Distributed Dynamical Omnicast Routing

Felix Schill
Uwe R. Zimmer

Research School of Information Sciences and Engineering,
Autonomous Underwater Robotics Research Group,
The Australian National University, ACT 0200, Australia

The omnicast problem addresses the need of uniform information dispersion in a group of individual nodes (robots), that is, every robot needs to receive (and process) the observations processed by everybody else. The time (or the number of schedule slots) it takes to exchange all distributedly collected information between all nodes/robots is a critical limiting factor for almost all practical swarm or distributed sensing applications. The establishment of a practical distributed scheduling scheme for this purpose is therefore crucial. Actual practical constraints of limited communication ranges, low bandwidth, asynchronous entities, and disturbances on the communication channels complicate this problem.

This paper proposes a method of distributed dynamical omnicast routing (DDOR) which converges and solves the given problem, and does so with high performance ratings, as demonstrated in simulations.

The problem of fast information dispersal in short range, and limited bandwidth communication systems in heterogeneous swarms of autonomous vehicles occurs in many applications. One example is the distributed control of large schools of underwater vehicles.

1. Motivation

A practical, and for the underwater technology community highly relevant, scenario for distributed control methods is a school of autonomous underwater vehicles with broadcast communication facilities of limited range. Specifically the communication range does not span over the whole school. The size and structure of the school of submersibles are not known a priori. Some information collected locally, such as a plume sample reading, a position estimation, or any other relevant piece of information, needs to be distributed to all neighboring robots and maybe even to all robots in the school. The only available means of communication is local broadcast. Colliding messages will lead to an unreadable signal which cannot be distinguished from noise on the communication channel by the individual nodes. Furthermore the communication chan-
Figure 1. One of the Serafina autonomous submersibles.

Figure 2. Long-wave radio transmitter inside Serafina.

channels are affected by stochastical external disturbances. A number of different channel models can be chosen, but for our purposes it only needs to be assumed that an infinitely repeated noncolliding message-transfer will lead to the reception of this message by all in-range nodes eventually.

40 cm long Serafina class autonomous submersibles (Figure 1 shows a graphical superposition of the outer hull and parts of the internal structure) serve as a test-bed for experiments of distributed dynamical omnicast routing (DDOR). One of the employed communication channels is long-wave radio with antennas and codec-interface as shown in Figure 2. The other employed mode is short-range optical communication as demonstrated in [1]. While individual Serafinas are ready-to-run and larger schools are currently built, they are not yet operational. Thus, results based on physical experiments concerning the omnicast message passing system are not yet available.

As given in [2] the omnicast problem and its optimal solution is technically specified as follows.

**Definition 1 (communication network)** Let \( G = (V, E) \) be a graph describing a communication network with \( n = |V| \) nodes and \( m = |E| \) edges. Each edge \( e_i = (v_1, v_2) \in E \) implies that \( v_1 \) is in communication range of \( v_2 \) and vice versa (symmetrical links).

**Definition 2 (local neighborhood)** The set of all nodes which are connected to an individual node \( v_1 \in V \) will be called the local neighborhood of \( v_1 \).

**Definition 3 (collisions)** If two nodes out of \( v_1 \)'s local neighborhood are sending at the same time, no message is received at \( v_1 \) and the situation is called a (communication) collision. This is also true if the messages sent from neighboring nodes overlap only partly in time.

**Definition 4 (the omnicast problem)** In the start state, every node \( u \in V \) has a set \( I_u(t_0) \) of information tokens, which contains exactly one unique token \( B_u \) of information. During the communication phase, a node \( v \) updates its set \( I_v(t + 1) = (I_v(t) \cup I_w(t)) \), if and only if it successfully receives a message from \( w \in V \) (i.e., no collision occurred with respect to \( v \)) in time step \( t \), and \( I_v(t+1) = I_v(t) \) otherwise. The end state \( t_f \) is reached when all nodes have the full set with all tokens \( I_u(t_f) = \{ B_v \mid V \in V \} \) for all \( u \in V \).

**Definition 5 (the optimal omnicast problem)** Find a schedule \( S_G = (T_1, \ldots, T_t) \), \( T_i \subset V \) for \( i = 1, \ldots, t \), with \( T_i \) being the set of sending nodes in time step \( i \), such that \( S_G \) solves the omnicast on the network graph \( G \), and \( t \) is minimal.

Now the task considered here is straightforward: Construct a global schedule which solves the given omnicast problem (although not necessarily the optimal omnicast problem) by means of distributed scheduling while considering a realistic communication system.

The solution presented guarantees solving the omnicast problem and also produces a highly efficient schedule.

## 2. Related work

In [3] Herman and Tixeuil present an algorithm for collision-free, distributed time division multiple access (TDMA) scheduling which is in many regards close to the approach presented here. The differences stem from the fact that Herman and Tixeuil assign schedules in a locally centralized fashion, that is, local groups (clusters) elect a leader which then works out a schedule for the whole cluster. The leaders themselves are globally ordered. While this scheme quickly comes up
with a collision-free schedule in the static case, it also exposes significant reorganization which can impact large network areas in the dynamical case. Thus the introduction of a single new node can change all schedules in the network. While the DDOR approach discussed here can (but does not need to) be slower in its initial schedule construction phase, it behaves more gracefully in the dynamical case and any local change can only affect a limited part of the network.

Many distributed or concurrent scheduling tasks have been proven to be NP-complete. This has not yet been done for the omnicast problem. For an overview of distributed scheduling tasks and some of their known solutions or approximations see [4, 5]. Another interesting and useful discussion about the limitations and possibilities of stochastical versus deterministic scheduling schemes are given in [6]. Due to assumptions of very limited bandwidth, a deterministic schedule is produced by the DDOR method. Still, unavoidable, nondeterministic factors during the build-up and adaptation of those schedules need to be taken into account (i.e., message losses due to initial message collisions and other disturbances).

3. Timing

After starting asynchronously, all nodes synchronize to a common time-slot pattern. This process is based on standard distributed clock synchronization (e.g., [7]) and is not discussed here. They also agree on a global logical clock which identifies all time-slots uniquely, that is, at any given time a node corrects its own logical clock if it receives a signal from another node which shows a more advanced time. Individual time-slots are composed of two major parts. First, the main message, containing the application-level message contents as well as a logical time stamp and the current local schedule (as assumed by the sending node). Second, a short request packet consisting only of a requested time-slot number and the index of the requesting node. The rest of the time-slot structure is given by the constraints of the communication channel and the speeds of the communication controllers. See Figure 3 for the precise schedule slot structure.

4. States and data structures

Each node \( i \) is in one of three different states \( T_i \) at any given time: \( T_i \in \{ \text{listen, request, run} \} \). The initial state of all nodes is “listen.” After each listen phase it is determined if the node \( i \) is represented in its own schedule \( S_i \). If this is the case then the mode is changed to “run” and the schedule is executed. Otherwise a “request” is composed, broadcasted, and the mode is changed back to listen. Changes in the network (or message eliminations during the distributed schedule construction)
might lead to eliminating node $i$ out of its own schedule at any time. In that case, $i$ falls back to listen mode. All nodes $j$ from which messages have ever been received by node $i$ are included in the set of visible nodes $V_i \subseteq A$, where $A$ is the set of all active nodes. The actual listen-deadlines depend on the communication model and assumed levels of interconnectivity, that is, the average number of nodes in broadcast range.

The most central data-structure in each node $i$ is the local schedule $S_i$ that consists of a list of ordered schedule slots $k \in [1, \ldots, S^i_L]$ which can be empty “e,” blocked “b,” or occupied by a node $j \in V_i$. Schedule length $S^i_L$ varies and is defined to include all slots which are not empty in the schedules of the visible neighborhood (including the node itself). Since the local schedule is attached to every outgoing message, it can be assumed in the following that a node $i$ has access to all schedules $S_j$ for $j \in V_i$ after the schedule has been executed once (or multiple times, if stochastical message loss is considered as well).

5. Routing method

This section introduces the actual DDOR method. The first part will motivate the scheduling strategies, while the second part denotes the exact formulation in predicate logic terms.

Here are the assumptions which are employed during requests.

- Prefer earlier scheduling slots when requesting.
  It is assumed that a ranking of slots produces “denser” schedules. As it turns out in the experimental section, this seems justified.

- Request slots which will not lead to collisions.
  (Assuming the current set of schedules is collision-free.) This enables a consistent embedding of the local node into the surrounding schedules. During the execution of the schedule there may be conflicting send operations (in order to increase overall efficiency), but at least one slot is guaranteed to be collision-free for each node.
- **Break existing collisions.**
  If a collision is detected within the schedules of the visible neighbors, then the already colliding slot might be additionally requested by the observing node, in order to make the potentially hidden conflict obvious to the local neighborhood \( V_i \). In such cases both (or multiple) conflicting nodes withdraw from this slot and the requesting node gains the currently overloaded slot.

These three concepts guide the actual construction of the local schedule.

- **Accept external requests immediately.**
  The external request is either based on complete and sound information about the current schedules (then the request will embed consistently), or it is based on wrong or incomplete information, or the request is meant to correct an existing inconsistency. In all cases (which cannot be distinguished locally) the request can or should be incorporated into the local schedule immediately. In the rare case that an ill-informed request breaks an otherwise consistent schedule, it needs to be accepted. As in all cases of inconsistent partial schedules, all involved nodes are forced to reschedule. This could only be overcome if an external observer (which does not exist in this setup) could decide which part of the schedule is most worthwhile to keep, and thus reduce the dynamics of rescheduling to some degree.

- **Copy agreed slot assignments.**
  Schedule slots which share a common assignment across all known schedules from the visible neighborhood are copied into the local schedule. Besides explicit requests, all slots (including empty slots) are exclusively assigned via this method. This implies that no node can add its own id into the local schedule unless confirmed by the visible neighbors.

- **Block known slot assignments of invisible nodes.**
  Nodes that are not contained in the visible neighborhood but are mentioned in neighboring schedules are considered blocked (\( b \)) slots in the local schedule. No message is expected to be received in those slots, but a local sending operation might result in a collision in neighboring nodes. Therefore those slots are not available for requests.

A more precise formulation of these principles is given in the remainder of this section.

The schedule slot \( k \) in each node \( i \) (denoted by \( s_{i,k} \)) is given below in equation (1). In the first case that a node is found in the same slot in the schedules of the visible neighborhood and there is no conflict with other assignments (\( \neg C_{i,k} \)) then this node is entered into the current schedule. An empty slot \( e \) is assigned if all slots \( k \) in the visible neighborhood \( V_i \) of \( i \) are empty (\( E_{i,k} \)), or two entries in slot \( k \) are occupied by different nodes out of \( V_i \) (\( C_{i,k} \)), or the slot is currently occupied by the node \( i \) itself and there exists a slot \( k' < k \) which is empty (\( O_{i,k} \)), or there are

*Complex Systems, 16 (2006) 299–316*
multiple entries of the same node in the current schedule \((M_{i,k})\). If no entry in the visible neighborhood \(V_i\) can be found \((\neg V_{i,k})\), but there are entries belonging to the invisible neighborhood of \(i (S_{i,k})\), then this slot is blocked \((b)\). All three cases are only applicable if the node is not in listen mode \((\neg L_i)\). This is important since otherwise recent requests would be eliminated from all schedules before they had a chance to establish themselves. Finally any occurring request is immediately entered into the current schedule. If none of the given cases apply then the entry \(s_{i,k}\) is left unchanged. The schedule \(S_i\) (together with every sent message) is broadcast by node \(i\) to its local neighborhood:

\[
s_{i,k} = \begin{cases} 
  s_{i,k}; & \neg L_i \land (\exists j \in V_i, s_{j,k} \in V_i) \land \neg C_{i,k} \\
  e; & \neg L_i \land (E_{i,k} \lor C_{i,k} \lor O_{i,k} \lor M_{i,k}) \\
  b; & \neg L_i \land S_{i,k} \land \neg V_{i,k} \\
  j; & (\exists r_j, r_j = k) \\
  s_{i,k}; & \text{otherwise}
\end{cases}
\] (1)

with

\[
E_{i,k} = (s_{i,k} = e, \forall j \in V_i)
\] (2)

\[
C_{i,k} = (\exists j_1, j_2 \in V_i, \mathcal{U}(s_{j_1,k}) \land \mathcal{U}(s_{j_2,k}) \land s_{j_1,k} \in V_i \land s_{j_2,k} \in V_i \land (s_{j_1,k} \neq s_{j_2,k}))
\] (3)

\[
S_{i,k} = (\exists j \in V_i, \mathcal{U}(s_{j,k}) \land s_{j,k} \notin V_i)
\] (4)

\[
V_{i,k} = (\exists j \in V_i, \mathcal{U}(s_{j,k}) \land s_{j,k} \in V_i)
\] (5)

\[
O_{i,k} = (s_{i,k} = i \land (\exists k' < k, s_{i,k'} = e))
\] (6)

\[
M_{i,k} = (s_{i,k} = j \land (\exists k' \neq k, s_{i,k'} = j))
\] (7)

\[
L_i = (T_i = \text{listen})
\]

\[
R_i = (T_i = \text{request})
\]

\[
A_i = (T_i = \text{run})
\]

\[
\mathcal{U}(s) = (s \neq b \land s \neq e).
\] (9)

Equation (1) holds at all times, but is evaluated in practical implementations only before every potential message sending operation and with any request which is received or transmitted.

Each node which is not in listen mode and finds itself not represented in its own schedule, formulates a request for a specific slot \(r_i\)

\[
r_i = \begin{cases} 
  f_i; & \neg L_i \land \neg I_i \land V_i \neq \emptyset \\
  \emptyset; & L_i \lor I_i \lor V_i = \emptyset
\end{cases}
\] (10)

\[
f_i = k \mid (s_{j,k} = e \land (\exists k' < k, s_{i,k'} = e))
\] (11)

\[
I_i = (\exists k, s_{i,k} = i)
\] (12)

before it falls back to listen mode (again).

If a node is not in listen mode and is represented in its own schedule it will employ this schedule for sending. More precisely, if the slot identified as currently active (see section 6 for the selection process)
contains the local node, then this node will use the active slot for message transmission. Those transmissions are guaranteed to be collision-free. Moreover if the currently active slot is empty it is also employed for sending, but in this case it might collide with other messages.

6. Expandible schedules

The DDOR method is based on the concept of listening to an existing set of schedules in proximity of a new node, and composing a subjective suggestion (request) for integrating this node into the set of perceived schedules. In order to achieve global convergence, this change needs to be as local and nondestructive as possible. As the received schedules can be densely packed (no free slots), the solution to this issue is not directly obvious.

One way of addressing this issue would be to ensure that no schedule will ever become dense, unless all possible nodes are integrated into a schedule. This can be achieved by setting the minimum number of local schedule slots to a global value which covers all possible schedules anywhere in the network and to step linearly through all slots (possibly filling empty slots with additional, but not necessarily collision-free broadcasts). For strictly homogenous networks this might be a reasonable way, but for dynamical and heterogeneous networks this implies that a vast number of schedule slots are left empty at all times.

If the possibility of densely packed schedules is considered (or even encouraged) then there must be a way to change those schedules with the smallest possible impact on the surrounding schedules. As at least one node needs to change its sending pattern, a simple implementation would be to ask one node to halve (or otherwise reduce) its sending frequency and therefore make room for another node. Such a change has no destructive impact on the surrounding schedules, since no node has been shifted out of its original schedule slot.

There are many ways to ensure such behavior. One of the most canonical ways is to always halve the broadcasting frequency of the last node in the schedule. Expanding schedules in this exponential pattern would therefore always be dense and will not shift existing schedules. The active slot $s_a^i$ in node $i$ is thus determined at logical time $L$ by

$$s_a^i = \max\{k \leq S^L_i \mid L \mod 2^{k-1} = 0\}$$

so that the scheduling patterns for growing $S^L$ will appear as

11111111111111111111111

**Ex**121212121212121212121212

**Ex**121312131213121312131213

**Ex**121312141213121412131214

... (14)

The numbers represent schedule slot numbers, which means that the number of schedule slots in a local schedule has no impact on the density or global timing pattern of a neighboring schedule possibly having a different number of slots. The major drawback of this approach is that the sending frequencies in higher slot numbers are reduced exponentially.

This problem can be resolved, if a minimal number of schedule slots in each schedule can be reasonably assumed. Then a linear schedule is implemented for the first $S_{Li}^{L}$ slots, where all actual schedules carry at least one more slot available for exponential expansion, $S_{Li}^{L} > S_{Li}^{L}$:

$$s_{i}^{L} = \begin{cases} k \mid (L + k) \mod (S_{Li}^{L} + 1) = 0; & \text{Lin} \\ \max \{k < S_{Li}^{L} \mid L \mod 2^{k-S_{Li}^{L}-1} = 0\}; & \neg \text{Lin} \end{cases}$$

$$\text{Lin} = (L \mod (S_{Li}^{L} + 1) \neq 0) \tag{15}$$

so that the scheduling patterns for growing $S_{Li}^{L}$ now appear as

$$\begin{align*}
1 & - - - 1 - - - 1 - - - 1 - - - 1 - - - 1 - - - \\
& \text{wr} 12 - - - 12 - - - 12 - - - 12 - - - 12 - - - \\
& \text{wr} 123 - - - 123 - - - 123 - - - 123 - - - 123 - - - \\
& \text{wr} 123412341234123412341234123412341234 \\
& \text{wr} 12341235123412351234123512341235 \\
& \text{wr} \ldots
\end{align*} \tag{16}$$

The linear schedule length in this example is set to three. Reasonable linear schedule lengths can be estimated based on the knowledge of a mean or a minimal local node density. This gives a meaningful lower bound because any local schedule will contain at least as many nodes as found in its local neighborhood plus the node itself. As “two-hop” neighbors also have an impact on a local schedule, the practical minimal schedule length is even (slightly) larger.

### 7. Analysis

**Lemma 1.** DDOR converges.

**Proof.** As long as new nodes are added monotonously to the schedule, a schedule with all nodes included will be achieved. Therefore the two cases in which nodes are removed from or not included in the schedule need to be considered. First, nodes are removed from the schedule if a local collision $C_{i,k}$ in slot $k$ is detected by node $i$. As the collision is caused by a lack of information about the local network structure (i.e., the existence of the involved conflicting nodes was unknown to each other at the time of schedule establishment) and this missing information is now broadcast to all involved nodes, this collision cannot occur again. As the number of possible collisions is limited by the size of the network, this node deletion will not endanger convergence. The second case of...
nodes not including themselves after the request phase is caused by lost or colliding request packets. As it is assumed that not all packets are lost at all times, and unsuccessful nodes will enter an increasingly long listing time with an additional random component, each node will eventually encounter a situation where packets are received correctly and no other node will try to request the same slot at the same time. Thus both cases will only delay but not violate convergence and the overall schedule will converge.

**Lemma 2.** DDOR converges to an omnicast solution.

*Proof.* As all nodes will be included in a converged global schedule at least once with a collision-free sending slot and this schedule is repeated infinitely, at least one bit of new information is received by one node in the network with every schedule hypercycle, and thus this schedule implements omnicast.

**Lemma 3.** DDOR does not necessarily converge to an optimal omnicast solution.

*Proof.* (By counter example.) Consider a network of four nodes as given in Figure 4. An optimal omnicast solution takes four steps to complete (lower schedules) while a solution generated by DDOR (might) take five steps (upper schedule). The solution space accessible by means of the DDOR method is limited by the structure of the underlying schedules as introduced in section 6. In the case of the example given in Figure 4, the optimal solution is not in the syntactical solution space of DDOR schedules, so it can never be achieved.

As optimal solutions are not necessarily to be expected, it is of course relevant how close the solutions will come to the optimal one. Although an analytical answer has not been found at the current time (see section 8 for experimental answers), it still can be shown that dense schedules are achieved. “Dense schedules” in this context means that no slot is
left unused unless there is an immediate danger of information loss by message collision of the possibly only message sent by a specific node during the complete schedule.

**Lemma 4.** DDOR produces dense schedules.

**Proof.** In a converged solution every node has one assigned slot which is guaranteed to be collision-free. Besides the local node itself and all immediate neighbors (every one exactly once) there can also be empty slots (only with higher slot-numbers than the local node’s slot) and blocked slots in the local schedule. All empty slots are also used for sending (even if those slots can lead to message collisions) while blocked slots ensure that the one collision-free slot for a node in the two-hop neighborhood is not violated. Thus besides the single guaranteed collision-free slot for each node in the two-hop neighborhood, all slots are employed and the schedule is dense. ■

In practice DDOR produces almost or completely collision-free schedules as demonstrated in section 8. But collisions are not a problem *per se* because a few collisions in the schedule might even enhance the overall performance, as many optimal omnicast solutions do include collisions as well.

While a connection between the denseness of a local schedule and the overall efficiency with respect to the omnicast problem seems obvious, a direct analytical relation is not known yet.

### 8. Experiments

The optimal omnicast problem as specified is measured globally and only once, that is, the periodicity of the optimal schedule is exactly the number of steps of the optimal solution. As it is assumed that there is no global communication between nodes, this criterion needs to be reformulated for the distributed and continuous, that is, the realistic case. Each node maintains and broadcasts a token vector containing token-version numbers of all nodes:

\[ I_i = \{ v_j \mid j \in A \} . \quad (17) \]

For every received token vector \( I^R = \{ v^R_j \} \) the individual maximum version numbers are determined and stored:

\[ \forall j \in A : v_j = \max [v_j, v^R_j] . \quad (18) \]

Each node switches to a new version number if all locally known version numbers are at least as large as the local one:

\[ v_j = \begin{cases} v_j + 1 ; & v_k \geq v_j, \forall k \\ v_j ; & \exists k, v_k < v_j . \end{cases} \quad (19) \]

*Complex Systems, 16 (2006) 299–316*
The mean number of steps between version updates $R^M_i$ is measured locally. This number converges to the same value for all nodes.

The simulation is implemented on a real-time basis employing separate asynchronous processes for each node. An underlying message distribution system accepts messages by any node and delivers them according to damping values in the adjacency matrix (not accessible by the individual processes which are emulating the network nodes) and disturbance assumptions. Multiple messages and requests which are overlapping in time are eliminated locally by each node (and are not passed to the local implementations of the DDOR algorithm). The Ada95-based implementation of DDOR (and the simulation environment) is available from the authors on request. All data structures are static and all computational complexities are constant, so that the transfer of the code to actual communication microcontrollers is a straightforward process.

Random generators are employed in the calculation of listen times and the occasional omission of sending slots. On average every tenth scheduled sending slot is left unused.

The central consideration of all simulations is to find experimental upper and lower bounds as well as variances for the number of round trip steps with various topologies and networks of different sizes. Therefore schedules for each investigated topology are established for more than 40 times from scratch and executed until the average number of round trip steps stabilizes within adequate bounds (considered stable to the second digit after the decimal point). Variances in these measurements stem from the fact that (significantly) different schedules are generated depending on the order of successful request calls. The schedules also need to be executed for a long time (>100 round trips) in order to reliably establish an average, since a single random send omission or message loss can multiply the measured individual number of round trip steps by a factor of two. Note that a complete round trip has to incorporate the latest token versions of all nodes.

First a set of five 12-node networks (Figure 5) is investigated in detail (“line” topology not shown as it is the same as the “ring” topology, but with one connection removed). Specifically networks with different density distributions (in terms of number of nodes in range for each individual node) are tested. As can be seen in Figure 7 the line and ring topologies provide the best and worst cases—even though the difference is just one connection. The performance differences are explained by the fact that DDOR schedules—although generated in a distributed fashion—tend to implement specific symmetries in the schedules as induced by the network. That is, there is usually one preferred direction of information progression, while the opposite direction is only served at half the speed. See, for example, the schedule produced for the homogeneously distributed 12-node network in Figure 8. The step pattern on the right-hand side expresses a preference of information dispersion in...
one direction. In a ring topology case this does not restrict the solution space, but in a line topology case the information stemming from one of the outmost nodes is progressing with only every second schedule step towards the opposite end. This explains the approximate factor of two between the ring and the line performance.

The more relevant topologies are symmetric/structured networks (Figure 5(d)), homogenous networks (the connection density is similar or the same all over the network), and heterogeneous networks (meaning the connection density varies) (Figures 5(a) and 5(c)). As can be seen in Figure 7 all three practically relevant cases show a comparable performance over all test-runs, and specifically there are no outliers. This may not seem astonishing, but in fact it is very easy to come up with schedules, which seem dense and compact, but that slow down the round trip performance by factors. This became apparent in early development stages of distributed schedulers (leading to the DDOR scheme eventually) which showed performances beyond 50 or even 60 steps usually.

The homogeneous 12-node topology (Figure 5(a)) deserves an even closer look as it is the closest to our practical application, namely almost uniformly distributed submersibles trying to establish a means of global information exchange. The networks show cycles of different length and in different coupling positions. This poses a specifically hard problem for any distributed scheduling system, as schedules which are constructed locally and step-by-step need eventually to be harmonized/synchronized with schedules that have been created concurrently—and which will most likely run on a different base frequency (given by the hypercycle of the local schedules). This situation has to be resolved many times over in the topology given in Figure 5(a), thus the slightly larger variations in comparison with other network types. Still it behaves extremely well in all experiments, as the assumed optimum number of round trip steps for this specific network is 11 for the collision-accepting case, and 13 for the collision-free case. While the actual optimum can no longer be calculated by means of extensive search for networks of that size [2], a reasonable solution is delivered by the distributed scheme suggested in this paper after approximately 100 to 300 exchanged messages. A general upper bound of \(2n - 2\) for optimal omnicast on a connected network with \(n\) nodes could be proven (see [2]). The measured results actually stay below this upper bound for optimal solutions in all experiments.

Different schedules are generated for the homogeneous topology based on random orders of messages and requests as well as message and request collisions during the start-up phase (see the schedules in Figures 8 and 9)—darker slot entries represent possible collisions (redundant transmissions), while the very dark entries represent definite collisions. The schedule in Figure 10 demonstrates a frequent phenomenon during schedule generation, namely the creation of “blind spots” in the network.
Figure 5. Test networks with 12 nodes—connected nodes are in broadcasting range.

Figure 6. Test network with 48 nodes—connected nodes are in broadcasting range.

Figure 7. Round trip steps for 12-node networks.

Figure 8. Generated schedule 1.

by synchronized neighbors with regularly colliding messages, such that the node between those synchronized senders will not receive any message. In the example here nodes 5 and 7 will not receive any messages during the schedule slot in which nodes 3, 6, and 9 send synchronously. This situation (which might or might not prevent omnicast) will only be resolved after one of the synchronized nodes eventually skips one of its sending slots and thus enable the middle node to receive a message. After the middle nodes collect enough information to detect this conflict, they will request exactly the slot that was problematic, forcing the formerly sending nodes to request another slot. Therefore this situation is resolved for good. The sending omission frequency is of crucial impact here, setting it too high will reduce overall performance, and setting it too low will not resolve such conflicts quickly. In systems which are considered stable, this frequency can be decreased after a successful schedule has been implemented. For the purpose of this paper, it is assumed that nodes leave or enter at any time. Thus there cannot be specific parameter settings for specific phases.
Finally the critical test of scaling. Will the DDOR algorithm scale well with larger networks? For this purpose a more complex, realistic, homogenous density network has been generated (Figure 6). Comparing the results as given in Figure 11 to those of the 12-node networks (Figure 7), it is a pleasant surprise that the average performances are even better (with respect to the number of nodes) and the variance does not increase. The number of round trip time steps are now almost always smaller than the number of nodes in the network, which is very close to the estimated optimum for this network (calculating the actual optimum for a network of 48 nodes by exhaustive search is impossible). Even though the given measurements only consider specific kinds of networks, they are a highly relevant class of networks for practical purposes.

9. Conclusion

This paper presents a distributed scheduling scheme which solves the omnicast problem under practical considerations of limited communication ranges and uncertainties. The performance measurements are close to the presumed optima and the algorithm itself is well suited for computationally low-power embedded systems.

Alternative scheduling methods might include spontaneous methods, which do not promote convergence to a static schedule, but do provide the earliest possible start of message exchanges. Those methods did not perform as well in terms of overall round trip times in preliminary experiments, but they skip the start-up phase, or reconfigure phases, which are required in the method discussed here. So for specific purposes where short start-up times are more important than overall efficiency, those methods might need to be discussed in more detail. For the
applications considered in the group of the authors (distributed control of schools of submersibles), distributed dynamical omnicast routing (DDOR) has been identified as the method of choice. Practical tests with schools of submersibles will follow as soon as a larger fleet of vehicles become operational. The autonomous underwater research group at ANU is currently also working towards this technical goal. Reliable and short information round trip times in self-organized or distributed controlled swarms of vehicles under realistic constraints are considered a fundamental requirement for useful applications in this field—the existence of adequate methods has been successfully demonstrated in this paper.

References


