Supporting Relative Location Constraints in Actor Systems

Hua Zhang  
Xinghui Zhao  
School of Engineering and Computer Science  
Washington State University  
14204 NE Salmon Creek Ave.,  
Vancouver, WA 98686  
{hua.zhang, x.zhao}@wsu.edu

Nadeem Jamali  
Department of Computer Science,  
University of Saskatchewan  
176 Thorvaldson Building, 110 Science Place,  
Saskatoon, Canada, S7N 5C9  
jamali@agents.usask.ca

Abstract
The Actor model supports mobility of computations, and in actor migrations, absolute destinations must be explicitly specified. However, in many real-world applications, relative location relationships are more meaningful. In this paper, we propose a middleware which supports relative location constraints of actors. We use a case study to illustrate that our middleware greatly enhances the flexibility and programmability of actor systems. In addition, experimental results show that our middleware platform is scalable.

Categories and Subject Descriptors D.1.3 [software]: Programming Techniques—Distributed Programming; D.3.4 [software]: Processes—Run-time Environments

General Terms Languages, Experimentation, Performance

Keywords Actors, Constraints, Mobility

1. Introduction
The growing ubiquity of big data applications results in renewed interests in the mobility of computations. This is because moving required data to computations is no longer a viable solution, instead, computations themselves should be migrated to the location where the required data reside. The Actor model [1], which supports live migration of computations, provides a convenient way to program such mobile computations. However, the fact that an absolute destination location must be specified in an actor migration limits the flexibility of actor systems. In this paper, we address this challenge by developing a middleware which supports various types of relative location constraints for actors, and enforces those constraints at run-time.

2. Constraint Management Middleware
Separation of concerns has long been a widely used design principle in software engineering. It has been shown that spatial relationships of computations can be specified and maintained separately from the computations [4] for enhanced programmability. Inspired by this work, we have designed and implemented a middleware for AKKA [3] actor systems, as shown in Figure 1. The Constraint Management Middleware (CMM) consists of three components: a list of Constraints that are currently enforced, a Yellowpage which maintains the topology of the physical nodes/machines in the runtime system, and a CSP Solver, the key component which models the problem as a constraint satisfaction problem (CSP), solves it, and generates feasible deployment plans for actors.

As shown in Figure 1, CMM takes as input actor programs and user-defined constraints, which can be specified in either a static or a dynamic way. A static constraint specification is in the form of a separate constraint configuration file, which lists actor names and their relative location constraints. This form is suitable for specifying predictable, fixed constraints. For instance, a computation actor should be collocated with the data that it requires. In contrast, a dynamic constraint specification can be combined in the actor program, so that constraints can be dynamically generated at runtime. This form is particularly useful when the location constraints evolve during the course of the computation, i.e., location constraints need to be programed. For example, a data aggregation actor should initially be placed at a location close to the actors from which it collects data; however, after the data has been collected, it is free to move elsewhere for other purposes. Table 1 shows a list of possible relative location constraints for actors.

<table>
<thead>
<tr>
<th>Constraints</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>separate</td>
<td>$a_1, a_2$</td>
<td>$a_1 \neq a_2$</td>
</tr>
<tr>
<td>collocate</td>
<td>$a_1, a_2$</td>
<td>$a_1 \neq a_2$</td>
</tr>
<tr>
<td>within</td>
<td>$a_1, a_2, d$</td>
<td>$\text{distance} \leq d$</td>
</tr>
<tr>
<td>outof</td>
<td>$a_1, a_2, d$</td>
<td>$\text{distance} &gt; d$</td>
</tr>
<tr>
<td>isolate</td>
<td>$a_1$</td>
<td>$a_1$ is isolated from other actors</td>
</tr>
</tbody>
</table>

The CSP Solver uses a customized backtracking-search algorithm [5] to solve the constraints, as shown in Algorithm 1. Note that heuristic approaches can be used in the algorithm to choose...
potential locations for an actor. In addition, the complexity of the algorithm can be reduced by deriving inferences from certain types of constraints, e.g., collocate, as suggested in [4].

Algorithm 1: CSP_Solver(constraints, yellowpage, map)

if all actors have been deployed then
    return map;
else
    get the next actor a;
    for each node in possible locations(a, map, constraints) do
        if place a to node is consistent with map then
            place actor a to node;
            derive the inferences of the deployment;
            if inferences do not lead to failure then
                add inferences to map;
                result = CSP_Solver(constraints, yellowpage, map);
                if result != failure then
                    return result;
            end
        end
    end
    remove deployment of a and its inferences from map
end
return failure

Once a valid solution map is generated by the CSP Solver, CMM informs AKKA runtime system to deploy actors based on the solution. CMM is extendable in terms of constraints. In other words, users can easily accommodate more user-defined constraints by modifying the consistency checking method in CSP Solver.

3. Experimental Results

Experiments have been carried out to evaluate the effectiveness and scalability of our approach.

3.1 A Case Study: Word Count

In this case study, we use AKKA actor system and CMM to implement word count, a Map-Reduce application. The application counts the number of occurrences of each word in a file. We implement the application as follows. A dataActor is responsible for maintaining the files, and sending the files to other actors which require them. The actual computation is carried out by a number of mapActors and reduceActors, as would be done in Map-Reduce [2].

We first run the computation in the original AKKA platform with 3 physical nodes. In this case actors are randomly distributed on 3 nodes. We then run the same computation on CMM, with one location constraint which is collocate(mapActor, dataActor). In this way, the computations are collocated with the data that they require. Figure 2 shows the results of the experiments with different file sizes. When the file size increases, we can achieve an order of magnitude of performance gain using CMM, with no added programming complexity. Only a static constraint is needed.

3.2 Scalability Analysis

The case study only uses one single constraint. To evaluate the scalability of CMM, we have carried out another set of experiments. In these experiments, we randomly generate a number of constraints, and use CMM to generate and enforce a deployment plan. We then measure the performance of CMM, and the results are shown in Figure 3, which demonstrates that CMM scales well when the number of constraints increases.

4. Conclusion and Future Work

The Actor model supports mobility of computations; however, its implementations often require users to explicitly specify the destination of actor migration, which unnecessarily limits the flexibility of actor systems. In this paper, we present a Constraint Management Middleware (CMM), which enables users to program relative location relationships among actors, and enforces these location constraints at runtime. CMM on AKKA actor systems shows good scalability, as well as potentials of enhancing programmability and performance of actor systems. Work is ongoing to extend CMM by accommodating other types of constraints, such as real-time constraints. In addition, we will use the extended CMM to solve problems in mobile clouds and other environments.

References