is service-oriented architectures [9] where the major part of application development boils down to assembling domain-specific services that comply with a set of declaratively specified policies. The complexity of flexible composition and runtime adaptation of services in the face of the crosscutting nature of functionality is the subject of active research [26, 95].

5.1.2 Middleware in Multi-Agent Systems

We now look at how typical middleware support for multi-agent systems maps on the different middleware layers:

- Distributed and host infrastructure middleware. Multi-agent system engineers generally consider distributed middleware services (RMI, CORBA, SOAP, etc.) as a basic platform to build multi-agent systems. The services provided by the bottom layer are not the main focus of research on agent environments but are typically considered as given infrastructure.

- Common middleware services. In multi-agent system development, common middleware services such as security, persistency, transactions are often considered minimally. For lab prototypes, there is a tendency not to consider these domain-independent services. Since the number of deployed multi-agent systems is rather limited, there is little experience with integrating common middleware services in multi-agent systems. Some platforms provide basic support for particular common middleware services such as Retsina [156] (security, monitoring, and logging) and Living Systems of Whitestein Technologies [171] (among others, transactions, persistency, and Web service access).

- Domain-specific middleware services. Support for agent interaction such as communication services for message exchange and infrastructures for coordination are part of the domain-specific middleware services layer. These infrastructures
are built on top of the distributed middleware platform and comprise programming abstractions and services that can be reused across multi-agent system applications [62]. Almost all agent platforms offer some form of domain-specific middleware service. The types of support are very different and include support for distributed message communication such as Jade [23], electronic institutions [54], artifacts [137], pheromone infrastructure [35], and infrastructures based on tuplespaces [113, 106]. Some examples of more specific approaches are delegate multi-agent systems [72], tag-based interaction [128], and communication filters [144].

Domain-specific middleware can help multi-agent application developers by simplifying and accelerating common development tasks [146]. Middleware simplifies application development by offering programming abstractions that hide lower level details from the application developer. It accelerates application development by encapsulating generic, reusable functionalities to support the programming abstractions. In particular, middleware encapsulates the tedious management tasks associated with distribution. As such, middleware offers conceptual and technical tools to support the application developer in dealing with the distributed aspect of the multi-agent system.

5.2 Case Study

The case study gives an extensive description of a domain-specific middleware for multi-agent systems and its application to the AGV transportation system. This middleware, called ObjectPlaces, supports the development of distributed, decentralized applications that are deployed in a mobile network. We start this section by characterizing the target systems of the middleware and derive requirements for the coordination middleware. Then, we introduce the basic building blocks of the middleware: objectplaces, views, and coordination roles. In the two following sections, we give a description of the software architecture of the middleware and explain how we have applied ObjectPlaces to solve the coordination problem of collision avoidance in the AGV transportation system.

5.2.1 Scope of the Middleware and Requirements

The ObjectPlaces middleware targets mobile applications with the following three characteristics:

1. Context Awareness. The applications have a strong connection with their context and actively need to take their context into account when coordinating. Typically, coordination solutions are expressed in terms of the current context properties of
5.2 Case Study

application components,\(^1\) in particular with respect to a components’ interaction partners. For example, to execute a transport from a particular location, an AGV is selected among the AGVs within a range of 30 m.

2. \textit{Dynamics}. The applications are subject to unexpected dynamics originating from the environment. These dynamics may be the result of the mobility of the nodes or of other changes in the application’s context. As a result of dynamics, and the need for application components to be aware of changes in their context, application components need to be aware of the changes in interaction partners. For example, AGVs may move in and out of collision range of a particular intersection.

3. \textit{Decentralization}. The applications we consider consist of distinct application components that cooperate as peers to reach the overall goal of the application. No single component has global control over or knowledge about the system. Decentralization of control typically increases both the importance and the complexity of coordination in the application.

These characteristics and the associated problems motivate the following requirements for middleware for mobile applications:

1. \textit{Discovery of Interaction Partners by Properties}. Interaction partners should be discovered based on their properties, such as location of a node, status of the node. The identification of interaction partners should be expressed by using a declarative constraint on node properties.

2. \textit{Management of Changes in Interaction Partners}. The supporting abstractions should allow the middleware to encapsulate the management of the group of components with which a particular component interacts, thereby removing this burden from the application developer.

3. \textit{Decentralized Architecture of the Middleware}. The middleware should not introduce a centralized element in its architecture, as this would make the middleware unusable for decentralized applications.

In addition, the middleware should be efficient, i.e., it should consume a reasonable amount of bandwidth. We do not consider the middleware’s overhead in computing space and time: bandwidth is the scarcest resource.

\textbf{5.2.2 Objectplaces}

Objectplaces are essentially containers of data objects. Objectplaces are not meant to be used by themselves, but the two main abstractions, views and coordination roles (explained in the following sections), are both used in conjunction with objectplaces. Hence, it is important to gain a basic understanding of objectplaces before explaining views and coordination roles.

\(^1\) We use the term application component in its general meaning, i.e., a modular and independently describable entity that is part of an application. An AGV local virtual environment is an example of an application component.
5.2.2.1 Conceptual Model

An objectplace is a collection of objects that can be safely manipulated by concurrent processes using operations such as put and read and is as such a variant of a tuplespace [42]. The main motivation for developing a specific tuplespace variant is the need for asynchronous operations. Typical tuplespace operations are synchronous, i.e., a read operation reads a tuple from a tuplespace and blocks until the tuple is available. Due to the dynamic conditions in a mobile network, an asynchronous interface is needed. Objectplace operations return control to the caller immediately, and results are returned when they are available via a callback. This allows an event-driven style of interaction with the objectplace, which in the case of synchronous operations should be handled using polling.

Objectplaces can be created by application components. Each node maintains its own set of objectplaces, each of which can be given a name unique on the node. An objectplace can be accessed by other application components using its name. This is summarized in Fig. 5.2.

An objectplace by itself is not accessible from nodes other than the node on which it is created. Instead, views and coordination roles are used as a structured way to access and manipulate objectplaces on remote nodes.

5.2.2.2 Basic Operations

The three basic operations of an objectplace are put, take, and watch. These three operations add objects to, remove objects from, and observe objects in the objectplace, respectively. All three operations are asynchronous: an application component that executes an operation does not wait for the result, but gives a callback as a parameter. When the objectplace has processed the operation, it returns the result of the operation to the callback. Multiple results may be returned over time.
An object place is thread-safe: multiple concurrent application components can use the same object place safely.

In more detail, the three basic operations on an object place are represented as the following methods:

- **put(Collection, Callback)** adds the given collection of objects to the object place. When finished, the value true is returned to the callback if all objects were successfully added and false otherwise.
- **take(ObjectTemplate, Callback)** removes the objects matching with the template from the object place and returns the matching objects to the callback.
- **watch(ObjectTemplate, EventTemplate, Lease, Callback)** observes the content of the object place and returns copies of objects matching the object template to the callback according to the given event template.

An object template is a function that takes a set of objects and returns a boolean value. An object for which the object template returns true is said to match with the object template. For the watch operation, application components can select which events are returned using an event template. An event template is a function from the set of possible events to a boolean value. Supported events are isPresent, isPut, isTaken. The watch returns all events for which the event template given by the caller returns true, i.e., a sequence of \((event, collection)\) pairs are returned to the callback, where \(event\) is one of the supported events and \(collection\) is a collection of objects. A **Lease** serves to unregister watch operations. An application component uses the lease to discard the watch for which the lease was given as argument.

In addition to the basic operations, an object place offers one extra operation, **executeAtomically**, to allow the execution of a series of basic operations atomically. For a discussion of this composed operation we refer to [146].

### 5.2.3 Views

In this section, we describe **views**, the first abstraction supported by the Object Places middleware. Views enable coordination of application components based on information exchange. Application components declaratively specify in which information they are interested. The middleware builds a view by collecting the required information from object places on remote nodes and maintains the information as nodes move in or out of the view and as the information on remote nodes is changed by other application components.

#### 5.2.3.1 Conceptual Model

A view is a collection of objects that are copies of objects in object places on connected nodes in the network. The middleware builds and maintains a view based on a declarative specification given by an application component. The specification
Fig. 5.3 Conceptual model of views

determines the objectplaces that are to be included in the view and the objects that are gathered from those objectplaces. Figure 5.3 shows the concepts and their relations in a conceptual model.

The model shows that application components can use any number of objectplaces to share objects in, and each application component can have any number of different views to observe objects with. The objects in a view are gathered from objectplaces from one or more nodes. These objectplaces or nodes contribute to the view.

The applications are decentralized, i.e., they consist of a set of application components that cooperate as peers. Therefore, the middleware is built as a set of decentralized, cooperating middleware components. Each node hosts one middleware component, which is responsible for providing the necessary services to the application components on that node and coordinating with other middleware components in order to guarantee the middleware’s functionality.

Views and objectplaces contribute to the realization of the requirements for middleware for mobile applications (see Sect. 5.2.1) as follows:

- **Allowing Context Awareness.** A view is built by the middleware based on an application-specific constraint on the nodes and objects that should be gathered in the view. A view is thus a representation of the information in the network that is of interest to an application component. Views allow application components to select the information they currently need declaratively.

- **Dealing with Dynamics.** By allowing the application to specify the information it wants to gather by means of a constraint on node properties, the view can be
maintained by the middleware. The application does not need to be concerned with managing changes in interaction partners, as views are kept up-to-date by the middleware.

5.2.3.2 View Management

The main access point of the middleware to start and stop a view is the View Manager. To request the construction of a view, an application component uses the `startView` operation that is provided by the view manager. The operation requires three parameters:

1. A **node constraint**, which determines from which nodes in the network the objects in the view are gathered.
2. An **objectplace name**, which determines from which objectplace the objects in the view are gathered.
3. An **object template**, which determines which objects are gathered from the objectplaces on the nodes determined by the previous two constraints.

The node constraint determines which nodes are to contribute to the view based on node properties. Node properties are application-specific properties of a network node, e.g., a node’s position. More precisely, a node constraint is a function that takes as arguments the current values of the properties of the viewing node and the current values of the properties of a candidate viewed node, and it returns true if the candidate viewed node should contribute to the view, and false otherwise. The two arguments enable the expression of constraints relative to the viewing node. For example, a view on all nodes within a certain distance from the viewing node needs a node constraint that is a function of both the viewing node’s position and the other node’s position.

Given the parameters of a view request, the middleware searches the network for nodes satisfying the node constraint. On these nodes, the objectplace whose name is the same as the given objectplace name is found. If the objectplace exists, the objectplace contributes to the view. If the objectplace does not exist, the node does not contribute any objects to the view. This mechanism implies that the objectplace’s names are known to all application components building a view and that objectplaces are present during the lifetime of the application.

If an objectplace is found on the node, a watch operation is executed on the objectplace by the middleware component on the viewed node. The watch operation’s event template matches with all events. The results of all these watch operations are events indicating the presence, arrival, or removal of objects in an objectplace. The middleware component on the viewed node sends the events to the middleware component on the viewing node.\(^2\) In this way, the middleware

\(^2\) The *viewing node* is the node on which an application node has requested that a view be built. The *viewed nodes* are the nodes that contribute to the view built on the viewing node.
component on the viewing node can keep the view up-to-date with respect to changes in the content of the viewed objectplaces.

Changes in the viewed nodes are handled by the middleware by managing the watch registrations on the objectplaces in the view. Only objects from objectplaces on nodes that satisfy the node constraint remain in the view. When a node moves out of the view, the watch operation on the objectplace of that node is unregistered, and the viewing node is notified that the node moves out of the view. All of the objects that were sent from that node are removed from the view. When a node moves into the view, a watch on its objectplace is registered and the viewing node is notified of the arrival of the new node in the view. Results from the watch operation are sent and the view is updated.

In order to allow the middleware to build and maintain views based on node properties, an application or the middleware on each node maintains node properties for that node in the middleware. Node properties are name–value pairs and may be the result of a sensor readout on the node, e.g., position or another observable property of the node. The middleware imposes no constraint on the form of the values, so they can range from an integer to a complex XML description.

A view is actively maintained by the middleware until it is stopped by the application component. The view manager provides the `stopView` operation to stop a view.

### 5.2.3.3 Quality of Views

Two important quality attributes that an application developer needs to know about view building and maintenance are as follows:

- **Reliability.** A perfectly correct view at all times is impossible: at least a transmission delay needs to be taken into account to send the necessary update information to the viewing node. Reliability determines how well the view reflects the actual contents of the objectplaces contributing to the view.

- **Efficiency.** There is overhead associated with building a view, both computation and communication overhead. Resources used by the middleware cannot be used by the application. Efficiency determines how much overhead the middleware uses to offer its services.

Improvement of reliability is usually at the expense of efficiency: a more timely view needs more updates and more communication.

There is much variation in the quality of mobile networks. At one end of the scale, unpredictable and unreliable mobile ad hoc networks are connected without any network infrastructure besides each node’s own network card. On the other end of the scale, wireless LAN networks are supported by access points to relay and amplify communication signals and achieve higher levels of reliability.

An in-depth discussion of implementation strategies for view building and maintenance is out of the scope of this book. Schelfthout [146] discusses two different implementation strategies for different deployment environments. The first implementation strategy describes a protocol for reliable and higher bandwidth wireless
LANs. The second implementation strategy describes a protocol to form views in unreliable mobile ad hoc networks. For each of the implementations, quantitative statements are discussed for two quality attributes: reliability and efficiency.

5.2.4 Coordination Roles

We now describe coordination roles, the second abstraction supported by the ObjectPlaces middleware. Coordination roles support the application developer with the design and implementation of dynamic protocols in mobile networks. A coordination role is an abstraction representing the behavior of a component in a protocol. Coordination roles allow the middleware to take over the management aspects of executing a protocol, i.e., the initial discovery of interaction partners in the network and the detection of changes in interaction partners during execution of the protocol. Such management is a main problem of coordination in mobile networks.

5.2.4.1 Conceptual Model

The concepts related to coordination roles and their relations are presented in Fig. 5.4.

A coordination role is an abstraction that encapsulates the behavior of one application component engaging in a protocol. A coordination role instance is a runtime instance of a coordination role. One coordination role can have many coordination role instances at the same time. When a coordination role instance is executing a protocol on behalf of an application component, the component plays the coordination role.

An interaction session is the exchange of a series of messages in a protocol by a group of coordination role instances played by distinct application components. An interaction session is always started by one application component that starts to coordinate play a role by instantiating the coordination role. A coordination role instance that starts an interaction session is called an initiator. Coordination roles played by components in the interaction session that participate in an interaction session started by an initiator are called participants.

5.2.4.2 Interaction Setup and Maintenance

For the middleware a coordination role is a black box, a unit of behavior that is played by an application component when it is involved in an interaction session. The middleware supports the setup of interaction sessions and the maintenance of the group of coordination role instances in the interaction session as node properties change.

The main access point of the middleware to start and stop interaction sessions is the Role Activator. To start an interaction session, an application component can
use the `startInteraction` operation that is provided by the role activator. This operation requires three parameters:

1. An initiator role, which is instantiated by the application component.
2. A node constraint, which specifies the group of nodes on which participants have to be instantiated.
3. The name of a participant role, which will be instantiated on the nodes that satisfy the node constraint.

The node constraint allows constraints based on the individual properties of each participant node and constraints based on relations between properties of initiator node and participant nodes.\(^3\) The former enables the specification of a constraint that compares an arbitrary combination of node property values from a single participant node with a constant value, e.g., to select participant nodes based on their status. The

---

\(^3\) The **initiator node** is the node on which an application node starts an interaction session using an initiator role. The **participant nodes** are the nodes on which a participant role is activated that participates in the interaction session.
latter enables typical constraints based on differences between properties of nodes, e.g., to select nodes based on distance between initiator and participant nodes.

For all nodes that satisfy the node constraint, the participant role with the given name is instantiated, but only if there are application components on that node that are capable of playing the participant role. To that end, an application component should register the names of the participant roles that it is capable of playing on the node on which the application component is deployed. Whenever a coordination role enters the interaction session, initiator and participants are notified. When a participant enters the interaction session, an asynchronous communication channel is opened between initiator and participant, so that the protocol can be executed.

When a coordination role is instantiated (initiator or participant), the middleware generates a unique identifier that can be used to refer to that role. The initiator and the participants use the `receive` operation to receive messages from a coordination role in the interaction session. To send a message to a participant an initiator uses the `sendToParticipant` operation. The middleware uses the role identifier of the participant to deliver the message. To send a message to all participants, the initiator uses the `sendToParticipants` operation. Participants can send a message to the initiator using `sendToInitiator`.

To support this continuous change in interaction partners, the middleware continuously monitors the node properties and maintains the instantiation of participant roles on the appropriate nodes. Two events can occur in a group of role instances engaged in an ongoing interaction session. First, the properties of a node that is not in the group change, and as a result its node properties satisfy the node constraint. A new participant role is instantiated on the node, and the initiator of the interaction session is notified. The initiator can then take the necessary actions to incorporate the new participant in the interaction session. Second, the properties of a node that is in the group change, and as a result its node properties no longer satisfy the node constraint. The initiator of the interaction session is notified that a participant will be removed. The participant on the node to be removed is notified, so it can clean up. Then, the participant is removed from the interaction session by the middleware. Evidently, only protocols which are able to deal with addition or removal of interaction partners are supported.

In order to allow the middleware to set up a group of coordination role instances based on node properties, the application or middleware on each node maintains node properties for that node in the middleware. Similar to views, node properties are name–value pairs and typically the result of a sensor readout on the node.

The maintenance process continues until the interaction session is closed by the initiator. The role activator provides the `stopInteraction` operation to close the interaction session.

### 5.2.4.3 Group Membership Guarantees

Regarding the setup and maintenance of interaction sessions, the arrival and removal of a participant in a group are notified to the initiator with a best-effort guarantee.
The update frequency of the node properties and the delay imposed by the underlying communication medium determine the granularity of group updates. The application can control the update frequency of node properties, taking into account that more updates are likely to cause more overhead. Since the middleware guarantees group updates with the same frequency (i.e., the middleware handles every update to node properties), the application can choose the update frequency such that application requirements are met.

For example, if on every node, the node’s position is updated every second, node constraints based on position are updated about every second as well (taking into account jitter on communication delay). In case of mobile nodes, based on the maximum speed, an upper bound can be calculated on the distance a node can travel between two updates. This upper bound can be used to calculate the bounds of the area in which a node is located; this in turn may be important at the application level, e.g., for collision avoidance.

In case a node failure occurs for some reason (hardware or software fault, battery down, etc.), a node is no longer able to communicate. Such failures are in general difficult to handle. In mobile networks with a reliable infrastructure, i.e., with access points, it can be assumed that communication is reliable. A failure detector can then be put in place in order to detect if a particular node cannot be reached anymore. Such a failure can then be relayed to initiators that are in an interaction session with the failed node, as a specific failure event. In this case, the initiator can thus distinguish between a node simply moving out of range and a node that fails. Typically, a failing node requires special measures in a protocol than nodes that move out of range. For example, a protocol that needs to avoid collisions between moving vehicles needs to know whether a vehicle has moved out of collision range or has failed and is still standing approximately at its last known location.

5.3 Middleware Architecture

In this section, we give an overview of the software architecture of the ObjectPlaces middleware. We present the high-level module decomposition of the middleware. Next, we explain group formation, the basis module of the middleware. Then we zoom in on view management and role activation.

5.3.1 High-Level Module Decomposition

Figure 5.5 shows the high-level module uses view of the ObjectPlaces middleware situated in its context. We summarize the responsibilities of the different modules in turn.

Group formation is the backbone of the ObjectPlaces middleware, providing support for (1) the discovery of groups of nodes that satisfy a node constraint and (2)
the maintenance of this group in the face of changes in the properties of the nodes in the network.

Group formation modules on the various nodes use their local set of node properties to determine how the group is formed based on a node constraint. For example, if a group needs to be formed using a distance constraint, the set of node properties on each node contains the node’s current position. Node properties are maintained in an objectplace.

The group formation module supports star-formed group formations, where a single leader node forms a group with multiple members. The leader can communicate with all the members of the group, and the members can communicate with the leader. Consequently, the leader is notified of changes in membership of the group, and the members are notified when they join or leave the group. Any number of groups can be formed, and a node may participate in any number of groups simultaneously, both as a leader and as a member.

Star-formed group formation supports both view construction and coordination role activation. For view construction, a view is requested on one node and gathers data is gathered from a number of other nodes in the network. For coordination role activation, an initiator coordination role is activated on one node and interacts with participant coordination roles on other nodes in the network. In both cases, a group is started on one specific node at the initiative of the application, and there are multiple other nodes in the network that need to be part of the group. In neither of the two cases participants need to communicate among each other.

View management provides the service for building views. To build a view, an application component specifies a node constraint and an object template to determine which objects are gathered for the view. The view manager uses the group
formation to form and maintain the group. The view manager on the leader node is responsible (1) to send the necessary information to the view managers on the member nodes and (2) to build and maintain a view for the application component based on the data received from the members. The view manager makes the view available for application components in an object place. The view managers on the member nodes are responsible (1) to collect matching objects for the view and (2) to notify the view manager on the leader node whenever the situation with respect to the view on the member node changes.

**Role activation** provides the service for a protocol-based interaction. To start an interaction session, the application component supplies an initiator coordination role that the application component will play in the interaction, a node constraint, and the name of the participant coordination role to be activated. Role activation uses the group formation to find the nodes belonging to the group and keep informed about changes to membership. The role activation module on the node that started the session is responsible (1) to contact the member nodes to activate the desired participant coordination roles and (2) to inform the initiator role when a participant leaves or a new participant enters the interaction session. The role activation modules on the member nodes are responsible (1) to activate the participant coordination role if available and (2) to notify the initiator node when a participant leaves the interaction session or a new participant enters the interaction session.

**Design Rationale**

The main functional requirements of the ObjectPlaces middleware are the management of views and the management of coordination roles of interaction sessions. With respect to view management, the middleware must be able to build and maintain views in the face of network dynamics based on a node constraint and additional data such as an object template. With respect to management of coordination roles of interaction sessions, the middleware must be able to activate and deactivate coordination roles on the appropriate nodes in the network based on a node constraint and the name of the coordination role that should be activated. These two requirements show that the problem common to both is the resolution of a node constraint to a group of nodes whose properties satisfy the node constraint. This functionality is provided by group formation which is responsible for forming and maintaining a group of interacting nodes. Each of the basic functionalities is encapsulated in a module providing separation of concerns.

Besides functional attributes, the quality of group formation is the major influencing factor on the overall quality of view management and role activation. Important qualities of group formation are reliability and performance. Reliability is a measure of the guarantees that can be accomplished with group formation. Reliability measures how up-to-date the group is and how fast the group is changed in response to changes in the network and node properties. Performance measures the overhead associated with group formation, in particular communication overhead. The main influencing factors on the quality of group formation are the characte-
tics of the network. Chapter 7 zooms in on the efficiency and bandwidth usage of the middleware for the AGV transportation system.

### 5.3.2 Group Formation

The group formation module is the backbone of the middleware, providing

1. The discovery of a group of nodes in the network that satisfy a node constraint.
2. The maintenance of this group in the face of changes in the properties of the nodes in the network.

As explained above, the group formation module supports star-formed group formation, see Fig. 5.6.

A single *leader* node (the viewing node or the initiator node) forms a group with many *members* (the viewed nodes or the participant nodes). The leader can communicate with all the members of the group, and the members can communicate with the leader. Consequently, the leader is notified of changes in membership of the group, and the members are notified when they join or leave the group.

To explain the working of group formation (and view management and role activation in the following sections) we use a communicating processes diagram. Communicating processes show a system, or a part of a system, as a set of concurrently executing units and their interactions. The elements of the communicating processes diagram are concurrent units, repositories, and connectors. Concurrent units are an abstraction for more concrete software elements such as task, process, and thread.

---

**Fig. 5.6** Schematic representation of a group. The leader communicates with the members. On the *right-hand side*: the leader receives a notification of a node that left the group.
Repositories are abstractions of more concrete elements such as a buffer. Connectors enable data exchange between concurrent units and control of concurrent units such as start, stop, synchronization.

Figure 5.7 shows the group formation processes in connection with view management and role activation.

We explain the subsequent steps in setting up a group. The number of each step corresponds to the numbers in Fig. 5.7:

1. An application component starts a view or starts an interaction session by sending a request to the View Manager or the Role Activator, respectively.
2. The View Manager and Role Activator process delegate group formation to the Group Formation process (specific actions related to view setup and role activation are explained in detail below). At this point, the node becomes the leader of a new group.
3. The Group Formation process communicates with other nodes using the Message Handler, to determine which nodes are to become a member of the group. The Group Formation process keeps monitoring the group for changes in membership.

Fig. 5.7 Communicating processes focusing on group formation
4. The Group Formation process notifies the View Manager or Role Activator of the group members and afterward of any changes in group members. A Property Maintainer process keeps the Node Properties repository up-to-date, e.g., by reading out sensor values. Only property values of the node itself need to be maintained. The Group Formation processes on the various nodes use their Node Properties repository to determine which nodes satisfy a node constraint. The Node Properties repository contains the updated values of all node properties used by the application. For example, if a group needs to be formed using a distance constraint, the Node Properties repository on each node contains the node’s current position.

5.3.3 View Management

To build a view, an application component specifies a node constraint, an objectplace name, and an object template. A first step in the construction of a view consists of the resolution of the node constraint to the group of nodes that satisfy the constraint.
This task is handled by the Group Formation process. Then, the members of the group are contacted to gather the objects that each of the members contributes to the view.

Figure 5.8 shows the main processes involved in view management. Elements that are not directly relevant for view management, such as the maintenance of node properties, are omitted.

We explain the steps that occur when constructing a view:

1. The View Manager receives a request to build a view from an application component. The request contains the node constraint, object place name, and object template.
2. The View Manager passes on the node constraint to the Group Formation process.
3. The Group Formation process forms the group and keeps the View Manager up-to-date with respect to membership (the node on which the view is built becomes the leader of the group).
4. Using the group member information, the View Manager on the leader node sends the object place name and object template to the View Managers on member nodes.

**Fig. 5.9** Communicating processes focusing on role activation
5. The member View Managers that are contacted by the leader perform a watch operation on the object-place with the given object-place name (shown as Object-places Repository), if it exists. If not, no objects from the member node contribute to the view. The events resulting from the watch operation are sent to the leader View Manager.

6. The leader View Manager builds and maintains a view in the Views repository, based on the events received from the members.

7. The application component that requested the view can now observe it in the Views Repository.

Views are maintained continuously. If the View Manager is notified of a change in membership by the Group Formation service, the View Manager updates the view accordingly. Similarly, if View Managers on a member node receive events from a watch, the events are sent to the leader View Manager.

### 5.3.4 Role Activation

To start an interaction session, an application component supplies an initiator role that the application component will play in the interaction, a node constraint, and the name of the participant role to be activated. A first step in role activation consists of group formation, after which the members of the group are contacted to activate the appropriate role. If the role is activated, a communication channel is set up between initiator and participant roles.

Figure 5.9 shows the main processes involved in role activation.

We explain the main steps that occur during role activation.

1. An application component sends a request to start an interaction session to the Role Activator, specifying an initiator role, a node constraint, and the name of a participant role.
2. The Role Activator gives the node constraint to the Group Formation service.
3. The Group Formation service finds the nodes belonging to the group and keeps the Role Activator up-to-date with respect to membership. The node on which the interaction session is started becomes the leader of the group.
4. The Role Activator on the leader node contacts the Role Activator processes on the member nodes to activate the desired participant role.
5. The member Role Activators activate the participant role, if the role is deployed on the node. The member Role Activator confirms activation of the role to the leader Role Activator.
6. For each participant that is activated, the Role Activator notifies the initiator of the newly activated participant. A handler to the participant that allows the initiator to communicate with the participant is given.
7. The protocol between initiator and participants commences.

The Object-places Repository contains the object-places that coordination roles and application components on the same node use to coordinate. Application
components observe the results or influence the course of the interaction protocol using these objectplaces, and coordination roles use the information in the objectplaces to determine the responses they send.

5.4 Collision Avoidance in the AGV Transportation System

We now demonstrate how views and coordination roles have supported the design and development of the multi-agent system for the AGV application. In this section, we focus on the coordination problem of collision avoidance. We assume that other functionalities such as task assignment, routing, deadlock avoidance, and battery recharging are available. A detailed discussion of task assignment supported with views follows in the next chapter. A solution for deadlock avoidance supported with coordination roles is presented in [146].

5.4.1 Collision Avoidance

Collision avoidance for AGVs is a coordination problem that resembles a mutual exclusion problem. Mutual exclusion algorithms are used in concurrent and distributed programs to ensure that several processes do not concurrently use unshareable resources. The un-shareable resources are called critical sections. For AGV collision avoidance, critical sections are physical areas on the factory floor that cannot be driven over by several AGVs at the same time. The important difference with classical mutual exclusion problems and collision avoidance is that areas are continuous, so the critical sections are continuous and determined dynamically at run time. In traditional mutual exclusion problems, critical sections are determined at design time and are discrete, i.e., there is a fixed and known number of critical sections that need to be guarded.

While the problem of collision avoidance can be made discrete, for example, by using segments as critical sections (of which there are a known, discrete number on the layout), this solution is not satisfactory, since it does not account for the case where two AGVs need to cross each other on closely located segments. In particular, there may be different types of AGVs working together on the same floor, so two small AGVs may be able to cross at the same time, while two broad AGVs cannot. If maximal flexibility is desired, the best option is then to allow AGVs to describe exactly which area they intend to cross and reserve that area for the AGV, instead of relying on imprecise, worst case discrete critical sections.

As a result, well-known distributed mutual exclusion protocols [136, 155, 101] are not directly usable for AGV collision avoidance. However, the similarities between both problems are still greater than the differences. Consequently, the protocol presented below is a variant on a classical mutual exclusion protocol described by [136].
Research in AGV control systems has tackled the collision avoidance problem [135, 111]. In all approaches, however, collision avoidance is handled together with routing and deadlock avoidance, i.e., an integrated approach to move AGVs from an arbitrary starting point to an arbitrary end point, taking into account the routes and destination of all other AGVs on the floor. Because all this information is needed, the approaches are all implemented in a centralized way, i.e., one server calculates all routes for each AGV. Since we study a decentralized architecture, these approaches do not fit our problem.

We have developed a decentralized approach for AGV collision avoidance. The underlying protocol allows decentralized mutual exclusion for continuous critical sections and can be applied to other similar mutual exclusion problems that require fine-grained critical sections.

### 5.4.2 Collision Avoidance Protocol

To explain the decentralized approach for AGV collision avoidance, we first focus on how the AGV agents avoid collisions without being aware of the underlying collision avoidance protocol. Then, we explain the work behind the scene, i.e., the decentralized mutual exclusion protocol executed by the local virtual environments supported by the ObjectPlaces middleware.

#### 5.4.2.1 AGV Agent Exploits the Local Virtual Environment

In order to drive collision free, an AGV agent exploits the local virtual environment, taking the following actions:

1. The AGV agent determines the trajectory it intends to follow over the layout. The trajectory is determined by *Lock Ahead Distance* parameter that ensures that the AGV moves smoothly and stops safely.

2. The AGV agent calculates exactly which area it is going to occupy on the floor if it drives over its intended trajectory. This area is determined by an AGV’s *hull projection*, see Fig. 5.10. A *hull* is the physical area an AGV occupies on the floor. A hull projection is the union of a set of hulls, projected along the AGV’s intended path in small increments. The hull projection determines accurately the space an AGV will occupy if it would drive over the path; so, if a number of hull projections of a set of AGVs overlap, the AGVs are on collision course.

3. To avoid collisions, an AGV agent tries to reserve the area represented by the hull projection for exclusive use. Therefore the agent marks the path it intends to drive in the local virtual environment\(^4\) with a *requested hull projection*. This projection contains the agent’s identity and a priority that depends on the transport the AGV is handling.

---

\(^4\) For convenience, we use “local virtual environment” to refer AGV local virtual environment in the remainder of this chapter.
4. The agent perceives the local virtual environment to observe the result of its action.

5. The agent examines the perceived result. There are two possibilities:

   a. The requested hull projection is marked as a *locked hull projection*: it is safe to drive.
   b. The hull is not marked as locked: this means that the agent’s hull projection conflicted with that of another AGV agent. The agent may not pass; at this point the agent may decide to wait and look again at a later time or remove its requested hull projection and take another path altogether.

Since the AGV steering system, E’nsor, must be instructed to drive segment per segment (i.e., the level of granularity is one segment), an AGV’s requested hull projection spans at least one segment. When an AGV is driving, the AGV agent releases the parts of its locked hull projection behind it, so that other AGVs may pass. Note that AGVs cannot completely clear their locked hull projections, since an AGV at least needs to keep a lock on the area it is currently standing.

### 5.4.2.2 Decentralized Mutual Exclusion Protocol

We now shift our focus to the AGV’s local virtual environment which must resolve conflicts with the local virtual environments of other AGVs that intend to move and make sure that the requested hull projection becomes locked eventually. To this end, the local virtual environment of the AGV agent that requests a new hull
projection executes a mutual exclusion protocol with local virtual environments of nearby AGVs.

In order to guarantee safety and save bandwidth, the subset of local virtual environments with which a requesting local virtual environment interacts must include the local virtual environments of all AGVs with which the AGV of the requesting local virtual environment might collide. Figure 5.11 illustrates how safe subset of AGV local virtual environments is determined.

A requesting local virtual environment interacts with other local virtual environments whose hull projection circle overlaps with the hull projection circle of requesting local virtual environment. 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 between the AGV and the furthest point on its hull projection. So, overlapping circles indicate to a first approximation that two AGVs are within collision range. This approximation has the benefit that it narrows down the possible candidates for interaction significantly, while each AGV only needs limited knowledge about other AGVs to determine interaction partners (i.e., position and hull radius).

Due to the mobility of the AGVs, a new AGV entering collision range should be taken into account when executing the collision avoidance protocol, and an AGV leaving collision range can be disregarded. Using the middleware support, the collision avoidance protocol is modeled by two roles: a Requester and a Voter role. To lock a new hull projection, the local virtual environment activates a Requester role, asking the activation of Voter roles with a node constraint that selects all AGVs within collision range:

\[ c_{\text{node}}(V_{\text{init}}, V_{\text{part}}) = \text{dist}(V_{\text{init}.\text{pos}}, V_{\text{part}.\text{pos}}) \leq V_{\text{init}.\text{hull}} + V_{\text{part}.\text{hull}} \]

Fig. 5.11 Illustration of the hull projection circle
V. *pos* denotes the current \((x,y)\) position, *hull* the current hull radius. From this constraint, it is clear that the middleware needs the AGV’s positions and current hull radii to determine where voters should be activated, so the application updates this information in the node properties repository. On each AGV, the AGV’s position and hull radius are updated every second. The middleware takes care of disseminating positions and hull radii to other AGVs. So, a small amount of data is sent to all AGVs, in order to allow the AGVs to execute the collision avoidance protocol in smaller groups. To instantiate the necessary Voter roles, the middleware finds all the AGVs in the system whose properties satisfy the node constraint. The Requester role is notified of these Voter role instances, after which the collision avoidance protocol can be executed.

Once the group is settled, to lock a requested hull projection, the local virtual environment executes the following mutual exclusion protocol with the local virtual environments in collision range:

1. The requester sends a Request(*HullProjection*) message to voters.
2. The voters check whether the requester’s hull projection overlaps with their hull projection. There are three possibilities for each of the requested voters:
   a. No hull projections overlap. The voter sends an allow message to the requester.
   b. The requester’s hull projection overlaps with the voter’s hull projection, and the voter’s hull projection is already locked. The voter defers to send an allow message until the lock on the overlapping area is released.
   c. The requester’s hull projection overlaps with the voter’s hull projection, and the voter’s hull projection is not locked. Since each of the requested hull projections contains a priority, the voter can check which hull projection has precedence. If the requester’s hull projection has a higher priority than that of the voter, the voter replies allow; otherwise the voter defers until the lock on the overlapping area is released.
3. The requester waits for all votes to come in. If all voters have voted allow, the requested hull projection can be locked and the state of the local virtual environment is updated.

When a new AGV enters collision range while a collision avoidance interaction session is in progress, this is detected by the middleware and a Voter role is instantiated on that AGV. The Requester is notified, and in response sends a request to the new Voter, and also waits for the allow message from that AGV. When an AGV moves out of collision range, the Requester is notified, and so the Requester no longer waits for that Voter.

Intuitively, the protocol is safe, i.e., collision-free movement is guaranteed, because for each two AGVs with overlapping requested hull projections, exactly one request is allowed. However, a closer examination reveals that two problems must be solved to guarantee safety of the protocol:
1. Group formation may be out of date. The middleware sends update messages to inform AGVs of new positions and hull radius. However, this information is updated once per second, and there is a transmission delay. The information an AGV has about other AGVs thus may not reflect the current situation. As a result, an AGV may not send a request to another AGV that is within collision range and erroneously assume that it is safe to lock a hull projection. This problem is solved as follows. Given the update interval of 1s for position and hull radius, and a maximum message delay $t_{\text{delay}}$, every Requester must wait a minimal safe time of 1 s plus $t_{\text{delay}}$ before closing a session and locking a hull. This delay ensures that the middleware has had time to exchange the requesting AGV’s new position and hull radius with other AGVs, so that each AGV’s information is up-to-date with respect to the requesting AGV. In practice, since $t_{\text{delay}}$ is much smaller than 1 s, the safe time is set conservatively to 2 s.

2. Due to communication delays, group formation may be temporarily inconsistent. In particular, when two local virtual environments start an interaction session to execute the collision avoidance protocol an error may occur when a voter on an AGV that is also requesting sends an allow message to an AGV that is not in the AGVs group, Fig. 5.12 shows a scenario.

   To enforce consistency, we add the condition that a voter role may only allow a request if the requesting AGV is also in the collision avoidance group as seen by the AGV on which the voter is deployed.

   Appendix C describes the collision avoidance protocol in detail and provides a proof of safety.

### 5.4.3 Software Architecture: Communicating Processes for Collision Avoidance

We now illustrate how collision avoidance is dealt with in the software architecture of the AGV transportation system. Figure 5.13 shows the communicating processes diagram for collision avoidance.

The diagram presents the basic components of the AGV control system (AGV Agent, AGV Local Virtual Environment, and Middleware) and overlays them with the main processes and repositories involved in collision avoidance; compare the module decomposition view of the AGV transport system in Fig. 4.12, the collaborating components of the AGV agent in Fig. 4.14, and the collaborating components view of the local virtual environment in Fig. 4.19. We explain the subsequent interactions between the main processes involved in locking a requested hull projection for collision avoidance. The number of each step corresponds to the numbers in Fig. 5.13:

1. The Collision Avoidance process of the AGV agent, which is part of the decision making component, requests the Action Manager process a hull projection.
Fig. 5.12 A possible collision if group formation is inconsistent. Part (a) shows the initial situation: two AGVs driving toward each other, each with a locked hull projection. The circles show the AGV’s hull radius. AGV B has a pending requested hull projection but has not requested an allow from AGV A since AGV A is not within collision range. Part (b) shows what can happen if AGV A also requests a new hull projection that overlaps with AGV B’s requested hull projection. AGV A sends a request to AGV B, which, if AGV A’s priority is higher than AGV B’s, is allowed by AGV B. AGV A’s request message, however, has arrived faster than the update message indicating AGV A’s new hull radius to AGV B. At this point AGV B can decide to lock its own requested hull projection, since it is not aware that it should send a request to AGV A. Likewise, AGV A has received an allow vote from AGV B, so it too can lock its requested hull projection. Collision is then imminent.

2. Action Manager instantiates the Requester role and the corresponding Requester process. The requester role adds the requested hull to the Collision Avoidance Object place.
3. Action Manager requests Role Activator to start the collision avoidance protocol to lock the requested hull projection.
4. Role Activator uses Group Formation to start the group.
5. Group formation communicates with the Group Formation processes on the other AGVs to determine which AGVs are to become a member of the group.
6. Group Formation notifies the Role Activator of the group members, and afterward in case a member leaves the group or a new member joins the group.
7. Role Activator contacts the Role Activators on the member nodes, i.e., the AGVs that are in collision range, to activate the Voter role.
8. The Role Activators on the member nodes inform Role Activator that the Voter role is activated.
9. Role Activator in turn notifies Requester.
10. When all the Voter roles are activated, Requester starts the collision avoidance protocol sending requests to the Voters with the requested hull projection.
11. Each Voter sends an allow message when the requested hull projection does not overlap with their hull projection.
12. As soon as the Requester has received an allow message from all the voters, it locks the hull in the Collision Avoidance Object place repository.

Subsequently, the Hull Maintainer process, which is part of the synchronization module of the AGV local virtual environment (see Fig. 4.19), observes the hull change and updates the hull representation in the State repository of the AGV local virtual environment. Finally, the Collision Avoidance process uses the Perception process to sense the status of the hull projection. The Collision Avoidance process notices that the hull is locked and the AGV can move on.

![Communicating processes for collision avoidance](Image)
5.5 Summary

In this chapter, we discussed the crucial role of middleware support for multi-agent systems. We gave an overview of the role of middleware for supporting the development of distributed systems, and we discussed the multiple layers of middleware in distributed software systems in general. Then we discussed how typical middleware for multi-agent systems maps on the different middleware layers. We explained that the focus of middleware support for multi-agent systems is on the domain-specific middleware layer. Such domain-specific middleware simplifies application development by offering programming abstractions that hide lower level details from the application developer, and it accelerates application development by encapsulating generic, reusable functionalities to support the programming abstractions.

The case study presented ObjectPlaces, a middleware for mobile systems. ObjectPlaces targets applications that are characterized by context awareness, dynamic operating conditions, and decentralization of control. These characteristics closely connect with many applications targeted by multi-agent systems.

We presented the two programming abstractions offered by the middleware: views and coordination roles. A view is a representation of data objects shared by application components in objectplaces on other nodes in the network. A coordination role is an abstraction that encapsulates the behavior of an application component in a protocol. We discussed the software architecture of the middleware and motivated the rationale for the architectural design. We used communicating processes diagrams to precisely describe the internals of group formation, view management, and role activation.

Finally, we explained how we have applied ObjectPlaces in the AGV transportation system. Our particular focus in this chapter was on collision avoidance. To avoid collisions, an AGV agent coordinates with other AGV agents by projecting a requested hull in the local virtual environments that demarcates the area the AGV intends to drive. The local virtual environments of the AGVs in collision range resolve conflicts by executing a mutual exclusion protocol. Dealing with dynamics and context awareness is a difficult problem in the AGV application. By applying coordination roles, we showed that the application developer can abstract from low-level details; the tedious but important tasks, such as finding the AGVs in collision range handling, dealing with AGVs that leave, and new AGVs that enter collision range, are handled by the middleware. The middleware support has shown to be invaluable in the design and development of this real-world multi-agent system.
Architecture-Based Design of Multi-Agent Systems
Weyns, D.
2010, XVII, 224 p., Hardcover
ISBN: 978-3-642-01063-7