Collective operations as building blocks for agent cooperation

Nguyen Tuan Duc and Ikuo Takeuchi
The University of Tokyo
Akihabara Daibiru 13F,
Sotokanda 1-18-13, Chiyoda, Tokyo, Japan, 101-0021
duc@nue.ci.i.u-tokyo.ac.jp, nue@nue.org

Abstract

Cooperation is the process of synchronization and exchanging useful knowledge between agents in multiagent systems. Since cooperation associates agents into a collaborative team to reach the overall goals of the systems, it is a crucial requirement for creating intelligent multiagent systems . This paper presents an approach for building complex cooperation protocols from simple primitives called collective operations. Collective operations are implemented in a new agent oriented programming language named Yaccai. The syntax and semantics of collective operations are designed to be effective for description of cooperation protocols while maintaining the autonomous computational model of agents. Our experiment shows that, collective operations are flexible building blocks for realization of cooperative multiagent systems.

1. Introduction

Cooperation is the process of synchronization and sharing knowledge to achieve a goal or to reach an agreement between agents. It is a crucial process in multiagent system because without cooperation the system is simply a set of separated agents and has no ability of collaborating to reach the goal. Many frameworks for developing cooperation and communication protocols of agents have been proposed such as FIPA or KQML. Furthermore, cooperation model has become an important topic in multiagent system research. There are many studies on cooperation model for agent such as the joint-intentions model [2], teamwork [4] or ECM [5]. Yet there is little attention on method for constructing cooperation, coordination protocols from simple communication primitives (such as sending, receiving, broadcasting message among the agents). In this paper, we propose an approach for implementing complex cooperation protocols from simple elements called collective operations. The concept of collective operation in the Message Passing Interface (MPI) [3], a famous parallel programming interface, is utilized to define building blocks for realization of cooperation schemes. Collective operations are implemented in a new agent oriented programming language named Yaccai (Yet another concurrent cooperating agent infrastructure). The syntax and semantics of collective operations are designed to be effective for description of cooperation protocols while maintaining the autonomous computational model of agents. Our experiment shows that using collective operation, programmers can easily create multiagent systems with intellectual cooperation algorithms.

The rest of this paper is organized as follows. In Section 2, we explain our collective operations' syntax and semantics. Section 3 describes the method for constructing cooperation schemes from collective operation primitives. Section 4 shows the model for implementing collective operations while maintaining autonomy of agents. We give experimental results to confirm the effectiveness of our model in Section 5. Section 6 compares our approach to other existing agent cooperation models. Finally, Section 7 discusses about future work and concludes.

2. Collective operations

2.1. Building blocks for cooperation

Cooperation in multiagent systems involves the sharing of mental state (belief, desire, intention (the BDI model)) of agent and exchanging solutions of sub-problems. The main issues one has to cope with to build an intelligent cooperation protocol include division of the goal into smaller tasks, synthesis of solution from sub-problem results, coordination of agents for avoiding unhelpful interactions and maximizing effectiveness [1]. To make the above issues easier and more intuitive, we propose collective operations as building blocks for realization of cooperation schemes. Collective operation is a concept in the Message Passing Interface (MPI) [3], a parallel distributed programming interface, in which many processes simultaneously participate into the execution of some procedure. For example, broadcasting message among agents in a group is a collective operation where the initiating agent may send message to its neighbors and then these neighbors may forward the message to other agents in the group until all agents received the message. MPI defines many collective operations such as *broadcast*, *gather*, *scatter*, *reduce* ... as tools for synchronization and coordination of processes/threads of execution. The processes that participate into the collective operations must be in a same group called a "communicator" in MPI (thus, communicator is an agent group in our model). Since collective operations are used to describe the synchronization and coordination of processes in distributed parallel programs, they are suitable for modeling cooperation of agents.

There are two specific cooperative problem-solving activities that are likely to be presented: task sharing and result sharing [1]. Task sharing takes place when a problem is decomposed into smaller sub-problems and allocated to different agents. Result sharing involves agents sharing information relevant to their sub-problems. Collective operations fit very well to the description of these activities. For example, gathering data from all agents (the *gather* operation) can be applied for collecting result of sub-problems to an agent. On the other hand, the *scatter* operation (scattering data to all processes in a group) is appropriate for task sharing because after decomposing the problem into smaller tasks, they need to be delivered to agents which can solve the tasks. We will discuss details of algorithm for constructing cooperation schemes using collective operations in Section 3. The following subsection presents the set of collective operations that are extremely useful for cooperation of agents.

2.2. Set of essential collective operations

Collective operations that are likely to be used for building cooperation protocols include *barrier*, *broadcast*, *reduce*, *gather*, *scatter* and are briefly described in Table 1.
We use the concept of "communicator" in MPI to model agent group. An agent can participate into a communicator by invoking the following method:

```
comm = Environment.GetCommunicator(
               comm_name, msg_listener );
```

where `comm_name` is a string which is used to identify the communicator. Other agents can join this communicator by invoking the method above with the same communicator name. The second parameter, `msg_listener` is the user defined plan (procedure) for processing messages when there are messages from other agents in the communicator needed to be delivered to the agent. After getting reference to a communicator (the variable `comm` in the code above), an agent can invoke collective operations on that communicator as shown in Table 1. Collective operations in normal parallel distributed programming library such as MPI [3] must be invoked in parallel by all processes that are participating in the operation. This requirement ensures the efficiency for the execution of collective operations but it causes difficulty in maintaining the autonomy of each process since it requires all processes to invoke the operation at the same time. In our system, we allow collective operations to be invoked by just one agent and other agents will automatically participate in the operations.

The message listener (`msg_listener`) does not need to do anything for "barrier", "reduce" or "gather" operations because the desired actions are automatically performed by the underlying execution environment (see Table 1 and Fig. 1(a)(c)(d)). On the other hand, in "broadcast" or "scatter" operations, the message will be passed to message listener and it is the responsibility of the programmer to describe what to do when an agent received a message (see Fig. 1(b) (e)). By this way, an agent with reactive capability can be easily created: the action that the agent takes when an event occurred can be specified in the message listener.
An agent can leave from a communicator by invoking the method "Leave" on that communicator:

```
    comm.Leave();
```

Messages in the communicator will not be passed to the message listener any more after the agent left the communicator (even though, the agent may participate in the message passing process of the execution environment and act as a router to forward messages to another agent in the system without awareness of the programmer).

3. Building cooperation protocol from collective operations

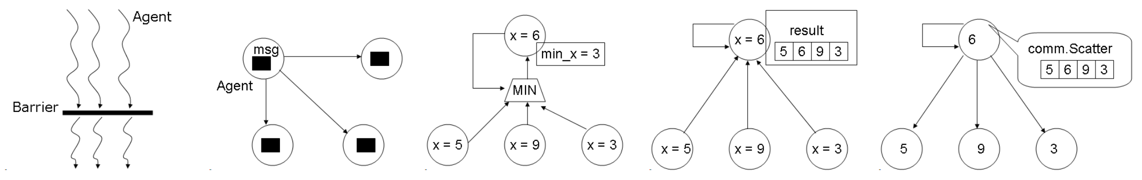
As stated in Section 2, cooperation of agents may be built from collective operations. Each collective operation is a building block for constructing cooperation schemes. In this section, we show how the process of reaching agreement and cooperation can be described by using collective operations as primitive actions.

3.1. Reaching agreement

Many-to-one negotiation is straightforwardly implemented by broadcast/scatter and reduce/gather operations. For example, in the auction scenario, the auctioneer (the agent who wants to sell goods) uses broadcast operation to broadcast the proposal to all bidders (the collection of agents that want to buy the goods), then it can use reduce or gather operation to collect the bidding result from the bidders. In English auction style, the auctioneer may want

Table 1. Collective operations

Operation	Meaning	Example
Barrier	Synchronizing all agents in the communicator	comm.Barrier(“barrier_name”);
Broadcast	Broadcasting message to all agents in `comm`.	comm.Bcast(msg);
Reduce	Evaluate `expression` at each agent and apply `operator` on the result set in round-robin manner	comm.Reduce(operator, expression);
Gather	Evaluate `expression` at each agent then gather the results to an agent	comm.Gather(expression);
Scatter	Scatter the array of messages to all agents in `comm`	comm.Scatter(array of msg);



(a) Barrier (b) Broadcast (c) Reduce (d) Gather (e) Scatter

Figure 1. Essential collective operations

```
procedure auction( ) {
   bidders = Environment.GetCommunicator( “bidders” );
   bidders.Broadcast( proposal );
   winner = bidders.Reduce( MAX_ID, @{price} );
   bidders.Send( winner, inform_win );
   bidders.Recv( winner, confirmation );
}
```

Figure 2. Implementation of an auction protocol by collective operations

to know the agent that bids the highest price to allocate goods to that bidder (first-price auction). In this case, it can directly use reduce operation on the agent group with the “MAX_ID” operator:

```
winner=bidders.Reduce(MAX_ID,@{price});
```

to identify the winner (the expression inside the “@{}” operator is evaluated at each agent in our language). Fig. 2 shows the pseudo-code for the auction scenario using collective operations.

Many-to-many negotiation can also be done with collective operations such as *AllGather*, *AllReduce* and *AllScatter* (these operations are defined in MPI), but we have not investigated them elaborately. We defer this to future work.

3.2. Cooperation using collective operations

The process of cooperative distributed problem solving contains of 3 stages: 1) Problem decomposition, 2) Sub-problem solution and 3) Solution synthesis [1]. Sub-problem solution is an issue related to the specific problem that can only be solved when we know about the domain of the problem. For problem decomposition and solution synthesis, we can use collective operation to achieve many complex strategies. For instance, a master agent can distribute tasks to slaves by putting tasks into an array and invoking “scatter” operation. Each agent has an integer identifier (called “rank”) so the master can use this identifier to arrange the task array such that the specified agent will receive the desired job (when scattering an array of data, we send each element to an agent in order of their ranks). This strategy is useful when the master knew the capability of each agent so it can assign appropriate task for each agent.

Even when the master does not know the capabilities of each agent, it may use broadcast and gather operations to determine which task should be assigned to which agent. It simply broadcasts each task to all agents and ask the agent for the ability of solving the task (by using gather operation). When it has response from all agents, it can assign task as desired.

Solution synthesis (result sharing) can be achieved by reduce/gather operation. The gather operation is used when the solution of the problem directly equals to the set of subproblem solutions (for example in QuickSort, the sorted list is the concatenation of sublists (in an appropriate order)).

The reduce operation provides a powerful function of deriving global solution from sub-problem solutions. For instance, in the auction problem above, the winner may be easily found by reduce operation with MAX_ID operator. Another example, by using SUM operator the master can directly find the distance of the route that is synthesized from many sub-routes that slave agents found.
The "barrier" operator is useful when the master needs to wait until all agents have completed solving the sub-problem. In this case, the master invokes barrier operation by providing a barrier name (identifier), other agents also invoke the operation with the same barrier name to inform the master about the completion of sub-problem solution.

```
procedure solve( ){
    for each task in TaskSet {
        workers.Broadcast( task );
        Solvable[task] = workers.Gather( @{is_solvable} );
    }
    task_array = create task assignment
                    array from Solvable;
    workers.Scatter( task_array );
    solution = workers.Reduce( OP, @{result} );
}
```

Figure 3. Cooperative problem solving using collective operations.

The skeleton of cooperative problem solving implementation by collective operations is provided in Fig. 3. In the figure, `Solvable` is a hash table that maps a task into an array of boolean values whose element i is `true` if the agent with rank i can solve the task.

4. Execution model

The execution of collective operation is not simple because it involves in all agents while the operation is invoked by just one agent. To accomplish this goal, we implemented an execution and communication model as shown in Fig. 4. Each agent is divided into two layers: the agent reasoning layer and the message passing layer. Reasoning layer contains user's code for the agent program while message passing layer contains code of the execution environment (in Fig.4, the circle "$\mathrm{a_i}$" denotes the reasoning layer of agent i while the circle "$\mathrm{m_i}$" is the message passing layer of the same agent). The reasoning layer contains 1 thread of execution; the message passing layer contains zero or more threads. When the reasoning layer's code invokes *send* or *broadcast* method of communicator, the message will be passed to the message passing layer of the same agent first, and it is actually sent to destination in this layer.

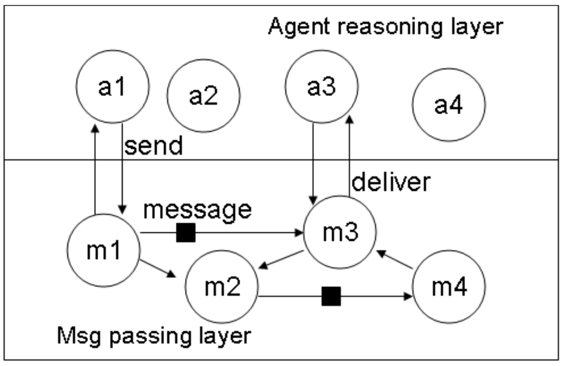


Figure 4. Execution and communication model.

The separation of agent reasoning and message passing layer supports the implementation of collective operations with different semantics from the semantics in normal parallel distributed programming: collective operations can be invoked by just one agent, not all agents in parallel because the message passing layer does the job of passing messages independently from agent's reasoning code.

Each agent is mapped to one process in the operating system which can be in a remote host. Agent can freely create new agent by calling the operator "create" and specify host name in which the new agent will be executed.

5. Evaluation

We use Vacuum Cleaner problem as a typical multiagent system benchmark to evaluate our language. The result shows that it is easy to realize a Vacuum Cleaner agents team with many complex cooperation schemes by using the framework. Moreover, the result also confirms the effectiveness of collective operation as a tool for cooperation.

Each Vacuum Cleaner is represented as a software agent whose capabilities are "move" and "clean". The simulation server constructs a virtual space which is a grid of 20 x 30 cells and contains many cells that have dirt. Agent can only move up, down, left or right (one step for each cycle) in the virtual space and it receives information about the current position (e.g., contains dirt or not) from the server. The agent can only clean a unit of dirt in each cycle by sending a "clean" command. A position (a cell) in the virtual space may contain many units of dirt. The agent is written in our language (complete source code for the agent can be viewed at the Yaccai's homepage[1]). At each cycle, the server waits until all agents have issued a command, then it advances the

[1]http://www.nue.ci.i.u-tokyo.ac.jp/%7Educ/yaccai

simulation one step (an agent can only issue one command for each cycle). A game lasts for 1000 cycles; at each cycle, the server reports the score of the game by the following formula:

$$Score = \frac{Total\ units\ of\ dirt\ cleaned}{Total\ units\ of\ dirt} \times 100 \quad (1)$$

In the first experiment, we created a multiagent system which contains many simple agents: the agents do not cooperate with each other, they only send/receive commands and information from the simulation server. The agents simply scan the virtual space by moving horizontally first, go up/down one row when they could not move in horizontal direction, then change the horizontal direction from left-right to right-left and vice versa. When an agent find dirt, it will immediately send "clean" commands until the dirt is cleaned.

In the second experiment, we created a multiagent system which contains agents that cooperate using broadcast operation. The agents use the same strategy in the first experiment to move around the virtual space. When an agent find a cell that has dirt, it will broadcast the position of the cell and the units of dirt contained there to all other agents. When received message from other agents, an agent will store the information into its belief-base and determine if it should go to the cell to clean or not. On the way to the dirt, if the agent found another place that contains dirt, it will clean the place immediately (and broadcast the position to other agents). When an agent successfully cleaned a position, it also broadcasts the information to other agents. If the agent is not in idle state, it simply stores the cell in its belief-base and when it becomes idle, it will query the belief-base for dirty cells and go to clean. An agent will remove the dirt's position from its belief-base when it received the clean message from other agents.

In the third experiment, we created a multiagent system which contains agents that use similar cooperation scheme to the agent in the second experiment except that when an agent found dirt, it uses reduce operation to know the idle agent nearest to it. Then it sends the information about the cell to that agent only (not broadcast to all agents).

We executed the simulation with 2 scenarios (maps): `map_dense` contains large amount of dirt that is broadly distributed across many cells in the virtual space (the map has many cells that contains dirt, so it is dense of dirt), `map_sparse` contains large amount of dirt that is comparatively concentrated on a region in the virtual space. The experiment is carried on a cluster of 20 machines to guarantee that each agent is executed on a machine. The score is the average score of 5 times of simulation reported at the end of each simulation (i.e., at the cycle 1000).

Fig. 5 and Fig. 6 show the score for the non-communication agents, broadcast agents and reduce agents with 2 maps: `map_dense` and `map_sparse` respectively. When the number of agents increased, the score also increased. Standard deviation is very small (as shown in the figures) because we do not use any random parameter.

In `map_dense`, the broadcast agents do not perform very well because they can not scan entire the space to find dirt and do too many useless moves. The broadcast agents seem to concentrate on a place at each time (because when received broadcast message they often go to the dirt place if they are in idle state or when they become idle, they will query the belief-base for the cell and go to clean). The reduce agents have the same performance with non-communication agents because when discovered a position has dirt, an agent does not broadcast message to all agents, only one agent is affected by the message. When there are many places with dirt, the agents are busy (not in idle state) and they will not react to messages from other agents immediately so the behavior of the team is similar to non-communication team. In the `map_sparse` map, the situation is different. The broadcast team has the best performance because they do not spend a lot of time in idle moves. They can concentrate on a dirt place immediately when an agent found the dirt place. The reduce team also has relatively good performance because when agents are in idle state, they will react to messages from other agents immediately so the behavior is similar to broadcast team. Agents team without cooperation has poor performance in this situation because agents have to find dirt independently and do many useless moves to scan the virtual space.

The result confirms that collective operations are good for cooperating agents: the agents that use collective operations to cooperate obtained higher score when cooperation is important to the problem. In situations where cooperation becomes very important (e.g., in the `map_sparse`), collective operation is very effective because it simplifies the description of cooperation of agents. Even in situations where cooperation is not important (e.g., in the `map_dense`), the agent team uses collective operation may also achieve good performance if it uses appropriate cooperation scheme to reduce the risk of "over-cooperating" (i.e., too concentrated on a region).

6. Related work

Cooperation models such as Joint-intentions [2] or Teamwork [4] allow the specification of global goal of the team, but it is difficult to use these models for multiagent systems where each agent is an autonomous entity. These models do not guarantee the autonomy of agents because all agents have the same mental state. Distributed cooperation model ECM [5] guarantees the independence of each agent and provides the ability to form agent groups. Still, ECM does not support the realization of cooperation from

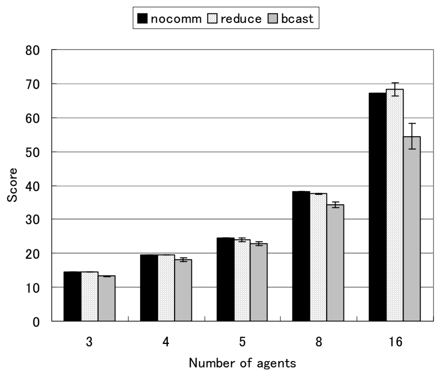


Figure 5. Average score of 5 times of simulation on `map_dense`

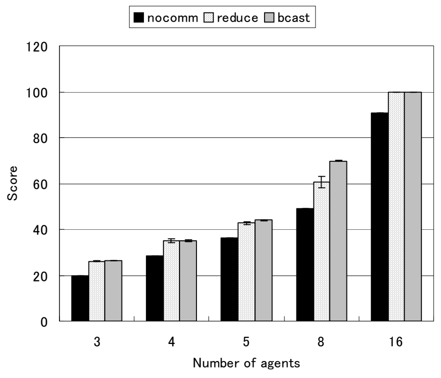


Figure 6. Average score of 5 times of simulation on `map_sparse`

powerful collective operations (ECM provides only one collective operations, i.e., the broadcast operation). There are cooperation models based on situation calculus [6] [7]. For instance, the requirement/service [6] model allows an agent to send request to another agent (service agent) and the service agent will serve the request. However, it is difficult to describe cooperation schemes with complex actions to synthesize the solution from sub-problem solutions since the cooperation here is restricted in the request and response actions. The Group situation model [7] defines cooperative group with situations (state of the group at a time) and allows specifying cooperation process for the group. Cooperation process describes the method to achieve the goal situation (i.e., what each agent needs to do). The model is intuitive to describe cooperative systems but it is difficult to generate real execution code from the model, especially code for distributed multiagent systems because the cooperation process is a centralized process which needs to be executed by a master agent (when the cooperation process failed, the system will not function). Our model guarantees that the cooperation process is decentralized, which promises the system's fault tolerance. Moreover, arbitrary complex cooperation protocols can be described in our language because wide range of cooperative actions can be specified by collective operation primitives.

7. Conclusion and future work

We have presented an approach for constructing agent cooperation schemes from simple primitives called collective operations. Collective operations can be considered as building blocks for creating complex cooperation protocols. We described our method for effective implementation of collective operations and provided experimental result to confirm that our model is suitable for modeling cooperative distributed multiagent systems. Future work is modeling many-to-many negotiation and other cooperative problem solving techniques by collective operations. Furthermore, implementation of real-world multiagent system problems such as Contract Net Protocol, RobocupSoccer, RobocupRescue team can help to evaluate the system.

References

[1] Michael Wooldridge, "An Introduction to Multiagent Systems.", John Wiley & Sons, 2002, pp 130 – 139, 190 – 197.

[2] Jennings, N. R, "Controlling cooperative problem solving in industrial multi-agent systems using joint intentions", Artificial Intelligence, 75(2): 195 – 240, June 1995.

[3] The Message Passing Interface, http://www-unix.mcs.anl.gov/mpi/

[4] Pynadath, D., et al., "Toward team-oriented programming", Proc. of the Agents, theories, architectures and languages workshop (ATAL99), pp 233 – 247, 1999.

[5] Michael Schumacher, "Objective coordination in multi-agent system engineering", LNAI 2039, 2001, pp 53 – 66.

[6] Y. Liu et al., "Cooperation Model of Multi-agent System Based on the Situation Calculus", Proceedings of the IEEE/WIC/ACM international conference on Intelligent Agent Technology (IAT), pp 424 – 427, 2006.

[7] Kim. M, Lee J., "Group Situation based Cooperation Model", International Conference on Convergence Information Technology (ICCIT07), pp 1372 – 1377, 2007.