Implementing a Multiagent Negotiation Protocol in Erlang

Álvaro Fernández Díaz Clara Benac Earle Lars-Åke Fredlund

Grupo Babel, Facultad de Informática, Universidad Politécnica de Madrid
{afernandez,cbenac,lfredlund}@fi.upm.es

Abstract

In this paper we present an implementation in Erlang of a multiagent negotiation protocol. The protocol is an extension of the well-known Contract Net Protocol, where concurrency and fault-tolerance have been addressed. We present some evidence that show Erlang is a very good choice for implementing this kind of protocol by identifying a quite high mapping between protocol specification and Erlang constructs. Moreover, we also elaborate on the added advantage that it can handle a larger number of agents than other implementations, with substantially better performance.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features

General Terms Languages, Performance

1. Introduction

Multiagent negotiation is a very active area of research in the Autonomous Agents and Multiagent Systems communities. In a negotiation, each agent modifies its local plans in order to achieve an agreement with other agents in the system. Several negotiation protocols have been proposed, among them, the Contract Net Protocol (CNP) [3] is probably the most popular. CNP is based on the bidding mechanism of a human market. However, as a task assignment or resource allocation mechanism, CNP cannot optimise system performance, especially when scheduling several orders. Also, CNP cannot detect failures in the agents participating in a negotiation process. An extension of the CNP to address these two issues has been proposed in [1].

The concurrent and fault-tolerant aspects of that extension to CNP, make Erlang an ideal implementation candidate. Indeed we have implemented the protocol in Erlang, with very encouraging results so far. In particular, we have been able to work with much larger systems than those reported in [1], i.e., using 3000 agents compared to 40, and with much better performance.

The rest of the paper is organised as follows. In the following section we describe the extended negotiation protocol. The implementation of the protocol in Erlang is discussed in Sect. 3, and the results of some experiments are shown in Sect 4. Finally, the conclusions and future work are summarised in Sect. 5.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Erlang'10, September 30, 2010, Baltimore, Maryland, USA.
Copyright © 2010 ACM 978-1-4503-0253-1/10/09. . . $10.00

2. The Contract Net Protocol extension

As mentioned in the previous section, the multiagent negotiation protocol presented in [1] is an extension of the Contract Net Protocol (CNP). The goal of the CNP protocol is to enable agents to negotiate the allocation of tasks among them in a fair way, avoiding the possibility of reaching a deadlock state. Such a deadlock state would prevent agents from performing other tasks they are committed to accomplish. The agents involved in the negotiation can be considered as self interested, as they are supposed to provide the best possible bids in order to get the task assigned to them. However, it is not important as the relevant outcome of the process depends only on the overall task allocation and on how efficiently it was performed. The CNP extension includes the addition of new negotiation phases, which: increase the probability for obtaining a more efficient allocation of tasks, permit an agent to handle negotiation of several tasks at the same time thus shortening the overall time required to finish the whole process, and support the detection of failures in agents and a mechanism for negotiation blockage avoidance.

The CNP extension protocol defines two roles for the agents involved in a negotiation process: the *Manager*, which announces the task to be accomplished, and the *Contractors*, which send bids to the manager.

The negotiation process is split into four phases:

1. **PreBidding:** first of all, the manager agent announces the task to all potential contractors. Then, the manager waits for bids from contractors. The contractor whose bid is the highest is promoted and considered as the *potential contractor*. A contractor whose bid is inferior to the highest bid is sent a *preReject* message from the manager. However, a contractor whose first bid failed can make new bids. The **PreBidding** phase ends once a time limit has expired, or all the informed contractors have bid at least once.
2. **PreAssignment:** in this phase, the manager informs the potential contractor that its bid was the highest one by sending a *preAccept* message.
3. **DefinitiveBidding:** once the potential contractor receives a *preAccept* message, it computes and sends a definitive bid, which can be different to its previous bid. If this definitive bid is lower than the bid of any other contractor, the current potential contractor is demoted and receives a *preReject* message. Then the contractor whose bid is now the highest is promoted to *potential contractor* and the negotiation protocol returns to *PreAssignment* phase. On the other hand, if the definitive bid is higher than the bids from all the other contractors, the *potential contractor* is assigned the task and, therefore, a contract is established.

4. **DefinitiveAssignment:** in this final phase, the manager sends a *definitiveAccept* message to the potential contractor and a *definitiveReject* message to all other contractors. After that, the negotiation for the task allocation has finished.

Despite the fact that there are four different phases, managers are the only agents that know the current phase for a negotiation. Hence, it is the only agent role that adapts its behaviour to the one associated to the negotiation phase. The behavior of the different contractors does not change with respect to the negotiation phase, as they just keep trying to get tasks assigned by improving their bids. Note that an agent can be simultaneously involved in several negotiations. This means that every contractor must define an ordering of the tasks it is negotiating.

Finally, in order to provide fault-tolerance to the negotiation process, the protocol defines an algorithm to end a multi-agent negotiation in case of a manager failure. In the first phase of this algorithm, all the contractors involved in the negotiation whose manager has crashed, namely fellow contractors, send each other a *manager_decision* message in which they specify the last answer received from the manager. If none was received, the agent does not send any message, which results in an inference of *definitiveReject*. Once a contractor has received a *manager_decision* from every fellow contractor, it infers its own definitive state for the task execution and forwards it again to all other fellow contractors. The decisions inferred by contractor agents are:

1. If any other contractor sends a *preAccept*, the agent infers a final decision of *preReject*.
2. If any other contractor sends a *definitiveAccept*, the agent infers a final decision of *definitiveReject*.
3. If all the other contractors send a *preReject*, the decision inferred for the task execution is *preAccept*.
4. If all fellow contractors send a *definitiveReject*, the agent infers a *definitiveAccept* final decision.
5. If any other fellow contractor is suspected of failure, and the contractor has received at least one *definitiveReject* message, the decision inferred is *definitiveReject*.
6. If any other fellow contractor is suspected of failure, and no other contractor has sent a *definitiveReject*, the decision inferred is *preReject*.

Note the contractors can only infer decisions by themselves when the manager has crashed. It occurs when this manager agent has not sent a message it is supposed to for a time that exceedes certain threshold, implementation dependent. The specification of this algorithm assumes that all the contractors know about a manager failure and that this failure really happens. Therefore, there is no need of an external agent deciding whether the manager failed or not, in order to start the negotiation termination algorithm.

Now that the main features of the protocol have been described, we will proceed to the details of its implementation in Erlang.

3. Implementing the Contract Net Protocol extension in Erlang

The implementation of a multi-agent environment performing the negotiation mechanism previously described seemed to be conceptually straight-forward. Nevertheless, some decisions had to be taken in order to obtain a fully working implementation of the CNP extension.

3.1 Process Orchestration

The first step was to define the process structure of the system. As the main goal of the implementation is to test the efficiency and performance of the negotiation protocol under Erlang, task execution is abstracted. We assume that each task completes successfully, and within any time limits specified in the contract. Consequently, the behaviour of each agent is relatively simple. Thus, the process structure of the system, depicted in Figure 1, has one Erlang process for each agent, manager or contractor, and a supervisor process which monitors the agents. The supervisor process receives a list containing several sets of tasks to be negotiated, and a set of contractor agents. The supervisor spawns a different manager agent for each set of tasks, and informs it of the contractors that should be involved in the negotiation. Finally, the supervisor is linked to all the managers, and the managers are linked to their associated contractors, to enable the system to be shut down cleanly and efficiently after a finished negotiation.

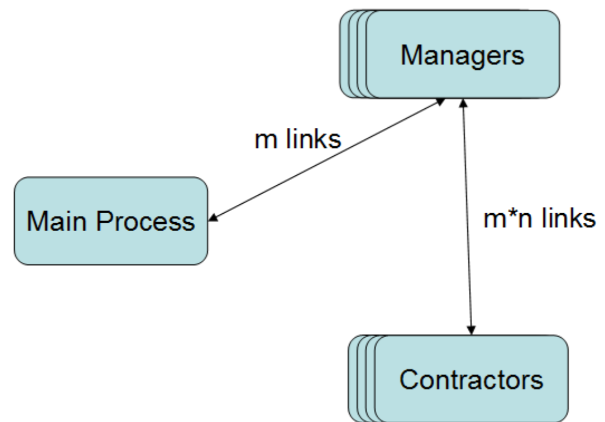

Figure 1. Process links orchestration for a system of *m* managers and *n* contractors

3.2 Information Representation

In order to represent a task, we decided to implement an Erlang record, named *task*, described in Figure 2. An instance of this record is generated by every manager for each task it handles and sent along to all contractors enclosed in an *announce* message during the start of each *preBidding* phase. The information required by a manager includes the name of the task being currently negotiated, the process identifier of the potential contractor, the bid of this potential contractor and the list of bids received from each contractor agent. This is represented by the *manager* record also shown in Figure 2. Analogously, every contractor maintains a record named *contractor* that contains information about the identifier of that agent, the next time it will be ready to execute new tasks according to its own schedule, a list of tasks it has committed to perform, a list of tasks it is currently negotiating and a list of tasks whose manager has crashed, for which the negotiation termination algorithm must be executed.

3.3 Simultaneous Negotiation

To complete the implementation of the protocol, that allows a contractor to be involved in several task negotiations, some decisions had to be taken. One such example was the “currency” used for computing task bids. As the negotiated tasks were not to be executed, but rather abstracted, we decided that a good currency candidate was the time an agent estimates that completing a task requires.

```
-record(task,{
  id=0,           % task name
  startTime = 0, % start time for task
  duration = 0,  % execution time for task
  endTime = 0,   % finish time for task
  manager=void,  % manager pid
  maxBid = 0,    % max bid received for task
  lastBid = 0,   % last bid received
  contractors=[],% list of contractors
  status,        % protocol status
                 % {announced,preAccepted,
                 % preRejected,Accepted}
  info = []}).

-record(contractor,{
  id=0,             % agent identifier
  nextFreeTime=0,   % time last task would finish
                    % (if assigned to this agent)
  tasksExecuted= [],% tasks already committed
  nextExecTime=0,    % first available time slot
                     % after task execution
  offeredTasks=[],   % tasks being negotiated
  blockedTasks=[]    % tasks with failed manager
}).

-record(manager,{
  task=void,   % task being negotiated
  winner=void,% pid of potential contractor
  maxbid=void,% bid of the potential contractor
  bids=[]      % all bids received
}).
```

Figure 2. Erlang records used in the protocol implementation

Thus a bid b_1 is considered to be higher (or better) than another bid b_2, if $time(b_1) < time(b_2)$, meaning that bid b_1 promises to complete a task before the bid b_2 promises to complete the task.

Next, the behaviour of each agent (manager, contractor) was defined. A manager agent first receives a sequence of tasks to negotiate and a list of process identifiers that correspond to contractor agents. Then, it starts negotiating the first task in the sequence, beginning by sending an *announce* message to all the contractor processes. After that it waits for bids from the contractor agents (in the *preBidding* phase), until either all of them have answered or a timeout expires. At that moment, a *potential contractor* has been identified and the manager process sends this process a *preAccept* message and enters the *definitiveBidding* phase.

The following messages can be received in that state:

- **preBid:** if this bid, from an agent *a*, is higher than the bids from all the other agents, including the one from the potential contractor, the current potential contractor is demoted, by sending it a *preReject* message, and *a* is promoted to potential contractor, consequently receiving a *preAccept* message.
- **definitiveBid:** There are two cases:
  - *This bid comes from current potential contractor: a* if it is better than the bids from all other contractors, the task being negotiated is assigned to *a* and a contract is signed via a *definitiveAccept* message to *a* and a *definitiveReject* message is sent to every other contractor. If there is a contractor *b* whose bid is higher that this definitiveBid, *a* is demoted while *b* is promoted, via *preReject* and *preAccept* messages respectively.
  - *This bid comes from a past potential contractor:* as Erlang does not guarantee message ordering from different processes, it is possible for a *definitiveBid* from a potential contractor to arrive to the manager when this process has been demoted. When this situation happens, *definitiveBid* message is processed like a *preBid* one.

A manager agent similarly negotiates all the tasks it has, and when all tasks have been successfully assigned among the contractors, the manager process finishes.

A contractor agents listens and responds to incoming messages sent by manager agents:

- **announce:** when this message is received, the contractor adds this task to its list of offered tasks. The list of tasks is sorted according to the preference for the contractor to serve a particular task. As previously indicated, as we are primarily interested in evaluating the suitability of Erlang for implementing such a multi-agent system, with regards to performance, we here abstract away from the decision regarding task preference, and simply order the task list in arrival order. Finally, a contractor agent computes the time required to finish the task according to its own schedule, and sends a *preBid* to the manager of the task.
- **preAccept:** the contractor computes again the time required to finish the task, as the calculation may have changed from its first prebid, and sends a *definitiveBid* to the manager.
- **preReject:** if the bid for the task this message refers cannot be improved, the preference from the contractor to serve this this task should be decreased. In order to simulate this, the task is removed from the list of offered tasks and added to its last position. It the bid can be improved, a new *preBid* message is sent to the manager.
- **definitiveAccept:** the contract is signed, and the contractor commits itself to complete the task, so the agent removes it from its list of offered tasks and adds its to its list of executed ones.
- **definitiveReject:** as there is no possibility for the agent to accomplish the task, it is removed from its list of offered tasks, which means that the contractor leaves the negotiation.

As it is a simulated environment, we introduced a new message with the only purpose to let the contractor processes finish when there are no more tasks to negotiate. Then, when a contractor receives a *finish* message from the main process and its list of offered tasks is empty, it terminates. If that list is not empty, it resends the finish message to itself, in order to process it again later.

3.4 Fault Tolerance

In [1], the means for detecting a failure in an agent is based on the absence of an expected communication. Concretely an agent is suspected to be unavailable for the negotiation when no message arrives from it during a certain time interval. This behaviour can be implemented directly in Erlang by letting receive statements time out. However, as Erlang provides a better mechanism for error detection, process linking, we decided to use it instead. Nevertheless, in a large multi-agent systems linking every process to each other is not efficient as this can generate a huge number of termination messages to be generated. Because of that, the processes are linked to each other in a dynamic way that varies with respect to the status of the negotiations. As depicted in Figure 1, every contractor process is linked to the managers of the negotiations they are involved in. Then, as the links are bidirectional, we can treat failures by taking into account the state of the negotiating state, and the type of process that has failed:

- **Contractor failure:** the actions to be taken by the manager depends on the status of the contractor that failed
  - *Potential Contractor:* if a failure is detected in the potential contractor, the manager deletes it and its bid from the list of

bids and possible contractors, respectively. Then, it chooses another potential contractor and continues the negotiation of the task.

    - *Not potential contractor:* the manager erases the bid this contractor sent, if any, and removes it from the list of possible contractors.

- **Manager failure:** if a manager fails during a negotiation, all the contractors linked to it perform the following steps in order to avoid getting stuck in the negotiation:

    - Every contractor generates a link to all the other contractors whose process identifiers were included in the task announcement, and sends them the last answer received from the manager, labelled *manager_decision*. If none was received yet, it sends *unknown* and infers a definitive rejection for itself in order to avoid the possibility of having several processes inferring acceptance simultaneously.
    - A contractor waits for an answer from all the processes that were successfully linked to it. The rest are assumed to be in a failure state, as it is not possible to create a link to a process already terminated. If all the other processes send a *manager_decision* belonging to set {*preReject*, *definitiveReject* }, the contractor infers a final status of *definitiveAccept* for the task and, thus, it behaves as if a contract had been signed with the manager for the execution of the task. If it receives any *unknown*, *preAccept* or *definitiveAccept* message or if another contractor involved in the stuck avoiding process fails, detected through the existing link to it, the contractor infers a *definitiveReject* for the task, as avoiding simultaneous acceptance inference is more important that not being able to allocate a task, a procedure that could be retried.

3.5 Differences from Specification

During the implementation of the protocol, we realised the specification of the behaviour of the agents is not complete, as it is not able to handle some situations that can happen during the execution of the protocol. The first of them is a *manager_decision* of type *unknown*. As it resembles the absence of an answer from the manager, this allows the possibility of representing either a *preAccept* or *preReject* status. This situation is discussed in [1] although the expected behaviour of a contractor that receives the message is not specified. Therefore, we decided to treat such an occurrence as an inferred decision of type *definitiveReject* in order to avoid more than one contractor inferring an acceptance status. Another required decision was how to deal with failures in a contractor, when it is the potential contractor, as the specified algorithm only elaborates on what to do when a non potential contractor agent fails. The solution was quite obvious but it yet represents a shortcoming of the protocol specification. In addition, concerning the blockage avoidance algorithm, we decided to only allow the inference of *definitiveAccept* and *definitiveReject* negotiation status. The reasons for this decision derive from the fact that an inferred status of *preAccept* would provide the same semantics as a *definitiveAccept* one, while a *preReject* inferred status only allows to degrade a task, which still will remain in the list of offered tasks. Then, as the inference about the negotiation status for a task, is only computed once, the task will remain in the list of offered tasks of the contractors indefinitely. This problem can be solved by inferring a *definitiveReject* status, which removes the task from the list of offered tasks and provides the expected semantics. Finally, as the specification does not indicate what to do when a *contractor_decision* message arrives, and we found no use for it, we simply removed that mechanism from the blockage avoidance mechanism, leading to the savings of a huge amount of messages with no observable semantic changes.

4. Experiments

In the previous sections we have shown how the protocol specification could be easily implemented in Erlang, with no major changes necessary, thus greatly simplifying the design and implementation task. However, our chief motivation for implementing the protocol in Erlang was to assess whether the performance of the protocol implementation would be improved, with regards to the size of multi-agent systems that could be handled.

In order to evaluate this, we conducted a series of experiments, also present in [1], to determine if the re-implementation in Erlang provided any performance benefits. The original implementation was written in Java but the architecture over which the experiments were run is unknown to us. It modelled a transportation application. Nevertheless, it is still comparable to our implementation as the protocol is the same, thus involving the same kind of communication, and the computations are not heavier than the ones our implemented agents perform. Moreover, the original implementation includes a feature, not imposed by the protocol, that can shorten the overall process of task allocation. This feature implies that if a manager agent tries to allocate identical tasks, the bid from a contractor for certain task is also taken into account for its identical ones. For instance, assume a scenario where manager *M* announces identical tasks t_1 and t_2, and contractor *C* bids for them with bids b_1 and b_2, respectively. If $b_1 < b_2$ and C is preRejected for task t_1, the manager takes b_1 as bid from *C* for t_2. In our implementation, agent *C* has to bid again for t_2 in order to improve its bid, requiring more messages to be exchanged, thus increasing overall process. We decided not to implement that feature as it is implementation dependend and not derived from the procotol specification.

In order to obtain relevant results, every experiment was repeated at least a hundred times. All of them were run under Ubuntu Linux version 10.04 in a computer with two 2.53 GHz processors. Our execution time measurements cover the whole negotiation process: from the moment the different agents start to be created to the time instant the last agent finishes its execution. As explained in Sect. 3.1, task execution is just simulated in our implementation by committing a contract to a contractor which does not allow overlapping of its assigned tasks. In fact, tasks are just defined by a unique identifier and a number representing the time units that task completion requires. The same simulation is performed in the implementation by the protocol designers.

The experiments are divided in two groups. In the first group the experiments are run with the same number of agents as in [1], to compare the two protocol implementations. In the second group of experiments, a larger number of clients has been considered, for the Erlang implementation.

4.1 Small Agent Population

Here we present the results of a series of experiments which are directly comparable to measurements reported in [1], as they do not measure protocol behavior under failure conditions. Therefore, our deletion of *contractor_decision* message has no impact in the comparison of both implementations. We consider the experiments in this series to measure protocol behaviour in rather small agent groups. We present the results in tables due to the big difference between execution times from both implementations.

The first experimental group is composed of 4 agents, 2 contractors and 2 managers. The results obtained for the Erlang implementation, and the original results reported by the protocol designers are:

Number of Tasks	Aknine et al. Average Execution Time (milliseconds)	Erlang Average Execution Time(milliseconds)
4	1320	0.122
10	2530	0.248
20	3460	0.518
30	5050	0.698
40	9140	0.974
50	14258	1.13

Table 1. 2 contractors and 2 managers

As we can see, the execution time required for the negotiation in our Erlang implementation is four orders of magnitude better than the timings reported by the designers of the protocol. As the results for such a small population are not very significant, we repeated the experiments with the maximum number of agents the protocol designers used to test their implementation. The results obtained for a population of 40 agents (14 managers and 26 contractors) were:

Number of Tasks	Aknine et al. Average Execution Time (milliseconds)	Erlang Average Execution Time(milliseconds)
4	3509	0.798
10	7901	2.027
20	10821	4.232
30	16941	7.292
40	18788	9.905
50	29872	1.2116

Table 2. 26 contractors and 14 managers

The results again indicate that our Erlang implementation runs approximately 1000 times faster. We repeated the experiment for another population of 40 agents, this time with 24 managers and 16 contractors. The results obtained were:

Number of Tasks	Aknine et al. Average Execution Time (milliseconds)	Erlang Average Execution Time(milliseconds)
4	3621	0.579
10	8189	1.374
20	10934	3.293
30	16991	4.743
40	18933	7.057
50	29978	9.686

Table 3. 16 contractors and 24 managers

The results for this scenario are consistent with early ones. Erlang clearly is an efficient platform for implementing multi-agent systems.

4.2 Large Agent Population

To measure the performance of the Erlang implementation of the protocol on larger, perhaps more realistic, multi-agent systems, we decided to repeat the previous experiments with much bigger agent populations. There is no possibility to compare the outcome of these experiments to previous works, as [1], for instance, contains no measurements for populations remotely similar in size to ours.

In the following table, we show how the execution time correlates with an increase in the number of manager agents in a negotiation of 1000 tasks among 100 contractor agents:

Manager Agents	Erlang Implementation Average Execution Time(sec)
10	0.914946
20	1.231567
40	1.869366
60	2.515276
80	3.094301
100	3.910985

Table 4. 1000 tasks to 100 contractors

Then, we repeated the experiment for a variable number of contractors, while the number of managers remains constant with a population of 10, as depicted in the table below:

Contractor Agents	Erlang Implementation Average Execution Time(sec)
100	0.853911
200	2.212398
400	6.691916
600	12.915213
800	21.532578
1000	32.297314

Table 5. 100 tasks and 10 managers

Next, we measured the impact on execution time in environments with a constant number of contractors and managers (100 and 10 respectively), and a variable number of tasks:

Number of Tasks	Erlang Implementation Average Execution Time(sec)
100	0.087581
200	0.172625
400	0.353451
600	0.525856
800	0.708959
1000	0.873533
2000	1.75532
5000	4.396946
10000	8.620578

Table 6. 100 contractors and 10 managers

Finally, in order to test the performance of the algorithm in an environment with few tasks to assign, only 10, we run a series of experiments that involved 10 managers and a sharply increasing number of contractors:

4.3 Performance under Failure

In the previous sets of experiments, the execution time measured was taken from executions where there was no failure in the agents involved in the negotiation. To evaluate the performance of the

Contractor Agents	Erlang Implementation Average Execution Time(sec)
100	0.009974
200	0.026022
400	0.80698
600	0.158065
800	0.302604
1000	0.413268
2000	1.531921
3000	3.467723
3500	4.342805

Table 7. 10 tasks and 10 managers

protocol implementation in failure scenarios, we designed a set of experiments where failures were introduced in the agents at certain points of the negotiation. The first experiment set measures execution time when the *potential contractors* fail as soon as they receive a *preAccept* message, except for one agent of the population who will not fail and, thus, will commit all contracts. The results obtained for with 10 tasks, 5 managers and a varying number of contractors were:

Contractor Agents	Erlang Implementation Average Execution Time(sec)
100	0.007226
500	0.070008
1000	0.230252
2000	0.91076
4000	3.7773
5000	5.662974

Table 8. 10 tasks and 5 managers under failure

We can observe how the execution time is significantly lower than earlier measurements, as the number of contractors is constantly decreasing, as the tasks are being assigned, until only one contractor remains alive. Besides, we wanted to test the implementation when there are failures in the managers, as this will increase the number of communications taking place. For instance, in a negotiation between one manager and *n* contractors, with $n > 1$, the number of messages sent once the announcement of the task has been finished, is very close to 1. However, if there is a failure in the manager, this number grows exponentially, as in the better case there are $2n^n$ messages. The results obtained for the allocation of 5 tasks, where there are 5 managers and 2 of them fail, immediately before sending a *definitiveAccept*, under a variable number of contractors were:

Contractor Agents	Erlang Implementation Average Execution Time(sec)
100	0.069875
500	1.48867
1000	9.858859
2000	67.838057

Table 9. 5 tasks and 5 managers under failure

These results show the aforementioned exponential growth in the number of messages exchanged. Nevertheless, they are still surprising, as the time required for such a significant amount of computations is relatively small, especially if compared to the results provided by the protocol designers. Once again, Erlang has proven itself a very capable platform for this type of process and communication intensive systems.

5. Conclusion and Future Work

In this paper, we have shown the benefits of using Erlang for the implementation of a task allocation protocol in a multi-agent system. One such significant advantage is the fact that mapping the original protocol specification into an Erlang program took very little effort, approximately one man month, as the assumptions made in the protocol matches well with the actor process and communication model used in Erlang.

We have further shown how the information the different agents handle can be represented as Erlang records. Moreover, it is significant how easily an event-driven system, as multi-agent systems usually are, can be implemented using Erlang message passing. Another useful mechanism Erlang provides is process linking which, compared to the protocol specification, provides a high-level failure notification mechanism. The use of this high-level mechanism contributed to enabling the agents to become more "aware of their environment", in terms of their knowledge about the status of other agents, which is a very desirable, if not mandatory, feature in collaborative multi-agent systems.

Regarding the execution of multi-agent systems, [1] states "*We understand it is difficult to increase the number of agents and tasks, because of the computational complexity problem*", referring to experiments that performed a task allocation of 8 tasks which involved a population of 8 agents. In this document, we have shown that by using the Erlang/OTP runtime system, which has a superior implementation of processes management and message passing, it is possibly to dramatically increase the number both of tasks allocated, and the number of agents involved in a negotiation.

Because of these two observations, we strongly believe that Erlang is a very good choice as programming language for the implementation of multi-agent systems, allowing the generation of huge agent populations and, thus, increasing relevance of both empirical experiments and real world applications.

As a future line of work, we intend to formally verify that our protocol implementation behaves correctly, and moreover, that the original protocol specification is consistent. Specifically, we plan to model check our Contract Net Protocol extension implementation using the McErlang [2] model checker. Having implemented the protocol we remain suspicious of a number of features of the protocol, and we are of the opinion that a verification is required to increase the trust, and the quality, of the original protocol specification.

References

[1] S. Aknine, S. Pinson and M.F Shakun. An Extended Multi-Agent Negotiation Protocol. *In Int. Journal of Autonomous Agents and Multi-Agent Systems*. Volume 8, pages 5–45. Kluwer Academic Publishers, 2004.

[2] L. Fredlund and H. Svensson. McErlang: a model checker for a distributed functional programming language. In *Proceedings of the 12th ACM SIGPLAN International conference on functional programming (ICFP 2007)*, Oct. 2007.

[3] R.G. Smith and R. Davis. Frameworks for co-operation in distributed problem solving. *In IEEE Transaction on System, Man and Cybernetics*. Volume 11, number 1, 1981.