ARTÍCULO

Octopus - Computation Agents Environment

Piotr Uhruski, Marek Grochowski, Robert Schaefer

Institute of Computer Science
Jagiellonian University
Krakow
{grochowski,schaefer,uhruski}@ii.uj.edu.pl

Abstract

This paper presents a platform named Octopus that facilitates the building and execution of mobile agent based applications. It presents the key ideas of how agents embed the computational task and how they cooperate to find the solution. The Octopus is presented with its key mechanisms used to sustain and execute the agents. The cornerstones of the Octopus design are described in detail, giving readers an overview on how to implement a computation problem within the platform. Finally, actual application examples are shown with a short discussion of the Octopus based implementation properties.

Keywords: mobile agents, migration, computational tasks, local scheduling.

1 The Structure of Agent Computational System

Parallel computing systems based on distributed network resources become the most powerful tools in high performance computing. To effectively utilize often heterogeneous network resources, a universal solution allowing for the easy design, implementation, deployment and finally control of the application is required. We propose an agent based computing system with two layer architecture that clearly divides such system required functionalities. The upper layer is the application while the lower is a platform sustaining agents - the Octopus.

Multi Agent Application
This layer provides the means and tools to wrap the computational task into agents. The layer is build from agents, but the agents themself are further decomposed into two sub-layers - a shell and an embedded task (see [10]). A task is the particular problem with the data required for computations. The task should have the ability to divide itself, but that is a natural property in parallel computing. The outer agent is a task container - it requires its execution environment to provide load information (including RAM and CPU utilization) to compute a local scheduling policy and autonomously take the decision to continue internal task computing or migrate to find better resources. Agents require the execution environment to let them communicate with each other for cooperation.

Agents Execution Platform
The application agents require an environment to execute them. The specific requirements for this layer originated from CAE computing and mainly included minimum runtime overhead with maximum performance. The platform provides the minimal set of functionalities clearly required by large computation problems, but these have to be performing well. Thus we did not require a universal solution that would fit any agent based applications. The platform has to be well scal-

able in terms of the utilized machines, and we required the ability to define virtual topologies. This feature lets us reflect the physical network characteristics in the virtual topology and thus influence the computation. The presented requirements for the execution platform made us create our own, specialized agent runtime environment instead of using available solutions [1, 2]. We would like, however, to reuse the well-known standards for such environments (as FIPA [3]).

Introducing these layers gave clear functionalities separation between the execution platform and the application's layer. First, the execution platform - named Octopus - became problem independent. It may be configured (e.g. by using the virtual topology parameters) to support a certain application with particular runtime characteristics, but the platform itself provides only basic mechanisms to run the application agents. Secondly, the agent application layer supports ready-made Smart Solid agents (see [9]) that implement required environment integration functionalities. Therefore the actual application implementation is done solely in the upper layer, which requires the embedded tasks to be partitionable, meaning the agents actually divide overall computational task into smaller sub-processes. That follows the basic processing models introduced by Lamport and Charrone [4]. This actually means that our architecture supports execution of any kind of parallel processes.

Beside the characteristic of overall system architecture, it influences the actual application design and its runtime properties. A single task mapping to agents depends on the particular problem. In this paper we present both one to one and one agent to many tasks mappings. This allows the application designer to change the executed entitie's size and thus influence the entities grain. On the other hand, thanks to the dynamic task's partitioning and dynamic scheduling executed by agents (see [10]), the application's execution is adapted to the changing environment properties. The Smart Solid agents have autonomous rights and supporting functionalities to change grain size (glue or split internal tasks) or to reschedule while computing. That makes the platform once more problem independent, as it's free to take autonomous decisions outside the problems scope.

2 Definitions

We start with an analytical model description of the Octopus and the application model as introduced in [10].

Environment
In our model, the execution environment for computation applications is modeled as a quarduple $(\mathbf{N}, B_H, perf, conn)$ where:

$\mathbf{N} = \{P_1, \ldots, P_n\}$, where P_i is a Virtual Computation Node (VCN). Each VCN may contain a number of agents.

B_H is the connection topology $B_H = \{N_1, \ldots, N_n\}, N_i \subset \mathbf{N}$ is an immediate neighborhood of P_i (including P_i as well).

$perf = \{perf_1, \ldots, perf_n\}, perf_i : \mathbb{R}_+ \to \mathbb{R}_+$ is a family of functions which describes the relative performance of all VCNs with respect to the total memory request of all agents allocated on the VCN.

$conn : \mathbf{N} \times \mathbf{N} \to \mathbb{R}_+$ is a function which describes up-to-date connection speed between two VCNs.

Application
The application agent is denoted by A_i where i stands for an unambiguous agent's identifier. An agent is independent in terms of its structure and life-cycle. Firstly, the Octopus does not require agents to have any particular internal structure, so the agent needs to carry all required data internally or know how to communicate to get it. For a life-cycle, the Octopus does not require the agent to follow any execution pattern. After the agent finishes its execution, the Octopus delivers it to its owner. One exception to this rule is the right of the Octopus to protect its resources - if there are too many agents which do not want to migrate from the host, the Octopus may block their execution. Finally, the application is a set of agents, and a computing application state is a triple $(\mathbf{A}_t, G_t, Sch_t)$, $t \in [0, +\infty)$ where:

$\mathbf{A}_t$ is the set of application agents, $\mathbf{A}_t = \{A_{\xi_j}\}_{\xi_j \in I_t}$, I_t is the set of indices of agents active at the time t,

G_t is the tree representing agents partitioning at the time t. All agents constitute the set

of nodes $\bigcup_{\xi\in\Theta} A_\xi$, $\Theta = \bigcup_{j=0}^{t} I_j$, while G_t edges show the partitioning history. All information on how to rebuilt G_t is spread among all agents such that each of them knows only its neighbors in the tree.

$\{Sch_t\}_{t\in[0,+\infty)}$ is the family of functions such that $Sch_t : \mathbf{A}_t \rightarrow \mathbf{N}$ is the current schedule of application agents among the MAS platform servers. The function is represented by the sets ω_j of agents' indices allocated on each $P_j \in \mathbf{N}$. Each ω_j is locally stored and managed by P_j.

3 Octopus Key Tasks

The Octopus serves as a low level execution mechanism; it does not implement any application-related logic. It boosts the application with a simple and robust execution model and gives it the decision to implement more complex tasks like scheduling or tasks partitioning. The following sections describe the low level Octopus tasks in more details.

Agent Execution
The key element of the Octopus is the ability to execute agents. Each agent gets its own execution space - *execution context*, through which an agent examines the environment, communicates with other entities and requests certain actions.

Communication Means
Although agents act as stand-alone units and their execution is independent, they may relate to each other. To support such relations, the Octopus handles communication between agents and their owners. For the Octopus it is enough to provide asynchronous message communication. It is up to the agent's designer to build a communication schema, which would simulate both synchronous and asynchronous communication schemas if required.

Environment Information
An agent has a set of goals with one of them being completing the internal task in a shortest possible time. To do that, the agent needs to search for the best available resources in terms of available memory and computational power. This information is gathered and delivered by the VCN running the agent. The VCN delivers not only local information, but also the distant machine's details - most preferable for P_i is the information concerning $P_j \in N_i$

Virtual Network Topology
The agents perceive the execution environment as a virtual machine with VCNs interconnected in a graph structure. It is the role of the Octopus to build and sustain this topology.

Agents Migration
To complete the agent's goals they require the ability to move across VCNs to find better resources or to locate other agents. Again, however, the migration is requested by the agent itself and is performed by the VCN to assure the agent's integrity and completeness - the Octopus also handles communication between actively migrating agents.

Agent's Construction Kit
The final Octopus goal is to speed up and ease the new application's development. For that purpose, the Octopus features a an agent skeleton implementation to be used by the developer. In addition it contains various utility classes that may be used when running the application - like an application log parser used to analyze logs to get performance numbers.

4 Design Details

The Octopus was build from the ground up using object oriented techniques and best practices. Its internal structure constitutes blocks separated by interfaces which implement various functionalities. Each such block provides a particular set of services for executed agents by implementing previously presented tasks. Because each module interacts directly with agents or indirectly supports other modules we named these blocks *policies*. A *policy* is defined as a contract between the Octopus platform and an agent, which specifies the means to achieve a certain agent's goals. The Octopus platform's functionalities are defined and implemented by the Execution, Information, Communication, Migration and Internal policies. The last one groups all Octopus functionalities which are not exposed directly or indirectly to any agent. It contains all functionalities required to deal with internal issues. All Octopus tasks may be mapped to one of the presented policies. The following sections provide details of each policy, thus giving an overall design overview.

4.1 Execution Policy

The main task of the Octopus is to execute agents. For each agent the Octopus creates a *container* - a sandbox which guarantees that agents are executed independently and do not influence each other. All agent interaction with the environment is carried through the container's interfaces. The Octopus defines the agent's life-cycle, which is then used by the container to control the agent at runtime. The life-cycle guarantees that when the agent gets migrated, it will not execute from the beginning, but from the point where it got migrated. This is defined by the *execution stages* concept - an agent may slice its execution into as many stages as it wants, and after migration, the Octopus resumes the agent's execution from the last completed stage. The agent may also state that its execution has finished and needs to be delivered to its owner. This is also handled by the execution context. The execution policy implements the agent's execution Octopus task.

4.2 Information Policy

Agents require a certain amount of information describing the current environment with its state. The Octopus does not define how the agent interprets this data; it is responsible for delivering it. The information is delivered through the agent's context, but it is gathered by a separate Octopus block. Therefore the Octopus is solely responsible for data gathering, analyzing and delivering. This separates the agent from the physical runtime environment (machine, operating system and network) configuration and gives the Octopus the advantage of being able to execute agents in a heterogeneous environment. The current implementation delivers all data required by the diffusion based scheduling implemented by the Smart Solid agents (see [10]), namely local and neighborhood host load in terms of active agent amount and virtual topology path information with the cost of communication to a given VCN. Additional information sources may be integrated with the Octopus and delivered transparently to the agent through its context.

4.3 Communication Policy

However, agents execute independently. But for collaborative work they need a means of communication to pass messages to each other. It is this communication that actually let's agents cooperate with each other. The Octopus does not define how and when messages should be exchanged between agents; rather it is responsible for finding the destination for a properly addressed message and delivering it. As stated when defining the Octopus tasks, the communication is based on asynchronous message queues. Each agent's context has incoming and outgoing message priority queues. When creating a message, the agent is responsible for setting its unique sender and receiver identification tokens and the message data that could be any type of data serializable into a stream. In addition, the message priority may be set. The agent creates a message and puts in into its outgoing queue. Queues are asynchronous, so this operation is not blocking and the agent may continue its execution. When receiving a message, the agent may actively check the incoming messages queue (non-blocking check and continue processing) or yield its execution until a new message is available (blocking message check). This gives the agent developer the possibility to implement any type of messaging schemas required - synchronous or asynchronous. To properly route the messages, the Octopus uses the virtual topology built up when establishing connections between VCNs. This is a very broad topic and is still under active development. The current implementation let's agents communicate with their owner or with agents residing on the same VCN. The Octopus also synchronizes incoming messages and keeps the queue complete despite the agent's migration ability. This is achieved by integrating the communication with migration policy and making message queue transfer a part of the migration procedure.

4.4 Migration Policy

This is the key agent's ability, which allows an application (being a set of agents) to execute more effectively - agent's may implement local scheduling algorithms based on neighborhood analysis, which may lead the agent to search and execute in a less loaded environment, thus making the whole application more effective. The Octopus platform's role in migration is to provide an transaction-like process for the migration process. It is based on the two-phase commit protocol with the following steps:

1. The agent examines the neighborhood and

internally elects a destination host with a set of desired destination host parameters (for example: maximum acceptable machine load). Migration will succeed only when distant VCN parameters conform to these numbers.

2. The agent invokes the migration on its execution context passing the desired destination VCN. The control is passed from the agent to its context, which prepares the agent for migration by serializing it. The migration policy starts the two phase commit protocol by querying the distant machine to lock a place for the new agent. When the distant VCN confirms the lock, its load is immediately increased with the new agent - this allows the migration processes to execute simultaneously from different nodes.

3. The destination machine is asked to confirm the agreed load numbers and the agent is transferred to the remote node. The distant machine may be doing other migrations at that time, but since it confirmed the required load parameters, the migrated agent should find an acceptable environment when the migration is completed.

4. All messages from the agent's incoming queue are transferred to the distant node where the new agent's context is created with new message queues.

5. The agent's owner is notified that the agent has moved - this assures the communication policy will be able to locate the migrated agent.

After these steps the migration is complete. If any of them fails, migration is rolled back, and the control is passed back to the agent with the appropriate error code. If migration succeeds, the agent starts execution on a distant node, but thanks to the execution stages concept, it continues its execution rather than starting from the beginning.

4.5 Internal Policy

The Octopus has a set of internal functionalities that are not exposed to the agents, but they are used by the platform to control the agent's execution. The policy includes the agent's serialization and the manual agent's migration. Serialization may be used by a VCN when it is too loaded and the system's performance is endangered. In such a case, an agent may get elected and serialized to the hard disk. As soon as the system resources are freed, its execution may be resumed. The manual agent's migration is the system administrator's tool to move a particular agent to a manually selected machine. This is similar to the agent-triggered migration, only the destination is imposed over an agent. Other functionalities are foreseen to be implemented for this policy with security (the agent's authentication and authorization) amongst others. This is a broad topic and requires further explorations.

5 Building Octopus-based Applications

From the very beginning, the Octopus development was driven by particular computational needs (see [9, 10]). Therefore it features a set of components that help building up an application which benefits from agent-based processing. This support is referred to as Agent SDK. It features the Octopus platform client component, abstract agent classes and other utilities not directly related to the agent or application.

Platform's Client
The Octopus platform runs agents, but these need to first be injected into one of the platform's nodes - VCNs. This agent's injector is referred to as the *requester* and has the following functionalities that cover the Octopus connection, agent delivery and communication. First, the requester connects to a given VCN and establishes a communication channel with it. The communication between the requester and the VCN is organized in the same way as within the Octopus - it is based on asynchronous messages passing. After the connection is established, it may start delivering agents to the VCN it is connected to. On the other hand, when the agent is finished, it is also delivered back to the requester by the Octopus. Finally, the agents may communicate back to the requester and may send messages to agents.
Thanks to these features, different application execution schemas are possible depending on the type of problem domain. The original problem is usually assembled by the requester, which then

feeds the data into agents who start computations. For example, if the application actually requires large data amount processing but the task may not be well partitioned (see [10] for such an example), it is better to create lightweight agents, which after injection into the Octopus, migrate to find the best execution environment. After it is found, agents send to their requester a message to get the actual computation data, which is then also delivered back in the form of a message. This guarantees fast agent scheduling. On the other side, some problem domains allow the task to divide extremely well (see [9] and the container concept). Applying the previously described procedure would result in tremendous network communication overhead, so it is better to create the agent initially with all its data encapsulated inside.

6 Technical Details

As mentioned before, the Octopus has been designed and implemented with the usage of object oriented techniques. Technically, the Octopus is implemented on the Sun Java platform with communication realized by CORBA services. Java is platform-independent since compiled Java classes (byte code) are not executed directly by the host OS, but by the Java Virtual Machine (JVM), which is customized to the operating system. Java also gives fairly a easy and adoptable mechanism for transferring any objects via network in a binary form. This is achieved by the object's serialization support. We have chosen CORBA for its definition of remote services and support for service discovery (Naming Service). Using these patterns made the Octopus code well separated from the actual communication schema and allows us to change the CORBA into another similar communication framework (for example: RMI). On the other hand, CORBA is far more complicated than low level protocols (RMI), which could impose significant overhead on the implemented applications. This was considered in the design phase, and therefore we separated the communication layer in such a way that it would be feasible to substitute CORBA with another implementation. Please see the examples section for a discussion of communication effectiveness.

7 Examples And Tests

The Octopus has been used as a base platform to implement applications for different types of problems. Up to now, we have successfully realized the applications in the CAD/CAE (this was the original field of research for our team) and genetic based computations domains. These both require parallel computations upfront, and it would be best if the solution would be realizable with a PC's network since this kind of environment is most widely available. To successfully build such applications, we require a virtual environment which would not restrict the application's design and have small overhead above the application's computation time and memory requirements. In the application's design we additionally included different scheduling and communication schemas. The environment should not restrict that since different applications may have different requirements resulting from their problem domains - the amount of data needed for computation and the possible granularity level amongst others. Additionally, non- critical features included the ease of application creation. The following subsections present the example applications with an Octopus-related results discussion.

7.1 Linear Equations Solver (SBS PCG)

The solver was implemented as a part of the CAD/CAE process as described in [8]. The SBS PCG solver divides the initial problem matrix into a set of sub-matrices that may be computed separately (see [6]). The computation algorithm first requires the initial matrices to be reversed, and then a central algorithm follows an iterative approach to find the final solution vector. The Octopus has been used to encapsulate the sub-matrix tasks into the agents, which implemented a simple local scheduling strategy - based on the Octopus's Information policy features available on each VCN - to distribute in the network. After distribution had been accomplished, the agents started their computation to invert the sub-matrices. Then, the Octopus's communication was used to communicate partial results to the agent's owner, which started iterations to get the final result. Iteration requires communication with sub-tasks. The Octopus message based communication schema was used to implement that. The Octopus's fundamental functionalities were

used for this application, including the agent's construction kit, migration, communication and information policies. The local scheduling implemented by SBS agents had enough information from the Octopus to successfully distribute agents while the communication was efficient enough not to slow down the application significantly.

7.2 Mesh Generator

The regular mesh generation (see [5]) was a second appliance in the field of CAD/CAE. The details of the application are presented in [10] with a detailed discussion of the results. The Mesh application followed the same design schema as the SBS PCG one; however, the agent internal design is more sophisticated, including a diffusion based local scheduling paradigm and the Smart Solid concept.

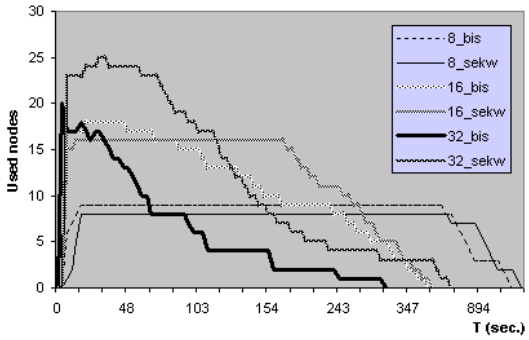

Figure 1. Dynamic of sample Mesh application runs.

The mesh agents were produced by the requester and migrated within the network to find a suitable environment. After local results were complete, the Octopus's communication was used to return them to the central application. The experiment showed that the Octopus puts small overhead on the application's total computational time. The mesh agents were distributed instantly with no delay coming from the platform itself.
Figure 1 shows the amount of actively computing agents over time for different runs of the Mesh application. Each function shows the amount of active agents in time for different input data set run - '8 bis' or '16 sekw' are example data set names. Please note that the agent's amount increases mostly at the beginning - this is because Mesh agents hold all information required to start computations. They migrate and instantly start computations. There are also no synchronization points implied over the application. Mesh tasks are independent, and so they were executed simultaneously by the Octopus - we observed small tasks being finished while others were still migrating in the network.

7.3 Hierarchic Genetic Strategy (HGS)

Genetic computations present far different requirements from the CAD/CAE ones. Mainly the problem's granularity level is much higher, thus raising the communication level tremendously. In our case, the HGS (see [7]) genetic algorithm was applied to optimize a function value in a given domain. The genetic populations may be evolved simultaneously, so they are a perfect candidate for agent based processing. However, the amount of populations prohibited us from using one agent per one population. We introduced a container concept, where a single agent contains multiple populations and evolves them. Such a container was wrapped with the Smart Solid agent implementing diffusion based local scheduling to get a fully functional, independent computing unit.

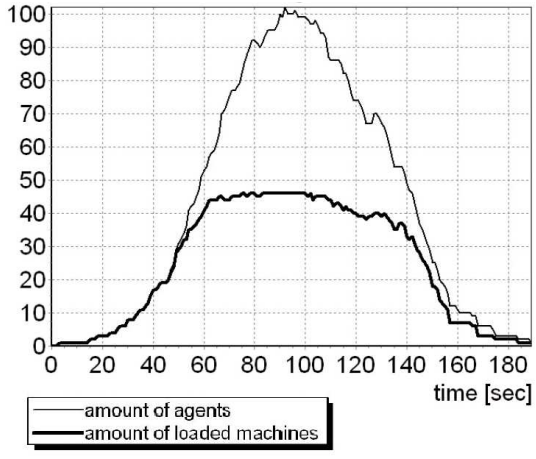

Figure 2. Dynamic of the HGS application run for a particular function.

The HGS application dynamic is also different from the SBS/Mesh ones. Here, the application starts with a single agent which evolves the internal computations until the container is filled up. When this happens, the agent's populations are split into two sets. One set is send back to the owner, and the second is further evaluated. When the owner receives the set, it creates a new

agent and puts it into the Octopus. This behavior is described in detail here [9] and its impact on the application's dynamic is illustrated by Figure 2. The amount of agents increases over time as they are created during computation. This effect is different from the Mesh application one, as observed in Figure 1.

In addition, we found that to speed-up the computation, we could introduce a cut-off factor for populations which do not seem to give any promising results. The goal was to achieve this without introducing any centralized communication requirement which would make the system unusable. The solution was based on a second type of agents which traversed the VCN network and communicated locally with agents on the node asking them to kill certain populations. This required the Octopus to introduce the agent's inter-operability. In conclusion, the following additional Octopus characteristics have been proved in this experiment:

– The ability to sustain large amounts of agents - up to 400 agents on 50 machines were used in the HGS experiments.

– Effective communication that performed well under these circumstances and showed no significant delays in agent execution.

In addition, the HGS Octopus implementation was compared with a low-level Round Robin scheduling strategy (see [9]). The agent based implementation running on the Octopus showed moderate performance losses with a small to medium agent amount (up to around 200), whereas with a bigger amount of agents, the HGS Octopus was faster.

8 Conclusions

We have shown a platform for creation and running multi-agent computing applications. The Octopus is a living project with a set of applications already implemented on it.

It is a stable runtime environment for independent agents. It delivers enough information to let agents integrate with the environment to utilize resources in a very efficient way. It is able to sustain various amounts of agents - starting from 1 up to 400 actively computing agents on 50 machines.

The Octopus does not impose significant requirements over applications; it allows for the implementation of different design schemas - varying from large agents armed with all required information, to small, robust ones using extensive communication to achieve their goals. Despite the presented examples, the Octopus is not limited to executing applications from such domains only. The current appliance results from our team research fields, but our solution may be used in a much broader set of domains, mainly thanks to its low complexity level and small amount of limitations imposed over an application's structure.

References

[1] IBM, http://aglets.sourceforge.net, IBM Aglets.

[2] Java Agent Development Framework (JADE), http://jade.tilab.com/

[3] The Foundation for Intelligent Physical Agents, http://www.fipa.org/

[4] Charron B., Delporte-Gallet C., Fauconnier H.: How to model a distributed computation. Institute Blaise Press, Paris, 1993

[5] Georg P.L.: Automatic Mesh Generation. John Wiley & Sons, 1991

[6] Golub G., Ortega J. M.: Scientific Computing. An Introduction with Parallel Computing. Academic Press, 1993.

[7] Schaefer R., Kołodziej J.: Genetic search reinforced by the population hierarchy. in *De Jong K. A., Poli R., Rowe J. E. eds. Foundations of Genetic Algorithms 7* Morgan Kaufman Publisher 2003, pp. 383-399.

[8] Schaefer R., Flasiński M., Toporkiewicz W.: Optimal Stochastic Scaling of CAE Parallel Computations. *Lecture Notes in Computer Intelligence*, Vol. 1424, Springer 1998, pp.557-564.

[9] Momot J., Kosacki K., Grochowski M., Uhruski P., Schaefer R.; Multi-Agent System for Irregular Parallel Genetic Computations. *Lecture Notes in Computer Science*, Vol. 3038, Proceedings of the ICCS 2004 conference, Springer 2004, pp. 623-630.

[10] Grochowski M., Schaefer R., Uhruski P.: Diffusion Based Scheduling in the Agent-Oriented Computing Systems. *Lecture Notes in Computer Science*, Vol. 3019, Proceedings of the PPAM 2003 conference, Springer 2004, pp. 97-104.