Using Coloured Petri Nets to Compare Mobile Agent Design Patterns

Emerson Ferreira de Araújo Lima\textsuperscript{1,2}, Jorge César Abrantes de Figueiredo\textsuperscript{3} and Dalton Dario Serey Guerrero\textsuperscript{4}

Grupo de Métodos Formais
Coordenação de Pós-Graduação em Informática
Departamento de Sistemas e Computação
Universidade Federal de Campina Grande
Av. Aprígio Veloso, 882, Caixa Postal: 10.106
CEP: 58.109-970, Campina Grande - Paraíba - Brasil

Abstract

Concurrent distributed systems modelling is a complex task. When we add mobility, the scenario is still worse, as specific problems are added. Petri nets is a largely used formalism to model and analyze complex systems, specially distributed and concurrent ones. Petri nets models are executable, allowing the simulation of abstract specifications. For this reason, Petri nets models are executable, allowing the simulation of abstract specifications. For this reason, Petri nets have been widely used as a validation and specification tool. This paper presents results of a performance comparative study of three mobile agents design patterns. The Itinerary, Star-Shaped and Branching migration patterns were investigated as solutions to a distributed information retrieval system. The three solutions were modelled with timed Coloured Petri Nets. The models were validated using simulation and occurrence graph analysis techniques. Experiments were conducted using further simulations and data were collected in order to evaluate the performance of each design pattern. The obtained data allow us to conclude that, for the studied scenarios, the Branching Pattern presents the best performance.

Keywords: Petri Net, Mobile Agents, Design Patterns.

\textsuperscript{1} This work and its authors are supported by CNPq – Brazilian Research Council, MOBILE Project, process 552190/2002-0. First author is supported by CAPES.
\textsuperscript{2} Email: emerson@dsc.ufcg.edu.br
\textsuperscript{3} Email: abrantes@dsc.ufcg.edu.br
\textsuperscript{4} Email: dalton@dsc.ufcg.edu.br
1 Introduction

In the context of distributed systems, mobility has been pointed out as a prominent and fundamental concept. It has been successfully used in complex Internet applications, because it can provide the necessary degree of configurability, scalability and customizability required by such applications [4]. However, mobility introduces non-existent problems in the traditional distributed computing: communication conditions variations, energy management and resources restriction. We can say that mobile computing is a hard scenario of distributed systems, where communication problems and disconnections are constants, and environment topology is dynamic.

Mobile agents is a paradigm based on code mobility, whose applicability has been shown in many case studies [2,5,11]. Mobile agents are autonomous software entities that can migrate to different physical locations and resume their execution at the point where they stopped before migration.

The use of mobile agents can bring several advantages to applications and their users: (1) reducing network traffic, as interactions can be carried out locally, independently of network latency; (2) asynchronous and decentralized execution, allowing the user to disconnect from the network when agents are performing a task; (3) ability to detect changes in the execution environment and react autonomously, simplifying the development of distributing systems that are more robust and fault tolerant. Despite the fact that these advantages can be obtained using other paradigms, mobile agents combine them all in a single and more abstract paradigm [8].

The paradigm presents some drawbacks. Firstly, it is necessary to install a support platform in each host the agents need to visit. Agents code is usually interpreted in such platforms. This can impact considerably on the performance of systems. Hence, it is recommended to avoid moving agents unnecessarily, since this can increase network traffic. Besides, agents code and data must be as short as possible if we want to achieve the benefits of the technology[15]. An agent functionality can be factored in a way it carries exactly what is needed. Information that is not going to be used or can be easily recovered can be discarded. Furthermore, there are security issues as well. However, this is a hard problem that is faced by all distributed systems technologies [3,19]. The more secure, the lower the performance of the system is.

System performance is an important factor, and when we deal with mobile agent-based distributed systems, the performance analysis is even more complex, since additional aspects should be considered as, for example, agent size and net capacity. Performance issues are relevant during the develop-
ment of systems and can influence, for instance, the way a given module is implemented. Therefore, in a mobile system, questions concerning the way how migration is defined are of utmost importance and should be addressed in early phases of system development. A wrong decision could affect the performance of the implemented system.

Petri nets [14] are a powerful formal, graphical, and executable tool that emerged in the beginning of the 60’s. Their major strength is the way in which the basic aspects of distributed systems are identified, both conceptually and mathematically. Petri nets have been used to model and analyze various types of concurrent systems. Many kinds of extensions have been proposed to support some requirements in specific applications areas. For example, Petri nets have been extended with time concept in order to cope with timed systems modelling and also to allow the performance analysis of systems.

There are some works that apply Petri nets to analyze mobile agents. In [16] and [17], the mobile agents design patterns Meeting and Master-Slave were modelled and validated with stochastic Petri nets. Merseguer [13] used stochastic well-formed coloured nets (SWNs) to model two information retrieval systems. One of the systems employs mobile agents and the other one, the client/server paradigm. From the comparative analysis, Merseguer concluded that, for low-bandwidth scenarios, the mobile agents solution presented better performance.

In this paper we present a comparative performance study of three solutions to a mobile agent-based distributed information retrieval system. We show how each solution behaves under different scenarios. Each one is an application of a mobile agent migration design pattern. We use Coloured Petri nets (CPN) [9] to model the three patterns. The Design/CPN tool [9] was used to edit, simulate and collect data from the CPN models.

The remainder of this paper is organized as follows: In Section 2, the distributed information retrieval system is shortly presented. Also the three mobile agents migration design patterns are introduced. The corresponding CPN models are explained in Section 3. In Section 4, we present the comparative study and the obtained results. Section 5 presents concluding remarks and pointers for further work. It is assumed that the reader has basic knowledge of CPN and mobile agents.

2 Mobile Agents Migration Design Patterns

Let us consider a distributed information retrieval system, in which an user requests the search for some information. The system receives the request and begins searching for the information that is distributed in many peers over the
net. Finishing the search, the system shows the results to the requesting user.

Search for the information is the critical point in the system, and is decisive to its performance. Focusing on the searching execution strategy, in this paper we use a mobile agents approach. In this approach, a mobile agent migrates to the data repository, here called agency, to locally execute its search and retrieval task. Nevertheless, this migration can be executed in several manners. For instance, an agent can travel sequentially through all the agencies; or, two agents can visit different agencies.

Not too many solutions to the mobile agent migration problem have been proposed [1,10,18]. These solutions are presented as design patterns. For the system presented here, we have that the problem is the migration, the context is distributed information retrieval, and the possible solutions are given by the mobile agents migration design patterns. The choice of one of the design patterns is not a trivial task. It is important to know the consequences of using them, considering the performance impact of their application.

In this paper we considered three migration design patterns proposed in [18]: Itinerary, Star-Shaped and Branching patterns. In the sequence we detail each pattern. We used a message sequence diagram to show an overall picture of the design patterns. The CPN models for these patterns are presented in the following section.

**Itinerary**

This pattern provides a way to execute the migration of an agent, which will be responsible for executing a given job in remote hosts. The agent receives an itinerary on the source agency, indicating the sequence of agencies it should visit. Once in an agency, the agent executes its job locally and then continues on its itinerary. After visiting the last agency, the agent returns to its source agency. This pattern is a good solution to agents that need to execute sequential jobs. In [6] and [12], case studies that apply this pattern are shown.

In Figure 1, we present a possible execution sequence for this pattern. We use a notation that is equivalent to the one presented in [11]. In this notation, an object is used to represent an entity that controls agents execution in a given agency (creation, destruction, migration) and indicates their location. Migrations are represented by message passing from one agency entity to the other. The message is labelled as \textit{MIGRATING AGENT}. Before migration, agent execution is interrupted (arrow labelled as \textit{destroy()}). Execution is continued in the target agency (arrow labelled as \textit{initialize()}).\footnote{This notation is used throughout this paper.}
As we can see, there are three agencies: a SourceAgency and two search agencies (DestinationAgency1 and DestinationAgency2). Following the diagram, we see that there is an agent (ItineraryAgent) that sets its itinerary, moves to the first search agency, where it executes its job, then it moves to the second one, executes the job, and then it returns to the source agency.

**Star-Shaped**

On the Star-Shaped pattern, the agent receives a list of agencies that it has to migrate to. Initially, the agent migrates to the first destination agency in the list. After migration is completed, it executes the relevant job and resumes migration going back to the source agency. The agent repeats this cycle until the last agency on its list is visited. The advantage of this pattern is that the agent stores the results of its job in the source agency and do not need to migrate to the others agency with them. Depending on the application, the results can be shown to the user as soon as the agent store them in the source agency. In this way, the user can already know the partial results before the agent finishes its migration through all search agencies.

In Figure 2, we can see an execution sequence for the Star-Shaped pattern. In this diagram, we have the same configuration of the sequence diagram showed for the Itinerary pattern: three agencies and one agent. Following
the diagram, we observe that the agent sets its itinerary, and then travels to the first search agency. After executing its job, the agent returns to the source agency, where it stores the job’s result. After that, the agent travels to the second search agency, executes its jobs, and returns to the source agency, storing the results obtained.

**Branching**

In the Branching pattern, the agent receives a list of agencies to visit and clones itself according to the numbers of agencies in the defined itinerary. Each clone is assigned an agency from the received list. Each clone has to migrate to its corresponding agency, execute its job and notify the source agency when the job is completed. The importance of this pattern is that it splits the tasks that can be executed in parallel. The treatment of the final results is an issue not covered by this pattern. For instance, the clones can put the result of the task in an user interface or send it to another agent.

In Figure 3, is shown an execution sequence for this pattern, in a scenario where there are three agencies and one agent. Following this figure, we can see
that the agent sets its itinerary, and then clones itself. After that, each agent (the original and the clone) migrates to a search agency, where they execute the job, and then return to the source agency.

Fig. 3. Sequence Diagram for the Branching Pattern

3 Design Patterns CPN Models

We have used timed hierarchical coloured Petri nets as the formalism to model the three migration design patterns presented in the last section. In Figure 4, we show the top most level model which is common for the three design patterns. The model consists of two places and three substitution transitions. Place $SA$ (Source Agency) is used for initialization purpose. Place $MA$ (Migrating Agent) centralizes the agents flow migration between agencies. The flow is controlled by the agencies as will be detailed later on. The three substitution transitions $SourceAgency$, $DestinationAgency$ and $Net$ represent, respectively, the agency from where the agent is sent to execute the search, the agencies where the search will be executed, and the net through which the agents are going to migrate.

The main difference in the modelling of the Itinerary, Star-Shaped and Branching patterns is in the model that details the substitution transition $SourceAgency$. Remember that, in the source agency occurs the definition of the agencies the agent should visit, and a different procedure is adopted in the
three design patterns. The detailed models for the SourceAgency substitution transition are presented latter, in the subsection of each pattern. The detailed models for substitution transitions Net and DestinationAgency are similar for the three patterns.

The modelling of the net is shown in Figure 5. Transition Transmit is associated with a time function. Whenever the timed transition Transmit fires, the model global time is increased according to the function migrateTime(agent). This function gets agent size (code + data) to calculate, according with the net capacity, the time delay of the migration. The token in the MA place is updated accordingly – agent itinerary and localization are updated (function updateAgency(agent)), and its state is changed from migrating to executing (function changeState(agent)). Notice that the agent is only going to be transmitted, i.e., the transition Transmit fires, if the agent state is migrating. This condition is guaranteed by the guard [not(isExecuting(agent))].

Whenever the agent arrives at the DestinationAgency (Figure 6), the agent is received by the transition Arrive and executes its jobs (transition DoJob, that retrieves one of the data of the place Data. This data is added to the agent by the function addData(agent, data). After that, agent migrates to the next agency of its itinerary (transition Migrate), and its state is changed from executing to migrating (function changeState(agent)). Notice that the agent is just received by the agency, i.e., the transition Arrive is just fired, if its state is executing and its current localization is not the SourceAgency, (guard [(isExecuting(agent)), not(isSource(agent))]).
3.1 Source Agency Model for the Itinerary Pattern

In Figure 7 we detail the model for the source agency, considering the itinerary pattern. Initially, the agent that will perform the search is represented by a token in place $SA$. The agent firstly sets its itinerary (transition $SetItinerary$), i.e., it receives the agencies list where it must execute the search. In this model the itinerary is stored in the place $Itinerary$, and the function
setItinerary(agent, itinerary) informs the itinerary to the agent. Then it migrates to the first agency in its itinerary (transition Migrate), and the function changeState(agent), changes its state to migrating. As soon as it finishes the migration through all the agencies in its list, the agent returns to the source agency (transition ReceiveAgent), the retrieved data is taken to the place Result (transition GetResult and function getData(agent)). Notice that the agent is only received by this agency, i.e., the transition ReceiveAgent fires, if the agent state is executing and its current localization is the SourceAgency (guard \( [(\text{isExecuting}(\text{agent})), \text{isSource}(\text{agent})] \)).

**Fig. 7. Source Agency Model for the Itinerary Pattern**

### 3.2 Source Agency Model for the Star-Shaped Pattern

The difference in this pattern is that, as shown in this pattern description (Section 2), the agent always returns to the source agency after visit each agency in its itinerary. In the SourceAgency model for the Star-Shaped pattern (Figure 8) transition ReceiveResult collects the retrieved data (function...
getData\((agent)\), and removes these data from the agent \((\text{resetData}(agent))\). This means that the agent does not need to migrate with the already stored data. This transition also changes the agent state to migrating \((\text{changeState}()\) function), and sends it to the next agency in its itinerary. Notice that the agent is only going to be received by this agency for the data collecting, i.e., the transition \(\text{ReceiveResult}\) fires, if the agent state is \(\text{executing}\) and its current localization is the \(\text{SourceAgency}\) (guard \([(\text{isExecuting}(agent)), (\text{isSource}(agent))]\)).

In order to finalize its migration, i.e., for the transition \(\text{ReceiveAgent}\) to fire, it is necessary that the agent state is \(\text{executing}\) and its itinerary is finalized (guard \([(\text{isExecuting}(agent)), (\text{isEnd}(agent))]\)).

### 3.3 Source Agency Model for the Branching Model

As explained in Section 2, in this pattern, the search agent clones itself according to the number of agencies in its itinerary. The cloning process, whose net is shown in Figure 9, is modelled by the timed transition \(\text{Clone}\), that increases the global time according to the \(\text{cloningDuration}\) variable.
value. The function \(removeFirstAgency(agent)\) removes the first agency of the itinerary, such that no other agent receive this agency as destination. On the other hand, function \(doClone(agent)\) creates a new agent, clone of the original search agent, but with the itinerary containing only the first agency of the original agent itinerary. Guard \([\text{itinerarySize(agent)}>1]\), of transition \(Clone\), guarantees that only the necessary clones will be created. The clone agents migrate through the transition \(Migrate\_Clones\), despite the original one only migrates (transition \(Migrate\)) when all needed clones are created (guard \([\text{itinerarySize(agent)}]=1]\).

### 3.4 Validation of the CPN Models

We have used both formal and informal methods to analyze the models. The informal analysis was accomplished by conducting simulations of the model.
Simulation is a very efficient way to debug a CPN model. We have mixed interactive and automatic simulations during the validation phase. Through simulations, it was possible to investigate details of the model. By choosing appropriate scenarios, it was possible to gain insight about the behaviour of the modelled patterns, enhancing our confidence that our model is correct. Various scenarios of itinerary, including different numbers of agencies to be visited by the agents were considered.

The formal analysis was carried out using the occurrence graph (OG graph) method. For each design pattern model, an occurrence graph was generated. Table 1 summarizes the main results of the OG analysis. Considering 5 agencies, it was possible to generate the full OG graph for all the three design patterns models. The Occurrence graphs for the Itinerary and the Star-Shaped models have 26 and 34 nodes, respectively. The OG graph for the branching model has 44,994 nodes. Considering the configuration with 15 and 45 agencies, it was not possible to generate the full OG graph for the branching design pattern model. We also used the standard report generated by the OG Graph tool to analyze some properties of the model. We focused on proving that the model consistently stops when all the results are returned to the source agency.

In Figure 10 we present part of the report generated by the Design/CPN tool, considering the branching model with 5 agencies to be visited. From the report, we can observe that there is only one home marking and one dead marking. This home marking corresponds to the final state when the results are sent back to the source agency. Since the final state is a home marking, it is guaranteed that the final state is always reachable. From the boundness property it is possible to observe that only 4 agents were cloned, as defined by the branching design pattern.
4 Migration Patterns Comparative Study

We have used the Design/CPN performance facilities to collect data and conduct a comparative study on the performance of the three design patterns modelled. The comparative study is an ongoing work, where we expect to detail more the models and use others patterns, and in this paper we present the initial results. Many configuration scenarios were defined, allowing set the involved parameters: agent size (code), collected data size, net capacity, number of agencies to agents go through, and cloning time. The results obtained are related with the time spent by each one of the approaches in order to complete the search in the agencies. In this paper we present the obtained results for the values shown in Table 2.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent Size</td>
<td>10Kb</td>
</tr>
<tr>
<td>Net Capacity</td>
<td>56Kbp/s</td>
</tr>
<tr>
<td>Cloning Duration</td>
<td>0.5s</td>
</tr>
<tr>
<td>Number of Search Agencies</td>
<td>5, 15 and 45</td>
</tr>
<tr>
<td>Data Size</td>
<td>0...75Kb</td>
</tr>
</tbody>
</table>

Table 2
Values of the parameters considered for the analysis.

In the comparison shown in this paper, we varied the number of agencies (itinerary size), and the collected data size. It is important to notice that the data size concerns to the data collected per agency. For example, if the data size is 5Kb, then, for three agencies, the total data size collected in the end
of search will be 15Kb. The data collection, needed to generate the graphics that helps comparison, was performed by means of automatic simulations, together with three collect functions: Predicate function, Observation function and Create function. The first function states when the data for comparison must be collected, the second one indicates what data must be collected, and the latter indicates where the collected data must be stored. In this study, the amount of data collected by the agent, and the time spent in order to complete the search were stored in a log file, whenever the agent completed the search (transition GetResult of the SourceAgency model from the patterns). Graphics were generated from the log file.

In the first situation, shown in the graphic of the Figure 11(a), we have a search including five agencies. In it, we can notice that when the collected data size is 0Kb (a remote job execution with no information retrieval, for instance), the Itinerary pattern presents the best performance, followed by the Star-Shaped pattern, and finally we have the Branching pattern with the worst performance. As the data size increases, the performance difference among the patterns decreases. With little bit less then 5Kb of data, Itinerary and Star-Shaped already present the same performance. When data size exceed 5Kb, Itinerary and Branching patterns present the same performance. Star-Shaped and Branching performance become equal when data size is close to 15Kb. Since then, the performance difference among the patterns increases considerably, having the Branching pattern with the best performance, followed by the Star-Shaped.

In the graphics of Figures 11(b) and 11(c), we have the same comparison with alteration of the number of agencies (15 and 45, respectively). We can notice that the number of search agencies in the agent’s itinerary is inversely proportional to the performance difference between the patterns when collected data size is 0Kb.

Other scenarios can be created and analyzed varying each one of the values shown in Table 2. We can, for instance, check the patterns behaviour for higher or lower net capacities. Of course, in a more accurate system analysis, many others factors must be considered. The net model is simple and is being detailed in order to consider reliability levels and net traffic.

With the analysis presented we could already perceive the influence of the parameters on the system performance. Also, we can note that there is no ideal solution to a given system, that is, the choice of the pattern to apply depends on many factors of the system. Considering the information retrieval system presented in Section 2, the retrieved searched data type has direct influence on the system performance, once the difference between textual and binary files, for instance, is considerably high. Another factor is the type of
Fig. 11. Time spent by the agent to finish the information retrieval in the Itinerary, Star-Shaped and Branching patterns, considering: 5 agencies (a), 15 agencies (b) and 45 agencies (c).
job that is going to be performed by the agent, because if it cannot be divided in order to be executed in parallel, the Branching pattern cannot be used. We could also consider the possibility of the search agents filters the data before retrieving it, what could improve the performance, mainly for the Itinerary pattern.

Although we cannot define the best pattern to apply in a given a kind of system, comparative studies are important to help the developer know which pattern behave better under specific conditions. This can help the pattern choice even before implementing the system.

5 Concluding Remarks

Distributed applications development is a complex work. When we add mobility feature the scenario is worse, due to the add of new problems, e.g., communication and connection conditions variation. In this sense, it is important to analyze properties of this kind of system before implementing it, verifying possible fails in its specification and looking for its improvement.

Petri nets are a very useful tool for modelling concurrent distributed systems, besides presenting many techniques for structural and behavioral analysis. Thus, we can use it in the modelling of mobile agents systems.

Besides the development problem, one must know which solution apply in given situations, looking for a better system performance. These solutions are given by the design patterns, and some of them are applicable in similar problems. In this manner, studies that compare the application of these patterns in many situations, presenting how each of them behave in these conditions, are important.

In this paper, we presented a comparative study of three mobile agents migration design patterns in the context of a distributed information retrieval system. With this study, we validated the Itinerary, Star-Shaped and Branching pattern, verifying how they behave under stated conditions, e.g., specific data size and net capacity. We argue that there is no better pattern to a kind of system, since the pattern performance depends on many issues present in the system scenario. From the graphics presented we can note, considering the parameters used, that the all three solutions are asymptotically linear in the size of data. We can also see that the Branching patterns has the better performance as the data size increases.

Many other comparisons can be performed considering others values involved in the system performance. Others patterns can be compared, e.g., communication patterns, once message interchange is also a relevant factor in the performance. Without the help of models and collect functions, it would
be hard to compare the patterns. Intuitively, we could say that Branching pattern has the best performance, however, for different values as cloning time and including the factor of net reliability, it gets hard to assess how this pattern performance would be.

With respect to the models, CPN was adequate to model most mobility concepts. However, the system decomposition style is not the ideal, for it demands a different vision of the modules by the developer in order to organize them as a hierarchy. An interesting approach is the use of object-oriented concepts to deal with decomposition, instead of hierarchies. In fact, this is an ongoing work in which we use an object-oriented Petri net formalism, called RPOO [7], to model and validate mobile agents patterns. This allows us to use the existing OO models as a base for the Petri net model. Furthermore, it provides for complete independence among modules.

References


