Heterogeneous Distributed Cooperative Problem Solving System *HELIOS* and Its Cooperation Mechanisms (Draft)

Akira Aiba, Kazumasa Yokota, and Hiroshi Tsuda
Institute for New Generation Computer Technology
{aiba, kyokota, tsuda}@icot.or.jp

1 Introduction

HELIOS is a system for constructing heterogeneous distributed cooperative problem solving systems. When we consider knowledge information processing, it is very difficult to cope with real large-scale applications in a single programming language and a single paradigm. That is, constructing those applications requires combinations of multiple heterogeneous problem solvers that may spatially distributed. Besides heterogeneity, we have to consider that those applications also require resource bound computation.

Considering those issues, we propose a heterogeneous distributed cooperative problem solving system *HELIOS*. Basic concepts of *HELIOS* are an agent and an environment. An agent is an encapsulated problem solver by a module called capsule, and an environment is a common space in which agents are placed. In *HELIOS*, agents can communicate with each other even if problem solvers have different communication protocols, and agents can cooperate even if problem solvers have no functions for cooperation. That is, those functions are implemented by capsules and an environment.

In this paper, we describe functions in *HELIOS* mainly focusing on cooperation functions. In *HELIOS*, cooperation is defined by providing negotiation protocols and negotiation strategies. We consider negotiation in *HELIOS* as transaction since negotiation can be considered a logical unit of message sequences. We call it transaction-based negotiation. Furthermore, since negotiation can be nested, transaction-based negotiation corresponds to nested-transactions. This protocol can represent several negotiation protocols such as the contract net and multi-stage negotiation protocols.

2 *HELIOS*

2.1 Motivations

We can find many applications which require multiple *heterogeneous* problem solvers [1]:

- Modeling Heterogeneity
  The complexity of a given problem requires a combination of multiple heterogeneous problem solvers,

[1] Here, we use a *problem solver* as a general term for a database system, a knowledge-base system, a constraint solver, an expert system, an application program, and so on.

- Spatial Heterogeneity:
  Spatially distributed problem solvers are required to process a given problem.
- Temporal Heterogeneity:
  For a new problem, a new problem solver is not necessarily developed: that is, multiple existing problem solvers must be reused for a new problem.

There have been some approaches: an arithmetic calculator in Prolog and a constraint logic programming language with a single constraint solver. Such a restricted approach seems to be neither flexible nor promising for most applications.

Further, considering the spread of distributed environments, there might be similar resources, each of which does not have complete information. In such an environment, we can frequently get better results by accessing and merging multiple information resources or multiple problem solvers. In other words, cooperation among distributed resources are frequently required. Considering such applications and environments, heterogeneous, distributed, cooperative problem solvers will become more important and can play a role of a large-scale applications in knowledge information processing.

2.2 *HELIOS* Model

A basic concept in *HELIOS* is an *agent*, defined as follows:

agent := (capsule, problem-solver) | (capsule, environment, {agent_1, ..., agent_n})

A *simple* agent is defined as a pair of a *capsule* and a problem solver: intuitively, a problem solver is wrapped with a capsule as in Figure 1.

A *complex* agent is defined as a triple of a capsule, an *environment*, and a set of agents (agent_1,...,agent_n), where an environment is a field where agent_1,...,agent_n can exist and communicate with each other. Intuitively, as a pair of an environment and a set of agents can be considered also as a problem solver, a new agent can be defined by wrapping them by a capsule. That is, an agent can be also hierarchically organized. Figure 1 shows such structures.

A capsule and an environment is defined as follows:

capsule := (agent-name,methods,self-model, negotiation-strategy)
environment := (agent-names,common-type-system, negotiation-protocol, ontology)

An agent name in a capsule is an identifier of the corresponding agent and *agent names* in an environment specifies what agents exist in the environment. *Methods* in a capsule define *import* and *export* method protocols of the corresponding agent. An agent with only import methods is called *passive* and an agent with both methods is called *active*: that is, only an agent which send new messages through export methods can negotiate with other agents. A *common type system* in an environment enforces all agents under the environment to type all messages strongly. A *self model* in a capsule defines what the agent can do. An environment extracts necessary information from self models in agents to dispatch messages among agents. Under a *negotiation protocol* in an environment, each agent defines a *negotiation strategy* to communicate with other agents. An *ontology* defines the transformation of the contents of messages among agents, while a capsule convert the syntax and type of messages between the common type system and the intrinsic type system of the corresponding problem solver. These information is defined in *CAPL* (CAPsule Language) and *ENVL* (ENVironment Language).

Although various information is defined in each environment and each agent, a *message* among agents is in the form of a global *communication protocol* consisting of the message identifier, the identifier of

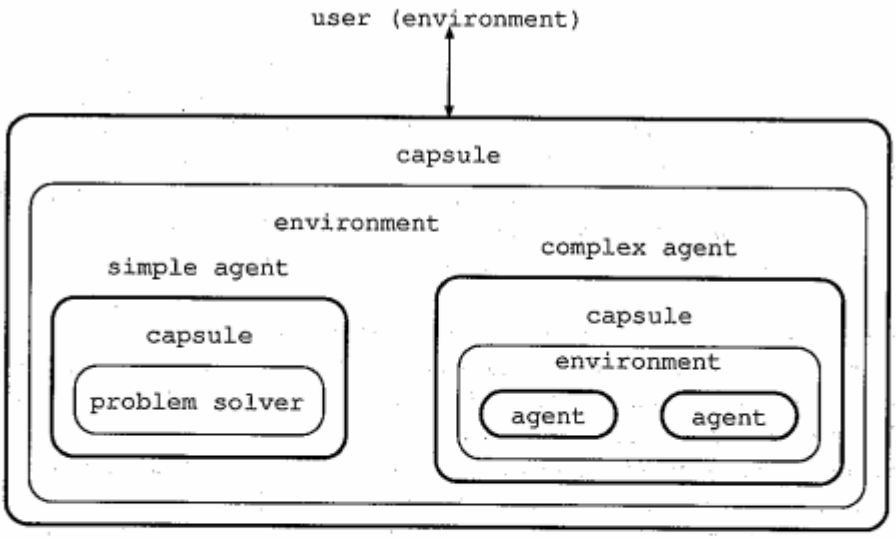

Figure 1: Configuration of an Agent

a sender agent, the identifier of a receiver agent, a transaction identifier, and a message. A message identifier is common in a query message and answer messages. A transaction identifier is used to identify a negotiation process as a transaction, which can be nested.

2.3 Message Dispatching in *HELIOS*

Any active agent can send a message to its environment. How is the message dispatched by the environment?

First, during the initialization of agent processes in the environment, the environment constructs a map of a logical agent name and a physical process address (or IP address). Secondly, the environment gathers method information and function information in self models from each agent and constructs two kinds of maps: a method and an agent name; a function name and an agent name. Such maps work for dispatching messages among agents.

As a method or a function does not necessarily corresponds to an agent uniquely, a message is possibly sent to multiple agents. This mechanism is useful for the followings:

- It is unnecessary to specify an agent name in problem solvers explicitly.
- It is possible to send simultaneously a message to possible agents.

An environment decides to send a message sequentially or in parallel to candidates listed by the maps, and processes answers sequentially or by grouping as a set. In the case of set grouping, aggregation functions can be specified in an environment. Such a mode can be selected in a query message.

How can agents communicate with each other? We consider three kinds of modes:

- simple communication,
- negotiation-based communication, and

- schedulable communication.

When communication among agents requires neither negotiation nor query plans, it is called simple.

For negotiation among agents, negotiation protocol and negotiation strategy in *HELIOS* can be defined in ENVL and CAPL, respectively, differently from conventional systems. The protocol is based on the *transaction-based protocol*, which comes from the similarity between transaction and negotiation processes: negotiations can be nested and controlled by begin-, end-, and abort-transactions. Various negotiation protocols such as contract net and multi-stage negotiation can be written by the transaction-based protocol. Negotiation strategies can be written in a logic programming language with the above protocol.

For efficient negotiation, an agent can use its proxy. In *HELIOS*, we introduce a proxy agent. An agent can send its proxy to other agent, and an agent can send it to an environment. The former decreases communication between agents, and the latter decreases communication between an environment and agents, and decreases functions of an environment.

Another kind of communication is applied when a message can be partitioned into sub-messages and their execution plan can be generated. A messages is analyzed and its corresponding processing plan is constructed as a dependency graph. A query in conventional distributed databases is such an example. Synchronization information between sub-messages is attached to each sub-message and controlled by the capsule of each agent.

An answer message can be processed by *global constraints*, a *constraint solver*, or aggregation functions. Global constraints pare used for restricting special values embedded in messages. For example, you can see the number of columns and rows in n-queen as unchanged one, and a blackboard as changeable one. A constraint solver is used for evaluation of results. An environment sends them to the related agent if necessary. Depending on their evaluation, the environment decides whether alternative message processing is necessary or not.

When all agent cannot solve a query, the environment sends the query to the outer environment, which may be a user, through its capsule.

3 Negotiation in *HELIOS*

In *HELIOS*, negotiation is realized by describing *negotiation protocol* in each environment, and *negotiation strategy* in each capsule. Based on those functions, we propose a transaction-based negotiation by considering a negotiation as a transaction. In the following subsections, we describe transaction-based negotiation protocol that can be defined as a protocol in an environment. Then, we show how this protocol can be used to represent various negotiation protocols such as contract net protocol, multi-stage negotiation, etc. Then, we show how other negotiation such as bargaining over the price and seat reservation can also be represented by the protocol.

3.1 Messages Protocols in *HELIOS*

A message between agents contains the followings:

- **Message identifier**
  An identifier used for identifying a message.
- **Message type**
  A message is either a method invocation or an answer. The former message is called a *query*

message, and the latter message is called a *reply message*. This field is used to distinguish a query message from a reply message.

- **Sender agent identifier**
  This field contains a logical name of the agent that sends this message.
- **Destination agents**
  As described in section 2.3, an environment can dispatch query messages by logical names, method names, and functions. This field contains such information to dispatch this message.
- **Transaction identifier**
  If update of a content of a destination problem solver is attendant on invocation of a method, then a transaction identifier is required. This field contains a transaction identifier. For nested-transactions, a transaction identifier with a nested structure is used.
- **Status**
  This field contains information on the status of invoked methods for error handling.
- **Message content**
  In a query message, this field contains a method invocation, and, in a reply message, this field contains the answer to the invocation.

3.2 Basic Policy: transaction-based protocol

A negotiation can be considered as a transaction since it is a logical unit of a message sequence, and it is required to ensure intermediate internal states of agents participating in the negotiation.

In the transaction-based negotiation, each unit of negotiation is described as a transaction. Since negotiations may be nested, transactions are also nested. Thus, we have to use nested-transactions in negotiation in *HELIOS*. That is, when a proposed plan is partially accepted then the negotiation becomes deeper, and transaction is nested to achieve agreement on issues that are not yet agreed.

When negotiation succeeds, then it corresponds to *commit*, while a breaking down on a negotiation corresponds to *rollback*. However, if there might be a possibility of re-negotiation, then a two-phase commit should be used for a breaking down of a negotiation. Note that both sides of the negotiation can send a commit message or a rollback message.

Transaction-based negotiation protocol can be defined in Figure 2.

As in Figure 2, presenting a plan is distinguished from asking a query. Presenting a plan is followed by its evaluation value, or rejection. On the other hand, asking a query is followed by its answer, acceptance, rejection, or conflict.

3.3 Various Negotiation Protocol in Transaction-based Protocol

As we described in section 2.3, an agent can send its proxy to the other agent or to the environment. In the following, we present several negotiation protocols as examples of transaction-based negotiation protocol. In all protocols, an environment plays an important role of a manager. An agent that asks to execute a job can send a proxy to the environment for decreasing communication, and the proxy plays a role of a manager.

Contract Net Protocol

The contract net protocol was proposed as one of protocols for dispatching a job by a manager to one of contractors [Smith 1980]. This protocol consists of *Task announcement*, *Bidding*, *Announcement of award*, etc. In the following case, an agent that has a job sends its proxy to the environment, and the

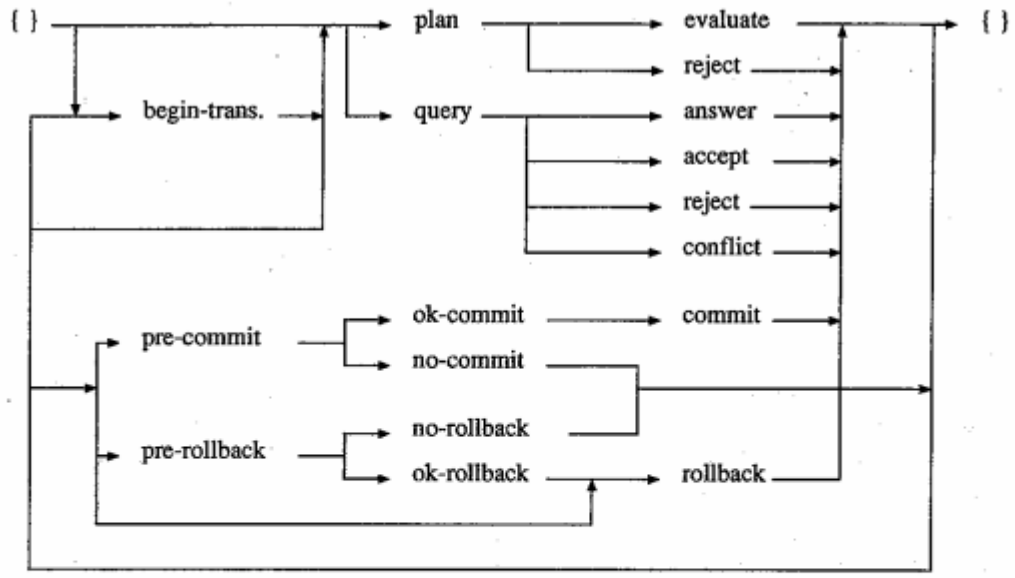

Figure 2: Transaction-based protocol

proxy plays a role of a manager. The contract net protocol is implemented using the transaction-based protocol as follows:

Message flow	Content	Messages	Corresponding message in [Smith 1980]
$a \rightarrow m$	Asking a job	*query*	
$m \rightarrow c \times n$	Presenting the job	*begin-trans* + *plan*	task-announcement
$c \times k \rightarrow m$	Bidding	*evaluate*	bid
$m \rightarrow c$	Asking the job	*query*	announced-award
$m \rightarrow c \times (k-1)$	Announcement of rejection	*rollback*	
$c \rightarrow m$	Receiving the answer	*answer*	final-report
$m \rightarrow c$	Termination	*commit*	
$m \rightarrow a$	Sending the answer	*answer*	

where a is an agent, m is a manager, c is a contractor, $c \times n$ and $c \times k$ represent n contractors and k contractors, respectively.

Before negotiation, an agent that has a job sends its proxy to the environment. Then the agent sends a job using a *query* message to the proxy in the environment. The proxy multicasts the job to each of candidates for contractors by functions of the environment using a *begin-transaction* message followed by a *plan* message. This corresponds to "task-announcement".

Some of contractors bid for the job. Evaluation of the job may be carried out by asking other agents that have functions for evaluation. Bids are send to the proxy using *evaluate* messages. This corresponds to "bidding".

Then, the proxy selects one of the contractors. Selection of a contractor may be carried out by asking other agents that have functions for comparing bids. Then the proxy sends an award to the selected

agent using a *query* message. This corresponds to "announced-award". The proxy sends messages to agents other than the selected one for giving up taking the job. These are realized by *rollback* messages.

The answer from the agent is sent to the proxy using an *answer* message, and the proxy sends a *commit* message to the agent to terminate the transaction. Then, the proxy sends the answer to the agent that originally asks the job to the environment.

On the other hand, the direct award can be implemented using a *begin-transaction* message followed by a *plan* message. In this case, there is only one contractor that is designated by the manager. Thus, the agent can accept or refuse the job. This can be implemented using an *evaluate* message or using a *reject* message. When the agent accepts the job, then the answer is sent to the proxy using an *answer* message.

Selection among Answers

Besides the contract net protocol, there are several other protocols that are exchanged between a manager and contractors. In the contract net protocol, one contractor is selected by estimated costs reported from contractors. However, there is a situation that the answer produced by each contractor is a criterion of selection. We call this *selection among answers*. In this case, an agent can also send its proxy to the environment, and the proxy plays a role of the manager. This protocol can be represented using the transaction-based protocol as follows:

Message flow	Content	Messages
$a \rightarrow m$	Presenting a job	*query*
$m \rightarrow a \times n$	Presenting the job	*query*
$a \times k \rightarrow m$	Collecting answers	*answer/reject*
m	evaluation	
$m \rightarrow a$	Answer	*answer*

In this protocol, an agent sends a job using *query* message to the manager. Then the manager multicasts the job to candidates for contractors using *query* messages. An agent that can accept the job sends the answer to the manager using an *answer* message, while an agent cannot accept the job sends a *reject* message. Time-out is checked for collecting answers from contractors. The manager selects one of the answers using functions of other agents, then the selected answer is sent to the agent that originally asks the job.

Aggregate Answers

Selection among answers can be generalized to aggregation.

Message flow	Content	Messages
$a \rightarrow m$	Presenting a job	*query*
$m \rightarrow a \times n$	Presenting the job	*query*
$a \times k \rightarrow m$	Collecting answers	*answer/reject*
m	aggregation	
$m \rightarrow a$	Answer	*answer*

In this protocol, the answer is produced using collected answers by certain aggregate functions. Just same as in "selection among answers", an agent sends its proxy to the environment, and the proxy plays a role of the manager. Almost all of this protocol are same as those in the "selection among answers" except after collecting answers from agents by the manager. After gathering answers, the manager

asks certain agent with aggregate functions to aggregate answers. Then the agent sends its aggregated answer to the manager. The manager sends the answer to the agent that originally asks the job.

4 Concluding Remarks

As described in the above, the major feature of negotiation in *HELIOS* is that problem solvers without an ability of negotiation can participate in negotiation by defining negotiation strategies in its capsule, and negotiation protocols in its environment.

As for the transaction-based protocol, there was a research on a transaction model [Ishida 1992]. Our transaction-based protocol for negotiation has different viewpoint from Ishida's transaction model. Our intention is to describe negotiation using a formal language by making element technologies of negotiation clear. For instance, from logic programming and deductive database points of view, the contract net protocol requires *set grouping*, *aggregation function*, *constraint solving*, *nested-transactions*, and *long-term transaction* as its element technologies. By making those element technologies clear, more flexible negotiation/cooperation strategies can be realized. Using the transaction-based negotiation protocol, some variations of the contract net protocol can also be easily represented.

Using the transaction-based negotiation protocol, we describe several problems requiring negotiation as examples. Those examples include bargaining over the price, seats reservation, and meeting scheduling. In the future, we should investigate the usefulness of the transaction-based negotiation protocol to other negotiation protocols. We also require *real* large-scale applications for verifying the effectiveness of *HELIOS* and the transaction-based protocol.

References

[Aiba *et al.* 1994] A. Aiba, K. Yokota, and H. Tsuda, "Heterogeneous Distributed Cooperative Problem Solving System HELIOS," In *Proceedings for International Symposium on Fifth Generation Computer Systems 1994*, Tokyo, December 1994.

[Ishida 1992] T. Ishida, "A Transaction Model for Multiagent Production Systems," In *Proc. Eighth Conf. on Artificial Intelligence for Applications*, Monterey, California, March 2–6, 1992.

[Smith 1980] R. G. Smith, "The contract net protocol: High-level communication and control in a distributed problem solver," *IEEE Transaction on Computers*, Vol. C-29, No.12, pp.1104 - 1113, December 1980.