

# Component Placement in Distributed Environment w.r.t. Component Interaction<sup>\*</sup>

Barbora Zimmerova

Faculty of Informatics, Masaryk University  
Brno, Czech Republic  
zimmerova@fi.muni.cz

**Abstract.** The paper identifies a lack in existing solutions to the component placement problem (optimal placement of system components on a given set of servers). The point is the insufficient attention of the interaction aspect of the problem (components with dense interaction should be placed on the same server). In the text, we formulate the problem of interaction-based component placement (as minimization of inter-server communication) and propose a solution to the interaction aspect of the CPP while allowing integration with existing algorithms for solving the non-interaction aspect of CPP (resource optimization). The algorithm is based on the Component-interaction automata language, which allows us to analyse the density of communication between two components, thanks to the information about communicating components in labels.

## 1 Introduction

The component-based paradigm of software development enables developers to construct large-scale systems, which can be assembled of heterogeneous components (with diverse functionality, programming language, platform), and distributed across network. The task of component distribution among available computational nodes (servers), called Component Placement Problem (CPP), is often realized manually based on experience of the developer. Research in the last decade led to investigation of the automatic resolving of the CPP as a *resource allocation* type of problem, with respect to resource capacity of particular servers, system constraints, and resource requirements of components.

The existing approaches [1–3] provide elaborated theories and practical results, but they do not cover the whole CPP because their techniques do not take into consideration the density of communication among components (components with dense interaction should be placed on the same server). Note that the information about density of inter-component communication is not provided by the bindings between component interfaces, because the bindings do not contain the information about the frequency of their use for communication.

In this paper, we complement the approaches by proposing a solution to the interaction aspect of the CPP. We first identify a specification formalism for

---

<sup>\*</sup> The research has been financially supported by the Czech Science Foundation within the project No. 102/05/H050, and the grant No. 1ET400300504.

capturing the inter-component communication in the system, which allows us to analyse the density of communication between two components. Then, we define the problem of interaction-based component placement, and propose a solution which integrates existing solutions to the non-interaction aspect of the CPP with our solution to the interaction aspect of the CPP. Therefore our technique provides a general solution to the CPP which concerns both interaction and non-interaction properties of the system.

The remainder of the text is organized as follows. In Section 2, we outline existing lines of research related to our work and discuss how they compare to our technique. In Section 3, a language for component-interaction specification is briefly described including basic definitions. Section 4 provides the statement of the interaction aspect of the CPP, and Section 5 proposes a solution to the CPP with respect to both interaction and non-interaction properties of the problem. In Section 6, we summarize the results and outline the directions for further research.

## 2 Related work

A lot of work has been already done in the area of component placement in distributed environment. This section outlines the approaches that are close to our work, and discusses how our work relates to these.

Several existing studies (by Stewart et al. [1], Kichkaylo et al. [3], Karve et al. [2]) have shown that the CPP can be defined and resolved via resource consumption optimization on the servers which enables the maximal system throughput. Stewart et al. [1] design component placement on the basis of component resource consumption profiles. These are given as functions mapping workload characteristic (average request arrival and request types) to resource consumption (CPU, memory, network bandwidth). Given available system resources and runtime characteristics, the profiles can be employed to resolve the CPP and achieve the maximal system throughput. Kichkaylo et al. [3] use component placement properties for this purpose. The properties are given in an XML (eXtensible Markup Language) description, which includes information about resource consumption, environment constraints, and component interfaces (trustworthiness). Given the placement properties, the placement problem is reduced to the planning problem addressed by the Artificial Intelligence community, which has several existing solutions. Karve et al. [2] address dynamic placement of parts of web applications while distinguishing load-dependent (CPU) and load-independent (memory) resource requirements of system parts. The objective is again to maximize the amount of resource demand that may be satisfied using the placement, and keep resource utilization balanced across all server machines. All the approaches focus on optimization of resource consumption, with no respect to interaction among components. Our work aims to supplement these solutions in offering a method for resolving the interaction aspect of component placement. In Section 5 we propose a solution that integrates our work with the work discussed here.

Another line of research (Coign [4], ABACUS [5]) related to our work investigates monitoring-based inter-component communication optimization. Coign [4] resolves component distribution of COM (Component Object Model) applications. Given an application in a binary form, Coign simulates application execution on profiling scenarios. During the execution, all service calls going through COM interfaces are logged with the amount of data transferred. It allows Coign to construct a graph model of component communication (components in vertices, amount of data on edges) and to apply a graph-cutting algorithm to partition the application on given computational nodes to minimize network use. A similar approach is used by dynamic function placement of ABACUS [5]. ABACUS monitors data transfers among application functions at run-time and proposes a placement strategy that decides when to relocate functions to more optimal locations based on logged values. The monitoring-based approaches have certainly high practical importance. However, they base the solution on a limited sets of system-execution scenarios received from monitoring. We complement these solutions by providing exhaustive analysis of all possible system’s interaction behaviours.

For the component-interaction specification, we use the Component-Interaction automata language [10, 11]. There is a variety of other languages that could be used for this purpose. It includes I/O automata based languages (Interface automata [6], Team automata [7]) and behaviour description sub-languages of architecture description languages (Tracta [8], SOFA Behavior protocols [9]). To the best of our knowledge, none of these languages has ever been used to address the CPP. Our choice of Component-Interaction automata for this task was motivated by the fact, that the language naturally captures information about communicating components (in structured labels) and preserves this information through the composition as well, which allows us to estimate the density of inter-component communication from the system’s model.

### 3 Component interaction specification

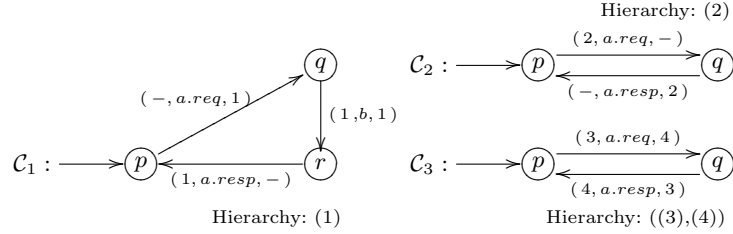
This section presents a notation used for capturing interaction specification of the system. We have chosen a *Component-interaction automata* language [10, 11] for this purpose. The Component-interaction automata language captures each component as a labelled transition system with structured labels (to remember components which communicated on an action) and a hierarchy of component names (which represents the architectural structure of the component). The essential definitions are briefly reminded in this section.

A *hierarchy of component names* is a tuple  $H = (H_1, \dots, H_n)$ ,  $n \in \mathbb{N}$ , of one of the following forms,  $S_H$  denotes the set of component names corresponding to  $H$ . The first case is that  $H_1, \dots, H_n$  are pairwise different natural numbers; then  $S_H = \bigcup_{i=1}^n \{H_i\}$ . The second case is that  $H_1, \dots, H_n$  are hierarchies of component names where  $S_{H_1}, \dots, S_{H_n}$  are pairwise disjoint; then  $S_H = \bigcup_{i=1}^n S_{H_i}$ .

A *component-interaction automaton* (or a *CI automaton* for short) is a 5-tuple  $\mathcal{C} = (Q, Act, \delta, I, H)$  where  $Q$  is a finite set of states,  $Act$  is a finite set of

actions,  $\Sigma = ((S_H \cup \{-\}) \times Act \times (S_H \cup \{-\})) \setminus (\{-\} \times Act \times \{-\})$  is a set of labels,  $\delta \subseteq Q \times \Sigma \times Q$  is a finite set of *labelled transitions*,  $I \subseteq Q$  is a nonempty set of *initial states*, and  $H$  is a hierarchy of component names.

The labels have semantics of input, output, or internal, based on their structure. In the triple, the middle item represents an action name, the first item represents a name of the component that outputs the action, and the third item represents a name of the component that inputs the action. Examples of three CI automata are in Figure 1. Here for example,  $(-, a.req, 1)$  in  $\mathcal{C}_1$  signifies that a component with a numerical name 1 inputs an action  $a.req$  (a request for a service  $a$ );  $(1, a.resp, -)$  in  $\mathcal{C}_1$  signifies that a component 1 outputs an action  $a.resp$  (a response for a service  $a$ );  $(4, a.resp, 3)$  in  $\mathcal{C}_3$  represents an internal communication between components 4 (sender) and 3 (receiver) on  $a.resp$ .



**Fig. 1.** Example of CI automata

Component-interaction automata offer a parameterizable composition operator, which preserves the information about communicating components during composition. We briefly outline its definition here. Given a parameter  $T$ , the operator composes a given set of CI automata into a product automaton (consisting of transitions respecting one-to-one synchronization<sup>1</sup>) and removes all transitions from the product that do not agree with  $T$ . An example of a complete composite automaton (with removal of an empty set of transitions) for the automata  $\mathcal{C}_1, \mathcal{C}_2$  (depicted in Figure 1) is in Figure 2.

## 4 Problem statement

In our view, the Component Placement Problem (CPP) consists of two aspects: an interaction and a non-interaction aspect. The *non-interaction aspect* is referred to by existing definitions of the CPP [1–3] which concern resource and other system constraints, but overlook interaction among components.

<sup>1</sup> Components cooperate either by interleaving of their original transitions, or simultaneous execution of two complementary transitions (with labels  $(n_1, a, -)$ ,  $(-, a, n_2)$ ) which results into a new internal transition (with label  $(n_1, a, n_2)$ ).



the application (system). Without loss of generality, suppose that the transition graph of  $\mathcal{A}$  does not contain unreachable states.

Note that the system can be either open or closed. A *closed system* is closed to an environment (its execution consists of internal actions only). Distribution of such systems is intuitively clear (all communicating parties are known). An *open system* is open for interaction with an environment (its execution contains external, i.e. input and output, actions). For distribution of open systems, we will suppose that the environment is placed outside the set of servers  $\{\mathcal{N}_i\}_{i \in \{1, 2, \dots, n\}}$ , and that there is no restriction on actions it provides or consumes. If we knew the environment's behaviour, we could capture it as a CI automaton, compose it with  $\mathcal{A}$  and study the result as a closed system.

## 2. Compute a set of candidate component placements

In this step, we determine a set of feasible component placements on given servers. That is a subset of the initial set of all placements (which is finite) that fulfills given constraints. For example, a set of feasible component placements can be limited by capacity of the servers and size of the components. The *capacity of a server* can be defined as an amount of provided resources (CPU, memory, etc.), and the *size of a component* as an amount of required resources. The size of each component can be given explicitly, or can be computed with respect to the component's interaction behaviour in an overall system (different actions – used with different frequency – may have different resource requirements).

The main intent of this step is to enable integration of existing solutions to the non-interaction aspect of the CPP [1, 2], which focus on resource consumption optimization and overlook interaction among components. Roughly, the approaches use various kinds of cost functions that assign a specific cost to each (feasible) component placement and then search for the optimal (minimal-cost) solution. This allows us to use the cost functions to select *a set of relatively cheap* component placements (with the cost close to the minimal value by some distance  $d$ ), and work with the set as a set of candidate placements in the next step. Hence both resource and interaction efficiency is considered in a portion determined by the distance  $d$ .

## 3. Evaluate each of the candidate placements

Suppose there is a cost function  $f$  that, given a CI automaton  $\mathcal{C}$  and a label  $l$  returns a value  $f(\mathcal{C}, l)$  characterizing the frequency of the label's occurrence in execution of  $\mathcal{C}$ . Various functions, with different fineness, could be used for this purpose. The basic is  $f(\mathcal{C}, l) = t_l/t$  where  $t_l$  is a number of transitions labelled with  $l$  in  $\mathcal{C}$ , and  $t$  is a number of all transitions in  $\mathcal{C}$ . A more accurate one can be based on the occurrence of the label in paths of execution of  $\mathcal{C}$  as the position of the label in the transition graph influences the frequency of its occurrence during system execution. We do not restrict ourselves to any of possible cost functions. The choice can be based on character of the system and preferences of a user.

Now we define the frequency of inter-component communication  $f(\mathcal{C}, n_1, n_2)$  in the following way. Let  $\mathcal{C} = (Q, Act, \delta, I, H)$  be a CI automaton and  $n_1, n_2 \in \mathbb{N}$ ,

be numerical names of components in  $\mathcal{C}$ . Then  $f(\mathcal{C}, n_1, n_2) = \sum_{l \in L} f(\mathcal{C}, l)$  where  $L = \{(n_1, a, n_2) \mid a \in Act\} \cup \{(n_2, a, n_1) \mid a \in Act\}$ .

Back to the concrete problem. Let  $\mathcal{A} = (Q, Act, \delta, I, H)$  be the CI automaton of the application, which represents interaction among given components  $\{C_i\}_{i \in \{1, 2, \dots, m\}}$ . For each component placement  $p : \{C_i\}_{i \in \{1, 2, \dots, m\}} \rightarrow \{\mathcal{N}_i\}_{i \in \{1, 2, \dots, n\}}$ , we create a new CI automaton  $\mathcal{A}_p$  representing interaction among servers  $\{\mathcal{N}_i\}_{i \in \{1, 2, \dots, n\}}$  as follows. To each server, we assign a unique name from  $\mathbb{N}$  (denote  $S$  the set of server names). Then  $\mathcal{A}_p = (Q, Act, \delta_p, I, H_p)$  where  $\delta_p$  and  $H_p$  result from  $\delta$ , resp.  $H$ , by replacing every occurrence of any component name with the numerical name of the server it is placed on (by  $p$ ).

Now  $f(\mathcal{A}_p, n_1, n_2)$  represents the frequency of communication between servers  $n_1, n_2$ . Hence the cost of component placement  $Cost(\mathcal{A}_p)$  is given as the sum of all inter-server communication  $Cost(\mathcal{A}_p) = \sum_{n_1, n_2 \in S, n_1 < n_2} f(\mathcal{A}_p, n_1, n_2)$ .

Note that  $Cost(\mathcal{A}_p)$  includes only the cost of inter-server communication within  $\{\mathcal{N}_i\}_{i \in \{1, 2, \dots, n\}}$ . It includes neither the cost of inner-server communication (as  $n_1 \neq n_2$ ), nor the cost of communication with system's environment<sup>2</sup> in case the system is open. It exactly follows the assumption that the environment is not placed on any of the servers. In such case, the cost of interaction with the environment is the same for any candidate placement and hence it is not necessary to take it into account.

#### 4. Select the optimal component placement

The set of candidate component placements is finite. Hence the optimal placement (the placement  $p$  with the minimal value of  $Cost(\mathcal{A}_p)$ ) can be found by exhaustive search. Investigation of more effective techniques for searching the minimal-cost placement is a subject for further research.

## 6 Conclusions and future work

The paper identifies a lack in existing solutions to the Component Placement Problem (CPP). The solutions evaluate component placements with no respect to the interaction among components. In this text, we proposed a solution to the interaction aspect of the CPP while allowing integration of existing algorithms for solving the non-interaction aspect (resource optimization for instance). The paper therefore provides a solution to a general CPP with respect to both interaction and non-interaction properties. The algorithm for solving the interaction aspect of the CPP is based on the Component-interaction automata language, which allows us to analyse the density of communication between two components, thanks to the information about communicating components in labels. Then, each component placement (from the finite set of all candidate placements) can be evaluated with a cost function with respect to the amount of inter-server communication of the placement, and we can select the placement with the minimal cost as the optimal one.

<sup>2</sup> Represented by the set  $\{(n, a, -) \mid n \in S, a \in Act\} \cup \{(-, a, n) \mid n \in S, a \in Act\}$ .

The work is intended as a proposal for extension of existing solutions to the CPP. For this reason, we have not implemented it ourselves. Besides the implementation, there are several other directions for further work. It includes the design and evaluation of a set of cost functions for calculation of one-to-one interaction density in CI automata, and a technique for getting the minimal cost solution that is more efficient than exhaustive search.

## References

1. Stewart, C., Shen, K., Dwarkadas, S., Scott, M.L., Yin, J.: Profile-driven component placement for cluster-based online services. *IEEE Distributed Systems Online* **5**(10) (2004) 1–6
2. Karve, A., Kimbrel, T., Pacifici, G., Spreitzer, M., Steinder, M., Sviridenko, M., Tantawi, A.: Dynamic placement for clustered web applications. In: *Proceedings of the International Conference on World Wide Web (WWW'06)*, Edinburgh, Scotland, ACM Press, USA (2006) 595–604
3. Kichkaylo, T., Ivan, A., Karamcheti, V.: Constrained component deployment in wide-area networks using ai planning techniques. In: *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'03)*, Nice, France, IEEE Computer Society, USA (2003) 3.1
4. Hunt, G.C., Scott, M.L.: The coign automatic distributed partitioning system. In: *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI'99)*, New Orleans, Louisiana, USA, USENIX Association, USA (1999) 187–200
5. Amiri, K., Petrou, D., Ganger, G.R., Gibson, G.A.: Dynamic function placement for data-intensive cluster computing. In: *Proceedings of the USENIX 2000 Annual Technical Conference*, San Diego, CA, USA (2000)
6. de Alfaro, L., Henzinger, T.A.: Interface-based design. In: *Proceedings of the 2004 Marktoberdorf Summer School*, Germany, Kluwer, The Netherlands (2005)
7. Beek, M., Ellis, C., Kleijn, J., Rozenberg, G.: Synchronizations in Team automata for groupware systems. *Computer Supported Cooperative Work — The Journal of Collaborative Computing* **12**(1) (2003) 21–69
8. Magee, J., Kramer, J., Giannakopoulou, D.: Behaviour analysis of software architectures. In: *Proceedings of the 1st Working IFIP Conference on Software Architecture (WICSA'99)*, San Antonio, TX, USA, Kluwer, The Netherlands (1999) 35–50
9. Plasil, F., Visnovsky, S.: Behavior protocols for software components. *IEEE Transactions on Software Engineering* **28**(11) (2002) 1056–1076
10. Brim, L., Černá, I., Vařeková, P., Zimmerova, B.: Component-Interaction automata as a verification-oriented component-based system specification. In: *Proceedings of the ESEC/FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS'05)*, Lisbon, Portugal, Iowa State University, USA (2005) 31–38  
Published also in *ACM SIGSOFT Software Engineering Notes*, Volume 31, Issue 2 (March 2006).
11. Vařeková, P., Zimmerova, B.: Component-Interaction automata for specification and verification of component interactions. In: *Proceedings of the IFM 2005 Doctoral Symposium on Integrated Formal Methods*, Eindhoven, The Netherlands, Technische Universiteit Eindhoven (TU/e), The Netherlands (2005) 71–75