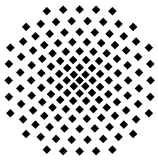

Universität Stuttgart
Fakultät Informatik

An Agent-Based Framework for the Transparent Distribution of Computations

Authors:
Dipl.-Inform. M. Straßer
Dipl.-Inform. J. Baumann
Dr. Ing. M. Schwehm

Institut für Parallele und Verteilte
Höchstleistungsrechner (IPVR)
Fakultät Informatik
Universität Stuttgart
Breitwiesenstr. 20 - 22
D-70565 Stuttgart

A Protocol for Preserving the Exactly-Once Property of Mobile Agents

M. Straßer, J. Baumann, M. Schwehm

Bericht 1999/06
Juni 1999

An Agent-Based Framework for the Transparent Distribution of Computations

Markus Strasser, Joachim Baumann, Markus Schwehm

Abstract *A mobile agent based framweork for the transparant distribution and concurrent execution of computations is presented. The framework uses design patterns like the master-slave, abstract factory or the strategy pattern. The architecture of the framework is built on top of a mobile agent system. A performance model allows to identify performance bottlenecks and unbalanced situations within the framework. The Framework has been implemented and tested on top of the mobile agent system Mole*

Keywords: mobile agents, distributed computation, performance model, load balancing

1 Introduction

Mobile Agent Systems have received great attention in the last years as a new programming paradigm for widely distributed and heterogeneous systems. The basic concepts of agent systems are *places* and *agents*. An agent system consists of a number of places where computation can take place and where various services are provided. Agents are active entities which may move from place to place to meet other agents or to access services provided there. The mobility of the agents - i.e. their ability to migrate from one place to another - is the basic difference from other approaches for distributed systems.
Major advantages of mobile agents are seen in the possibility of reducing (expensive) global communication costs by moving the computation to the data [Chess et al. 1997] and in the possibility to easily distribute complex computations onto several, possibly heterogenous, hosts. [Straßer and Schwehm 1997] deals with the first of this two advantages and presents a performance model regarding network load and execution time which can help to identify situations for which agent migration is advantageous compared to remote procedure calls. This paper discusses some aspects of the second advantage, the distribution of complex computations using mobile agents.

In general, the distribution of complex computations using mobile agents requires the application developer to explicitly deal with the distribution, i.e. he has to explicitly code how and on which node to distribute and how to communicate. In this paper, we present a small framework for the transparent distribution of computations over a network of mobile agent systems. Using this framework, all computations which can be split into smaller, autonomously computable parts can be distributed automatically. Furthermore, to help the application developer to decide whether to distribute a computation or not, a performance model for the framework is developed.
The paper is organized as follows: Section 2 describes the agent based framework. Section 3 introduces a performance model for this framework. Section 4 outlines some measurements performed using an implementation of the framework.

2 The Framework

In this section we present the framework developed for transparently distributing problems over a network of mobile agent systems. First we give a short description of the patterns used for the design, then we present the agent model (including system and fault model), and present the architecture of the framework.

2.1 Used Patterns

Patterns are a way of describing in a simple and elegant manner solutions to specific problems in object-oriented software design. In our framework

we used some well-known patterns given in [Gamma et al. 1994] and [Buschmann et al. 1996]. In this section we shortly describe the patterns we used in the design of the framework.

Master-Slave Pattern. "Divide and Conquer" is a common solution to many kinds of problems in computer science. The work is partitioned by the master into several independent jobs which then are given to different slaves. The results returned by the slaves are then combined and a global result is computed. In our framework the master is the coordinator that dispatches different jobs, and the slaves are the worker agents computing the results of the jobs.

Strategy Pattern. A strategy encapsulates an algorithm and its algorithm-specific data in a way that allows to avoid detailed knowledge in the object using it. In our framework each worker receives a strategy used to compute one part of the problem. The worker does not have to know the intrinsics of the strategy, it does not even have to understand the job objects sent to it. The job objects contain the context needed and interpreted by the strategy.

Abstract Factory Pattern. An abstract factory provides an interface for creating objects without specifying their concrete class. This allows to defer the specification of the actual object until runtime. In our case the abstract factory produces the strategies that will be sent out with the worker agents, without any component, besides the factory, knowing the specifics of the strategy.

2.2 The Agent Model

In this section we present a very simplified agent model, which describes the minimal concepts needed for our framework. Agent models for existing agent systems are normally much more complex; one example for this is the agent model used for Mole (see [BaumEA98] for a description). Furthermore we present the fault model we assume for the underlying distributed system.

The Agent Model. The agent model used throughout this paper is based on the concepts of mobile agents and places. An agent system consists of a number of (abstract) places providing an infrastructure for agents. Places provide the environment for safely executing local as well as visiting agents. An agent system consists of a number of (abstract) places, being the home of various services. Agents are active entities, which may move from place to place to meet other agents and access the places' services. In our model (see Figure 1), agents may be multi-threaded entities, whose state and code is transferred to the new place when agent migration takes place. Each agent is identified by a globally unique agent identifier. An agent's identifier is generated by the system at agent creation time. A place is entirely located at a single node of the underlying network.

Fault Model. We can distinguish node and network failures. We assume that nodes suffer from crash-recovery failures only. This type of failure is an extension of the original crash failure, in which no failure is assumed permanent (see Aguilera, Chen and Toueg (1998) for details). The failure causes the node to halt and to lose its internal volatile state. The stable storage survives failures. We assume a communication protocol is used that supports full connectivity between the nodes, and the delivery of messages in order, correct (i.e. the message is not garbled), and exactly-once as long as no network fault occurs. Furthermore, we assume the

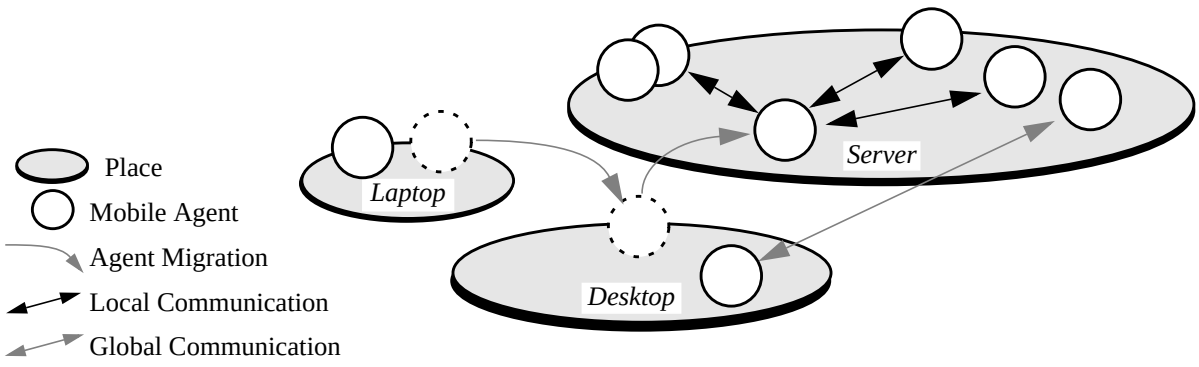

Figure 1: Mole System Overview

communication protocol to be fail-aware. Many protocols providing this type of reliable datagram service are known, i.e. this assumption is close to reality. Consequently the following can be assumed: the communication network is fully connected and it provides reliable communication channels as long as no network fault occurs. Communication networks can suffer from crash failures that may cause the network to be partitioned. In the case of a network partition the communication channel between sender and receiver in different partitions fails, but continues to work between participants within the same partition. Node and network failures are detectable, but not distinguishable.

2.3 Architecture

Our architecture consists of an application, a coordinator and workers. All of these are agents. Optionally, the coordinator may be realized as an object being attached to the application. A programmer using our framework has to provide only the strategies (containing the algorithm and the global strategy identifier) and the application (integrating the results) to allow transparent distribution.

The normal operation is as follows (numbers in braces correlate to the numbers in Figure 2): the application is started and in turn starts a coordinator (provided by the framework) with a strategy factory as a parameter (0). In the next step the application begins to give different jobs to the coordinator (1). Every job contains an identifier for the strategy needed for this job. The coordinator examines its list of free workers for one containing this strategy (in the beginning none), and sends the job to the worker. If no worker exists containing the needed strategy, or if all workers are busy, and there are still places left to which workers can be sent, then a new worker is created with the needed strategy. The needed strategy is requested from the factory with the help of the strategy identifier (2). The worker is now sent to a place where it computes the result of the job (3). As soon as the result is sent back (4), the coordinator removes the job from the worker's queue and transfers the next job in the queue (5). The local queue in the coordinator contains time-stamped entries with the jobs scheduled for one worker. Different policies can be used to send the jobs to the workers, or to reschedule if either a job needs an unforeseen time, or if the worker has crashed (detected via regular "alive" messages). The result received by the coordinator is then returned to the application, which integrates this part into the overall result (6).

This continues until the application signals that it has no more jobs. Now the coordinator removes all workers as soon as their queue is emptied and the last result reported. The final step for the coordinator is to remove itself to finish the clean-up.

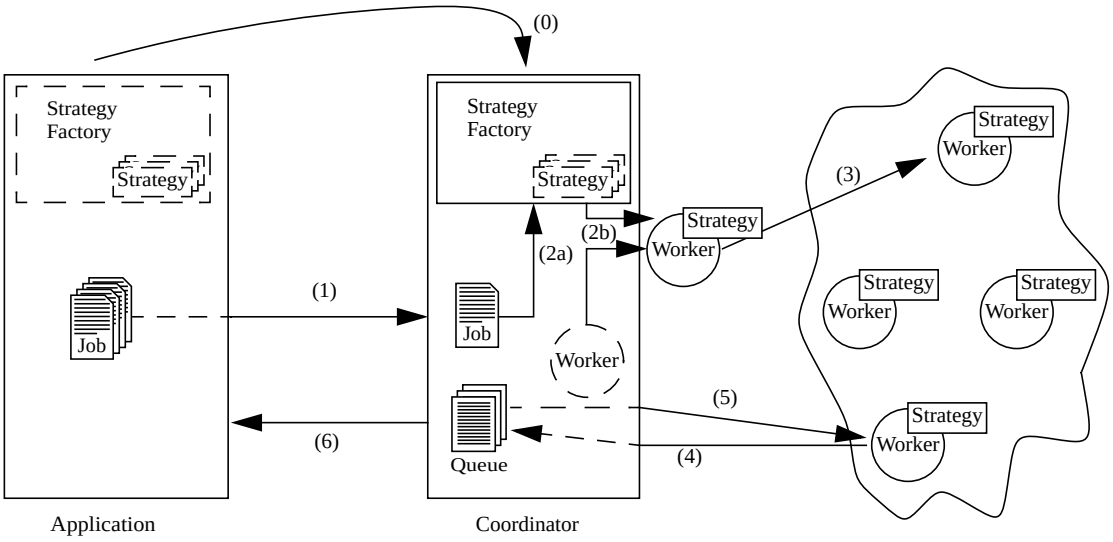

Figure 2: Architecture of the Framework

3 Performance Model

In this section, we will discuss some performance issues regarding the presented framework. The following simplifying assumptions are made:

- The coordinator itself is no agent. It is integrated (as an object with own threads) into the application agent. Therefore, no (global) communication between application and coordinator has to be considered.
- All places execute jobs with the same speed, the average execution time t_{exe} of a work package and the average integration time t_{int} of a work package result are known.
- The time t_{start} needed to start a new worker agents is known.
- Network delay δ and throughput τ are equal between all places. Therefore, the transfer times of work packages t_{snd}, result packages t_{rec} and the migration time for worker agents t_{mig} are constants.
- Only one type of worker functionalities is assumed

3.1 Local Computation

To calculate the time needed for a local execution of a job, two approaches have to be considered: The first approach is to time the execution of the job. In this case, the job is done and you don't have to think about distribution. But you can use this measurement as a base for similar jobs. The second approach is to use the (known) average times for the execution of a work package and the integration of results to compute the time for a local execution.
To be able to deal with the problem in general, we use the second approach and calculate the time t_0 for the local execution by

$$t_0 = n(t_{exe} + t_{int})$$

where t_{exe} and t_{int} are the average work package execution and integration times and n is the number of work packages of the job. In this case, the calculated time is the "worst case time" for the local execution of the job including the overhead for dividing the job into work packages and for integration of the work package results.

3.2 Distributed Computation

To calculate the time needed for the distributed computation of a job consisting of n work packages with m worker agents, we first have a look at the sequence of actions that have to be executed by the coordinator and the worker agents respectively. The coordinator performs the following algorithm:

1. Repeat for each worker agent (m times)
   a. Start worker agent
   b. Migrate worker agents to destination
   c. Send work package
2. Repeat until all work packages are sent (n-m times)
   a. Receive result
   b. Send new work package
   c. Integrate result
3. Repeat until all results are received (m times):
   a. Receive result
   b. Integrate result

Meanwhile the worker agents does

4. Repeat until terminated (about n/m times):
   a. Receive work package
   b. Execute work package
   c. Deliver result

A lower bound for the execution times for each algorithmic step is computed as follows. The time for the initialization phase is approximated by

$$t_{ini} = m(t_{start} + t_{mig} + t_{snd})$$

The main loop of the coordinator computes to

$$t_{coord} = (n-m)(t_{snd} + t_{rec} + t_{int})$$

The wrap up time at the end of the computation is

$$t_{fin} = \frac{m}{n}(t_{rec} + t_{int})$$

On the other hand, a lower bound for the execution time of the worker agents is

$$t_{work} = \frac{n}{m}(t_{rec} + t_{exe} + t_{snd})$$

The equations above are only lower bounds for the execution time because additional time might become necessary for the synchronization between coordinator and the worker agents. For example, the coordinator normally has to wait for results

from the worker agents before it can continue sending work packages. But it can also happen that worker agents have to wait for work packages, e.g. because the coordinator has to process too many small work packages or is busy with the integration of previously delivered results. In order to estimate the overall execution time, we have to know whether the coordinator is overloaded with work or not. The coordinator can be expected to be overloaded if

$$t_{coord} > t_{work}$$

i.e. the inner loop of the coordinator takes longer than the average worker agent cycle: In this case the coordinator is the bottleneck of the computation.

The overall computing time using *m* worker agents now computes to

$$t_m = t_{ini} + max(t_{coord}, t_{work}) + t_{fin}$$

3.3 Evaluation

Using the above equations we have computed the expected performance of the framework for two scenarios. In the first scenario we consider the execution of a fixed number of 500 work packages using zero to four worker agents. The execution times for the diagram in Figure 3 are given by $t_{start} = 200$, $t_{mig} = 100$, $t_{snd} = t_{rec} = 1$, $t_{int} = 3$, while the execution time for a work package t_{exe} is varied between 1 and 20. It can be observed that the use of worker agents yields a performance gain only for large work packages. For example, using one worker agent yields a performance gain only if the execution time for the work package is larger than 2, i.e. if the remote execution of the work package is faster than the time needed for sending the work package request and for receiving the results. In this case the performance gain can be explained by the concurrent execution of the work package by the worker agent and the integration of the results by the coordinator. If more worker agents are used, the overhead for the initial starting and migarating of worker agents increases and the coordinator has to prepare more work packages per time unit. So the coordinator remains a bottleneck unless larger work packages are sent to the worker agents.

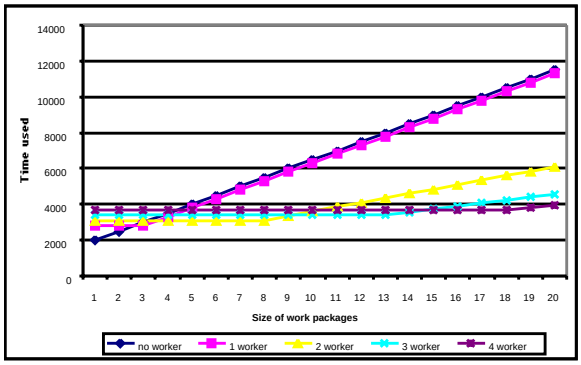

Figure 3: Performance of fixed number of work packages

In the second szenario, a fixed overall work load of 5000 units are executed using zero to four worker agents. Using the same execution times as above, the graph in Figure 4is normalized using the execution time for the local coordinator-only execution t_0. It can be observed that using a single worker agent there is a clear optimal work package size with $t_{exe} = 3$.Similarly a larger number of worker agents also need for a larger work package size.

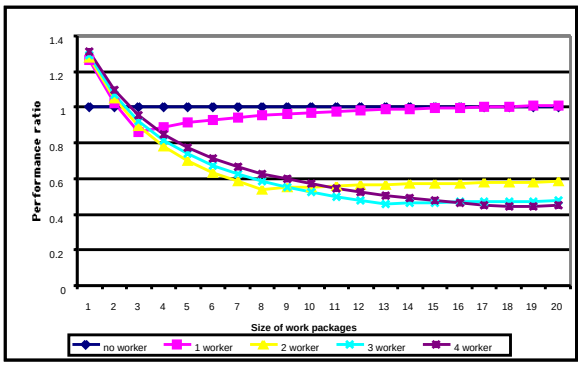

Figure 4: Relative performance for fixed problem size

4 Implementation

A prototype of the framework has been implemented in Java using Mole [BaumEA98]. The implementation provides a set of interfaces which the agents serving as application, coordinator or worker have to implement. In addition, basic implemen-

tations for each of the three agent types are provided.
This approach allows to simply replace one agent, e.g. the coordinator by a similar agent with enhanced capabilities. For example, in the first prototype, the crucial question of how to find places for the worker agent has been solved by simply providing the coordinator with a list of available places. Replacing the coordinator by another agent which collects load information of available places and schedules the jobs according to this information would provide a very powerful tool for load balancing.
To show the validity of the concept, we just started to make meassurements using the implemented prototype. For the meassurements, an application calculating the Mandelbrot set [Peitgen et al. 1992] has been implemented.
The table shows average times (in milliseconds) for the distributed calculation of a Mandelbrot set

	2	3	4
16	33452	29110	27095
32	18065	14475	11353
64	15028	12724	9573

(384x256 image pixels) with different numbers of workers (2, 3 and 4 workers, one cpu per worker) and different job sizes (16x16, 32x32 and 64x64 image pixels per job). The coordinator and the application reside on the same place. The parameters of the calculated set (especially the parameter which specifies “infinity”) are chosen to generate a very heavy load.

5 Conclusions

We have presented a mobile agents based framework for the transparent distribution and concurrent execution of computations. A performance model for this framework indicates that a performance gain is possible despite the overhead introduced by the creation and coordination of agents. The performance model furthermore allows to tune work package sizes such that the workload is balanced between coordinator agent and worker agents. First measurements performed on the implementation of the framework on top of the mobile agent system Mole confirm these evaluations. Further measurements and their discussion will be included in the final version of this paper.

References

Aguilera, M. K. and Chen, W. and Toueg, S. (1998), “Failure detection and consensus in the crash-recovery model”, Technical Report TR98-1676, Cornell University, Computer Science Department.

Baumann, J. and Hohl, F. and Rothermel, K. and Straßer, M. (1998), “Mole - Concepts of a mobile agent system”, WWW Journal 1, 3, Baltzer Science Publishers, pp. 123 - 137.

Buschmann, F. and Meunier, R. and Rohnert, H. and Sommerlad, P. and Stal, M. (1996), A System of Patterns, John Wiley & Sons, England.

Chess, D. and Harrison, C. and Kershenbaum, A. (1997), “Mobile Agents: Are They a Good Idea?”, In: Vitek, J. and Tschudin, C. (ed) Mobile Object Systems. Towards the Programmable Internet. Second International Workshop, MO’96. Selected Presentations and Invited Papers, Springer, Berlin, Germany, pp. 25-47

Gamma, E. and Helm, R. and Johnson, R. and Vlissides, J. (1994), Patterns: elements of reusable object-oriented software, Addison-Wesley, Reading, Massachusetts.

Jalote, P. (1994), Fault Tolerance in Distributed Systems, PTR Prentice Hall.

Peitgen, H.-O. and Jürgens, H. and Saupe, D. (1992), Fractals for the classroom, Springer, New York

Straßer M. and Schwehm M. (1997), A Performance Model for Mobile Agent Systems. In: H. Arabnia (ed.), Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'97).Vol II, CSREA, pp. 1132-1140