Constrained Component Deployment in Wide-Area Networks using AI Planning Techniques

Tatiana Kichkaylo, Anca Ivan, and Vijay Karamcheti
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University, New York, NY 10012
{kichkay,ivan,vijayk}@cs.nyu.edu

Abstract

Component-based models represent a dominant trend in the construction of wide-area network applications, making possible the integration of diverse functionality contained in modules distributed across the network. Although linkages between modules have traditionally been specified statically, a growing number of frameworks are investigating approaches where appropriate components are dynamically selected and deployed in the network. Dynamic approaches enable flexible adaptation of the application to achieve load-balancing, satisfy QoS requirements, and in general, enable customization to changing client and network characteristics.

A key element of dynamic component-based frameworks is the deployment plan, which describes the choice of components, their locations, and linkages. The problem of finding a valid component deployment is harder than traditional optimization problems because one needs to decide on the set of components while satisfying various constraints resulting from application semantic requirements, network resource limitations, and interactions between the two. Partly because of this complexity, existing component frameworks typically address only a limited case of the general deployment problem.

In this paper, we propose a general model for the component placement problem (CPP) and present an algorithm for it, which is based on planning algorithms developed by the Artificial Intelligence community. These algorithms have benefited from several decades of research and offer substantial expressibility with good efficiency. We validate the effectiveness of our algorithm by demonstrating its scalability with respect to network size and number of components in the context of deployments generated for two example applications – a security-sensitive mail service, and a webcast service – in a variety of network environments.
1 Introduction

The explosive growth of the Internet and the development of new networking technologies has been accompanied by a trend favoring the use of component-based models for construction of wide-area network applications. This trend, exemplified in grid frameworks such as Globus [7], as well as component frameworks such as CORBA [21], J2EE [26], and .NET [20], enable the construction of applications by integrating functionality embodied in components possibly running across multiple administrative domains. Although most such frameworks have traditionally relied upon a static model of component linkages, a growing number of approaches (e.g., Active Frames [18], Eager Handlers [30], Active Streams [3], Ninja [10, 25], CANS [9], Partitionable Services [11], Conductor [28, 24], and recent work on Globus [8]) have advocated a more dynamic model, where the selection of components that make up the application and their location in the network (“deployment”) are both decisions that are deferred to run time.

Dynamic component-based frameworks allow distributed applications to flexibly and dynamically adapt to variations in both resource availability and client demand. A static application model is often unable to deliver good performance in such changing environments. To take an example, security-sensitive applications may wish to trade-off concerns of security and efficiency depending on whether or not the execution environments they run in consist of trusted nodes and links. Similarly, an application that relies on high-bandwidth interactions between its components is not efficient either when the available bandwidth on a link drops or the application is accessed by a resource-limited client. The dynamic frameworks mentioned above enable adaptation to the above changes by dynamically deploying application-aware components that can achieve load-balancing, satisfy client QoS requirements (e.g., by transcoding), and enable higher throughput (by replicating appropriate components), in essence customizing the application to its resource and usage conditions.

The benefits of dynamic component frameworks are fully realizable only if the deployment of components in response to changes is done as automatically as possible. To enable this, most such approaches rely on three elements: (i) a declarative specification of the application, (ii) a trigger module, and (ii) a planning module. The trigger module monitors application behavior and network conditions and chooses the moments when adaptation is required. The planning module makes decisions on how to adapt, by selecting and deploying components in the network to best satisfy application requirements as dictated by the declarative specification. This paper focuses on the planning aspect of such frameworks.

In general, the planning problem in dynamic frameworks is complicated by the fact that to compute a valid deployment, one needs to (i) decide on a set of components, and (ii) place these components on network nodes in the presence of application (type) constraints (e.g., that linked components can consume each other’s outputs), resource constraints (e.g. node CPU capacity and link bandwidth), and interactions between the two (e.g., that an insecure link might affect the security characteristics of application data). The need to simultaneously achieve both these goals makes the planning problem computationally harder than traditional mapping and optimization problems in parallel and distributed systems, which tend to focus on a subset of the concerns of requirement (ii) above. This complexity is also the reason that existing dynamic frameworks have either completely ignored the planning problem [18, 30, 3], or have a solution only for a very limited case [10, 25, 9, 11, 28, 24, 8].

This paper addresses this shortcoming by proposing a model for the general planning problem, referred to as the Component Placement Problem (CPP), and describing an algorithm for solving it. The model aims for expressibility: component behavior is modeled in terms of implemented and required
interfaces [11], and application, resource, and their interaction constraints are all represented using arbitrary functions. Our algorithm for solving the CPP, called Sekitei, leverages several decades of research on planning techniques developed by the Artificial Intelligence (AI) community. Sekitei overcomes the scalability restrictions of state-of-the-art AI planning techniques (e.g., RIFO [16]) by exploiting the specific characteristics of CPP. The Sekitei planner has been implemented in Java as a pluggable module, allowing its use in component-based frameworks such as Smock [11]. We characterize the run-time and nature of deployments generated for two example applications – a security-sensitive mail service, and a webcast service – in a variety of network environments. Our results validate the scalability of the algorithm, both with respect to the network size and the number of application components.

The rest of this paper is structured as follows. As background in Section 2, we briefly discuss existing approaches adopted for the component placement problem, overview AI planning techniques, and introduce an example mail application that serves as a running example throughout the paper. Sections 3 and 4 present CPP and our algorithm to solve it. In Section 5 we present and analyze experimental results. Section 6 discusses limitations of the current implementation of the algorithm and future work, and we conclude in Section 7.

2 Background and Related Work

2.1 Component-based frameworks

From a planning point of view, there are two classes of dynamic component-based frameworks: (i) systems that assume the existence of a planner (Active Frames [18], Eager Handlers [30], Active Streams [3]), and (ii) systems that implement their own planner (GARA [5, 8], Ninja [10, 25], CANS [9], Smock [11], and Conductor [28, 24]). The planners implemented by the second class can be further divided into two subclasses: (i) planners that deploy a fixed set of components (GARA), and (ii) planners that both choose and deploy a subset of components, while satisfying application and network constraints (Ninja, Smock, CANS, Conductor).

GARA (Globus Architecture for Reservation and Allocation) [5, 8], the module responsible for planning in the Globus [7] architecture, belongs to the first subclass. GARA relies on a pre-established relationship between the tasks of the application and deploys the tasks such that the resource consumption is minimum. GARA provides resource discovery and selection (based on attribute matches), and allows advance reservation for resources like CPU, memory, and bandwidth. However, it does not consider application specific properties, such as that some interactions need to be secure.\(^1\)

Approaches that both select and deploy components can be further subdivided into two categories: planners that exhaustively search the solution space (Smock [11]) and planners that simplify the assumptions of the component placement problem (Ninja [10, 25], CANS [9]) to perform directed searches.

The Smock system implementing the Partitionable Services framework [11] permits network services to be constructed as a flexible assembly of smaller components, permitting customization and adaptation to network and usage situations. The component and network descriptions in Smock are fairly general: components can implement and require multiple interfaces (these define “ports” for linkages), can specify resource restrictions, and additionally impose deployment limitations based on application-dependent properties (e.g. privacy of an interface). This generality comes at a cost: the current Smock

\(^1\)Globus sets up secure connections between application components, thereby satisfying this particular constraint. However, there is no mechanism to specify component properties that are affected by the environment.
planning module performs exhaustive search to infer a valid deployment. Ninja [10, 25], CANS [9], and Conductor [28, 24] all provide solutions for stream-based applications, enabling the deployment of appropriate transcoding components along the network path between clients and servers so as to overcome limitations of client devices and network connectivity. The Ninja planning module focuses on choosing already existing instances of multiple input/output components in the network so as to satisfy functional and resource requirements on component deployment. CANS restricts itself to single input, single output components, but can handle constraints imposed by the interactions between application components and network resources, and additionally can efficiently plan for a range of optimization criteria. For example, the CANS planner can ensure that node and link capacities along the path are not exceeded by deployed components, while simultaneously optimizing an application metric of interest (e.g., response time). Conductor adopts the same component restriction as CANS, but has a less expressive component model that focuses only on resource constraints.

2.2 General planning approaches

The high-level objective of the component placement problem closely resembles the long-studied classical planning problems in the Artificial Intelligence community. In classic AI planning, the world is represented by a set of boolean variables, and a world state is a truth assignment to these variables. The system is described by a set of possible operators, i.e. atomic actions that can change the world state. Each operator has a precondition expressed by a logical formula and a set of effects (new truth assignments for some variables of the world state). An operator is applicable in a world state if its precondition evaluates to true in that state. As the result of an operator application the world state is changed as described by the operator’s effects. A planning problem is defined by a description of the set of possible operators, an initial state (complete truth assignment to all variables), and a goal (logical formula). The goal of the planner is to find a sequence of applicable operators that, when executed from the initial state, brings the system to a state in which the goal formula evaluates to true.

Classic planners attempt to perform directed search in the space of possible plans and can be divided into four classes based on their search method: regression planners (Unpop [19], HSPR [2]) search from the goals, progression planners (GraphPlan [1], IPP [17]) start from the initial state, causal-link planners (UCPOP [23]) perform means-ends analysis, and compilation-based planners (SATPLAN [14], ILP-PLAN [13], BlackBox [12], GP-CSP [6]) reduce the planning problem to a satisfiability or optimization problem, e.g. integer linear programming. Some planners, e.g. BlackBox, use a combination of the above techniques to improve performance. McDermott [19] suggests that a regression planner can benefit of using some planning graph (progression) techniques; however, we are not aware of any implementation of this idea.

One of the extensions of classic planning is planning with resources. Most existing resource planners (e.g. RIFO [16], LPSAT [27], ILP-PLAN [13]) are limited to linear expressions in preconditions and effects. Zeno [22] can accept more complicated expressions, but delays their processing until the expressions become linear due to variable bindings.

Classic planners are capable of solving a very broad class of problems, including the component placement problem (given that all resource restrictions are expressed as linear equations). However, the performance of such planners on the CPP can be improved. The problem we are aiming to solve is very structured, which allows us to introduce optimizations not possible in a general AI planner.
2.3 Mail Application

Throughout this paper, we highlight different aspects of the planning algorithm using the example of a component-based security-sensitive mail service, originally introduced in [11]. The mail service provides the classic functionality — user accounts, folders, contact lists, and the ability to send and receive e-mail. In addition, it allows a user to associate a trust level with each message depending on its sender or recipient. A message is encrypted according to the sender’s sensitivity and sent to the mail server. The mail server transforms the ciphertext into a valid encryption corresponding to the receiver’s sensitivity and saves the new ciphertext into the receiver’s account. The encryption/decryption keys are generated when the user first subscribes to the service.

The mail service is constructed by flexibly assembling the following components: (i) a mail server that manages e-mail accounts and can be accessed using (ii) mail client components of differing capabilities, (iii) view mail server components that replicate the mail server as desired, (iv) encryption/decryption components that ensure confidentiality of interactions between the other components, and a (v) key repository that holds cryptographic keys. These components allow the mail application to be deployed in different environments. If the environment is secure and has high available bandwidth, the mail client can be directly linked to the mail server. The existence of insecure links and nodes should trigger the deployment of an encryptor/decryptor pair to protect the privacy of sent messages. Similarly, the view mail server can act as a cache close to the client if the links have low available bandwidth. Ideally, such decisions should be made automatically, but this requires both a better specification of application requirements, and a planning module that can take these requirements to produce the desired deployment. In an earlier paper [11], we have described a novel declarative model for specifying the component behavior (see Appendix A for a detailed example of the XML-based specification of the mail application). This paper focuses on the planning module.

Figure 1 illustrates an example of a possible component deployment given the mail application specification and a very simple network configuration. The network comprises four nodes (N0, N1, N2, N3) connected by four links ((0,1),(1,2),(2,3),(3,0)). The first three links have good bandwidth (200Mb/s) and the last links is very slow (50Mb/s).

For clarity, we assume that each node has CPU power to execute only one component. The goal of the planner is to deploy a MailClient component on node N0. One way to achieve this goal is to deploy a KeyRepository component on node N3, a MailServer on node N2, and connect the MailClient to the KeyRepository and the MailServer. In the next two sections, we shall describe how our planning algorithm automatically achieves this deployment.

3 Modelling the Component Placement Problem

Many component-based systems solve the Component Placement Problem (CPP) in one form or another. However, the specific formulation differs along one or more of the following dimensions: mobility
of components (fixed locations in Ninja vs. arbitrary deployments), arity of components (single input -
single output components in CANS vs. arbitrary arity), support for resource constraints, etc. As one of
the contributions of this paper we present a general model for the CPP that allows us to unify the different
variations of this problem and enables use of the same planning algorithm in various component-based
frameworks.

Formally, a component placement problem is defined by the following elements:

- **The network** topology is described by a set of nodes and a set of links. Each node and link has
tuples of static and dynamic properties associated with it. The dynamic properties are non-negative
real values that can be changed, e.g. node CPU, link bandwidth. The static properties are assumed
fixed during the life time of an application. Static properties might be represented by boolean
values or real intervals, e.g. security of a link and trust level of a node.

- **The application framework** is defined by sets of interface types and component types, similar to an
object-oriented language such as Java. For each component type, sets of required and implemented
interfaces are specified.\(^2\) A component can be deployed on a node only if all of its required
interfaces are present on that node, and as a result of such a deployment each of the implemented
interfaces becomes available.

- **Link crossing behavior** is described by interface specific functions. For each interface type, these
functions describe how the interface properties are affected by the link properties when crossing
the link, and how dynamic properties of the link are changed in result of this operation. For
example, the security of an interface after link crossing can be computed as a conjunction of the
security of the interface at the source and the security of the link; the link bandwidth after the
link crossing is the original bandwidth minus the consumed bandwidth, which is computed as
minimum of the original link bandwidth and the bandwidth of the interface at the source.

- Similarly, **component deployment behavior** is specified by component type-specific functions.
The function parameters are the properties of the node and all the required interfaces. The func-
tions describe: (i) the resource requirements (preconditions) of the component, (ii) the properties
of each of the implemented interfaces, and (iii) the dynamic properties of the node after the com-
ponent deployment.

- **The goal.** In the simplest case, the goal is to put a component of a given type onto a given node.
Other goals can include, for example, delivering a particular set of interfaces to a given node; this
can be useful for repair of deployments in case of change in resource availability in the network.

Appendix A contains an XML description of CPP for the example problem discussed in Section 2.3.
The XML notation was originally introduced in the context of the Smock system [11].

The above model of the CPP is very flexible and allows capturing of a variety of application properties
and requirements. In particular, most models we found in literature can be expressed in our formalism.
In addition, the above model of the CPP enables easy compilation into a planning problem as described
in Section 4.1.

\(^2\)The counterparts for these concepts in a statically-linked Java/RMI application is as follows: implemented interfaces are
identical to their namesake, while required interfaces correspond to remote references.
4 The Planning Algorithm

Figure 2 describes the structure of our planning module. The representation of the component placement problem is framework specific. The compiler module transforms this representation of the problem into an AI-style planning problem which can be solved by the planner. The decompiler performs the reverse transformation, converting the AI-style solution into a framework specific deployment plan.

![Process flow graph for solving the CPP.](image)

4.1 Compilation of the CPP into a planning problem

The component placement problem can be mapped to the AI planning problem in the following way. The state of the world is described by the network topology, the existence of interfaces on nodes (boolean values), and the availability of resources (real values). The operators correspond to placing a component on a node and crossing a link with an interface. An operator schema has the following sections (line numbers refer to the code fragment below):

- logical precondition of the operator, i.e. a set of boolean variables that need to be true for the operator to be applicable (line 2);
- resource preconditions described by an arbitrary function that returns a boolean value (line 3);
- logical effects, i.e. a set of logical variables made true by an application of the operator (line 4);
- resource effects represented by a set of assignments to resource variables (lines 5,6,7).

For example, the following schema describes the placing of a Decryptor component on a node. This component requires that a secure MailServerInterface is available on the node. As a result of the component placement, part of the CPU capacity is used up, and an instance of DecryptorInterface becomes available on the node. The properties of the implemented interface are computed as functions of the properties of the required interface.

```
1 placeDecryptor( ?n: node )
2 pre: avMailServerInterface(?n)
3 cpu(?n) >= 3 AND secure(MailServerInterface, ?n)==True
4 eff: avDecryptorInterface(?n), placedDecryptor(?n)
5 cpu(?n) := cpu(?n)-3
6 bw(DecryptorInterface, ?n) := bw(MailServerInterface,?n)
7 trust(DecryptorInterface, ?n) := trust(MailServerInterface,?n)
```
Given the CPP definition discussed above, the compilation of the CPP into a planning problem is straightforward. For each of the component types, the compiler generates an operator schema for a placement operator. In addition, an operator for link crossing is generated for each interface type. The initial state is created based on the properties of the network. The goal of the CPP is translated into a boolean goal of the planning problem.

4.2 The planning algorithm

The general planning problem is computationally very hard (PSPACE-complete), and complete algorithms that solve it usually do not scale well. Various AI planning algorithms achieve good performance on practical problems by effectively pruning different parts of the search space, even though the worst case scenarios take exponential time. In the case of the CPP, scalability concerns stem from two sources: the size of the network, and the number of components, both of which affect the number of operators in the compiled problem. Since we do not expect practical problems to require use of all possible operators, what distinguishes a good CPP solution is its ability to scale well in the presence of large amounts of irrelevant information. Our solution combines several techniques from AI planning and exploits the structure of the problem, allowing us to drastically reduce the search space.

Our algorithm consists of the following four phases (see Figure 3).

- First, a regression phase determines what operators are relevant for the goal. An operator is relevant if it can participate in a sequence of actions reaching the goal. The set of relevant operators is further reduced to a set of possible operators. An operator is possible if it is relevant and belongs to a subgraph of the regression graph rooted in the initial state.

- Second, a progression phase finds all states possibly reachable from the initial state given the set of possible operators, thus performing some additional pruning of the regression graph.

- Third, an exhaustive search in the resulting graph is performed to achieve completeness. An algorithm is complete if it finds a solution when a solution exists.

- Plans found in the third phase are symbolically executed to ensure soundness in the presence of arbitrary expressions in resource preconditions and effects.

Regression phase. The regression phase takes into account only logical preconditions and effects of operators and builds a regression graph. This graph is a very optimistic representation of all operators that might be useful for achieving the goal.

The regression graph contains interleaving facts and operator levels, starting and ending with a fact level. This graph is constructed as follows.

- Fact level 0 is filled in with the goal.

- Operator level \(i\) contains all operators that achieve some of the facts of level \(i - 1\).

- Fact level \(i\) contains all logical preconditions of the operators of the operator level \(i\).
Initially, the regression graph is constructed until it is possible to extract a DAG containing the goal whose leaves are all present in the initial state. We say that a proposition (or operator) is possible in \( n \) steps if it belongs to the possible DAG of a regression graph with \( n \) operator levels. Appendix B gives an example of the RG for the problem presented in Section 2.3.

**Progression phase.** The regression graph provides a basis for the second phase of the algorithm, the planning graph construction. The planning graph is also called a progression graph, because it describes all states reachable from the initial state in a given number of steps.

The progression graph also contains interleaving operator and fact levels, starting and ending in a fact level. In addition, this graph contains information about the mutual exclusion (mutex) relations, e.g., that the placement of a component on a node might exclude placement of another component on the same node (because of CPU capacity restrictions). The mutex condition for operators used by our planner is a simplified version of that used by other planning algorithms [16]. This condition also takes into account resource expressions, and therefore the second phase of the algorithm is less optimistic than the regression graph construction.

The \( 0^{th} \) level of the planning graph contains facts that are true in the initial state. Then, pairs of levels are added according to the following algorithm:

- For each of the propositions of level \( i - 1 \) a copy operator is added to level \( i \) that has that fact as its precondition and effect, and consumes no resources.

- For each of the possible operators contained in the corresponding layer of the regression graph, an operator node is added to the planning graph if none of the operator’s preconditions is mutex at the previous proposition level.

- The union of logical effects of the operators of the level \( i \) forms the \( i^{th} \) fact level of the graph.

- Two operators of the same level are marked as mutex if (i) some of their preconditions are mutex, (ii) one operator changes a resource variable used in an expression for preconditions or effects of the other operator, or (iii) their total resource consumption exceeds the available value.
Two facts of the same level are marked mutex if all operators that can produce these preconditions are pairwise mutex.

Since the progression graph is less optimistic than the regression graph, it is possible that the last level of the progression graph does not contain the goal, or some of the goal propositions are mutually exclusive. In this case a new step is added to the regression graph, and the planning graph is reconstructed.

**Plan extraction phase.** If the planning graph contains the goal and it is not mutex, then the plan extraction phase is started. At this phase, an exhaustive search in the planning graph is performed, as introduced in [1]. A memoization technique is used to prune the search (i.e. bad sets of facts are saved and are not explored on subsequent iterations).

**Symbolic execution.** In our work, we aim to support arbitrary functions in resource preconditions and effects. For this reason, symbolic execution is the only safe way to ensure soundness of a solution. It is implemented in a straightforward way: a copy of the initial state is made, and then all operators of the plan are applied in sequence, their preconditions evaluated at the current state, and the state is modified according to the effect assignments. Note that correctness of the logical part of the plan is guaranteed by the previous phases of the algorithm, only resource conditions need to be checked.

### 4.3 Decompilation

The planner produces a plan as a sequence of operators `placeComponentNode` and `crossInterfaceFromTo`. In addition, information about logical support is easily extractable from the plan (e.g. `placeMailServerN0` produces proposition `avMailServerInterfaceN0` required by `crossMailServerInterfaceN0N1`). A deployment plan required by the framework should consist of `(component, node)` pairs and linkage directives, e.g. `(Decryptor, n1, DecryptorInterface, Encryptor, n5)` (send the DecryptorInterface implemented by the Decryptor component located on node n1 to the Encryptor component on node n5). The algorithm for obtaining the latter from the former is straightforward, and we do not present it here.

### 5 Results

The main goal of our experiments is to characterize the run-time and nature of deployments produced by the planner for different kinds of application behaviors and for a wide range of network conditions. To facilitate this evaluation, we built a prototype Java-based implementation of the Sekitei planner. All measurements reported in this section were taken on an AMD Athlon XP 1800+ machine, running Red Hat 7.1 and the J2RE 1.3.1 IBM Virtual Machine.

To model different wide-area network topologies, we used the GT-ITM tool [29, 4] to generate eight different networks $N_k$ (for different $k \in \{22, 33, \ldots, 99\}$ nodes). Figure 4 illustrates the $N_{99}$ network. Each topology simulates a WAN formed by high speed and secure stubs connected by slow and insecure links. The initial topology configuration files (.alt) created by the GT-ITM tool were augmented with link and network properties and transformed into XML files by the Network EDitor tool [15]. We present a simplified XML description of the network in Appendix A.
The performance of the planner was evaluated in the context of two applications. The first application is the **mail server** application described in Section 2.3. As mentioned there, the application comprises of five components – *MailServer, ViewMailServer, Encryptor, Decryptor, and MailClient* – each defined by its implemented and required interfaces, and the deployment conditions. Every interface is characterized by a set of properties *Trust, Secure, and Bandwidth* (see Appendix A). The second application is a **webcast** service, consisting of a *Server* that produces both images and text, a *Client* that consumes both, and additional *Splitter, Zip/Unzip, and Filter* components for splitting the stream and reducing the bandwidth requirements for the text and image data respectively. The goal in both applications was to locate the client components on specific nodes. In both cases, the “best” deployment was defined as the one with the fewest number of components.

We tested our planner by running six different experiments. The next paragraphs present in more detail the goal, the description, and the results of each experiment.

**Experiment 1: Planning under various conditions.**  The purpose of the first experiment is to show that the planner finds a valid component deployment plan even in hard cases, and usually does so in a small amount of time. The experiment, involving the mail server application, is conducted as follows. For each network topology $N_k$, where $k \in 22, 33, ..., 99$, and for each node $n$ in the network $N_k$, the goal is to deploy a *MailClient* component on the node $n$ given that the *MailServer* is running on some node.

The algorithm indeed finds a solution when it exists. The data points on Figure 5 correspond to the following cases. When the client and the server are located at the same stub, the algorithm essentially finds the shortest path between two nodes, which takes a very short time. Placement of a client in a different stub requires inserting some components into the path, and therefore takes longer (about 2 seconds). The reason for the bigger average time for networks $N_{22}$ and $N_{77}$ will be discussed in Section 6.

**Experiment 2: Scalability with respect to the network size.**  To see how the performance of the algorithm is affected by the size of the network, we ran the following experiment. Taking the $N_{99}$

---

3The algorithm does not distinguish any special cases. “The shortest path” is only a characterization of the result.
network topology (Figure 4) as our reference and starting with a small network with only two stubs, we added one stub at a time until the original 99-node configuration was achieved. For each of the obtained networks we ran the planner with the goal of placing MailClient on a fixed node.

As shown on Figure 6, the running time of the planner increases very little with the size of the network. Moreover, the graph tends to flatten. Such behavior can be explained by the fact that the regression phase of the algorithm considers only stubs reachable in the number of steps bounded by the length of the final plan. Even this set is further pruned at the progression stage. Therefore, our algorithm is capable of identifying the part of the network relevant for the solution, with no additional preprocessing necessary.

Figure 5. For each node of the network the planner tries to put a MailClient on that node given that a MailServer is running somewhere in the network. The graph shows the running time of the algorithm for each of the cases where a solution exists.

Figure 6. Scalability of the planner with respect to the size of the network.

Figure 7. Planning time for the webcast application with the increasing network size.
Experiment 3: Complex application structure. The mail application used in the above experiments requires only a chain of components. An important feature of our algorithm is that it can support more complicated application structures, i.e. DAGs and even loops. To verify that the planner behavior is not negatively affected by DAG-like structures, we generated deployments for the webcast service (the DAG structure arises because of the splitting and merging of image and text streams). The goal for the planner was deployment of the Client component on a specific node, given that the Server was separated from it by links with low available bandwidth. Figure 7 illustrates the running time of the algorithm as a function of the network size and validates our assertion.

Experiment 4: Scalability with respect to irrelevant components. Normally, a component-based framework can support more than one application at a time, and some components, such as Zip and Unzip may be reused in several applications. Therefore, the total number of components can be big, and it is desirable that the planner can effectively select a relevant subset of components. Given a component deployment problem (i.e. a state of the network and a client component to be deployed), all components can be divided into the following categories: (i) absolutely useless components that can never be used in any configuration involving the client, (ii) components useless given the availability of interfaces in the network, (iii) useful components. The following experiments investigate scalability of the planner with increasing number of component types.

First, the mail application is augmented with ten components that have no relation with the original components (absolutely useless components) and the first experiment is run. Next, the original mail application is augmented with ten components that implement interfaces meaningful to the application, but require interfaces that none of the components can provide. Figure 8 shows the results of these experiments. The absolutely useless components are rejected by the regression phase of the algorithm and do not affect its performance at all. Components whose implemented interfaces are useful, but required interfaces cannot be provided can be pruned out only during the second phase, which also takes into account the initial state of the network (the required interfaces might be available somewhere from the very beginning). The increase in the running time is due to processing of the useless components in the first phase of the algorithm.

Experiment 5: Scalability with respect to relevant components. Figure 9 shows how the performance of the planner depends on the increasing number of useful components. Initially, the planner runs the first experiment on the original $N_{99}$ network topology and deploys all five components of the mail application. Next, the network properties are modified such that all links (including stub-transit ones) become fast. In this case, the planner deploys at most four components (without the ViewMailServer). The third case is when the original network is transformed to be secure and the planner deploys at most three components. The last case considered in this experiment is when the network is both secure and fast, so that at most two components are necessary.

A component is considered useful if it implements an interface relevant for achieving the goal and all its required interfaces are present in the network or can be provided by other useful components. The choice of whether a useful component is actually used in the final plan is made during the third phase of the algorithm, which in the worst case takes time exponential in the length of the plan. The bigger the number of useful components, the bigger the branching factor of the progression graph, and therefore the

---

4The slight fluctuation is due to measurement artifacts, such as garbage collection.
Figure 8. Performance of the planner in presence of absolutely useless components and components useless given the network state.

base of the exponent. This means that in hard cases (very strict resource constraints, multiple component types implementing the same interface, highly connected networks) the initial planning can take long (a few minutes). However, adding new components to the system is much quicker.

Figure 9. Dependence of the planning time on the number of useful components.

Experiment 6: Reusability of existing deployments. In real-life scenarios, by the time a new client requests a service some other clients may already be running. It is desirable to reuse existent deployments, and thus minimize both the planning time and the total number of components deployed in the network (and therefore resource consumption). To demonstrate that our planner can take advantage of existent component deployments, we ran the following experiment.

Taking the webcast application and the $N_{99}$ network topology with the server running on a fixed node, we ran the planner with the goal to put a client on each of the nodes of the network in turn. The network
state is saved between the runs, so that the clients can join existing paths. We assume that clients are using exactly the same datastream, and there is no overhead for adding a new client to a server. Figure 10 shows a possible component deployment for the webcast application.

As expected, it is very cheap to add a new client to a stub that already has a client of the same type deployed (this corresponds to the majority of the points on Figure 11), because most of the path can be reused. The problem in this case is effectively reduced to finding the closest node where the required interfaces are available.

Figure 10. Logical component deployment for the webcast application

Figure 11. Reuse of existent deployments.

6 Limitations and Future Work

The increase of the running time in Experiment 5 and the big average time for networks \( N_{22} \) and \( N_{77} \) in the first experiment can be explained by the fact that many resource conflicts can be identified only during the last phase of the algorithm. The two networks mentioned above have a bigger number of low-bandwidth insecure links between stubs than the other networks. Because of this, the algorithm constructs and checks many logically correct plans that fail during symbolic execution due to resource restrictions. Currently the algorithm is not capable of identifying such bottlenecks. Such an ability will significantly improve the algorithm’s performance. One way of achieving this objective is to compile some resource restrictions into logical form,\(^5\) or by learning logical restrictions from failed symbolic executions.

In addition to improving the presented four-phase algorithm, we also plan to evaluate the effectiveness of other approaches for solving the CPP (to address the few cases that Sekitei does not handle well). Among the possibilities include replacing the planning graph phase with compilation into an optimization problem. This might require us to put tighter restrictions on the form of expressions used in preconditions and effects. The right balance between the expressibility of the expressions and the performance of the algorithm is an interesting research question. A completely different algorithm can

\(^5\)This would increase the size of the planning problem, but we expect it to significantly improve the performance of the planner
also be constructed based on causal link planners. Although such planners show low performance on classic planning problems compared to planning graph and compilation-based planners, they might be a good choice for the CPP because of the structure of the problem.

We also plan to extend the presented problem model to take into account the actual load on components, e.g., the number of clients connected to the server. This would require changing the formulae describing component placement; in particular, the parameters of the implemented interfaces need to be added to the list of arguments.

Sekitei is capable of reusing existent component deployments for new clients. We plan to investigate the application of our planning algorithm for incremental replanning in case of a change in resource availability. Another issue that arises in real-world applications is the need for decentralized planning. It is desirable that each trust domain have its own planner, and the planners collaborate when necessary, each planning only for nodes in its domain. We plan to address this issue in our future work.

7 Conclusions

In this paper we have presented a general model of the component placement problem and an algorithm for solving it. The component placement problem arises in most component-based frameworks, which are gaining increasing popularity. Our model allows for specification of a variety of network and application properties and restrictions, and is general enough to be used in many existing frameworks.

The Sekitei algorithm for the general CPP is based on AI planning techniques. It provides good expressibility in both the problem specification and the plans that can be generated. In particular, complex application structures and general expressions in resource preconditions and effects are supported. As demonstrated by the experiments presented in Section 5, Sekitei is capable of identifying the relevant information and scales well with respect to the size of the network and the total number of components supported by the framework. For reasonable sizes of the network and the application, the algorithm takes a few seconds to generate a valid deployment plan, significantly smaller than the expected overhead for actually deploying these components in the network.

Sekitei is implemented as a pluggable module and is currently being integrated into the Smock framework [11], which would allow us to test the performance of deployed plans.

References


A XML specification of the CPP

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trust</td>
<td>Interval</td>
<td>$Interf.Trust := Node.Trust$ (Deploy)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$Interf2.Trust := Interf.Trust$ (Crosslink)</td>
</tr>
<tr>
<td>LinkBandwidth</td>
<td>Float</td>
<td>$Interf2.LB := \min(\text{Link.LB}, Interf.LB)$ (Crosslink)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{Link.LB} := \min(\text{Link.LB} - (\text{Link.LB}, Interf.LB))$ (Crosslink)</td>
</tr>
<tr>
<td>NodeCPU</td>
<td>Float</td>
<td>$\text{Node CPU} := (\text{Node CPU} - \text{Comp.Node CPU})$ (Deploy)</td>
</tr>
</tbody>
</table>

Figure 12. Declarative specification of properties. Each property is described by a name, type, and formulae describing the link crossing behavior and default assignment during component placement (the latter can be overridden by a component specification). $Interf$ refers to interface properties on the source node, $Interf2$ to the interface properties on the destination node during the link crossing.
Figure 13. Declarative specification (incomplete) of the example mail service. A component is specified by a name, sets of required and implemented interfaces, deployment conditions, and formulae describing resource effects of the deployment.
Figure 14. Declarative specification (incomplete) of the network. The network description contains lists of nodes and links, with static and dynamic properties specified for each of them.

B Regression and progression graphs
Figure 15. The levelled regression graph for the example application discussed in Section 2.3. Propositions are shown in italic, and operators in normal font. Lines correspond to support relations between propositions and operators. The bold lines correspond to the possible subgraph with 3 levels, solid lines to the possible subgraph with 4 levels. Nodes connected with dashed lines are not possible within 4 steps.

Figure 16. The planning graph for the 4-level LRG shown on Figure 15. Straight lines correspond to support relations, frame support is shown in dashed lines. The arcs correspond to mutex relations. Mutex relations are defined only between pairs of operators or propositions at the same level.