Regional Synchronization for Simultaneous Actions in Situated Multi-agent Systems

Danny Weyns and Tom Holvoet

AgentWise, DistriNet, Department of Computer Science
K.U.Leuven, B-3001 Heverlee, Belgium
{danny.weyns,tom.holvoet}@cs.kuleuven.ac.be

Abstract. Agents of a multi-agent system (MAS) must synchronize whenever they want to perform simultaneous actions. In situated MASs, typically, the control over such synchronization is centralized, i.e. one synchronizer has the supervision on all agents of the MAS. As a consequence, all agents are forced to act at a global pace and that does not fit with autonomy of agents. Besides, global synchronization implies centralized control, in general an undesirable property of MASs. In this paper we present an algorithm that allows agents to synchronize with other agents within their perceptual range. The result of the algorithm is the formation of independent groups of synchronized agents. The composition of these groups depends on the locality of the agents and dynamically changes when agents enter or leave each others perceptual range. Since in this approach agents are only synchronized with colleagues in their region, the pace on which they act only depends on the acting speed of potential collaborating agents. The price for decentralization of synchronization is the communication overhead to set up the groups. In the paper, we discuss experimental results and compare the benefits of regional synchronization with its costs.

1 Introduction

Whenever agents of a multi-agent system (MAS) interact by performing *simultaneous actions* they need to synchronize. With simultaneous actions, we mean a set of interfering actions that are executed together and that produce a compound result[1]. We distinguish between three kinds of simultaneous actions: *joint actions*, *influencing actions* and *concurrent actions*. Joint actions are actions that must be executed together in order to produce a successful result. An example of joint actions is two or more agents that pick up an object that none of them can pick up by itself; or agents that carry such object to a certain location together. Influencing actions are actions that positively or negatively affect each other. An example of influencing actions is two agents that push together the same object. When they push the object in the same direction it likely moves faster, however when they push it in opposite directions the object might not

[1] We do not take *independent actions* that happen together into account. Independent actions do not interfere with one another and as such do not affect each other.

V. Mařík et al. (Eds): CEEMAS 2003, LNAI 2691, pp. 497–510, 2003.

move at all. Finally, concurrent actions are simultaneously performed actions that conflict. An example of concurrent actions is two or more agents that try to pick up the same object at the same time. When only one of the involved agents can get the packet, synchronization resolves which of the agents this will be, i.e. typically a non-deterministic selection. Other researches make a similar distinction between different types of interacting actions. We give a number of examples. Allen and Ferguson [1] differentiate between 'actions that interfere with each other' and 'actions with additional synergistic effects'. Bornscheuer and Thielsher [3] define the notion of 'compound actions', i.e. 'a non empty, finite subset of a given set of unit actions'. The execution of a compound action is modeled as the manipulation of the composing subset of unit actions. Boutilier and Brafman [4] distinguish 'concurrent actions with a positive or negative interacting effect'. Griffiths, Luck and d'Iverno [6] introduce the notions of a 'joint action that a group of agents perform together' and 'concurrent actions, i.e. a set of actions performed at the same time'. These definitions are build upon the notions of 'strong and weak parallelism' described by Kinny [8].

Most of the work regarding simultaneous actions in MAS is done in the context of languages for action description and planning for agents. In this paper we focus on simultaneous actions for situated MASs, where agents execute *situated actions*. A situated action is an action selected by an agent on the basis of the position of the agent, the state of the world which he perceives and limited internal state[2]. For planning–agent systems, simultaneity of actions is realized through planning. In situated MASs however, agents do not coordinate through planning but select their actions according to the actual situation. In order to allow agents to coordinate their interactions, support for synchronization is required. Such synchronization guarantees, that simultaneous actions that conceptually must happen together, but that are executed separated in time[3], are treated as if they happened together.

The focus of this paper is on the *achievement* of synchronization. The formation of groups of synchronized agents determines the *granularity* of synchronized groups, i.e. the number of agents that belong to a group of synchronized agents. Typically, synchronization for simultaneous actions is organized for the entire group of agents of the MAS, examples are Ferber's synchronizer [5] or Look, Talk and Do Synchronization [12]. The major advantage of this approach is its simplicity: there is no overhead to setup synchronization, a single synchronizer controls the synchronization of all agents. However centralized synchronization has a number of disadvantages. A single synchronizer centralizes control and that conflicts with the distributed nature of MAS. Since all agents are synchronized with one another, the activity in the MAS evolves at the pace of the slowest agent. Agents that have concluded their action selection are blocked by

[2] Wavish and Connah [10] adopted the concept of situated action in MAS for the stimulus/response–like actions of an agent that are only related to the agent's external perception. In this paper we allow a situated action to be influenced also by limited internal state of the agent, e.g. a commitment in an ongoing collaboration.

[3] E.g. on a single/sequential processor system.

the synchronizer until all other agents of the MAS have concluded, even if their current state or situation does not require synchronization with the other agents. Besides, centralization of control makes the system more vulnerable to errors. When e.g., the synchronizer invokes the agents to select their next action, and for some reason one of the agents fails, without special provisions the synchronizer waits for the completion of the action selection of the failing agent for ever, leaving the system in a deadlock. In this paper we present an algorithm that allows agents to dynamically synchronize with other agents only if they are located within each others perceptual range. Agents organize themselves in synchronized groups, resulting in a much finer grained synchronization. The composition of the groups depends on the locality of the agents and dynamically changes when agents enter or leave each others perceptual range. Since in this approach agents are only synchronized with colleagues in their region, the pace on which they act only depends on the neighboring and thus potential collaborating agents.

The price for decentralization of synchronization is an overhead of communication to set up groups of regional synchronized agents. In the paper, we discuss experimental results and compare the gain stemming from decentralizing the synchronization with its costs.

The rest of the paper is structured as follows. In section 2 we discuss the algorithm for regional synchronization in detail. Section 3 reports simulation results and evaluates the algorithm. Finally, we conclude and look to future work in section 4.

2 Discussion of the Algorithm

In this section we discuss the algorithm for regional synchronization. First we give a high–level overview of the algorithm and zoom in on the goal of the algorithm. Then we discuss the main challenges we have to deal with. Next, we introduce a number of definitions for the major concepts used in the algorithm. Finally we present the algorithm and discuss each step in detail.

2.1 Goal of the Algorithm

Synchronization of simultaneous actions must ensure that conceptually simultaneous actions are treated as if they happened together. Typically, the synchronization of simultaneous actions follows two major phases: the phase of *synchronization setup* and the *acting phase*. During synchronization setup, an agent synchronizes with all agents within his perceptual range. When no other agent is visible at that moment, the agent can act asynchronously with respect to all other agents. Otherwise the agent starts synchronizing with the visible agents. During synchronization setup, the agent blocks his activity until all agents in his region are synchronized. Agents synchronize by exchanging synchronization messages. When all agents within their perceptual range have concluded synchronization, they act together during the acting phase. Fig. 1 illustrates some synchronization situations. The left part of the figure depicts a snapshot of the

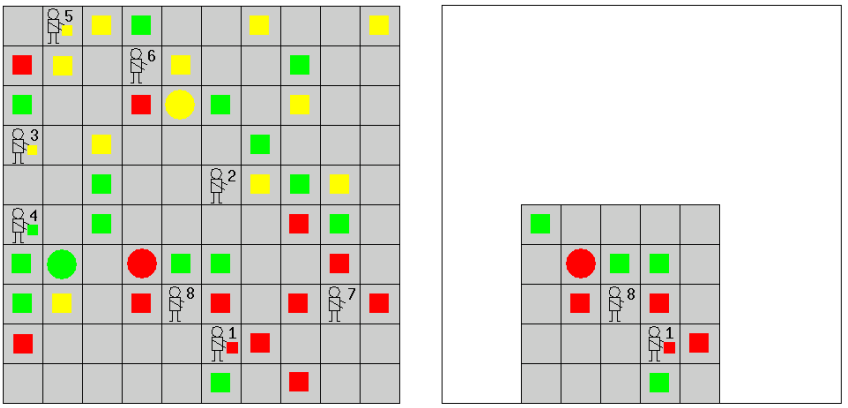

Fig. 1. Examples of synchronization situations in the Packet–World

Packet–World, i.e. a simple MAS we use as a case in our research[4]. In the Packet–World agents have to bring colored packets (rectangles) to their corresponding colored destination (circles). Agents are allowed to make one step at a time to a free neighboring field or pick up a packet from a neighboring field. An agent is able to carry one packet that he can deliver at any free field or at the destination of the packet. The job of the agents is to deliver all packets efficiently, i.e. with a minimum steps, packet manipulations and message exchanges. In the Packet–World each agent has only a limited view on the world. In the example, we suppose a view size of 2, illustrated for agent 8 in the right part of Fig. 1. To optimize the job execution agents can cooperate in different ways. They can for example, request each other for information or set up a chain to pass packets. For more information about the Packet–World we refer to [11].

Synchronization is necessary if we allow agents to interact by means of simultaneous actions, e.g. when we allow direct transfer of packets between agent 1 and agent 8 in Fig. 1. Synchronized agents are able to execute simultaneous actions since they act together, i.e. during the same acting phase. Applied to the example: the packet transfer only succeeds when agent 1 passes a packet and agent 8 accepts that packet during the same acting phase. With the synchronization algorithm we present in this paper, agents synchronize with colleagues within their perceptual range. With a view size of 2 we have, besides agents 1 and 8, two other groups of synchronized agents in Fig. 1: agents 5 and 6, and agents 3 and 4. Each group of synchronized agents act on its own pace. For the first group with agents 5 and 6, synchronization benefits as soon as agent 5 makes a step in the direction South–East (SE[5]), after which he can transfer

[4] The Packet–World is based on an exercise proposed by Huhns and Stephens in [7] as a research topic to investigate the principles of sociality in MASs.

[5] We denote a particular neighboring field of a field with the first capital letter(s) of the direction from the field to that neighboring field. E.g., agent 8 is positioned NW to agent 1.

a number of packets to agent 6 for direct delivering. For the third group with agents 3 and 4, the actual synchronization contributes little since both have no collaborating intentions (they carry a packet of different color). Anyway, as soon as agent 3 steps NE toward the delivering field of the packet he carries, he will no longer be synchronized with agent 4, but synchronizes with agent 5 and 6 enabling collaboration between these three agents. The other agents in Fig. 1, agents 2 and 7, are not synchronized with any agent (there is no other agent inside their perceptual range), so they act asynchronously.

Summarizing, the goal of the synchronization algorithm is to ensure that: (1) during the synchronization setup each agent synchronizes with the colleagues inside his perceptual range and hence also with all agents within the perceptual range of the latter, and so on; and (2) during the action phase, synchronized agents act together, allowing them to perform simultaneous actions.

2.2 Main Challenges of the Algorithm

The algorithm to set up regional synchronization is not trivial. To illustrate the complexity we briefly discuss the main challenges. Since agents have only a limited view on the environment it may be the case that two agents positioned inside each others perceptual range see different candidates to synchronize with. So at the end of synchronization setup an agent typically knows only a limited number of the agents of the synchronized group to which he belongs. This property hides a hard problem the algorithm has to tackle: avoiding deadlock when a sequence of overlapping perceptual regions of agents form a cycle. An example: in Fig. 1 agent 8 and agent 1 have overlapping perceptual regions and so they act synchronized. Suppose that in the given situation agent 2, (who act in the depicted situation asynchronously, remember the view size is 2) decides to make a step in the direction S and agent 7 (also acting asynchronously) makes a step in the direction NW. Now the perceptual regions of the agents 1, 8, 2 and 7 form a loop. The algorithm must ensure that, in whatever order the involved agents in such a loop enter synchronization setup, synchronization of the next action may not lead to deadlock. Note that since the synchronization between agents is reached by message exchange and the agents of a MAS run in separate processes there is no guarantee for the algorithm when these message are delivered.
Another problem the algorithm has to deal with, arises when an agent A requests two agents, X and Y, of a group of synchronizing agents of which one, e.g. X, has concluded synchronization, while the other, Y, is still busy with synchronization. If the conditions for synchronization between A and Y are satisfied, the two synchronizers establish synchronization. However, since synchronized agent X is unable to handle incoming messages, agent A will not receive an answer to his request from X and so he is unable to conclude synchronization. Since agents A, X and Y belong to the same group of synchronizing agents, this scenario leads to deadlock. Therefore the algorithm must offer requesting agents a mains to detect and handle such blockades.

2.3 Definitions for the Algorithm

In order to cope with the problems discussed in the previous section we developed an algorithm that combines a distributed 2-phase commit protocol with a logical clock. Before we discuss the different steps of the algorithm in detail, we first give an overview of the definitions we use in the remainder of the paper and then explain them in an intuitive description of the algorithm.

- S_i is the synchronizer of agent A_i of the MAS with view size VS
- $V_{S_i}^{t_i}$ is the view–set of S_i composed by the environment at logical clock time t_i: $V_{S_i}^{t_i} = \{S_j : dist(A_i, A_j) \leq VS \; at \; time \; t_i\}$; we call t_i the synchronization–time of S_i during synchronization setup with the synchronizers of $V_{S_i}^{t_i}$
- $mset_i^{t_i}$ is the member–set of S_i: $mset_i^{t_i} \subseteq \{M_{j \prec i} : M_{j \prec i}.S_j \in V_{S_i}^{t_i}\}$ where $M_{j \prec i} = (S, s, t)$ is a member j of S_i, i.e. the representation of S_j that is maintained by S_i with $S = S_j$, $s \in \{ini, req, ack, com, sync\}$ and $t \in \{t^0, t_j\}$ ($t^0 = 0$, and we denote S_j of $M_{j \prec i}$ as $M_{j \prec i}.S$ etc.)
- $sset_i^{t_i}$ is the synchronization–set of S_i: $sset_i^{t_i} = (S_i, mset_i^{t_i}, t_i)$
- $synchronizedWith$ is the equivalence relation over a set S of synchronizers: $\forall \, S_i, S_j \in S : ((\exists p \in N : S_0, \ldots S_p \in S) :$ $(\exists M_{i \prec 0} : M_{i \prec 0}.S = S_i \wedge M_{i \prec 0}.s = sync) \wedge (\exists M_{0 \prec 1} : M_{0 \prec 1}.S = S_0 \wedge M_{0 \prec 1}.s = sync) \wedge \ldots \wedge (\exists M_{p \prec j} : M_{p \prec j}.S = S_p \wedge M_{p \prec j}.s = sync))$
- $region_{S_i}^{t_i}$ is the region of S_i based on the view–set composed at time t_i, i.e. the equivalence class of synchronizers related with the equivalence relation $synchronizedWith$ whereof S_i is a representative.
- $msg_{from \rightarrow to} = (from, to, perform, time)$ is a synchronization message sent from S_{from} to S_{to} with $perform \in \{req, ack, nack, com, sync\}$ and $time \in \{t^0, t_{from}\}$

Each agent is equipped with a synchronizer who is responsible for handling synchronization. Synchronization setup starts when the synchronizer receives its view–set, together with the synchronization–time from the environment. The view–set is the initial set of candidates for synchronization, containing all synchronizers within the perceptual range of the associated agent. The synchronization–time is the value of the logical clock when the synchronizer's view-set was composed. This logical clock is a counter maintained by the environment[6]. Each time a group of synchronized agents has concluded the acting phase, the value of the logical clock is incremented and new view–sets for the agents are composed. With the information the synchronizer receives from the environment it composes a synchronization–set. Besides its own name and synchronization–time, such synchronization–set contains a member–set. In the member–set each synchronizer in the view–set of the synchronizer is represented by a member. A member is a triplet, containing the name of the candidate for synchronization, a state and a time stamp. Initially each member of the member–set is in the initial state denoted by *ini*, while the time stamps have the value t^0 that stands

[6] Notice that the value of the logical clock is not a global variable. In a distributed setting, the local environment of the MAS on each host maintains its own local clock.

for the initial value, zero. During the execution of the algorithm, synchronizers progressively try to synchronize with the members[7] of their synchronization–set by means of sending messages back and forth. During this interaction, negotiating synchronizers pass two phases. During the first phase they decide whether they agree about synchronization and subsequently during the second phase they mutually commit to the agreement. During this process, synchronizers exchange the value of there synchronization–time and mutually register the received values for the member of that particular synchronizer. Throughout the algorithm, the state of each member evolves from *ini* to *ack* (synchronization accepted), *com* (committed) and finally *sync* (mutually synchronized). The decision whether a synchronizer continues synchronization with a particular synchronizer depends on (1) the membership of a requesting synchronizer; and (2) the comparison between the value of the synchronization–time of the member and the value of the synchronizers own synchronization–time; and finally (3) the combination of states of all members of the member–set. In case synchronization can not be achieved, the rejecting synchronizer informs its colleague. As far as they belong to each others member–set, both synchronizers then remove the corresponding member from their member–set. As soon as all members of the member–set of a synchronizer have reached the state *sync*, the synchronizer concludes synchronization setup and activates its associated agent to enter the acting phase.

2.4 The Algorithm in Detail

In this section, first we describe the algorithm in Java–liked code and elaborate on each step in the algorithm. Then we discuss an example scenario in the Packet–World, and show how the algorithm deals with the challenges described in section 2.2. We conclude with a brief discussion how our algorithm integrates existing mechanisms from distributed algorithms.

The Algorithm. Fig. 2 depicts the algorithm for regional synchronization in Java–liked code. When a synchronizer enters synchronization setup, he first executes *makeSynchronizationSet()*, composing a new synchronization–set with the last received view–set and synchronization–time. Next, the synchronizer checks his mailbox, verifying whether there are pending requests, i.e. requests received by the synchronizer during the previous acting phase. In *handleMail()* the synchronizer handles requests according to the following rule:

> **R1.** A request is accepted (i.e. an *ack* message is sent and the state as well as the synchronization–time of the corresponding member is updated with the received information in the message) if the requesting synchronizer belongs to the synchronizer's member–set; otherwise the request is rejected (a *nack* message is sent).

[7] In the remainder of the paper we use the term member for the concept we formally have defined as well as for the synchronizer of a member that belongs to a particular member–set. The interpretation follows from the context where we use the term.

```
private void setupSynchronization() {

  makeSynchronizationSet();
  if(not mailbox.isEmty())
     handleMail();

  if(not toActAsynchronously()) {
     sendRequests();
     while(not synchronized()) {
       handleMail();
       if(blockedToCommit())
          unBlock();
       sendCommits();
       if(readyToSendSyncs())
          sendSyncs();
     }
  }
}

private void handleMail() {

  while(not mailbox.isEmty()) {

    Message msg = mailbox.pickMessage();
    Performative perform = msg.getPerformative();
    Synchronizer from = msg.getFrom();
    int time = msg.getTime();

    if(isRequest(perform)) {
       if(belongsToSynchronizationSet(from)) {
          sendAck(from,synchronizationTime);
          updateMember(from,"ack",time);
       } else
          sendNack(from);
    }
    else if(isAcknowledge(perform))
            updateMember(from,"ack",time);
    else if(isNack(perform))
            removeMember(from);
    else if(isCommit(perform))
            updateMember(from,"com");
    else if(isSync(perform))
            updateMember(from,"sync");
  }
}
```

Fig. 2. The algorithm for regional synchronization in Java–liked code

R1 ensures that a pending request is rejected if a synchronizer detects that, since the time of the request, he has left the requesting synchronizer's perceptual range. Furthermore, R1 ensures that a synchronizer only synchronizes with known colleagues, i.e. the synchronizers belonging to its member–set.
After the pending requests are handled, the synchronizer verifies *toActAsynchronously()*. This method returns true if the member–set of the synchronizer is empty. In that case the remainder of the algorithm is skipped and the agent immediately enters the acting phase to act asynchronously. Otherwise the synchronizer sends requests to the members of his member–set according to the second rule:

R2. To every member of the member–set in the state *ini*, the synchronizer sends a request to synchronize, i.e. a *req* message.

Subsequently, the synchronizer enters a while loop in which he stays until *synchronized()* returns true. This condition is determined by rule 3:

R3. As soon as all members of member–set of a synchronizer have reached the state *sync*, the synchronizer concludes synchronization setup and activates its associated agent to enter the acting phase.

Inside the loop, the synchronizer starts checking his mail. Besides requests (R1), the synchronizer handles the other messages according the following rules:

R4. For every received *ack* message the state as well as the synchronization–time of the corresponding member is updated with the received information in the message.

R5. Every member from which the synchronizer receives a *nack* message is removed from the member–set.

R6. For every received *com* or a *sync* message, the state of the member is updated according to the information of the received message.

After handling mail, the synchronizer verifies whether he is *blockedToCommit()*. This state is described by rule 7:

R7. A synchronizer is *blockedToCommit* if (1) there is at least one synchronizer in his member–set in the state *com*; and (2) there is at least one synchronizer in his member–set in the state *req*; and (3) all other members are in the state *sync*.

In this state the synchronizer is allowed to remove blocking members of his member–set, described in rule 8:

R8. If a synchronizer is *blockedToCommit*, he is allowed to eliminate the blocking synchronizers of his member–set; blocking synchronizers are in the state *req*; during unblocking a synchronizer removes subsequently these blocking members from his member–set and informs them with a *nack* message.

This rule is necessary to deal with the blocking situation we briefly discussed in section 2.2. We discuss a concrete example of R8 below.
An interesting side effect of R8 is that it gives the algorithm some degree of robustness. Since synchronizers, due to R8, reject colleagues that do not react to a request in time, they also reject synchronizers that have failed and no longer are able to react to requests.
Subsequently, the synchronizer sends commits, according to rule 9:

> **R9.** To every member in the state *ack* with a synchronization–time younger or equal to the synchronizer's own synchronization–time, the synchronizer is allowed to send a commit, i.e. a *com* message.

In the last step of the loop, the synchronizer verifies whether he is able to conclude synchronization with the members of his member–set. The conditions are described in rule 10:

> **R10.** A synchronizer is allowed to confirm a commitment with the members of his member–set (by means of sending them a *sync* message and updating their state) if all the members of his member–set are *synchronizable*, i.e. their state is (1) *com* or *sync*; or (2) the state is *ack* and the synchronizer's synchronization–time is younger or equal to the member's synchronization–time.

Subsequently, the synchronizer verifies whether he has concluded synchronization setup (R3). If this is the case, he activates its associated agent to enter the acting phase, otherwise he starts a new cycle in the loop.

Discussion. We now apply the algorithm to an example situation in the Packet–World that deals with the challenges described in section 2.2. Suppose in Fig. 1, agents 8, 1, 7 and 2 all are executing the acting phase. Now agent 7 makes a step NW, entering the perceptual range of agents 1 and 2. S_7 starts synchronization setup by requesting S_1 and S_2 for synchronization. Subsequently, agent 1 and 8 conclude their action (suppose they transferred a packet) and enter synchronization setup. Now things can evolve in different ways. We look to three scenarios:

- Agent 2 steps W and enters synchronization setup before S_7, S_1 and S_8 have concluded synchronization setup.
- Agent 2 steps S and enters synchronization setup when S_8 already has concluded synchronization setup, while S_7 and S_1 are still busy synchronizing.
- Agent 2 steps S and enters synchronization setup while S_7, S_1 and S_8 are still busy synchronizing.

First scenario. This scenario is rather simple. When S_2 enters synchronization setup, he detects that S_7 do not belongs to his member–set and according to R1, he rejects the pending request of S_7, sending him a *nack* message. Since S_2's member–set is empty, no further synchronization is required and A_2 immediately can enter the acting phase. When S_7 receives the rejection, he removes S_2 from his member–set (R5) and subsequently concludes synchronization setup with S_1.

Second scenario. In this scenario, S_2 confirms the pending request of S_7. Subsequently, S_2 synchronizes with S_8 and S_7 (R9, R10). In the end S_8, S_1 and S_7 have concluded synchronization setup, while S_2 still waits for a response of S_8. Fortunately, in this blocked situation (R7), S_2 can apply R8, liberating himself from the non-responding S_8 and subsequently conclude synchronization setup.
Third scenario. If in this scenario S_8 receives S_2's request too late (i.e. S_2's request message is scheduled after S_8 concludes synchronization setup) we have the previous case. Otherwise S_8 rejects, according to R1 (S_2 does not belongs to S_8's member–set), the request. S_2 then removes S_8 from his member–set, after which the four synchronizers normally can conclude synchronization setup. Note that in this case, S_2 and S_8 not have synchronized directly, although in the end they are indirectly synchronized via the chain of synchronizers between them.

Integration of Existing Mechanisms from Distributed Algorithms. To conclude this section, we briefly discuss how our algorithm for distributed synchronization integrates the two building blocks we have used to design it: two-phase commit (2PC) and a logical clock (LC). The goal of standard 2PC is to reach a full agreement between a set of processes (participants) whether or not to perform some action. The result is all–or–nothing, i.e. if a commitment is reached the action should be executed by all participants, otherwise the operation as a whole is aborted. The protocol is normally initiated by one process, i.e. the coordinator. The coordinator collects votes from the participants and decides about the outcome of the interaction. For a detailed description of 2PC, see [2]. On the other hand we use a logical clock. Lamport [9] invented logical clocks to capture numerically causal ordering of events within process groups.

In our algorithm, synchronizers are peers and can play both the role of participant as well as coordinator during one ongoing synchronization setup. Which role one synchronizer plays with respect to the other depends on the comparison of the values of both their synchronization–time, i.e. the value of the logical clock they received when they entered synchronization setup. As for 2PC, the result of our algorithm is a set of synchronizers that have reached an agreement, i.e. execute their next action phase synchronized. However, during synchronization setup, some of the candidates for synchronization might be shut out from the synchronizing group.

3 Evaluation

Evaluation compares the gain from regional synchronization with its costs. The major advantage of the algorithm is that agents, after regional synchronization, only need to wait for agents of the region to which they belong. The algorithm tunes the granularity of synchronized groups to the number of agents that are candidates for simultaneous interaction. For centralized synchronization, agents act on the pace of the slowest agent of the entire MAS. Thus the size of the region in comparison with the total number of agents in the MAS is a measure for the gain of the algorithm. The cost of using the algorithm is an overhead

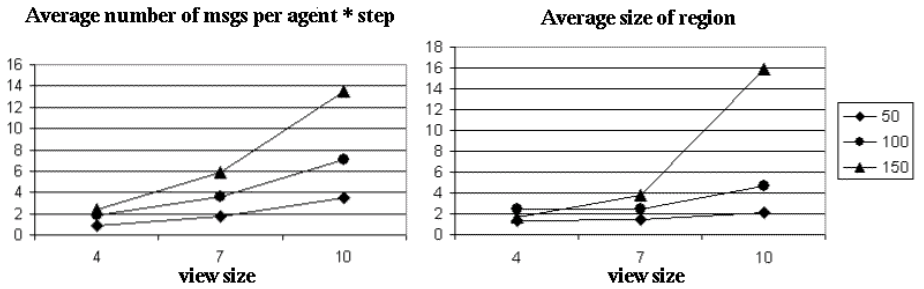

Fig. 3. Quantitative simulation results for populations of 50, 100 and 150 agents

to setup the groups of synchronized agents. This cost includes three parts: (1) the number of sent messages; (2) the cost for sending the messages; and (3) the computational overhead induced by handling the messages.

We did a great number of tests on a MAS with a 2D–grid environment, sized 100x100. First, we selected one number of agents of the set {50,100,150}, with one view size of the set {4,7,10}[8]. For each such combination, we did three separated tests. In the first test we used random moving agents, in the second test the agents attracted each other and in the third test we used agents that repulsed each other. Each test started with a random distribution of the agents, and subsequently the agents run for 1000 cycles through the algorithm. Each time a region was composed, the agents of that region acted simultaneously, i.e. each agent made a step according to its behavior. During the runs, agents are randomly scheduled. Fig. 3 depicts the results of our measurements. The left graph depicts the average number of sent messages per agent and per step, for different numbers of agents and different view sizes. The right graph shows the average size of regions for the same parameters. Both graphs combine the results for the three kinds of agents used in the tests. The results of the right graph confirm the intuitive expectation that regions grow with (1) increased density of agents in the MAS and (2) increased view size of the agents. The results of the left graph show that the number of exchanged synchronization messages for the algorithm is proportional to the size of the regions. Moreover, the results for our tests show that an agent sends as an average about one message for each agent of the region to which he belongs for each action he performs. Note that a higher average number of synchronization messages is not a disadvantage on itself, since more sent messages corresponds to higher sizes of regions and thus increased possibilities for simultaneous interaction.

Whether the gain of the algorithm, for a particular MAS, outweighs the costs, is application dependable. For example, for a MAS populated with 150 agents with a view size 7, the average number of sent messages is 6, while the average size of the region is about 4, see the upper points for view size 7 in Fig. 3.

[8] The view size is the number of squares and agent is able to perceive in each direction, similar to the view size in the Packet–World, see Fig. 1.

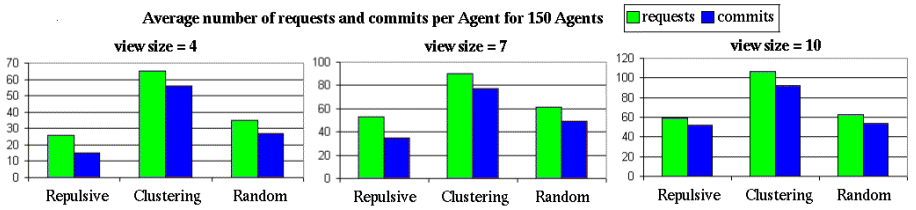

Fig. 4. Simulation results for success of synchronization

For this particular MAS, the pace on which agents are able to act depends on only 3 neighboring agents. In comparison with centralized synchronization, where the pace of each action for each agent depends on the entire population, in this case 150 agents, this gain appears to be very significantly. In practice however, the order of this gain will be influenced by the heterogeneity of agent activity in the MAS. If some of the agents act quickly while others are very slow, the first will be no longer concerned about the latter. Opposite to the gain, there is for each agent the cost associated with the handling of an average of 6 messages for every performed action. Since the communication between the synchronizers is defacto regional, the cost induced by synchronization will mainly be computational. Fig. 4 gives an idea about how successful the agents establish synchronization. The graphs depict for the three kinds of agents in the tests, in a MAS populated with 150 agents for view sizes 4, 7 and 10, how much requests lead to synchronization. The results show that most of the communication lead to synchronization. For the depicted results, an average of 80 % of the requests finally result in synchronization.

4 Conclusions and Future Work

In this paper we have presented an algorithm that allow agents to synchronize with the agents in their region, enable them to perform simultaneous actions. The algorithm combines a distributed two phase commit protocol with a logical clock. The gain of the algorithm is a much finer grained synchronization in comparison with centralized synchronization, increasing the efficiency of acting for the agents significantly. The cost for regional synchronization is an overhead of communication to setup regional groups of synchronized agents. Since synchronizing agents communicate regional, the overhead of sending messages are minor to the computational cost. Therefore, the gain appears to outweighs the costs, however in practice the balance must be made according to the characteristics of the application.

Future work includes formal verification of the algorithm and integration in a multi-agent application. Actually, we are finalizing a Colored Petri–net to prove formally our algorithm is free of deadlock. We also are integrating the algorithm in a full Java implementation of the Packet–World. This will allow us

to investigate the value of the algorithm in the context of collaborating agents. Another interesting issue we intend to investigate is how the algorithm can be applied for other kinds of dynamical group formations.

Acknowledgments

We would like to thank the members of the AgentWise working group at the K.U.Leuven for the many valuable discussions that have contribute to the work presented in this paper. Also a word of appreciation goes to Wouter Joosen for his useful comments to improve this paper.

References

[1] J. F. Allen and G. Ferguson, *Actions and Events in Interval Temporal Logic*, in Journal of Logic and Computation, Special Issue on Actions and Processes, 1994. 498

[2] K. Birman *Building Secure and Reliable Network Applications*, Cornell University, Ithaca NY, 14853, 1995. 507

[3] S. E. Bornscheuer and M. Thielscher, *Explicit and Implicit Determinism: Reasoning about Uncertain and Contradictory Specification of Dynamic Systems*, TR–96–009, ICSI Berkeley, CA, 1996. 498

[4] C. Boutilier and R. I. Brafman, *Partial–Order Planning with Concurrent Interacting Actions*, in Journal of Artificial Research 14 p.105–136, Access Foundation and Morgan Kaufmann Publishers, 4-2001. 498

[5] J. Ferber, *Multi-Agent Systems, An Introduction to Distributed Artificial Intelligence*, Addison-Wesley, ISBN 0-201-36048-9, Great Britain, 1999. 498

[6] N. Griffiths, M. Luck and M. d'Iverno *Cooperative Plan Annotation through Trust*, in Workshop Notes of UKMAS'02, Eds. P. McBurney, M. Wooldridge, UK Workshop on Multi-agent Systems, Liverpool, 2002. 498

[7] M. N. Huhns and L. M. Stephens, *Multi-Agent Systems and Societies of Agents*, in G. Weiss ed., Multi-agent Systems, ISBN 0-262-23203-0, MIT press, 1999. 500

[8] D. Kinny, M .Ljundberg, A. Rao et al. *Planning with Team activity*, 4th European Workshop on Modeling Autonomous Agents in a Multi-Agent World, LNCS 830, pp. 227–256, S. Martino al Cimino, Italy, 1992. 498

[9] L. Lamport *Time, clocks and the ordering of events in a distributed system*, ACM, vol. 21, no. 7, pp.558-65, 1978. 507

[10] P. R. Wavish and D. M. Connah, *Representing Multi–Agent Worlds in ABLE*, Technical Note, TN2964, Philips Research Laboratories, 1990. 498

[11] D. Weyns and T. Holvoet, *The Packet–World as a Case to Investigate Sociality in Multi-agent Systems*, Demo presented at the Conference of Autonomous Agents and Multi-Agent Systems, AAMAS 2002, Bologna, Italy, 2002. Demo available at: `www.cs.kuleuven.ac.be/~ danny/aamas02demo.html` 500

[12] D. Weyns and T. Holvoet, *Look, Talk and Do: A Synchronization Scheme for Situated Multi-agent Systems*, in Workshop Notes of UKMAS'02, Eds. P. McBurney, M. Wooldridge, UK Workshop on Multi-agent Systems, Liverpool, 2002. 498