Reliable Real-Time Communication in Cooperative Mobile Applications

Edgar Nett and Stefan Schemmer

Abstract—Embedded systems are expected to provide increasingly complex and safety-critical services that will, sooner or later, require the cooperation of several such systems for their fulfillment. In particular, coordinating the access to shared physical and information technological resources will become a general problem. Examples are mobile robots in industrial automation or car-to-car coordination for future traffic control applications. In such applications, cooperation is subject to strong real-time and reliability requirements. In this paper, we present an architecture that allows autonomous mobile systems to schedule shared resources in real-time using their own wireless distributed infrastructure. In this architecture, there is a clear separation between the application-specific scheduling part and the application independent communication part that constitutes the real-time and reliability hardcore of the system. The latter provides clock synchronization, real-time atomic multicast, and real-time group membership based on an IEEE 802.11 Standard wireless LAN. An application prototype shows how the architecture can be used in future mobile cooperative applications.

Index Terms—Fault-tolerant real-time systems, distributed algorithms, mobile computing, wireless networks.

1 Introduction

Mastering and controlling the mobility of and wireless communication between autonomous systems will be of utmost importance for the design of future distributed embedded systems. Such systems are expected to provide more and more complex and safety-critical services, parts of which will be based on the reliable real-time cooperation of the involved systems. In particular, coordinating the access to shared resources will become a general problem that must be solved by the cooperation of such systems. For example, mobile robots will cooperate in industrial automation to achieve collision avoidance. This will allow for more flexible production systems and for more efficient employment of the mobile systems. More visionary, intelligent driving assistants will cooperate to coordinate their access to shared road space such as crossroads and merging roads, thus achieving increased driving safety and comfort as well as a more efficient usage of the existing road space. The cooperation of mobile systems is subject to strong real-time and reliability requirements. However, it must be based on wireless networks, which are intrinsically unreliable and therefore subject to message loss rates that are significantly higher than on wired media. Hence, achieving reliable and timely cooperation is a promising but challenging task.

We have designed an architecture for the reliable real-time cooperation of mobile systems [25] that is divided into four layers. On the application-specific layer, each system computes the schedule for the shared resource locally. Below, the so-called event service is called whenever an event requiring a new schedule to be built is detected. The event service is used to propagate events reliably and timely to the group of cooperating systems. It is based on the communication hardcore, which provides the communication semantics and Quality of Service (QoS) that are necessary for cooperation. The bottom layer is comprised of an IEEE 802.11 Standard compliant wireless LAN. This standard is commonly accepted for wireless communication and network hardware is commercially available.

Regarding reliability and timeliness, the development of the communication hardcore is the most challenging task. On the one hand, to be usable in mobile applications, it must be based on a wireless medium, which is intrinsically error prone and unreliable. On the other hand, it must provide strong reliability and timeliness guarantees as well as rich group semantics to serve as the basis for cooperative real-time applications. The communication hardcore provides fault-tolerant clock synchronization, real-time atomic multicast, and real-time membership to support the cooperation of groups of mobile systems. The protocols that implement these services will be described in this paper.

To demonstrate the feasibility of our approach, the general problem behind has been embedded in an application prototype that shows how the architecture can be used in future cooperative mobile applications. In the prototype, mobile robots use the presented architecture to schedule the intersection of two paths in real-time such that they cross the intersection efficiently and without collisions.

The remainder of the paper is organized as follows: In Section 2, our architecture is presented. The communication hardcore of this architecture is presented in Section 3. Section 4 describes our application prototype. Section 5 refers to related work and some concluding remarks are presented in Section 6.

• *The authors are with the University of Magdeburg, Institute for Distributed Systems, Universitätsplatz 2, D-39106 Magdeburg, Germany. E-mail: {nett, schemmer}@ivs.cs.uni-magdeburg.de.*

Manuscript received 1 Feb. 2002; revised 7 Sept. 2002; accepted 20 Sept. 2002. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 117448.

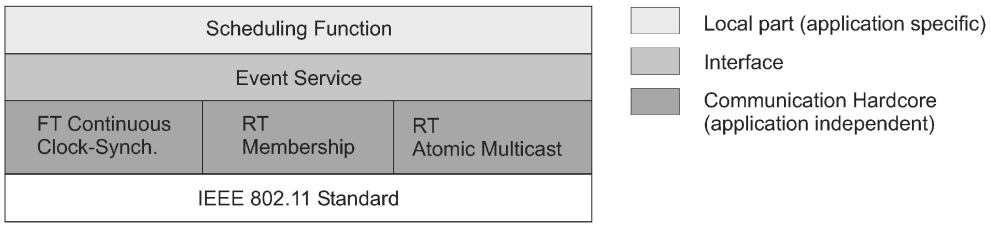


Fig. 1. The architecture.

2 Architecture

This section presents our architecture for real-time resource scheduling in groups of autonomous mobile systems.

As we are dealing with physically mobile entities, coordinating the access to a shared resource is subject to strong real-time and reliability requirements. In particular, if a potential conflict occurs, the shared resource must be scheduled reliably and in bounded time. Therefore, we avoid using sophisticated negotiation protocols for which predictable behavior is only achieved with great difficulty. Instead, our approach is to compute the schedule for the shared resource locally at the involved systems and to ensure through communication that the input of the local computations is consistent. To this end, we divide our architecture into four layers (see Fig. 1). On the highest, application-specific layer, the actual scheduling is performed locally. Each system computes the schedule for the shared resource as a function, the so-called scheduling function, that does not require any communication. The mobile systems can implement the computed schedule by local actions such that no further coordination is needed after execution of the scheduling function.

As each system computes the scheduling function locally, it is necessary to provide consistent input for it. This is the job of the event service, the next layer of the architecture. The input, called global state in the following, is comprised of the scheduling-relevant parameters of the mobile systems, such as position or velocity, with respect to the same point of time. Whenever a mobile system detects an event requiring that a new schedule be computed, it calls the event service to propagate the event. A typical example is a mobile system detecting that it is approaching a shared resource—say, a car that is approaching a crossing. It calls the event service to propagate this event to all mobile systems that are approaching the resource as well. Before the event service delivers the event, it determines the global state (in the example, the positions and velocities of all cars with respect to the same point of time) and delivers it, together with the event, to the scheduling function. In our design, the global state is determined by local computation of the event service as far as possible. To achieve this, the event service is based on a model in which the changes of the scheduling relevant parameters of the mobile systems can be predicted. The architecture is described in more detail in [25].

As event propagation and global state determination must be achieved reliably and in real-time, the event service uses the communication hardcore to propagate events to the group of mobile systems. This layer allows transmitting atomic multicasts in dynamically changing groups of mobile systems. It also provides a continuous global time base that is needed by the event service for global state computation.

3 Communication Hardcore

In Section 3.1, we describe the real-time features already provided by the IEEE 802.11 Standard and we exhibit the fault assumptions that are the basis of our design. Subsequently, we describe the protocols that constitute the communication hardcore. In Section 3.2, the fault-tolerant clock synchronization protocol is briefly summarized. In the following two sections, the real-time atomic multicast protocols and the membership protocols are presented.

3.1 The IEEE 802.11 Standard and Fault Assumptions

The IEEE 802.11 Standard [1] is commonly accepted for wireless local area networks. It specifies two medium arbitration schemes, which are applied in two alternating periods. One scheme is carrier sense multiple access (CSMA)-based and allows collisions to take place on the medium. Hence, periods in which this scheme is used are called "contention periods" (CP). The other arbitration scheme precludes collisions on the medium. Hence, the periods in which this scheme is applied are called "contention free periods" (CFP).

Medium arbitration during the CFP works by imposing the following communication structure: The access point (AP), a special fixed node, grants exclusive access to the medium by sending polling messages to the stations. Every station remains silent until the AP polls it. When being polled, it is allowed to send a single message. The AP is free to implement any strategy for polling stations during the CFP. Although there are no contentions during the CFP, the problem of message losses still has to be tackled. Actually, the number of message losses is considerably higher than for wired local area networks because the wireless medium is unshielded and thus prone to external interference. 802.11 specifies a mechanism to improve the reliability of message transmission by means of acknowledgments and retransmissions. However, this mechanism is only specified for point-to-point and not for broadcast messages.

We are considering a set $\mathcal{S} := \{s_1, s_2, s_3, \ldots\}$ of stations.

Property 1 (Crash failures and AP stability). *Stations may be subject to crash failures except of the AP, which is assumed to be stable, i.e., not subject to any kind of error.*

The assumption of the stability of the AP is already implied in the IEEE 802.11 Standard, which uses the AP as a

central coordinator during the CFP. The IEEE 802.11 Standard does not indicate how the stability is achieved. From our point of view, a virtual AP provides the coordination functionality. Wherever high reliability is necessary, the virtual AP is implemented with redundant hardware using well-known fault tolerance concepts.

At each time, the team $\mathcal{T} \subseteq \mathcal{S}$ consists of those stations that are cooperating to schedule a shared resource.

Property 2 (Bounded team size). *We assume that, at each point of time, the number of team members is not greater than N and that the rate at which stations join the team is bounded.*

As we are considering teams cooperating in a limited spatial area covered by a single wireless LAN, Property 2 is a reasonable assumption since such an area can only contain a bounded number of mobile systems.

For the following specifications, we assume that all messages sent can be distinguished, e.g., by a tag consisting of the address of the sender and a local sequence number. The properties of the medium during the CFP are defined as follows:

Property 3 (Timeliness). *There exists a known constant δ_m such that, for all stations s_i, s_j, messages m, and times t, t': If s_i sends m at time t and s_j receives m at time t', then $t' - t \leq \delta_m$.*

Property 4 (Integrity). *For all stations s_i, messages m, and times t: If s_i receives m at time t, there exists a time $t' < t$ and a station s_j such that s_j sent m at time t' either addressed to s_i or to "broadcast." Furthermore, for all stations s_i, messages m, m', and times t, $t' \neq t$: If s_i receives m and m' at times t and t', then $m \neq m'$.*

Property 5 (FIFO). *For all stations s_i, s_j and messages m, m': If s_i sends m before m' and s_j receives m and m', then s_j receives m before m'.*

We now define the reliability properties of the medium. As the AP plays a central role during the CFP, the reliability of communication between the AP and the stations is especially important. Since wireless media exhibit a poor and time-varying reliability and since the stations are mobile, not much can be said about this reliability independent of the times and stations considered. Nevertheless, our protocols will provide guaranteed timeliness and reliability properties for those stations and times, for which the reliability is "sufficiently good." To allow us to express this guarantee, the following predicate defines what "sufficiently good" actually means. Basically, it means that there exists an upper bound *OD* on the number of message losses affecting any information transferred between the AP and the station, which is a necessary precondition to achieve reliable and timely transfer of information. The details of the definition are related to the IEEE 802.11 Standard and our protocols; for example, point (3) ensures that reliable polling is possible. Given the constant *OD*, we define:

Definition 1 (valid). *For all stations s_i and intervals of time $[t, t']$, s_i is valid during $[t, t']$ if and only if:*

1. *s_i is correct throughout $[t, t']$.*
2. *If the AP transmits an information to s_i in at least $OD + 1$ messages sent during $[t, t' - \delta_m]$, s_i receives at least one of those messages.*
3. *If s_i answers each polling message it receives with a reply addressed to the AP, then, for each sequence of $OD + 1$ consecutive polling messages sent to s_i during $[t, t' - 2\delta_m]$, the AP receives at least one reply.*

We define a further predicate describing the situation in which communication between a station and the AP is completely impossible.

Definition 2 (invalid). *For all stations s_i and intervals of time $[t, t']$, s_i is invalid during $[t, t']$ if and only if*

1. *s_i is crashed throughout $[t, t']$ or*
2. *each message sent by s_i during $[t, t' - \delta_m]$ is not received by the AP and vice versa.*

The "invalid" predicate is not the negation of the "valid" predicate. There may be intervals during which a station is neither valid nor invalid. Furthermore, during intervals in which only few messages are transmitted, both predicates may be true. For example, as long as less than $OD + 1$ polling messages are sent to s_i during an interval I, s_i could be valid or invalid according to Definition 1.3. However, as long as the protocols ensure certain properties if a stations is valid during an interval of a specified length, it is not of so much interest that the station could also be viewed as being invalid during a fraction of that interval.

3.2 Fault-Tolerant Continuous Clock Synchronization

To provide a global time base for the mobile systems, we designed and implemented a continuous clock synchronization protocol that extends the IEEE 802.11 Standard, which already specifies a simple master/slave clock synchronization protocol. The following subsections highlight three essential aspects of the protocol; a detailed description has been presented in [18], [19].

3.2.1 Achieving High Precision

The IEEE 802.11 Standard specifies a clock synchronization protocol, which uses the AP as a clock master. This protocol has a long time-critical path, which, in particular, contains the transmission of the synchronization messages. This is a major drawback since the variance of the time-critical path restricts the achievable precision.

Our protocol exploits the tightness of message reception on broadcast media, i.e., the fact that stations receiving the same message receive it at approximately the same time, to achieve a short time critical path. This idea already proved to be effective in our clock synchronization for the CAN-bus [11]. It limits the length of the time critical path to the few deterministic actions needed to take a timestamp on reception of a synchronization message. In particular, the critical path does not comprise any message transmissions. The precision of the protocol has been determined analytically under worst-case assumptions. The empirical evaluation of our driver level implementation yielded a measured precision of 150μs.

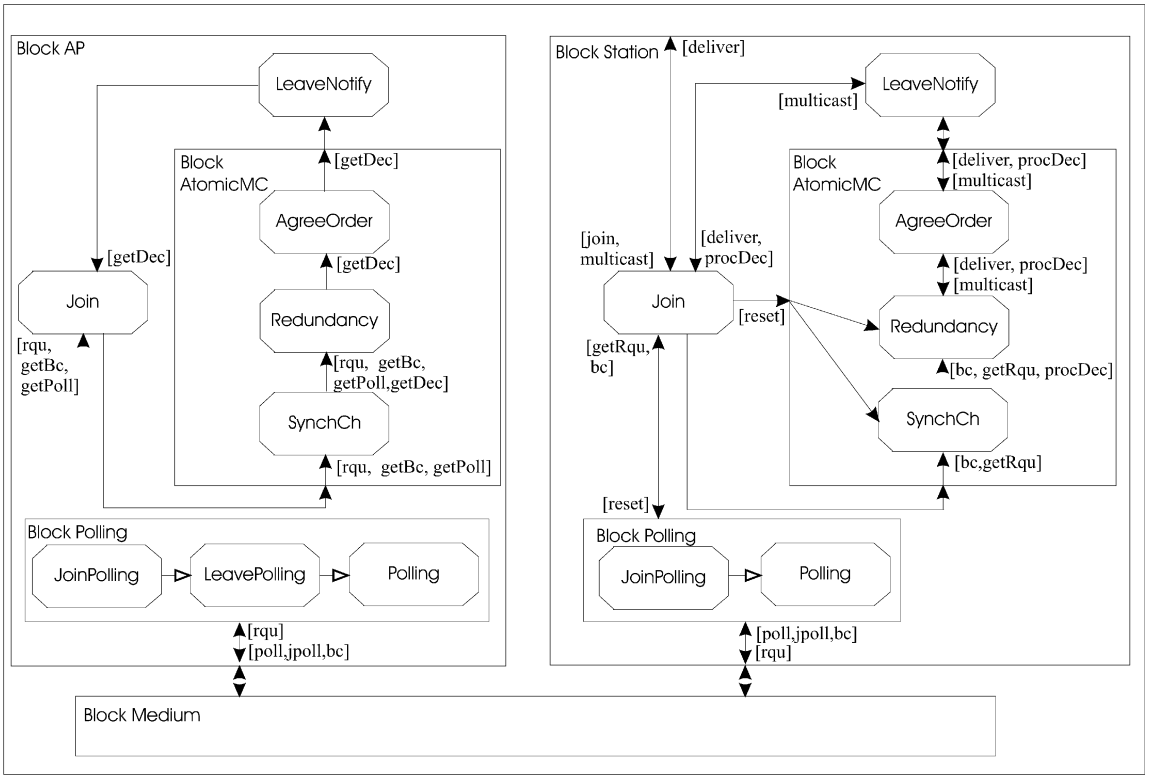


Fig. 2. The protocol stack.

3.2.2 Achieving Fault Tolerance

In our protocol, a station must receive at least two synchronization messages to be able to adjust its clock—the reception of the first constitutes a reference point and the second contains the master's timestamp for this reference point. However, due to the unreliability of wireless media, a station cannot be assumed to receive two consecutive synchronization messages always.

Our protocol tolerates up to OD consecutively lost synchronization messages. The master includes the timestamps for the last $OD+1$ synchronization messages in each synchronization message such that a valid station can adjust its clock on every $(OD+1)\text{th}$ synchronization message at least.

3.2.3 Achieving Continuous Clock Adjustment

Adjusting the clocks of the stations by simply changing their values has several drawbacks. It leads to gaps in the time base and can result in negative interval measurements. Therefore, our protocol realizes a continuous clock adjustment. As the physical clocks in our implementation environment cannot be adjusted, neither their value nor their frequency can be set, the protocol provides a virtual clock, which is a piecewise linear functions of the physical clock, at each station. The protocol adjusts the parameters of the pieces of the virtual clocks in such a way that each virtual clock is a continuous function (neglecting the discreteness of the physical clock). This also contributes to the precision of the protocol in the case of message losses since the rates of the virtual clocks are much closer to the master's rate than the rates of the physical clocks.

3.3 Real-Time Atomic Multicast

A family of microprotocols implements the real-time atomic multicast and membership services. By configuring stacks of these protocols, different services can be realized for different kinds of applications.

Fig. 2 shows the protocol stack. The eight-cornered boxes denote processes which communicate via signals and remote procedure calls. Solid arrows depict the routes of the signals and calls. Lines with an empty arrow denote an inheritance relation between processes. Blocks have been used to give additional structure by clustering parts of the protocol stack. Those parts realizing the atomic multicast service are contained in the blocks labeled "AtomicMC."

In the following, we describe the basic polling microprotocol and the microprotocols implementing the real-time atomic multicast service. Definitions of the properties of the atomic multicast service can be found in [20] (presented as broadcast properties there).

3.3.1 Polling

The IEEE 802.11 Standard specifies that polling shall be used to provide predictable medium access. In our protocol, the AP polls the members of the group in a round-based manner, each member exactly once during each round. In addition, the protocol uses the AP as a central router since each valid station is by definition able to communicate with the AP with bounded omission failures, but it is possibly unable to communicate with other valid stations. A member that receives a poll includes a message it wants to have multicast in a request and sends it to the AP, which includes the message in a broadcast and sends it to the group. The AP uses a timeout to avoid blocking if either the poll or the

request is lost. Members receiving a poll send a request to the AP, even if they have no message to transmit, in order to avoid having the medium remain unnecessarily idle while the AP is waiting for the timeout. The described sequence of three messages—poll, request, and broadcast—is called a "slot" in the following. The number of slots per interval of time is independent of the number of members. As a consequence, the AP does not become a bottleneck when the number of members increases.

3.3.2 Dynamic Time Redundancy

It is the task of the redundancy protocol to handle omission failures. The protocol uses a dynamic redundancy approach, where messages are only retransmitted when message losses are detected. Either the requests or the broadcasts containing a multicast message may be lost. To tolerate lost requests, the protocol uses implicit positive acknowledgments. The AP acknowledges the reception of a message by sending a broadcast including it. The protocol uses the round structure to efficiently detect lost broadcasts. To this end, a bit field *ack* is provided in the requests. When a member receives a poll, it uses the *ack* field of its request to acknowledge each broadcast it received since the AP last polled it. To relate the bits of the *ack* field to broadcasts, the AP assigns consecutive global sequence numbers to broadcasts and polls. A station determines the position in the *ack* field relating to a broadcast it receives as the difference between the broadcast's global sequence number and sequence number of the last poll it received. When the AP receives a request, it knows that, for each broadcast acknowledged therein, the sender of the request has received the included message. The AP knows that all valid members have received a message and that it need no longer transmit it if either they all have acknowledged the message or if it has transmitted the message $OD + 1$ times. (Here, valid means valid during $[t, t' + \delta_m]$, t being the time of the first and t' the time of the last transmission of the message. We will not state the intervals explicitly in the following for the sake of brevity.)

The presented acknowledgment scheme does not need extra messages for error detection. Members piggyback acknowledgments on the requests they sent anyway. The AP acknowledges messages implicitly by broadcasting them.

There are applications for which guaranteeing reliable transmission is less important than guaranteeing small delays. For such applications, the protocol provides a parameter *res* allowing the limiting of the number of retransmissions of a message to a value smaller than *OD*. Members transmit a message to the AP in no more than $res + 1$ rounds and the AP transmits the message in at most $res + 1$ broadcasts. Hence, it is possible that not all valid members receive it. Our agreement protocol (see Section 3.3.4) ensures that, in this case, no member delivers the message. Thus, the application can trade reliability for improved timing guarantees while atomicity is still guaranteed.

3.3.3 Synchronous Channel

The so-called synchronous channel allows the AP to transmit a small amount of information reliably and timely to the members. Before sending a broadcast, the AP can insert one out of a set of decisions into the synchronous channel. To keep the overhead of the synchronous channel small, we use only a small number of possible decisions, e.g., three in the following agreement protocol. To transmit the decisions, the synchronous channel protocol piggy-bags a field sy of $OD + 1$ decisions on each broadcast. It transmits a decision that the AP inserts into the synchronous channel at positions $0, 1, \ldots,$ and OD in the sy field of the following $OD + 1$ broadcasts. Each valid member receives at least one of these broadcasts no more than $\delta_m + OD \times 3 \times \delta_m$ time units after the decision was made. Thus, whenever the AP makes a decision d before sending a broadcast with sequence number sg on behalf of member s_i, each valid member will process the decision within bounded time. The members process the decisions in the order in which the AP made them and relate them to the same member (s_i) and the same sequence number (sg) as the AP.

3.3.4 Atomic and Totally Ordered Delivery

As not all valid members necessarily receive a message if it has a resiliency smaller than *OD*, members delay the delivery of a message until the AP notifies them whether they shall deliver it (*accept* decision) or not (*reject* decision). Whenever the AP is about to send a broadcast on behalf of a member, it decides if the message transmitted on behalf of this member during the last round can be accepted. This is the case, if the message has been acknowledged by all group members or has been transmitted $OD + 1$ times. Otherwise, the AP decides to retransmit if the resiliency limit is not yet exceeded or reject it. The AP inserts the decision into the synchronous channel to transmit it to the members.

This protocol achieves agreement among the members, even if not all valid members receive a message. As members process the decisions in the synchronous channel in the order in which the AP made them, it achieves total order as well. The analysis of the timeliness of the atomic multicast service yields a worst-case delay of $\Delta_{BC} = (2 \times res + 1) \times \Delta_{Round} + (OD + 1) \times \Delta_{Slot}$ (or $\Delta'_{BC} = \Delta_{BC} + \Delta_{Round}$ if the application is not synchronized with the protocol) for a message with resiliency res, being $\Delta_{Slot} := 3 \times \delta_m$ and $\Delta_{Round} := N \times \Delta_{Slot}$ [20]. The equation shows how reducing *res* improves the time bound. Measured delays will be presented in Section 4.

3.4 Real-Time Membership

In the preceding section, we described how timely and atomic multicast transmission is achieved assuming that the group is static. In this section, we explain how changes of the membership are handled. In Section 3.4.1, the properties of the membership service are defined. Section 3.4.2 then explains how invalid members are excluded from the membership. Afterward, in Section 3.4.3, we explain how stations join the group in real-time.

3.4.1 Properties of the Membership Service

The service provides two kinds of properties. First, safety properties, namely Agreement, Total Order, and Virtual Synchrony, which are guaranteed to hold always and for every station whether valid or not. Second, progress properties, namely timeliness of joins and exclusions, which can only be guaranteed for stations being valid for a sufficiently long time. Hence, properties of the second kind are formulated as conditional properties.

Safety Properties. The application interface of the membership service consists of two signals. The application sends signal *join* to start the membership service when the station becomes a team member (e.g., when a car detects that it is approaching a crossing). The membership service sends signal *deliver*(*m*) to deliver membership change messages to the application. The atomic multicast service uses the same signal to deliver messages to the application. The application observes a sequence of messages, each of which is either an atomic multicast or a membership change message. Membership change messages have the structure (*memch*, *memid*, *mem*), where *memch* indicates the type of the message, *mem* is the set of team members currently assumed to be valid (called the membership in the following), and *memid* is a unique identifier. If the protocol delivers the same membership several times—say, a station leaves and then rejoins the group—*memid* allows distinguishing the membership change messages. The membership protocol delivers a membership change message with an empty membership and a special identifier $\perp$ at station s_i to indicate that s_i is no longer part of the membership. We say that a station s_i delivers a message m if the membership or multicast service at this station delivers m to the application.

Property 6 (Agreement). *For all stations s_i, s_j, identifiers n, and memberships mem, mem': If s_i delivers $(memch, n, mem)$ and s_j delivers $(memch, n, mem')$, then $mem = mem'$.*

In the following, $succ(s_i, n, n')$ denotes the fact that n and n' are the identifiers of two membership change messages successively, i.e., with no other membership change messages between them, delivered by s_i.

Property 7 (Total Order). *For all stations s_i, s_j and all identifiers $n \neq \perp$, $n' \neq \perp$, $n'' \neq \perp$: If $succ(s_i, n, n')$ and $succ(s_j, n, n'')$, then $n' = n''$.*

The two properties together ensure that two stations, once they have delivered a membership change message with the same identifier, deliver the same sequence of memberships henceforth until one of them delivers an empty membership (i.e., becomes itself excluded from the membership).

Property 8 (Virtual Synchrony). *For all stations s_i, s_j and identifiers n, $n' \neq \perp$: If $succ(s_i, n, n')$ and $succ(s_j, n, n')$, then s_i and s_j deliver the same set of atomic multicasts between the membership change messages with identifiers n and n'.*

In the event service, a single member must be elected that reacts to the first atomic multicast of a new member. As, by Property 8, all members receive this atomic multicast in the context of the same membership, this property allows solving the problem by applying some deterministic rule to the membership locally at each member.

Progress properties. So far, we have defined properties ensuring that if stations deliver membership change messages, they do so in consensus. However, it is not yet ensured that valid team members become part of the membership and that invalid team members are excluded. In the following, we define two properties guaranteeing that if a team member is valid long enough, its view of the current membership reflects all changes in the set of valid team members with a known maximum delay. To this end, we denote by $mem(s_i, t)$ the last membership delivered by station s_i at a time no later than time t and, by $mem(n)$, the membership delivered in $(memch, n, mem)$.

Property 9 (Timeliness of exclusions). *There exists a known constant Δ_{Mem}^{excl} such that, for all team members s_i, s_j and intervals $[t, t']$: If s_i is invalid during $[t, t']$, then, for all $t'' \in [t + \Delta_{Mem}^{excl}, t']$, s_j is crashed or $s_i \notin mem(s_j, t'')$.*

Property 10 (Timeliness of joins). *There exists a known constant Δ_{Mem}^{join} such that, for all team members s_i, s_j: If s_i and s_j are both valid during $[t, t']$, then, for all $t'' \in [t + \Delta_{Mem}^{join}, t']$, $s_i \in mem(s_j, t'')$. Furthermore, let n be the identifier of the first membership change message— going backward the delivery order of s_i—with (s_i, n', n) and $\{s_i, s_j\} \subseteq mem(n)$ and not $\{s_i, s_j\} \subseteq mem(n')$. Let n'' be the corresponding identifier at s_j. Then, $n'' = n$.*

Note, that the first proposition in Property 10 ensures that $\{s_i, s_j\} \subseteq mem(s_i, t'')$ and $\{s_i, s_j\} \subseteq mem(s_j, t'')$ such that the existence of n and n'' in the second proposition is ensured.

3.4.2 Leaving Stations

Excluding a leaving station from the membership can be divided into two aspects: First, it must be detected that the station is leaving. Second, this information must be propagated to the remaining group members, which must change their membership accordingly and deliver it to the application.

Detecting leaving stations. To realize the first aspect, the AP counts for each member the number of consecutively failed poll-request pairs. The AP detects that either the poll or the request has been lost when its poll timeout expires. If $OD + 1$ pairs have failed, the polled member must have left the group.

Notifying the remaining members. When the AP detects that a member is leaving the group, it decides that the member shall be excluded from the membership and passes the decision to the synchronous channel. Each valid member processes the decision. When doing so, the member changes its membership and delivers a membership change message to the application.

As all members process the decisions in the order in which the AP made them, they exclude members in the same order. The protocol achieves virtual synchrony since each member processes the same *accept* decisions between two successive *exclude* decisions. When a member is leaving the group, the presented protocol ensures that the remaining members deliver a membership not including the leaving member within bound time. The protocol needs no extra messages additional to those of the atomic multicast protocols since it uses the poll-request pairs for the detection of leaving stations and the synchronous channel to notify the remaining members.

3.4.3 Joining Stations

The second kind of membership change that must be dealt with is the joining of new members. In our architecture,

team members must be able to join the group in real-time. The join protocol allows joining stations to transmit a first atomic multicast message while joining the group. To understand how joining works, it is best to think of it in the following way:

A station that intends to join the group uses the atomic multicast service to multicast a message (with resiliency *OD*) to the current membership plus itself. In our architecture, this atomic multicast message contains the scheduling-relevant parameters of the joining system. Its address is transmitted implicitly with the multicast message. When the atomic multicast service delivers the multicast message of the joining station, all members, as well as the joining station add the joining station to their membership, deliver a membership change message and then deliver the atomic multicast message. This way, joining the group essentially has the same delay as sending an atomic multicast message. Furthermore, as *accept* decisions for atomic multicast messages from joining stations and members as well as *exclude* decisions are all transmitted in the synchronous channel, total order and virtual synchrony are achieved by the properties of the synchronous channel.

However, a joining station cannot simply use the atomic multicast service to transmit a multicast message. First, the AP must be enabled to poll joining stations, meaning that the joining stations are allocated the necessary bandwidth to transfer their atomic multicast message (and their address implicitly) to the AP. Second, it must be ensured that not only the current members learn the address of the joining station, but that the joining station learns the current membership, too. Otherwise, it could not deliver a correct membership when its atomic multicast message is accepted.

Polling joining stations. The AP must allocate bandwidth to joining stations dynamically and predictably. Static allocation is not feasible for dynamically changing teams, whereas an unpredictable allocation would lead to unpredictable join delays. To achieve this by polling is a hard task. The AP cannot poll stations the addresses of which are unknown to it. Also, it cannot poll all stations that are potentially intending to join the group, as this would mean to poll all addresses possibly in use. Our solution exploits specific properties of the considered application. If, instead, a nonpredictable allocation scheme for non or soft-real-time applications or a static allocation scheme is acceptable, the upper layers of the presented membership protocol still can be used.

In the given application, the protocol has to deal with a restricted and known number of roads incident to a shared spatial resource. As joining the group is performed within bounded, short time, we can assume that, at each point of time, there is at most one joining system per road. (On each road, there may be several systems that are already members.) Each of the incident roads has an identifier that is known to the mobile systems driving on it. When a mobile system detects that it is approaching the shared resource, it starts the join protocol, providing the atomic multicast message it wants to transmit and the identifier of its road in the *join* signal.

The AP broadcasts for each road a special poll, a so-called jpoll, containing the identifier of that road. A joining station reacts to the jpolls containing the identifier of its road. Basically, it treats the jpolls in the same way as polls, namely, by sending requests including its current atomic multicast message to the AP. The only difference is that it transmits an incarnation number in the requests it sends after a jpoll, which allows the AP to detect if a station tries to rejoin the group before it is excluded from the membership. The current atomic multicast message of a joining station is the one provided in the *join* signal. The join protocol sends this message to the atomic multicast protocols when it receives the first polling message and, hence, simulates a behavior as if the application had multicast the message immediately after receiving the first poll.

When the AP receives the first request from a joining station, it learns the station's address, inserts it into the membership list, and polls the station directly, henceforth, as it does with members. However, it does not yet accept the station as a new member.

Transferring the membership to joining stations. So far, we have enabled a joining station to send an atomic multicast message to the AP. We now address how the joining station learns the current membership such that it is able to initialize the atomic multicast protocols and deliver the membership when it becomes a member.

Whenever the AP sends a broadcast on behalf of a joining station, it not only includes the station's atomic multicast message, but also a copy of the membership list. The AP constructs the copy before it sends the first broadcast on behalf of the joining station. Additionally, the AP includes the sequence number of this broadcast, allowing the joining station to reset the synchronous channel protocol such that it processes all decisions made by the AP after the copy was constructed. Thus, when a joining station has acknowledged its own atomic multicast message, the AP knows that the station has received the membership, together with the atomic multicast message, and has initialized its atomic multicast stack.

As long as a joining station has not initialized its atomic multicast stack, it stores the broadcasts it receives in a queue. When the station receives a broadcast sent on behalf of itself, it uses the membership list and sequence number provided therein to initialize its atomic multicast stack. The station then processes those broadcasts in its queue that the AP sent after it had constructed the copy of the membership list. Afterward, the station is up-to-date with the membership list of the AP and can directly process all broadcasts it will receive in the future.

According to our basic idea, when the atomic multicast service delivers a message of a joining station, the join protocol changes the membership, delivers a membership change message, and delivers the atomic multicast message afterward. When a joining station delivers its own atomic multicast message, it delivers a membership change message to the application and behaves as a member henceforth since the AP has accepted it as a group member.

Deciding when to accept a new member. Even if all members have acknowledged the atomic multicast message of a joining station, the job is not done yet. When a station

starts to join the group, several other stations might already be joining and several atomic multicasts might be in progress. Simply "overtaking" these activities could lead to severe problems with the semantics of the protocol.

Pending multicasts. When the AP inserts a further station into the membership list, several atomic multicasts (either from a joining station or a member) may be in progress. It is possible that the inserted station was not valid when the transmission of these multicast messages started. This implies that this station will possibly never receive them. Hence, the AP is not allowed to accept a station s_i as a new member until either s_i has acknowledged all atomic multicast messages with resiliency OD that have been in progress when it was inserted into membership list and are still in progress or s_i's own atomic multicast message has been transmitted $OD + 1$ times.

Pending joins. Suppose that station s_j is inserted into the membership list after station s_i. Suppose, as well, that all members acknowledge s_j's atomic multicast message before s_i's and that the AP accepts s_j as a new member. As it is possible that s_i will not receive a broadcast sent on behalf of s_j before it is accepted as a member, it could happen that s_i is not able to deliver the correct membership. Therefore, the AP cannot accept a station s_j as a new member until either each joining station that has been inserted into the membership list before s_j has acknowledged s_j's atomic multicast message or this message has been transmitted $OD + 1$ times. In the latter case, all valid stations that have been inserted into the membership list before s_j must have received it.

As the AP can always accept the atomic multicast message of a joining station after $OD + 1$ transmissions, the presented protocol ensures that a station is able to join the group in bounded time. Furthermore, since the AP transmits the *accept* decision in the synchronous channel together with other *accept* and *exclude* decisions, membership change notifications are delivered totally ordered and virtually synchronous.

3.5 Formal Description of the Atomic Multicast and Membership Protocols

The formal description presented in Table 1 uses an SDL-like syntax, but it does not claim to be a full SDL specification of the protocols, e.g., formal type definitions and variable declarations have been omitted. It is assumed that, whenever the AP transmits a broadcast on behalf of station *mem*[*cur*], it transmits the address of this station implicitly and the address is made available at the receiving stations in the variable *originator*. This is supported by the frame structure of the IEEE 802.11 Standard. It may be helpful to note that variables containing "SDU" in their names always represent data units that a protocol transmits on behalf of the next layer of the stack. If a variable *x* is an optional parameter of a signal, it may be present when the signal is received or not. For each such variable, a variable "*x*Present" is implicitly maintained, which is true if *x* is present and false otherwise.

The APs membership list *mem* is a list of tuples (*name*, *st*, *wait*, *tries*, *acked*, *lp*, *l*, *inc*, *pendBc*, *pendJo*,*jAcked*, *mem*, *memsg*). *st* is the state of the tuple, which may be *candidate*, *joining*, *normal*, or *dead*. If the state is *candidate*, *name* is the identifier of a road; otherwise, it is the *address* of a station. If the tuple represents a station, *wait* is a queue of tuples, each containing an SDU *rSDU* waiting to be transmitted, its resiliency *res*, and its local sequence number *sl*; *tries* is the number of transmissions of the first SDU in the wait queue (called the current SDU); *acked* the set of stations that acknowledged the reception of the current SDU; *lp* the sequence number of the last poll sent to the station; *l* the number of consecutively failed poll-request pairs; *inc* the incarnation number of the station. If a station is not yet a member, i.e., the state of the tuple is *joining*, *pendBc* is the set of members from which multicast messages are pending, *pendJo* the set of pending joins, *jAcked* the set of stations that acknowledged the reception of the current SDU, *mem* is the copy of the membership list that the AP transmits to the station, and *memsg* is the sequence number of the broadcast in which *mem* was transmitted for the first time.

4 Application Prototype

In order to evaluate how the communication hardcore performs in a real application, we build up an application prototype. This enables us to evaluate the performance of the hardcore in a real application context, under realistic traffic patterns. Furthermore, the performance can be assessed according to the real-time constraints imposed by a real application. There, mobile robots or automated guided vehicles (AGVs), which are already extensively used in factory automation, cooperatively coordinate their access to shared spatial resources using wireless communication. In particular, we are considering critical spots such as crossing or converging paths of the robots.

In the application prototype, a group of mobile robots uses the presented architecture to coordinate their access to a shared spatial resource. The robots are driving along traces that form two closed loops. The two loops overlap such that there is a shared space situation with the need for multirobot coordination. On each loop, there is a so-called approaching zone that starts at a certain distance in front of the shared zone. Robots detect that they are entering an approaching zone through a marking. Whenever a further robot passes a marking, each robot that is approaching the shared zone executes the scheduling function to compute a new schedule. To achieve this, the event service is used. When a robot detects a marking, it calls the event service providing its current position and velocity. The event service propagates the event to all robots that are approaching the shared zone. Furthermore, before it delivers the event, the event service determines the global state, i.e., the positions and velocities of all robots that are approaching the shared zone with respect to the same point of time. When the event service delivers the event together with the global state, the robots call the scheduling function to compute a schedule for the shared zone.

We have measured the end-to-end delay of the architecture, i.e., the time that elapses from the moment in which a marking has been detected to the moment in which the new schedule has been computed. To measure this delay, a robot takes a timestamp when it detects a marking and a second timestamp when it has computed the new schedule. The difference of the two timestamps yields the delay. Fig. 3

TABLE 1a
Formal Description of the Protocols

AP

```
process Polling
timer toPoll := 2*δm
start
  output poll(getPoll) to mem[0].name
  set toPoll

on rqu(pSDU)
    output rqu(pSDU)
l0: pSDU := getBc
    if (pSDU) output bc(pSDU) to broadcast
    call pollNext

on toPoll
  goto l0

procedure pollNext {
  cur := cur + 1 (mod size(mem))
  output poll(getPoll) to mem[cur].name
  set toPoll }
endprocess
```

```
macrodefintion
  M; {mem[j].name} 'ordered as mem'
  cm; mem[cur]
  cp; first(cm.wait)
  retryExceeded; cm.tries = cp.res+1
  acked; cm.acked ⊇ M
  id(x); 's.t. mem[id(x)].name = x'

process Redundancy

Evaluate the ack field. Store the SDU if it
has not yet been received
on rqu(ack,res optional,sl optional,
       rSDU optional)
  forall x in mem
    if (ack[x.lp - cm.lp])
      x.acked := x.acked ∪ {cm.name}
  if (rSDUPresent and (empty(cm.wait)
      or sl > last(cm.wait).sl))
    enq(cm.wait,(res,sl,rSDU))

If the current SDU needs no more retransmis-
sions, remove it from the wait queue. Transmit
current SDU.
export procedure getBc {
  if (retryExceeded or acked){
    dq(cm.wait)
    cm.acked := ∅, cm.tries := 0 }
  if (empty(cm.wait))
    enq(cm.wait,(0,0,empty))
  cm.tries := cm.tries + 1, cm.lp := sg
  if (cp.rSDU) return (sg,cp.sl,cp.rSDU)
  else return empty }

At the beginning of a slot, compute the cur-
rent round and global sequence no. and include
them in the poll.
export procedure getPoll {
  if (cur = 0) r := r + 1
  sg := sg + 1
  return (sg,r) }

'forward getDec'
endprocess
```

```
process SynchCh

Get a decision from the higher layers and in-
sert it into the synchronous channel. If there
is no SDU to transmit, just send 'sy' and the
global sequence no.
export procedure getBc {
  sy := (getDec(nodec),sy[0],…,
         sy[OD-1])
  sSDU := getBc
  if (sSDU) return (sy,sSDU)
  else return (sy,sg)
```

Station

```
process Polling

on poll(pSDU)
  output rqu(getRqu(pSDU))

on bc(pSDU)
  output bc(pSDU)
endprocess
```

```
process Redundancy

Set ack for the broadcast. Reset current SDU
if the AP implicitly ack'ed it. Deliver re-
ceived SDU, but only if it has not been re-
ceived before.
on bc(sg,sl,rSDU)
  id := id(originator)
  ack[sg-lp] := true
  if (mem[id].name = self and sl = cSl)
    cSDU := empty
  if (sl > mem[id].maxsl) {
    ouput deliver(id,rSDU)
    mem[id].maxsl := sl }

Accept 'rSDU' as the current SDU only if the
station is not busy with a preceding SDU and
if the AP will not be busy with a preceding
SDU when it must start transmitting 'rSDU'.
Assign a local sequence number to 'rSDU' and
compute the last round in which it shall be
transmitted
export procedure multicast(res,rSDU) {
  if (cSDU or lr + res < apfree)
    return overload
  cSDU := rSDU, cRes := res, cSl := cSl+1
  lrts := lr + res, apfree := lrts + res+1
  return ok }

Transmit the current SDU but not after round
'lrts'. Reset the ack field if the last poll
has not been received.
export procedure getRqu(sg,r) {
  if (r > lrts) cSDU := empty
  if (r > lr+1) ack[*] := false
  if (cSDU) rPDU := (ack,cRes,cSl,cSDU)
  else rPDU := (ack)
  lr := r, lp := sg
  if (r = lrts) cSDU := empty
  return rPDU }

export procedure reset(r){
  lr := r, apfree:=0, cSDU:=empty}

'forward procDec'
endprocess
```

```
process SynchCh
timer toSynch := 3*δm*(OD+1)

If more than OD broadcasts have been lost
raise an exception since the station has not
been valid. Otherwise, evaluate the decisions
in the order in which the AP made them
on bc(sy,sg,rem optional)
  if (sg - lsg > OD+1)
    raise amNotValid
  for (i := sg-lsg-1; i >= 0; i := i-1) {
    bcur := bcur + 1 (mod size(mem))
```

TABLE 1b
Formal Description of the Protocols (Continued)

```
'forward getPoll and rqu'
endprocess
```

```
    export csg := sg-i
    call procDec(sy[i],bcur)}
  lsg := sg
  mem[bcur].name := originator
  set toSynch
  if (remPresent) output bc(sg,rem)

on toSynch
  raise amNotValid

export procedure reset(c,s){
  bcur := c, lsg := s}

'forward getRqu'
endprocess
```

```
process AgreeOrder

If a SDU of the current member has been trans-
mitted in the last round, make a decision and
return it to the synchronous channel
export procedure getDec(d) {
  if (cp.sl != 0)
    if ((retryExceeded and cp.res = OD)
        or acked)
      d := accept
    else if (retryExceeded)
      d := reject
  return getDec(d)}
endprocess
```

```
process AgreeOrder

Save SDU until AP's decision is processed.
on deliver(id,rSDU)
    mem[id].sdu := rSDU

Act according to AP's decision. If an accepted
SDU is not present, the station has not been
valid.
export
procedure procDec(d,inout bcur) {
  if (d = accept){
    if (mem[bcur].sdu = empty)
      raise amNotValid
    ouput deliver(bcur,mem[bcur].sdu)
    mem[bcur].sdu := empty}
  else if (d = reject)
    mem[bcur].sdu := empty
  call procDec(d,bcur)}

'forward multicast'
endprocess
```

```
process LeavePolling extends Polling

Reset the counter of consecutive failed poll-
request pairs when a request is received
on rqu(pSDU)
  mem[cur].l := 0
  parent::on rqu(pSDU)

Increment the counter when 'toPoll' expires
on toPoll
  mem[cur].l := mem[cur].l+1
  parent::on toPoll

Exclude station from the membership
procedure pollNext
  if (mem[cur].l = OD+1) {
    remove(mem,cur)
    cur := cur - 1 (mod size(mem)) }
  parent::pollNext
endprocess

process LeaveNotify

Notify the group
export procedure getDec(d) {
  if (mem[cur].l = OD+1) d := exclude
  return getDec(d) }
endprocess
```

```
process LeaveNotify

on deliver(id,rSDU)
  output deliver(atomicmc,id,rSDU)

If member has been excluded, remove it and
deliver a membership change message. Adapt
'bcur' such that the next decision is related
to the correct member.
export procedure procDec(d,inout bcur) {
  if (d = exlude) {
    if (mem[bcur].name = self)
      raise amNotValid
    remove(mem,bcur)
    bcur := bcur - 1 (mod size(mem))
    output deliver(memch,import csg,M) }
  call procDec(d,bcur) }

'forward mulitcast'
endprocess
```

```
macrodefinition
Definitions of 'M' and 'id(x)' replace former
definitions
J; {mem[j].name | mem[j].st = joining}
   'ordered as mem'
M; {mem[j].name | mem[j].st = normal}
   'ordered as mem'
id(x); 's.t. mem[id(x)].name = x and
        mem[id(x)].st != dead'

process JoinPolling extends LeavePolling

If 'toPoll' expires after polling a candidate,
do not send a broadcast and not count failed
poll-request pairs.
```

```
process JoinPolling extends Polling

Transmit an incarnation number when answering
to a jpoll
on jpoll(id,pSDU) provided id = canId
  output rqu(inc,getRqu(pSDU))

Do no longer react to jpolls after receiving a
poll
on poll(pSDU)
  canId := ⊥, parent::on poll(pSDU)

Increase the incarnation number before start-
ing to join the group
export procedure reset(id){
```

TABLE 1c
Formal Description of the Protocols (Continued)

```
on toPoll
  if (cm.st = candidate) {
    sg := sg-1, call pollNext }
  else
    parent::on toPoll

on rqu(inc optional,pSDU)
  if (cm.st = candidate) {
If a station rejoins with a new incarnation
number, mark the old incarnation as dead.
    if (defined id(sender) and
        mem[id(sender)].inc != inc)
      mem[id(sender)].st := dead
If the sender of the request has not yet an
entry in 'mem', insert it and store the pend-
ing broadcasts and joins. Otherwise, ignore
the request
    if (sender not in J) {
      memInserted := true, cur := cur+1
      insert(mem,cur,(sender,joining,empty,
       0,∅,sg,0,inc,∅,∅,∅,∅,0))
      cm.pendBc := {mem[j].name |
          first(mem[j].wait).res = OD}
      cm.pendJo := J
      parent::on rqu(pSDU) }
    else{
      sg := sg - 1, call pollNext}}
  else parent::on rqu(pSDU)

Send polls to members and joining stations and
broadcast jpolls to candidates
procedure pollNext {
  if (mem[cur].l = OD+1) {
    remove(mem,cur)
    cur := cur - 1 (mod size(mem)) }
  cur = cur + 1 (mod size(mem))
  pSDU := getPoll
  if (cm.st = candidate)
    output jpoll(cm.canID,pSDU) to broadcast
  else if (cm.st != dead)
    output poll(pSDU) to cm.name
  set toPoll }
endprocess

process Join

export procedure getDec(d) {
  if (memInserted) {
Notify stations that a station was inserted
into 'mem'
    memInserted := false, d := insert}
  else if (cm.st = dead)
Exlude dead member
    d := exclude, cm.l := OD+1
  else if (cm.st = joining)
Decide if station can be accepted as a new
member
    if (cm.jAcked ⊇ M ∪ cm.pendJo
        and forall x in cm.pendBc:
          (cm.name in mem[id(x)].acked ∪
           mem[id(x)].jAcked)) {
      d := accept
      cm.acked := cm.jacked
      cm.jAcked := ∅ }
  if (d = accept)
    forall x in mem {
      x.pendBc := x.pendBc / {cm.name}
      x.pendJo := x.pendJo / {cm.name} }
  return d }

Store the acks of the redundancy protocol and
reset them to avoid that this protocol stops
transmitting the SDU of the joining station.
Transmit a copy of the membership list in
broadcasts sent on behalf of joining stations.
export procedure getBc {
  if (cm.st = joining) {
    cm.jAcked := cm.jAcked ∪ cm.acked
    cm.acked := ∅
  canId := id
  if (canId != ⊥)
    inc := inc+1 'inc is persistent' }

endprocess

process Join
exceptionhandler
  handle amNotValid
    goto l1
endexceptionhandler

start
l1: 'reset all protocols', M' := ∅
    output deliver(memch,⊥,∅)
    nextstate out

Do nothing until the join signal starts the
join protocol.
state out
  on join(sdu,canId)
    call reset(canId) to JoinPolling
    nextstate joining

  'ignore getRqu'
endstate

state joining

Store all broadcasts until the atomic multi-
cast stack is initialized. When a broadcast
sent on behalf of the station itself is re-
ceived, initialize 'mem' and the atomic multi-
cast protocols and process the stored broad-
casts.
  on bc(type,lsg optional,amem optional,
        jSDU)
    enq(rcvdBc,(originator,jSDU))
    if (originator = self){
      for i in [0,size(amem)-1] {
        mem[i] := amem[i]
        mem[i].sdu :=dummy
        mem[i].maxsl :=0 }
      call reset(id(originator),lsg)
        to SynchCh
      call reset(⊥) to JoinPolling
      while (not empty(rcvdBc)) {
        (originator,jSDU) := dq(rcvdBc)
        if (jSDU.sg ≥ lsg) output bc(jSDU) }
      nextstate waitAccept }
endstate

state waitAccept
  on bc(type,lsg optional,amem optional,
        jSDU)
    output bc(jSdu)

If an atomic multicast of a joining station is
delivered change the membership. If the multi-
cast is from the station itself, deliver a
membership change message and change to state
normal.
  on deliver(type,m)
    provided type = atomicmc
    if (mem[m.id].st = joining) {
      mem[m.id].st = normal, M' := M }
    if (mem[id].name = self) {
      output deliver(memch,import csg, M)
      output deliver(type,m)
      forall x in mem
        if (x.maxsl = 0) x.sdu := empty
      nextstate normal }
endstate

state normal
  on bc(type,lsg optional,amem optional,
        jSDU)
    output bc(jSdu)
```

TABLE 1d
Formal Description of the Protocols (Continued)

```
    if (cm.mem = ∅) {
      cm.mem := {(mem[i].name,mem[i].st) |
                  mem[i].st != candidate}
                  'ordered as mem'
      cm.memsg := sg }
    return (joining,cm.memsg,cm.mem,getBc) }
  else
    return (normal,getBc) }

'forward rqu and getPoll'

endprocess
```

```
Do not deliver a membership change message if
a joining station was excluded. Deliver a mem-
bership change message if the atomic multicast
of a joining station is delivered
  on deliver(type,m)
    if (type = memch and m.mem != M'){
        M' := m.mem, output deliver(type,m)}
    else if (type = atomicmc) {
      if (mem[m.id].st = joining) {
        mem[m.id].st = normal, M' := M
        output deliver(memch,import csg,M) }
      output deliver(type,m) }
endstate

Accept multicasts only if the station is a
member
export procedure multicast(res,rSDU) {
  if (state != 'normal') return notInGroup
  else return multicast(res,rSDU) }

When the first poll is received multicast the
SDU received in the join signal
export procedure getRqu(jSDU) {
  if (sdu) {
    call reset(jSDU.r) to Redundancy
    call multicast(OD,sdu), sdu:=empty }
  call getRqu(jSDU) }

Insert a station into the membership list if
the AP did so
export procedure procDec(d, inout bcur) {
  if (d = insert)
    insert(mem,bcur,(⊥,joining,empty,0)) }
```

shows the results of our measurements. The delays have been measured in a group of three mobile robots equipped with PC104 computers (AMD K6-266, Windows NT) and connected by an IEEE 802.11 Standard wireless LAN (2 Mbit/s for broadcast transmission). In the atomic multicast protocols, *OD* was set to 15, the resiliency equal to *OD*, and the poll timeout (corresponding to $2 \times \delta_m$) to 20ms. Because of the small number of robots and the limited physical extend of the prototype, only clock-synchronization and real-time atomic multicast have been considered in this first prototype.

The delays are distributed around a mean value of 42.76ms with a maximum of 81.90ms. It appears that the measured delays are acceptably small. For example, assume that a robot is driving at 3m/s and that the approaching zones have a length of 3m. In this setting, the robot must start braking no later than 0.6s after entering the approaching zone to be able to stop in front of the shared zone (considering a brake retardation of $3.79\mathrm{m/s}^2$). The measured delays are much smaller than this latest reaction time.

In addition to the end-to-end delay, the delays of the multicast messages that are used for event propagation have been measured. The measured delays are distributed around a mean value of 21ms with a maximum of 55.85ms. The mean value is approximately half the mean end-to-end delay. The relation of the delays can be explained by the fact that two multicast messages are needed for event propagation and global state determination in the event service. It shows that local computations have only a minor impact on the delay distribution of the architecture. It can be concluded from this observation that local computation is much cheaper than communication in terms of an increase in the overall delay.

We now consider how the end-to-end delay scales with the number of stations that the AP polls. The delay of atomic multicasts is approximately proportional to the duration of a communication round and, hence, to the number of polled stations. This also applies to the end-to-end delay of the whole architecture since, as explained above, it is mainly determined by the communication hardcore. To give an idea, for a number of 5, 10, or 20 robots, scaling the maximum (mean) measured delay yields an approximated end-to-end-delay of 136.5ms (71.25ms), 273ms (142ms), or 546ms (285ms), respectively.

5 Related Work

There are only a very few works on dynamic groups of autonomous mobile systems that use wireless communication to coordinate their access to shared resources in real-time. Wang and Premvuti [28] present an algorithm for distributed resource sharing based on a wireless medium. Exploiting the total order of messages on the medium, the systems coordinate their access to shared resources by managing a resource allocation table in a distributed manner. However, the approach is based on a special MAC protocol (CSMA/CD-W) that allows stations to detect collisions on the medium. Thus, special network hardware is needed. Furthermore, reliability is not considered. Message losses, especially asymmetric ones, may result in a failure of the resource allocation mechanism. There are other works, e.g., [3], [22], [24], in which coordination among a group of mobile systems is accomplished by the exchange of local state information, such as position or planned trajectory. Again, it is not considered how reliable and timely transmission of the information and a consistent

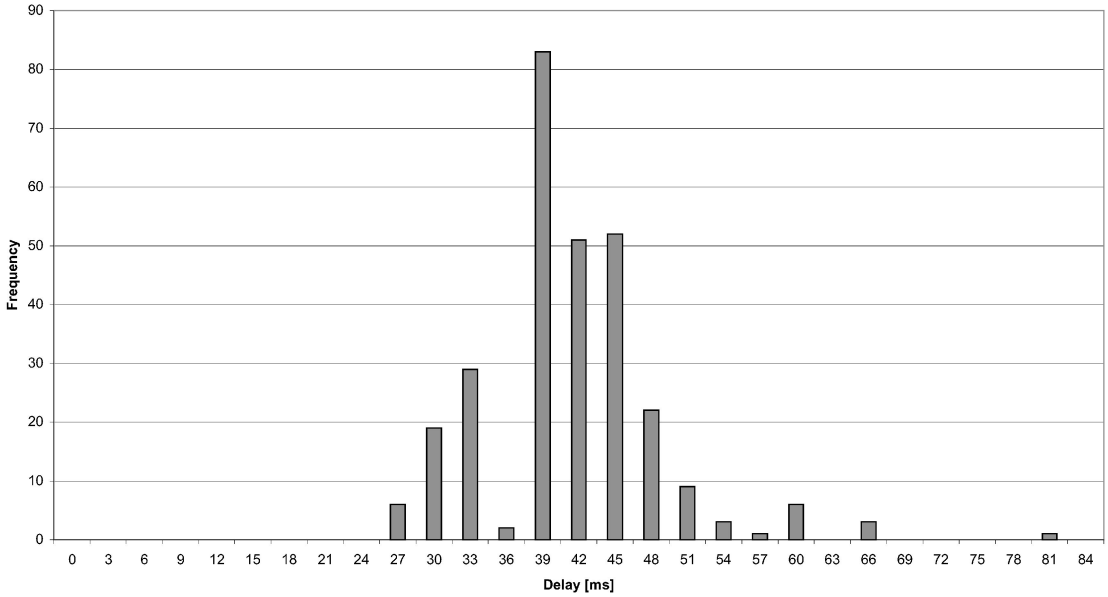


Fig. 3. Measured delay distribution of the architecture.

view of the global state among the mobile systems can be achieved. Other approaches, such as [13] and [5], completely do without communication and rely on local sensor information instead. We suppose, however, that approaches which are completely based on local information lead to less efficient solutions of the coordination problem.

In our architecture, all communication is handled by the communication hardcore, which provides fault-tolerant clock synchronization, real-time atomic multicast, and membership. Fault-tolerant algorithms for clock synchronization [8], [16], [17], [27] cope with message losses by introducing redundancy in form of extra messages. This would be a possible approach to guarantee that every system participates in each synchronization round. However, these protocols are implicitly designed for wired communication links that offer a sufficiently high bandwidth and a considerable reliability with respect to message transmission. Wireless media, however, offer only relatively low bandwidth and a poor reliability. As well, there is only little work addressing the special problems in providing reliable and timely multicasts on wireless media. There are some publications that consider reliability and real-time properties for local area wireless networks, for example, [6], [7], [12], [26]. None of these, however, provides reliable real-time group communication for wireless LANs. In [23], a hybrid token-based approach is described to accomplish totally ordered group communication for wireless networks. The work focuses on how total order can be achieved in multicell networks and does not consider time-bounded message delivery (see [20], [21] for a more detailed survey). In [2], [10], [14], [15], real-time membership protocols are presented which are similar to ours in that they are using a round-based communication structure. In [10], [14], [15], the underlying physical group is static and membership only changes because of station failure or recovery such that static bandwidth allocation can be used. This does not match to our wireless environment where group members can come from a huge set of mobile systems. Furthermore, the protocols are built on the assumption of a fault-free medium. Stability of the medium is achieved by static time and hardware redundancy in [14], [15], which cannot be applied efficiently for wireless media. The protocol presented in [2] provides a special slot in which joining systems are allowed to send join request messages to a group member. However, collisions are possible in this slot such that bandwidth allocation is not predictable and time bounded joining is not guaranteed. The protocol tolerates a bounded number of omissions. Membership protocols that have been applied in asynchronous systems are not designed to achieve predictable timing behavior. Even if the protocol presented in [4] guarantees time bounded agreement on membership, frequent message losses can cause frequent ring reformations resulting in an unpredictable timing behavior.

Our system model is similar to Cristian's timed asynchronous system model [9]. We do consider that communication between connected stations may still be subject to a high number of message losses. Assuming no message losses, as done in [9], would result in very short connected phases and, hence, in rather unstable system behavior.

6 Concluding Remarks

In this paper, we have presented an architecture that allows groups of mobile systems to schedule shared resources cooperatively based on wireless communication. In this architecture, there is a clear separation between the application-specific scheduling part that is locally executed and a general-purpose communication hardcore. An interface layer, called event service, provides a combined event and state-based interface that delivers events totally ordered together with the global state of the group at the time of event delivery. This approach has two main advantages. Regarding the application-specific requirements, it allows the designer to concentrate on optimizing the scheduling function according to the application-

specific criteria. Concerning the QoS requirements, ensuring the predictability and reliability of the architecture becomes much easier since the more sophisticated and error prone communication part is separated from the locally executed scheduling part. Thus, by isolating the communication in the hardcore, the reliability of the architecture can be improved.

The design of the communication hardcore has been the focus of the paper, giving special emphasis to real-time atomic multicast and membership services and the evaluation in an application context. The membership service allows stations to join and leave the group in real-time. Membership change notifications are delivered totally ordered and virtually synchronous. The join protocol is based on application knowledge to solve the problem of allocating bandwidth to joining stations dynamically, but yet predictably. However, it can be converted to a general solution if predictable join delays are not required or if bandwidth can be allocated statically. The membership protocols use the atomic multicast protocols and need only a minimal additional overhead.

To evaluate the communication hardcore in the context of a real application, an application prototype has been built. Successful experiments with the prototype show that the communication hardcore can be applied as the basis for the real-time cooperation of mobile autonomous systems. The measurements conducted in the prototype show that the end-to-end delay is largely determined by the delays of the communication hardcore and only to a minor extent by the local computations. This, on the one hand, justifies our decision to base global state determination in the event service on local computation as far as possible. On the other hand, it shows that, regarding the QoS of the architecture, in particular timeliness and reliability, the communication is the essential aspect. Thus, special emphasis must be put on the QoS of communication. For this purpose, we isolated all communication in the general purpose hardcore of our architecture. The protocols that constitute the hardcore have been carefully designed to work efficiently on the unreliable wireless medium. The end-to-end delays of the architecture are acceptable in our application context and it can be concluded that the delays provided by the hardcore are sufficiently small.

Our next step will be to port our Windows NT-based implementation of the hardcore to Ti Linux. We suppose that this will further improve the QoS of the hardcore. Furthermore, the possibility to modify the behavior of the operating system allows us to integrate the operation of the hardcore with the local CPU scheduling and to consider how, in addition to the hard real-time traffic, non and soft real-time traffic can be transmitted on the medium without jeopardizing the QoS of the hardcore.

Acknowledgments

The authors would like to thank their colleagues Michael Mock, Spiro Trikaliotis, and Reiner Frings, who have contributed to the results of this work, especially to the clock synchronization and atomic multicast protocols. This work is partially supported by DFG project no. NE 837/2-1.

References

[1] *IEEE Std. 802.11, Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.* New York: IEEE, 1997.

[2] T.F. Abdelzaher, A. Shaikh, F. Jahanian, and K.G. Shin, "RTCAST: Lightweight Multicast for Real-Time Process Groups," *Proc. Second IEEE Real-Time Technology and Applications Symp.,* 1996.

[3] M. Aicardi and M. Baglietto, "Use of Neural Networks in the Solution of the Control Problem for a Team of Mobile Robots," *Proc. Sixth IEEE Mediterranean Conf. Theory and Practice of Control and Systems,* 1998.

[4] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella, "The Totem Single-Ring Ordering and Membership Protocol," *ACM Trans. Computer Systems,* vol. 13, no. 4, pp. 311-342, 1995.

[5] R.C. Arkin, T. Balch, T.R. Collins, A.M. Henshaw, D.C. Mackenzie, E. Nitz, D. Rodriguez, and K. Ward, "Buzz: An Instantiation of a Schema-Based Reactive Robotic System," *Proc. Int'l Conf. Intelligent Autonomous Systems,* 1993.

[6] R.O. Baldwin, N.J.I. Davis, and S.F. Midkiff, "A Real-Time Medium Access Control Protocol for Ad Hoc Wireless Local Area Networks," *Mobile Computing and Comm. Rev.,* vol. 3, no. 2, pp. 20-27, 1999.

[7] S. Cavalieri and D. Panno, "On the Integration of Fieldbus Traffic within IEEE 802.11 Wireless LAN," *Proc. IEEE Int'l Workshop Factory Comm. Systems,* 1997.

[8] F. Cristian, "Probabilistic Clock Synchronization," *Distributed Computing,* vol. 3, pp. 146-156, 1989.

[9] F. Cristian, "Synchronous and Asynchronous Group Communication," *Comm. ACM,* vol. 39, no. 4, pp. 88-97, 1996.

[10] P.D. Ezhilchelvan and R. de Lemos, "A Robust Group Membership Algorithm for Distributed Real-Time Systems," *Proc. Real-Time Systems Symp.,* 1990.

[11] M. Gergeleit and H. Streich, "Implementing a Distributed High-Resolution Real-Time Clock Using the CAN-Bus," *Proc. First Int'l CAN-Conf.,* 1994.

[12] Y. Inoue, M. Iizuka, H. Takanashi, and M. Morikura, "A Reliable Multicast Protocol for Wireless Systems with Representative Acknowledgement Scheme," *Proc. Fifth Int'l Workshop Mobile Multimedia Comm.,* 1998.

[13] S. Kato, S. Nishiyama, and J.i. Takeno, "Coordinating Mobile Robots by Applying Traffic Rules," *Proc. IEEE/RSJ Int'l Conf. Intelligent Robots and Systems,* 1992.

[14] H. Kopetz, *Real-Time Systems.* Boston: Kluwer Academic, 1997.

[15] H. Kopetz and G. Grünsteidl, "TTP—A Time Triggered Protocol for Fault-Tolerant Real-Time Systems," *Proc. 23rd Int'l Symp. Fault-Tolerant Computing,* 1993.

[16] H. Kopetz and W. Ochsenreiter, "Clock Synchronization in Distributed Real-Time Systems," *IEEE Trans. Computers,* vol. 36, no. 8, pp. 933-940, 1987.

[17] J. Lundelius Welch and N. Lynch, "A New Fault-Tolerant Algorithm for Clock Synchronization," *Information Computing 77,* pp. 1-36, 1988.

[18] M. Mock, R. Frings, E. Nett, and S. Trikaliotis, "Clock Synchronization for Wireless Local Area Networks," *Proc. 12th Euromicro Conf. Real-Time Systems,* 2000.

[19] M. Mock, R. Frings, E. Nett, and S. Trikaliotis, "Continuous Clock Synchronization in Wireless Real-Time Applications," *Proc. 19th IEEE Symp. Reliable Distributed Systems,* 2000.

[20] M. Mock, E. Nett, and S. Schemmer, "Efficient Reliable Real-Time Group Communication for Wireless Local Area Networks," *Proc. Third European Dependable Computing Conf.,* 1999.

[21] M. Mock, S. Schemmer, and E. Nett, "Evaluating a Wireless Real-Time Communication Protocol on Windows NT and WaveLAN," *Proc. Third IEEE Int'l Workshop Factory Comm. Systems,* 2000.

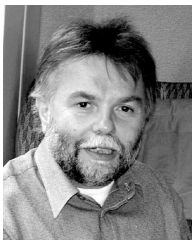
[22] F.R. Noreils, "Coordinated Execution of Trajectories by Multiple Mobile Robots," *Proc. IEEE/RSJ Int'l Workshop Intelligent Robots and Systems,* 1991.

[23] L. Rodrigues, H. Fonseca, and P. Verissimo, "Reliable Computing over Mobile Networks," *Proc. Fifth Int'l Workshop Future Trends of Distributed Computing Systems,* 1995.

[24] T. Rupp and T. Cord, "Kollisionsvermeidung mobiler autonomer Roboter durch koordinierte sensorgeführte Manöver," *Autonome Mobile Systeme,* F. Freyberger, ed., pp. 216-225, München: Springer, 1996.

[25] S. Schemmer, E. Nett, and M. Mock, "Reliable Real-Time Cooperation of Mobile Autonomous Systems," *Proc. 20th Symp. Reliable Distributed Systems,* 2001.

[26] J.L. Sobrinho and A.S. Krishnakumar, "Real-Time Traffic over the IEEE 802.11 Medium Access Control Layer," *Bell Labs Technical J.*, vol. 1, no. 2, pp. 172-187, 1996.
[27] P. Verissimo and L. Rodrigues, "A Posteriori Agreement for Fault-Tolerant Clock Synchronization on Broadcast Networks," *Proc. 22nd Int'l Symp. Fault-Tolerant Computing*, 1992.
[28] J. Wang and S. Premvuti, "Resource Sharing in Distributed Robotic Systems Based on a Wireless Medium," *Proc. IEEE/RSJ/GI Conf. Intelligent Robots and Systems*, 1994.



Edgar Nett obtained the PhD degree in computing science from the University of Bonn in 1978. He has led several research projects of the German national research center for Information Technology (GMD). In December 1991, when he habilitated at the University of Bonn, he also became a member of the faculty staff of its Department of Computer Science. At the same time, he was appointed head of the research division for responsive systems at GMD. Since the beginning of 1999, he has been a professor for technical computer science at the Otto-von-Guericke University of Magdeburg. At present, he is head of the Institute for Distributed Systems. He has done research in various areas of distributed system design, advanced information technology for concurrent engineering, and distributed real-time control. He has more than 80 refereed publications, especially in the areas of fault tolerance and distributed, real-time computing. He has served in leading committee positions for numerous workshops and conferences related to distributed, reliable, real-time, and object-oriented computing. He hosted the Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC01) and is presently the steering committee chair for the annual IEEE International Symposium on Reliable Distributed Systems (SRDS).

Stefan Schemmer obtained the Diploma in computing science from the University of Bonn in 2000. He worked as a research scientist at the German national research center for Information Technology (GMD). Since October 2001, he has been with the Otto-von-Guericke University of Magdeburg. His research interests include distributed real-time systems, wireless communication, and team robotics.