

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

Wide-area network applications are increasingly being built using component-based models, which enable integration of diverse functionality in modules distributed across the network. In such models, dynamic component selection and deployment enables an application to flexibly adapt to changing client and network characteristics, achieve load-balancing, and satisfy QoS requirements. Unfortunately, the problem of finding a valid component deployment while satisfying various constraints resulting from application semantic requirements, network resource limitations, and interactions between the two.

In this paper, we propose a general model for the component placement problem and present an algorithm for it, which is based on AI planning algorithms. 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 [6], as well as component frameworks such as CORBA [19], J2EE [24], and .NET [18], enables 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 [16], Eager Handlers [26], Active Streams [3], Ninja [23], CANS [8],

Smock [9], Conductor [22], and recent work on Globus [7]) have advocated a more dynamic model, where the selection of components that make up the application and their location in the network (“deployment”) are 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. For example, a security-sensitive application may wish to trade-off concerns of security and efficiency depending on whether or not its execution environment consists 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. Dynamic frameworks enable adaptation to the above changes by 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 components are automatically deployed in response to changes. To enable this, most such approaches rely on three elements: (i) a *declarative specification* of the application, (ii) a *trigger* module, and (iii) 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.

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 pres-

ence of application (type) constraints (e.g., linked components should consume each other’s outputs), resource constraints (e.g. node CPU capacity and link bandwidth), and interactions between the two (e.g., 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 [16, 26, 3], or have addressed only a very limited case [23, 8, 9, 22, 7].

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 expressiveness: component behavior is modeled in terms of implemented and required interfaces [9], and application, resource, and their interaction constraints are all represented using arbitrary monotonic 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 [14]) by exploiting the specific characteristics of CPP. The *Sekitei* planner has been implemented in Java as a pluggable module to allow its use in component-based frameworks such as *Smock* [9]. We report on its use to generate deployments 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 discuss existing approaches to 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 *Sekitei* implementation 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 [16], Eager Handlers [26], Active Streams [3]), and (ii) systems that implement their own planner (GARA [7], Ninja [23], CANS [8], *Smock* [9], and *Conductor* [22]).

The second class can be further divided into two subclasses. The first subclass includes systems such as GARA (Globus Architecture for Reservation and Allocation) [7], the planning module in the Globus [6] architecture, which assumes a pre-established relationship between application tasks to deploy them with minimal resource consumption. GARA supports 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

The second subclass of planners both select and deploy a subset of components, while satisfying application and network constraints. Systems such as *Ninja* [23], *CANS* [8], and *Conductor* [22], all of which enable the deployment of appropriate transcoding components along the network path between weak clients and servers, simplify the assumptions of the planning problem to perform directed search. 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. *Conductor* restricts itself to single input, single output components, focusing on satisfying resource constraints. *CANS* adopts similar component restrictions, 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).

More general are systems such as *Smock* [9], which permit network services to be constructed as a flexible assembly of smaller components, permitting customization and adaptation to network and usage situations. The *Smock* planner works with very general component and network descriptions: 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* planning module performs exhaustive search to infer a valid deployment. The work described in this paper grew out a desire to remedy this situation.

2.2 General planning approaches

The high-level objective of the component placement problem closely resembles long-studied planning problems in the AI community. In classic AI planning, the world is

¹Globus 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.

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 to variables of the world state). An operator is applicable in a world state if its precondition evaluates to true in that state. The result of an operator application is to change the world state as described by the operator’s effects. A planning problem is defined by a description of the operator set, an initial state (complete truth assignment to all variables), and a goal (logical formula). The planner finds 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 perform directed search in the space of possible plans and can be divided into four classes based on their search method: regression planners (Unpop [17], HSPr [2]) search from the goals, progression planners (GraphPlan [1], IPP [15]) start from the initial state, causal-link planners (UCPOP [21]) perform means-ends analysis, and compilation-based planners (SATPLAN [10], ILP-PLAN [12], BlackBox [11], GP-CSP [5]) 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 [17] suggests extending regression planners using progression techniques; however, we are not aware of any implementation of this idea.

An extension of classic planning is planning with resources. Most existing resource planners (e.g. RIFO [14], LPSAT [25], ILP-PLAN [12]) limit themselves to linear expressions in preconditions and effects. Zeno [20] can accept more complicated expressions, but delays their processing until variable bindings linearize the expressions.

AI planners are thus capable of solving a very broad class of problems, including the component placement problem (as long as all resource restrictions are expressed as linear equations), but suffer from scalability limitations. We address the latter issue in this paper, exploiting the structured nature of the component placement problem 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 [9]. The mail service provides expected 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, which transforms the

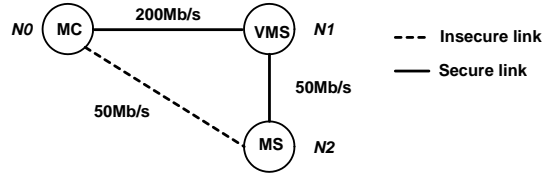


Figure 1. Component deployment.

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 MailServer that manages e-mail accounts, (ii) MailClient components of differing capabilities, (iii) ViewMailServer components that replicate the MailServer as desired, and (iv) Encryptor/Decryptor components that ensure confidentiality of interactions between the other components. These components allow the mail application to be deployed in different environments. If the environment is secure and has high available bandwidth, the MailClient can be directly linked to the MailServer. The existence of insecure links and nodes triggers deployment of an Encryptor/Decryptor pair to protect message privacy. Similarly, the ViewMailServer can serve as a cache to overcome links with low available bandwidth. Figure 1 illustrates a simple scenario where the MailClient can be deployed on node N_0 only if connected to a MailServer through a ViewMailServer. Directly linking the MailClient to the MailServer is not possible because the link between them does not have enough available bandwidth to satisfy the MailClient requirements. Satisfying these requirements automatically needs both a better specification of application requirements and a planning module to generate the deployment. An earlier paper [9] has described a novel declarative model for specifying component behavior; here, we focus on the planning module.

3 The component placement problem

Many 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 (fixed locations in Ninja vs. arbitrary deployments), arity (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 unifies different variations of this problem and enables use of the same planning algorithm in various component-based frameworks.

Formally, the CPP is defined by the following five elements: (i) the network topology, (ii) the application framework, (iii) the component deployment behavior, (iv) the link

crossing behavior, and (v) the goal of the CPP.

Network topology. 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.

Application framework. The application is defined by sets of interface types and component types, similar to an object-oriented language such as Java. Each component type specifies sets of *implemented* and *required* interfaces:² the former describe component functionality, while the latter indicate services needed by the component for correct execution. In addition, each interface is characterized by a set of component-specific *properties*. From the planning point of view, properties are defined as functions of other properties and have no semantics attached to them.

In general, applications can propagate properties either (i) from required to implemented interfaces – *publish-subscribe* applications, or (ii) from implemented to required interfaces – *request-reply* applications. In *publish-subscribe* applications, servers send data streams to clients. In *request-reply* applications, clients make requests to servers and servers send back replies. Although the planner can work with both types of applications, our description of the planning algorithm focuses on request-reply applications, e.g. the mail application described in Section 2.3.

Our example application contains three interfaces—MailServerInterface, EncryptedMailInterface, and MailClientInterface—corresponding to the normal and encrypted server interfaces, and the client interface respectively. Figure 2 shows the partial specification of the ViewMailServer component, which implements and requires MailServerInterface. This interface is associated with both application-specific and application-independent properties. Application-specific properties include the trust level (Trust) and message security (Sec), which indicate, respectively, the maximum message sensitivity level and whether or not the interface preserves message confidentiality. Application-independent properties include the number of incoming requests (NumReq), the maximum response size for a request (ReqSize), the request reduction factor (RRF), the amount of CPU consumed to process each incoming request (ReqCPU), and the maximum number of requests that can be processed by the component (MaxReq). The RRF attribute gives the ratio of requests sent to required interfaces in response to

²The counterparts for these concepts in a statically-linked Java/RMI application is as follows: implemented interfaces are identical to their name-sake, while required interfaces correspond to remote references.

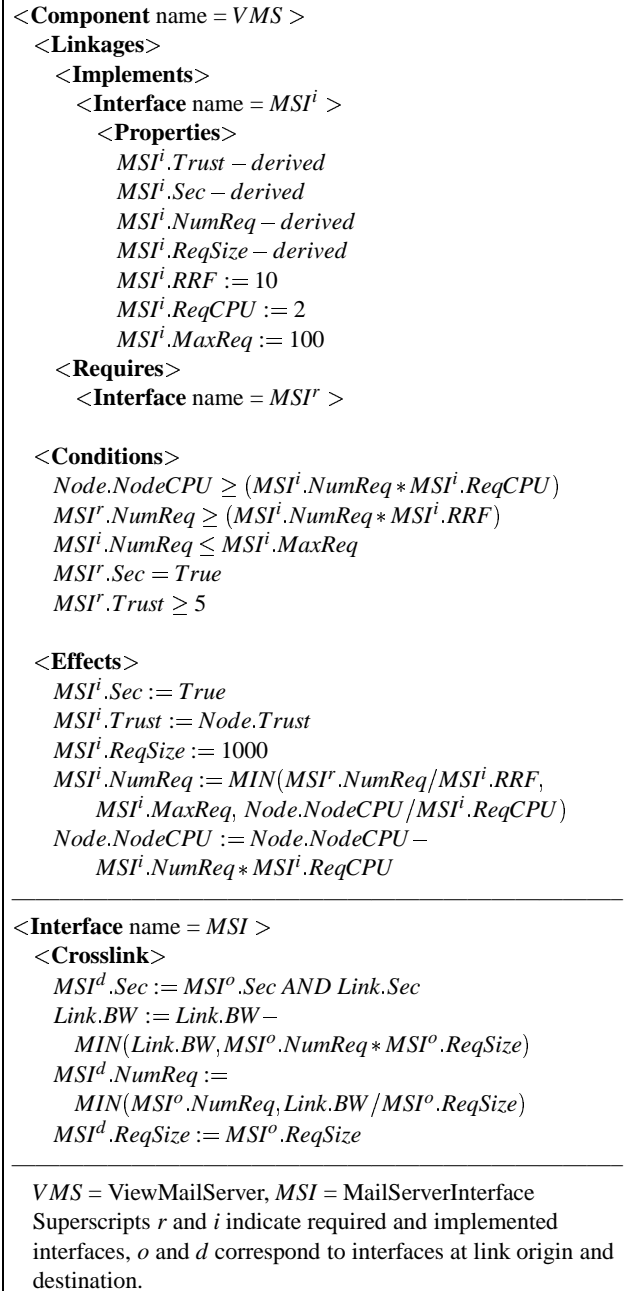


Figure 2. Component/Interface descriptions.

requests on the implemented interfaces. The use of these application-independent properties is described in the next paragraph.

Component deployment behavior. A component is deployed on a node only if the required interfaces are present on the node and there are sufficient node and link resources. After deployment, the implemented interfaces

become available on the node and the dynamic properties of the node are altered. The Sekitei planner can find a plan that satisfies both the application-specific and application-independent constraints. The former are expected to be written by the programmer. To simplify the task of writing application-independent constraints, we have introduced a small set of properties (NumReq , MaxReq , RRF , ReqCPU) that capture component resource consumption as shown in Figure 2. The conditions associated with `ViewMailServer` specify that (i) the node should have enough capacity to serve incoming requests, (ii) the number of incoming requests should not exceed a certain maximum, and (iii) the component should be able to forward the RRF portion of requests to the required interfaces. The effects of deploying the `ViewMailServer` component are to decrease the node’s CPU capacity and the number of requests to the implemented interface.

Link crossing behavior. The 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 as a result of this operation. For example (see Figure 2), 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 the smaller of the original bandwidth and the total size of processed requests.

CPP goal. In the simplest case, the goal is to put a component of a given type onto a given node. For example, the goal in Figure 1 is to place `MailClient` on node N_0 . Other goals can include, for example, delivering a particular set of interfaces to a given node; this can be useful for repairing deployments when network resource availability changes.

The above model of the CPP is very flexible and allows the expression of a variety of application properties and requirements. In particular, most models we have found in literature can be captured in our formalism.

4 Solving the CPP

Figure 3 describes the structure of our planning module. The compiler module transforms a framework-specific representation of the CPP 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.

4.1 Compiling the CPP into a planning problem

The CPP can be mapped to an AI planning problem in the following way. The state of the world is

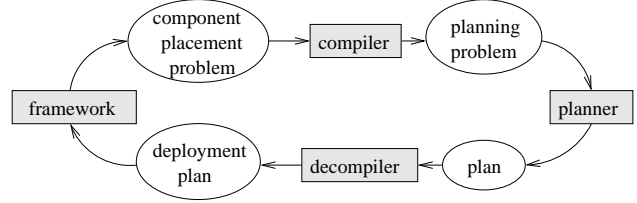


Figure 3. Process flow graph for solving CPP.

described by the network topology, the existence of interfaces on nodes (Boolean values), and the availability of resources (real values). There are two operators: `pl<component>(?n)` places a component on a node, and `cr<interface>(?n1, ?n2)` sends an interface across a link.

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 (propositions) that need to be true for the operator to be applicable (line 2);
- resource preconditions described by arbitrary functions that return Boolean values (line 3-6);³
- logical effects, i.e. a set of logical variables made true by an application of the operator (line 7);
- resource effects represented by a set of assignments to resource variables (lines 8-16).³

For example, the following schema describes the placing of the `ViewMailServer` (VMS) component on a node. The preconditions result from the conditions in Figure 2 and the fact that `MailServerInterface` (MSI) is a required interface. The effects come from the effects section of Figure 2, with MaxReq providing the upper bound on the NumReq parameter of the implemented interface.

```

1 plVMS(?n: node)
2 PRE: avMSI(?n)
3     cpu(?n) > MSIMaxReq*MSIReqCPU
4     numReq(MSI, ?n) > MSIMaxReq*MSIRRF
5     sec(MSI, ?n) = True
6     trust(MSI, ?n) > 5
7 EFF: avMSI(?n), plVMS(?n)
8     numReq(MSI, ?n) :=
9         MIN(numReq(MSI, ?n) / MSIRRF,
10            MSIMaxReq,
11            cpu(?n) / MSIReqCPU)
12     cpu(?n) := cpu(?n) -
13         numReq(MSI, ?n)*MSIRRF/MSIReqCPU
14     sec(MSI, ?n) := True
15     trust(MSI, ?n) := ntrust(?n)
16     reqSize(MSI, ?n) := 1000

```

³Sekitei currently does not support formulae involving parameters of implemented interfaces, and instead generates a conservative solution by using upper bounds on values of such parameters.

Given the operator definition 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 hard (PSPACE-complete), and complete algorithms, i.e. those that always find a solution if one exists, usually do not scale well. 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 CPP, scalability concerns stem from two sources: the size of the network, and the number of components, both affecting 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 multiple AI planning techniques and exploits the problem structure to drastically reduce the search space.

The algorithm uses two data structures: a *regression graph* (RG) and a *progression graph* (PG). RG contains operators relevant for the goal. An operator is *relevant* if it can participate in a sequence of actions reaching the goal, and is called *possible* if it belongs to a subgraph of RG rooted in the initial state. PG describes all world states *reachable* from the initial state in a given number of steps. Only possible operators of the RG are used in construction of the PG. The Sekitei algorithm consists of four phases shown in Figure 4 and described below.

Regression phase. The regression phase considers only logical preconditions and effects of operators in building the RG, an optimistic representation of all operators that might be useful for achieving the goal. RG contains interleaving facts and operator levels, starting and ending with a fact level, and 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 .

RG is initially constructed until the goal becomes possible, but may be extended if required. Figure 5 shows the RG for the problem presented in Section 2.3. Bold, solid, and dashed lines correspond to possible subgraphs with 3, 4, and 5 steps respectively.

Progression phase. RG provides a basis for the second phase of the algorithm, the construction of the progression graph. PG also contains interleaving operator and fact levels, starting and ending in a fact level. In addition, this graph contains information about mutual exclusion (mutex) relations [14], 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). Because of this reason, the PG is less optimistic than the RG. Figure 5(right) shows the PG corresponding to the RG in Figure 5(left), which is constructed as described below. (Straight lines show relations between propositions and operators, the dotted arc corresponds to a mutex relation.)

- Fact level 0 contains facts true in the initial state.
- 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 (marked with square brackets in the figure).
- For each of the possible operators contained in the corresponding layer of the RG, an operator node is added to the PG 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.

It is possible that the last level of the PG does not contain the goal, or some of the goal propositions are mutually exclusive. In this case a new step is added to the RG, and the PG is reconstructed.

Plan extraction phase. If the PG contains the goal and it is not mutex, then the plan extraction phase is started. This phase exhaustively searches the PG [1], using a memoization technique to prevent reexploration of bad sets of facts in

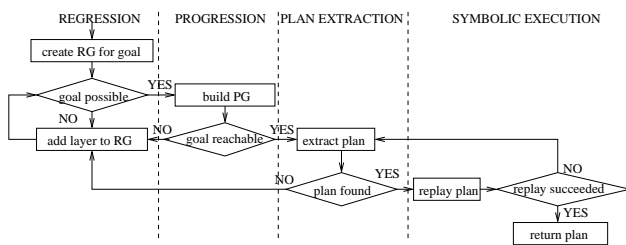


Figure 4. The algorithm. RG stands for “regression graph”, PG for “progression graph”

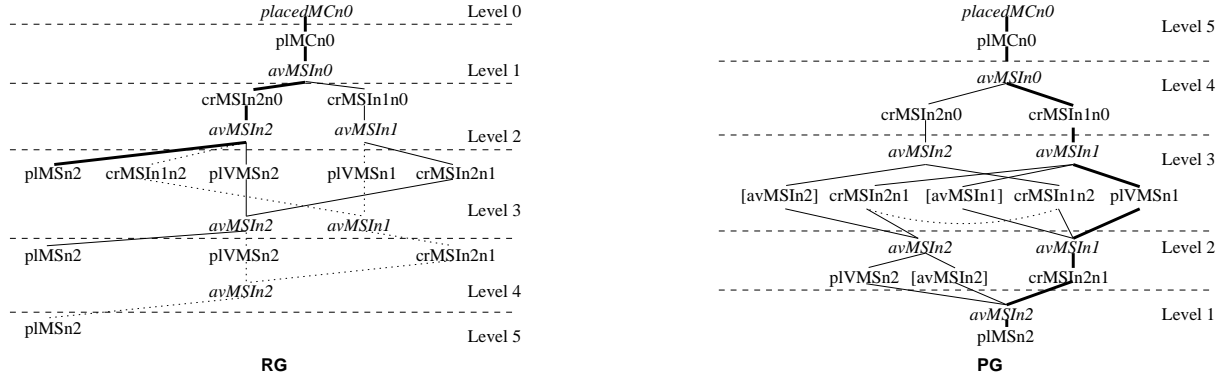


Figure 5. Regression and progression graphs.

subsequent iterations. The extracted plan is marked in bold lines in Figure 5(right).

Symbolic execution. Our work supports arbitrary monotonic functions in resource preconditions and effects. For this reason, symbolic execution is the only 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 modified according to the effect assignments. Note that correctness of the logical part of the plan is guaranteed by the previous phases; here, only resource conditions need to be checked.

4.3 Decompileation

The plan is a sequence of $pl\langle component \rangle\langle node \rangle$ and $cr\langle interface \rangle\langle from \rangle\langle to \rangle$ operators. In addition, information about logical support is easily extractable from the plan, e.g., that operator $plMSn2$ produces proposition $avMSIn2$ required by $crMSIn2n1$. Given this information, it is straightforward to obtain a framework-specific deployment plan, which consists of $(component, node)$ pairs and linkage directives, e.g. $(MS, n2, MSI, VMS, n1)$ (send the MailServerInterface implemented by the MailServer component located on node $n2$ to the ViewMailServer component on node $n1$).

5 Scalability analysis

We built a Java-based implementation of Sekitei to characterize the run time and nature of deployments it produces for different application behaviors and network conditions. The 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 [4] to generate eight different networks N_k (for different $k \in \{22, 33, \dots, 99\}$ nodes). Each topology simulates a WAN formed by high speed and secure stubs connected by slow and insecure links. The initial

topology configuration files (.alt) were augmented with link and network properties using the Network Editor tool [13].

The performance of the planner was evaluated using two applications — the **mail service** described in Section 2.3, and a **webcast** service, consisting of a Server that produces 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. Interfaces in the mail server application are characterized by a set of application-specific properties: Trust and Sec. The goal in both applications is to locate the client components on specific nodes. In both cases, the “best” deployment is 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 service application, is conducted as follows. For each network topology N_k , where $k \in 22, 33, \dots, 99$, 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 in Figure 6 represent the time needed

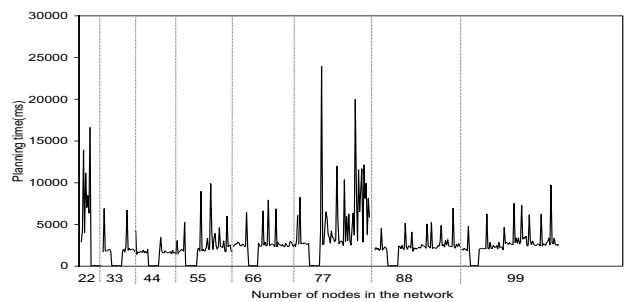


Figure 6. Planning under various conditions.

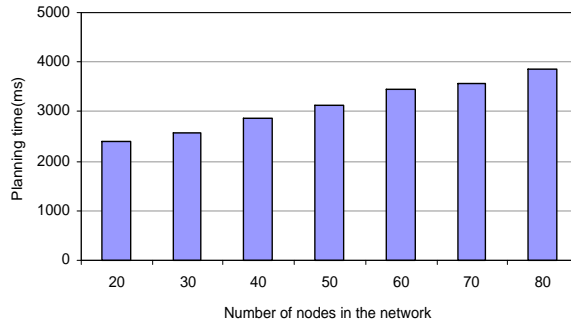
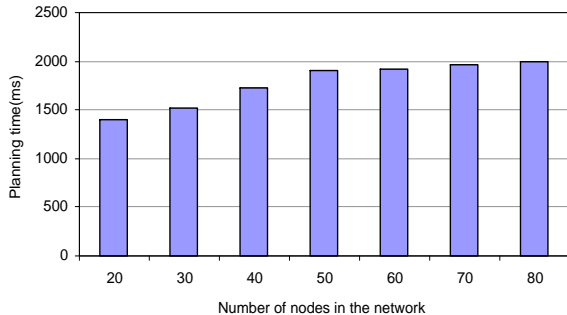


Figure 7. Scalability w.r.t. network size for the mail service (left) and webcast (right) applications.

to find a valid plan, and correspond to the following cases. When the client and the server are located in the same stub, the algorithm essentially finds the shortest path between two nodes, which takes a very short time.⁴ 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 w.r.t. 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} network topology 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 in Figure 7 (left), 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, without additional preprocessing.

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 planner behavior is not negatively affected by DAG-like structures, we generated deployments for the webcast service (the DAG structure arises because of splitting and merging the 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 (right) illustrates the run-

ning time of the algorithm as a function of the network size and validates our assertion.

Experiment 4: Scalability w.r.t irrelevant components.

To analyze the scalability of the planner when the application framework consists of a large number of components, we classify components into three categories: (i) absolutely useless components that can never be used in any application configuration; (ii) components useless given availability of interfaces in the network, and (iii) useful components, i.e., those that implement an interface relevant for achieving the goal and whose required interfaces are either present or can be provided by other useful components.

Figure 8 (left) shows the performance of the planner in the presence of irrelevant components. The two plots correspond to two situations: the mail service application augmented first with ten absolutely useless components, and then with ten components that implement interfaces meaningful to the application, but require interfaces that cannot be provided. The absolutely useless components are rejected by the regression phase of the algorithm and do not affect its performance at all.⁵ 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 running time increases as a result of processing these components in the first phase (polynomial in the number of components).

Experiment 5: Scalability w.r.t. relevant components.

Figure 8 (right) shows the planner performance with increasing number of useful components. The plots correspond to four cases. `5 comp` represents the first experiment on the original N_{99} topology, where all five components of the mail service may need to be deployed. The `4 comp`, `3 comp`, and `2 comp` cases represent situations where the network properties are modified such that all links become fast (i.e., `ViewMailServer` is not needed), secure (i.e., the `Encryptor/Decryptor` pair is not needed), or both secure

⁴The algorithm does not distinguish any special cases. “The shortest path” is only a characterization of the result.

⁵Slight fluctuations are a result of artifacts such as garbage collection.

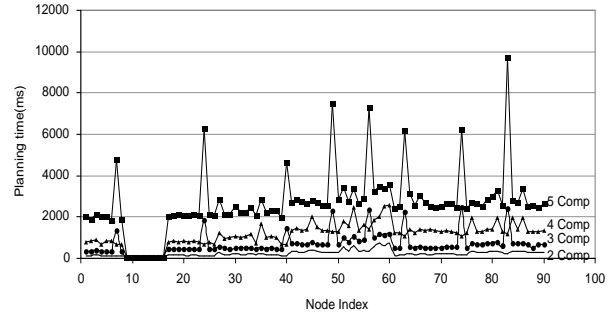
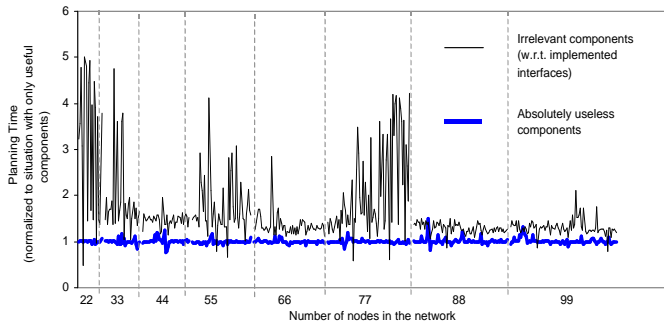


Figure 8. Scalability w.r.t. increasing number of irrelevant (left) and relevant (right) components.

and fast (i.e., only MailClient and MailServer need to be deployed) respectively.

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. Larger numbers of useful components increase the branching factor of PG, and therefore the 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. However, as we show below, new components can be added quickly.

Experiment 6: Reusability of existing deployments. In practical scenarios, by the time a new client requests a service, the network may already contain some of the required components. To see how the planning time is affected by reuse of existing deployments, we ran the following experiment. Starting with the webcast application and the N_{99} topology where the Server was present on a fixed node, we analyzed the planning costs for the goal of putting the Client on each of the network nodes in turn. The x-axis in Figure 9 represents the order in which the nodes were chosen. The network state is saved between the runs, so that clients can join existing paths. We assume that clients are using exactly the same datastream, and there is no overhead

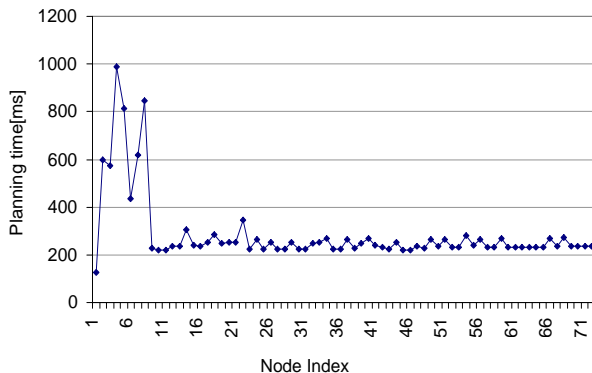


Figure 9. Reuse of existing deployments.

for adding a new client to a server.

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 in Figure 9), 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.

6 Limitations and future work

The large run-times for the N_{22} and N_{77} networks in Experiment 1 and the run-time increase in Experiment 5 can both be explained by the fact that many resource conflicts are identified only during the last phase of the algorithm. The two networks above have a bigger number of low-bandwidth insecure links between stubs as compared to the others. Because of this, the algorithm constructs and checks many logically correct plans that fail during symbolic execution due to resource restrictions. Currently, we are investigating the memoization of intermediate results to address this issue. Preliminary results look promising, however, space considerations prevent a detailed discussion.

The current Sekitei implementation does not take into consideration the actual load on components, e.g. the number of clients connected to a server. One way of capturing such incremental resource consumption in our current model is by introducing artificial components that can support a limited number of additional clients. We are exploring more general schemes, including changing the formulae describing component placement to consider parameters of implemented interfaces (as opposed to their upper bounds).

We also plan to extend Sekitei to support incremental replanning in case of a change in resource availability, and to support decentralized planning. The latter stems from the observation that is desirable for each administrative domain to have its own planner, which plans for nodes in its domain collaborating only when necessary.

In addition to improving the presented four-phase algorithm, we also plan to evaluate the effectiveness of other ap-

proaches for solving the CPP (to address the few cases that Sekitei does not handle well). In particular, the progression phase of the algorithm can be replaced with compilation into an optimization problem, or a completely different algorithm can be constructed based on causal link planners. Both these approaches will require putting tighter restrictions on the form of expressions used in preconditions and effects. The right balance between the expressiveness of the expressions and the performance of the algorithm is an interesting long-term research question.

7 Conclusions

We have presented a general model and algorithm for the component placement problem (CPP) arising in most component-based frameworks. Our model allows 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 expressiveness in both the problem specification and the plans that can be generated, supporting complex application structures and general expressions in resource preconditions and effects. As demonstrated by the experiments presented in Section 5, Sekitei is capable of identifying the relevant information and scales well with respect to network size and the number of application components. 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 being integrated into the Smock framework [9], which would allow us to test the performance of deployed plans.

Acknowledgements

This research was sponsored by DARPA agreements N66001-00-1-8920 and N66001-01-1-8929; by NSF grants CCR-9876128, CCR-9988176, and IIS-0097537; and Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA, SPAWAR SYSCEN, or the U.S. Government.

References

- [1] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.
- [2] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *ECP*, 1999.
- [3] F. Bustamante and K. Schwan. Active Streams: An approach to adaptive distributed systems. In *HotOS-8*, 2001.
- [4] K. Calvert, M. Doar, and E. Zegura. Modeling Internet topology. *IEEE Communications Magazine*, 35(6):160–163, June 1997.
- [5] M. B. Do and S. Kambhampati. Solving planning-graph by compiling it into CSP. In *AIPS*, pages 82–91, 2000.
- [6] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl. J. of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.
- [7] I. Foster, A. Roy, and V. Sander. A quality of service architecture that combines resource reservation and application adaptation. In *IWQOS*, 2000.
- [8] X. Fu, W. Shi, A. Akkerman, and V. Karamcheti. CANS: Composable, Adaptive Network Services infrastructure. *USITS-3*, 2001.
- [9] A. Ivan, J. Harman, M. Allen, and V. Karamcheti. Partitionable Services: A framework for seamlessly adapting distributed applications to heterogenous environments. In *HPDC-11*, 2002.
- [10] H. Kautz and B. Selman. Planning as satisfiability. In *ECAI*, 1992.
- [11] H. Kautz and B. Selman. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *AIPS*, 1998.
- [12] H. Kautz and J. Walser. Integer optimization models of AI planning problems. *Knowledge Engineering Review*, 15(1):101–117, 2000.
- [13] T. Kichkaylo and A. Ivan. Network EDitor. <http://www.cs.nyu.edu/pdsg/projects/partitionable-services/ned/ned.htm>, 2002.
- [14] J. Koehler. Planning under resource constraints. In *ECAI*, 1998.
- [15] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an ADL subset. In *ECP*, 1997.
- [16] J. Lopez and D. O’Hallaron. Support for interactive heavy-weight services. In *HPDC-10*, 2001.
- [17] D. McDermott. Using regression-match graphs to control search in planning. *Artificial Intelligence*, 109(1-2):111–159, 1999.
- [18] Microsoft Corporation. Microsoft .NET. <http://www.microsoft.com/net/default.asp>.
- [19] Object Management Group. Corba. <http://www.corba.org>.
- [20] J. Penberthy and D. Weld. Temporal planning with continuous change. In *AAAI*, 1994.
- [21] J. S. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *KR*, 1992.
- [22] P. Reiher, R. Guy, M. Yarvis, and A. Rudenko. Automated planning for open architectures. *OPENARCH*, 2000.
- [23] S. Gribble et al. The Ninja architecture for robust Internet-scale systems and services. *Computer Networks*, 35(4):473–497, 2001.
- [24] Sun Microsystems, Inc. Java(TM) 2 platform, Enterprise Edition.
- [25] S. Wolfman and D. Weld. Combining linear programming and satisfiability solving for resource planning. *Knowledge Engineering Review*, 2000.
- [26] D. Zhou and K. Schwan. Eager Handlers - communication optimization in Java-based distributed applications with reconfigurable fine-grained code migration. In *3rd Intl. Workshop on Java for Parallel and Distributed Computing*, 2001.