Resource-Driven Collaborative Component Deployment in Mobile Environments

Peter Rigole Yolande Berbers
Department of Computer Science, K.U.Leuven
Celestijnenlaan 200A,
3001 Leuven, Belgium
{Peter.Rigole,Yolande.Berbers}@cs.kuleuven.be

Abstract

Two techniques are presented in this paper that address the issue of hosting applications in a ubiquitous computing space: (1) context-aware component deployment and (2) self-adaptive component deployment configurations. The former focuses on deploying an application, which is composed of fine-grained components, in a mobile distributed computing environment based on resource information that is spread through context-data. The latter focuses on letting the component deployment configuration adapt itself when the resources in the environment change due to device mobility. Both techniques are solved using a single mechanism that is based on an extended version of collaborative reinforcement learning (CRL) in the design of our component deployment framework. In order to find local optimal solutions for deploying a component-based application in a distributed resource constrained environment, the CRL method balances the resource consumption of components on several nodes that are in each other’s proximity.

1 Introduction

In today’s distributed computing research, novel paradigms anticipate a continuous evolution of computing towards the idea of ambient intelligence: a surrounding ubiquitous computing space that is spontaneously and proactively supporting users. A main characteristic of this ubiquitous computing space is that its hardware infrastructure is mainly composed of resource-constrained mobile devices that are able to communicate and cooperate when located in each other’s vicinity. This ad hoc composed computing network is often backed up by more powerful computing devices that are integrated into local facilities and transport vehicles that enclose people. However, in order to deploy applications on these devices without distracting the user, a self-organizing approach is desired.

Our approach focuses on balancing the resource consumption of components on several nodes that are in each other’s proximity. At this moment, we pay special attention to memory and bandwidth consumptions as we believe these are the most vulnerable resources in mobile computing environments. Adaptations in the component deployment configuration to meet the resource requirements of applications are essential since the computing environment changes often due to the mobile nature of its composing devices. Our DiCOMPLY solution harbors a distributed learning mechanisms that support the component deployment mechanism in order to find local optimal solutions.

This paper is further organized as follows. After an introduction, the DRACO component model is briefly discussed, followed by a survey on decentralized self-adaptive collaborative reinforcement learning using a cost function in multiple dimensions. Next, we explain how we applied the learning strategy to design a self-adaptive distributed component deployment mechanism. Finally we conclude with experimental results and general conclusions on our approach.

2 The DRACO Component Model

The DRACO component model [6, 7] is developed to support component-based applications that are deployed in a distributed environment. Components are lightweight and behave as black-boxes, i.e. they only communicate through their ports and have no shared data with other components. Ports are explicit bidirectional communication gateways that can receive and send out messages. These messages are data containers that are identified by their type. Each message type has an associated method at the receiving end that processes the message when it arrives at the component port. The communication semantics allow asynchronous communication only. Ports can communicate when they are connected with a connector. Figure 1 represents a simple composition of two components that have ports that are connected with a connector.

The choice to work with asynchronous communication...
semantics only is driven by our need to deploy the components in a distributed computing environment. The semantics reduce the communication overhead that is typically caused by synchronous remote communication. The black box nature of a component and the explicit use of connectors reduces the dependencies between components in a composition. It allows a component framework to distribute components at runtime over a set of interconnected hosts. Mobility is supported as well so that changes in the deployment configuration can be realized easily without much interference with the performance of the application. State extraction from a component is implemented by the component designer and is performed automatically by the component framework when a component is relocated from one host to the other.

3 Multidimensional Collaborative Reinforcement Learning

Reinforcement learning is an unsupervised learning method that derives a way of behaving from interaction with an uncertain environment. As a consequence of interaction, the environment returns rewards that feed the reinforcement learning strategy [1, 2]. These reinforcements are used to learn to maximize the total reward received over a time horizon by selecting optimal actions. Actions that give a poor immediate payoff may be taken in the anticipation of a higher return in the longer term. Action selection in a RL strategy is characterized by certain degree of probabilism, dependent on the trade-off between exploration and exploitation. Exploration means that actions are preferred that are deemed to result in an optimal behaviour, while a exploration ensures that the learned experience covers all actions, even those that are not yet explored or those that were considered to give a bad pay-off in the past. This por-

tion of trial-and-error is a powerful way to deal with dynamic environments that are subject to frequent changes in their behavior. On the down-side however, this makes RL inappropriate for systems that are intolerant to suboptimal action-selection, such as real-time systems.

Reinforcement-learning problems are usually modeled as Markov decision processes (MDPs) [3]. An MDP consists of a set of states, \( S = \{s_1, s_2, \ldots, s_N\} \), a set of actions, \( A = \{a_1, a_2, \ldots, a_N\} \), a reinforcement function \( R : S \times A \rightarrow \mathbb{R} \), and a state transition distribution function: \( P : S \times A \rightarrow \prod(S) \), where \( \prod(S) \) is the set of probability distributions over the set \( S \).

3.1 Decomposing Optimization Problems

Collaborative reinforcement learning (CRL), presented in [2], is a technique that enables collaborating nodes to solve system optimization problems in dynamic decentralized networks. In order to do so, system optimization problems are decomposed into a set of discrete optimization problems (DOPs) (see [2]). One network node initiates the solution to a DOP, which terminates at some node in the network. Each of the collaborating nodes decides on which action to take to solve the DOP. The actions among which a node can choose are grouped into DOP actions, delegation actions and discovery actions. DOP actions try to solve the DOP locally. They are elements of the set \( A_{dp} \). Delegation actions, elements of the set \( A_d \), delegate the solution of the DOP to a neighbor. Where DOP actions and delegation actions are associated with specific states, discovery actions can be taken at any time from any state to attempt to find new neighbors. Obviously, discovery actions can not trigger state transitions.

3.2 Collaboration with Neighboring Nodes

Delegation actions are introduced in CRL to hand over the solution of the DOP to a neighboring node. In order to establish a relationship between two neighboring nodes that may delegate the solution of DOPs towards one another, the concept of causally connected states is used. There are two categories of states in CRL. They are either internal or external. An external state is a state that may be shared by neighboring nodes to delegate the solution of DOPs to. Two causally connected states consist of an external state mapped on an internal state of a neighboring node, see the example in Figure 2(b). This mapping represents a contractual agreement between neighboring nodes to support the delegation of DOPs from one to the other. An internal state on one node can be causally connected to external states on many different neighboring nodes.

For every neighbor node, \( n_j \), that is causally connected
to node \( n_i \), there exists a delegation action \( a_j \in A_j \) that represents an attempt by \( n_i \) to delegate a DOP to \( n_j \). On a successful delegation, a state transition is made to \( n_i \)'s causally connected state, which terminates the MDP at node \( n_i \). The DOP is then further handled by \( n_j \) by initiating a new MDP. The success rate of a delegation is dependent on many factors, such as the capabilities of each neighboring node, the available resources on each node and the quality of the network connections. This is precisely what a node has to learn from the reinforcements while interacting with its neighbors. Although the states are fixed for each node in CRL, the number of causally connected neighboring nodes may evolve over time, and thus also the set of delegation actions \( A_i \).

3.3 Multidimensional Cost Functions

In CRL, network connection costs are taken into consideration when the cost of a state transition to a neighboring node is estimated. Therefore, CRL includes a connection cost, \( D_i(s'|s, a) \in \mathbb{R} \) on top of the optimal value function, \( V_j(s') \), for the next state at node \( n_j \). \( D_i(s'|s, a) \) is only nonzero if \( a \in A_d \) in CRL.

mCRL introduces a cost function in more than one dimension by including other resource costs as well. From this, it follows that the connection cost, the \( V \) values and the \( Q \) values are now vectors with values corresponding to the different dimensions of the problem.

The general updating algorithm of the reinforcement learning algorithm used by mCRL becomes

\[
\bar{Q}_i(B, a_j) = \sum_{s' \in D, R, L, F} P_i(s'|B, a_j) (\bar{D}_i(s'|B, a_j) + \text{Decay}(\bar{V}_j(s'))) 
\]

which is very similar to the updating algorithm used in [2].

4 Distributed Component Deployment using MCRL

The distributed deployment mechanism presented here takes a decentralized approach using MCRLand is designed to deploy components that adhere to the DRACO component methodology. Each node that is offered the task of deploying a component composition uses information about itself and its neighboring nodes only in order to make its deployment decision. This way, only a small amount of information is considered during the deployment process which enables each node to make fast deployment decisions that still strive towards a solution that approaches a global optimal deployment.

4.1 Decentralized Distributed Component Deployment

The states and state transitions of the MDP representing this deployment mechanism is depicted in Figure 2(a). For each component in the composition, the node decides to either deploy the component locally or to delegate the deployment of the component to one of its neighboring nodes. Both decisions may result in either a deployment or a failed deployment of the components. If no deployment candidate for the component can be found, the node can decide to fail on the deployment of the component. The latter choice has a small probability in the MDP, though this probability will increase after taking successive failing deployment choices.

Figure 2(b) shows a node that has two neighboring nodes that have states that are causally connected with node A's remote deployment states. For each neighboring node, there is a delegation action available in node A's MDP that lead to an attempt to deploy components on the neighboring node.

4.2 DiCOMPLOY Characteristics

The distributed component deployment algorithm we designed using MCRL, called DiCOMPLOY, incorporates two cost functions. One cost function to express the cost of memory consumption on a local or remote node and one cost function to express the cost of bandwidth consumption caused by connectors that are linked over network connections. When evaluating the cost of a specific deployment of component \( c \), the resource vector of the component, represented by \( \tau \), is used to define the \( Q \) value as follows

\[
\bar{Q}_i(B, a_j) \cdot \tau = \sum_{s' \in D, R, L, F} P_i(s'|B, a_j) (\bar{D}_i(s'|B, a_j) + \text{Decay}(\bar{V}_j(s'))) 
\]

Notice that \( \bar{D}_i(s'|B, a_j) \cdot \tau \) has been replaced by \( D_i(s'|B, a_j, c) \), because the latter expression incorporates knowledge about the composition component \( c \) is part of. The bandwidth consumption of the connectors attached to the component’s ports may be reduced by deploying a component remotely if the neighboring component is already deployed on the remote host. Taking this into account results in better decisions because the cost is estimated more accurately.

The probability function \( P_i(s'|B, a_j) \) is constantly updated based on the ratio between the number of decisions that resulted in a failed deployment and the number of successful deployment decisions when taking action \( a_j \) from within state \( B \). The cost estimations used in \( \bar{D}_i \) take into account the ratio between the available amount and the total amount of each resource dimension.
5 Experimental Results

We compare DiCOMPLOY experimentally with centralized constraint based optimization, using traditional constraint solvers. Work in this direction has been done by [4, 5]. Centralized algorithms like these need all information in one node, which then seeks the optimal solution. Decentralized mechanisms such as DiCOMPLOY on the other hand solve subproblems based on information the obtain from their immediate neighbors only.

We conducted experiments in a simulation environment with three different network and composition configurations. In order to refer to the experiments, we call them the simple, advanced and complex experiment configuration respectively. The simple experiment consists of three components interconnected with two connectors and a network configuration with two hosts interconnected through one connection. The advanced experiment uses the network configuration depicted in Figure 3(a) and the component composition shown in the top half of Figure 3(b). The complex configuration uses the same network configuration as the advanced experiment and uses the component composition depicted in the bottom half of Figure 3(b).

The time needed to decide on the deployment of the component composition on the network nodes was the criterion of comparison between the two mechanisms. The results of the three deployment cases are depicted in Table 1. It shows the average time needed to find a valid deployment. The average was calculated over 100 experiments with each of the test cases. We learn from the experiment that DiCOMPLOY is much faster than the centralized algorithm, especially when the problem becomes more complex. From [4], we know that centralized constraint solving mechanism has an exponential complexity in function of the number of nodes and the number of components involved in the deployment problem. The time complexity of DiCOMPLOY on the other hand is linear in function of the number of components and nodes.

Table 1. Average deployment calculation times over 100 experiments, expressed in milliseconds.

<table>
<thead>
<tr>
<th></th>
<th>centralized algorithm</th>
<th>DiCOMPLOY</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>90</td>
<td>10</td>
</tr>
<tr>
<td>advanced</td>
<td>144</td>
<td>56</td>
</tr>
<tr>
<td>complex</td>
<td>604</td>
<td>69</td>
</tr>
</tbody>
</table>

Obviously, there are drawbacks in the DiCOMPLOY approach. First, the solution is not optimal and may be influenced by the order in which the components are deployed. In some cases, DiCOMPLOY cannot find a solution at all, where the centralized algorithm always finds a solutions if there is one, when given enough time. This phenomenon occurred a few times while solving the complex test case mentioned above. Second, the solution with DiCOMPLOY is subject to probabilities that inherently influence the solution. Sometimes, the algorithm can give up on seeking a solution candidate too soon.

Advantages of DiCOMPLOY, apart from its speed, over the centralized approach are plenty. First its decentralized nature does not require all information to be collected on one host and calculation is spread over several nodes in-

(a) The states and state transitions of the distributed deployment MDP.

(b) Node A with two neighboring nodes that have causally connected states. Node A can delegate the solution of an optimization problem either to Node B or to Node C.
instead of a single node. Second, changes in the deployment can be performed in an incremental way when the characteristics of the environment changes, which is interesting in mobile computing environments. The centralized approach would have to solve the initial deployment problem all over again when the environmental conditions (especially resource availability) change.

6 Conclusions

MCRL results in a much faster decision process than trying to look for an exact deployment solution, although we not necessarily obtain the optimal solution. Luckily this is considered acceptable in dynamic ubiquitous computing environments. Moreover, changes in the computing environment would often imply excessive adaptations in the deployment of the application if an optimal deployment would be the only goal. The impact this would have on deployment overhead justifies the choice for applying local deployment changes only, which are quickly decided by the MCRL algorithm and non-invasive because of the incremental nature of the adaptation it involves. Finding the optimal redeployment on the contrary, involves solving the global problem all over again, which is time-consuming and suffers from an exponential time complexity.

References